RWTH Aachen

Department of Computer Science
Technical Report

Model Checking of Software for Microcontrollers

Bastian Schlich

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2008-14

RWTH Aachen · Department of Computer Science · June 2008

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/`

Model Checking of Software for Microcontrollers

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
Bastian Schlich
aus
Essen/Ruhr

Berichter: Professor Dr.-Ing. Stefan Kowalewski
Professor Dr. Jan Peleska

Tag der mündlichen Prüfung: 04.06.2008

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.

Abstract

Software of microcontrollers is getting more and more complex. It is mandatory to extensively analyze their software as errors can lead to severe failures or cause high costs. Model checking is a formal method used to verify whether a system satisfies certain properties.

This thesis describes a new approach for model checking software for microcontrollers. In this approach, assembly code is used for model checking instead of an intermediate representation such as C code.

The development of [MC]SQUARE, which is a microcontroller assembly code model checker implementing this approach, is detailed. [MC]SQUARE has a modular architecture to cope with the hardware dependency of this approach. The single steps of the model checking process are divided into separate packages. The creation of the states is conducted by a specific simulator, which is the only hardware-dependent package. Within the simulator, the different microcontrollers are modeled accurately.

This work describes the modeling of the ATMEL ATmega16 microcontroller and details implemented abstraction techniques, which are used to tackle the state-explosion problem. These abstraction techniques include lazy interrupt evaluation, lazy stack evaluation, delayed nondeterminism, dead variable reduction, and path reduction. Delayed nondeterminism introduces symbolic states, which represent a set of states, into [MC]SQUARE while still explicit model checking techniques are used. Thus, we successfully combined explicit and symbolic model checking techniques.

A formal model of the simulator, which we developed to prove the correctness of abstraction techniques, is described. In this work, the formal model is used to show the correctness of delayed nondeterminism.

To show the applicability of the approach, two case studies are described. In these case studies, we used programs of different sizes. All these programs were created by students in lab courses, during diploma theses, or in exercises without the intention to use them for model checking.

Acknowledgments

First and foremost, I would like to thank Professor Dr.-Ing. Stefan Kowalewski for giving me the opportunity to join his group to write my Dissertation thesis. He provided me great degrees of freedom to find my own topic and assisted me making important decisions. He was always willing to listen to my problems and gave me many useful advices.

I thank Professor Dr. Jan Peleska for his willingness to be part of my Promotionskomission. Moreover, I thank him for the invitations to Bremen during which we had many fruitful discussions.

I sincerely thank all members of Informatik 11 for the friendly working atmosphere, which contributed a lot to this thesis. Particularly, I have to thank my friends and colleagues Daniel Klünder, Falk Salewski, and Dirk Wilking for many fruitful discussions and for answering a pledora of questions both related and unrelated to my Dissertation thesis. Furthermore, I thank Dr. Carsten Weise for many useful hints and advices during the final phase of my thesis. I thank my students Lucas Brutschy, Eduard Feicho, Dominique Gückel, Volker Kamin, Jann Löll, Michael Rohrbach, Florian Scheuer, and John Schommer for helping me to implement [MC]SQUARE.

I gratefully thank Dr. Thomas Noll for his support during the last year. He helped me to get a more formal insight of delayed nondeterminism and gave me many useful advices for my thesis. He always took the time to answer my numerous questions.

I thank Dr. Ralf Huuck for inviting me to Australia. We had many interesting discussions and his pragmatism helped me to solve some of the problems that arise when writing a Dissertation thesis. Furthermore, I thank all colleagues of the Managing Complexity program at the National ICT Australia in Sydney. Especially, I thank Timothy Bourke, Jörg Brauer, Dr. Ansgar Fehnker, Dr. Gerwin Klein, Rafal Kolanski, and Harvey Tuch for many discussions and their valuable comments about my thesis.

I appreciate Dr. Michael Weber's contribution in the initial phase of my Dissertation thesis. He helped me discovering the peculiarities of assembly code.

Last but not least, I would like to thank my family. They always supported me and helped my to find my way. Without them, this thesis would not have been possible.

Bastian Schlich
Aachen, June 2008

Contents

1 Introduction **1**
- 1.1 Objectives . . . 2
- 1.2 Contributions . . . 3
- 1.3 Outline . . . 4
- 1.4 Bibliographic Notes . . . 4
- 1.5 Notation . . . 5

2 Preliminaries **7**
- 2.1 Computation Tree Logic . . . 7
- 2.2 Model Checking . . . 9
  - 2.2.1 Explicit vs. Symbolic Model Checking . . . 10
  - 2.2.2 Global vs. Local Model Checking . . . 11
  - 2.2.3 Counterexamples & Witnesses . . . 11
- 2.3 Static Analysis . . . 11

3 Model Checking Software for Microcontrollers **13**
- 3.1 Model Checking C Code . . . 13
- 3.2 Model Checking Assembly Code . . . 16
- 3.3 Requirements for an Assembly Code Model Checker . . . 18

4 [mc]square **21**
- 4.1 Features . . . 21
- 4.2 Architecture . . . 23
- 4.3 Evaluation . . . 26

5 State Space Building in [mc]square **29**
- 5.1 Simulator Overview . . . 30
- 5.2 Simulator State . . . 31
- 5.3 Microcontroller . . . 32
  - 5.3.1 Core . . . 33
  - 5.3.2 Memories . . . 34
  - 5.3.3 Stack . . . 38
  - 5.3.4 Interrupts . . . 39
  - 5.3.5 I/O Ports . . . 40

5.3.6 Timers . . . 44
5.4 Program . . . 48
5.5 Instruction Simulator . . . 50
5.6 Determinizer . . . 53
5.6.1 Determinizer Overview . . . 53
5.6.2 Immediate Instantiation . . . 56
5.6.3 Delayed Nondeterminism . . . 58
5.7 Formal Model of the Simulator . . . 62
5.7.1 Handlers and Guarded Assignments . . . 63
5.7.2 Modeling the ATMEL ATmega16 . . . 65
5.7.3 Coping with Nondeterminism . . . 67
5.7.4 Establishing Correctness . . . 70
5.8 Related Work . . . 70
5.8.1 Model Checking Machine Code . . . 71
5.8.2 Delayed Nondeterminism . . . 73

6 Static Analysis in [mc]square 75
6.1 Challenges in Static Analysis of Assembly Code . . . 75
6.2 Static Analyzer Overview . . . 77
6.3 Static Analyses . . . 79
6.3.1 Control Flow Analysis . . . 79
6.3.2 Live Variables Analysis . . . 81
6.3.3 Reaching Definitions Analysis . . . 84
6.3.4 Stack Analysis . . . 86
6.3.5 Global Interrupt Flag Analysis . . . 88
6.4 Abstraction Techniques . . . 90
6.4.1 Dead Variable Reduction . . . 90
6.4.2 Path Reduction . . . 91
6.5 Related Work . . . 93

7 Model Checking in [mc]square 97
7.1 Model Checker . . . 97
7.1.1 Local Model Checking . . . 99
7.1.2 Global Model Checking . . . 103
7.1.3 Invariant Checking . . . 106
7.1.4 Counterexample Generation . . . 107
7.2 State Space . . . 109
7.3 Influence on Validity of Formulas . . . 112
7.4 Related Work . . . 113

8 Case Studies 115

8.1 Effects of Abstraction Techniques on Different Programs 115
8.1.1 Execution 115
8.1.2 Evaluation 120
8.2 An Automotive Microcontroller Application 121
8.2.1 Application 122
8.2.2 Execution 122
8.2.3 Evaluation 129

9 Conclusion **131**
9.1 Discussion 131
9.2 Future Work 133

1 Introduction

Embedded systems are widely used in our daily life. They are, for example, employed in airplanes, cars, mobile phones, and household appliances. Embedded systems consist of hardware and software. The importance of the software on these devices is increasing because more and more functionality is implemented within the software and no longer realized by the hardware. The software of embedded systems has to be tested extensively and validated because errors in the software may lead to severe or even fatal events, as in case of the Ariane 5 disaster [76], or high costs and loss of reputation, as in the case of the Toyota Prius bug [68].

Many embedded systems are based on microcontrollers, which are special purpose computers on a single chip. They are often specifically developed for single applications. The software for microcontrollers is mostly written in C or in assembly language.

Removing errors in microcontroller software is difficult in the field because deploying the updates is complicated and cost-intensive. In contrast to software run on general-purpose personal computers, it is often not possible for users to update microcontroller software themselves. For example, the software of a car can only be updated in a garage. In some cases, the software cannot be updated at all due to the type of memory used to store the program. This may render the affected microcontrollers useless.

Full or extensive testing of microcontroller software is often not possible because it is too time-consuming for the desired time to market or too expensive for the specific product. Also, testing alone is not sufficient for safety critical systems. There are standards such as IEC61508 [66] that strongly recommend the application of formal methods if a system requires a certain safety level. An example for safety levels are the safety integrity levels defined in the IEC61508 standard. The formal methods mentioned in this standard include verification techniques such as model checking [5, 15, 33, 78, 108] and automatic theorem proving.

Model checking is able to automatically verify systems. It uses an exhaustive search over all reachable system states to check whether the system (model) satisfies a given property (specification). If the system does not satisfy the property, the model checker provides a counterexample, which details the error. Model checking can be applied to systems that do not have a full specification.

Automatic theorem proving is a verification technique used to prove the correctness of a system. Depending on the complexity of the problem, an automatic theorem

prover is able to prove that something is true, but it may be unable to disprove something, which is not true. In this case it is possible that the theorem prover does not terminate. Often, the theorem prover requires user interaction and hence, developers have to be educated to use theorem provers. In contrast, developers familiar with simulators can use model checkers more easily as their application is similar [33].

Model checking is used by companies such as AMD, Infineon, Intel, and Siemens for the analysis of hardware systems. After the Pentium FDIV bug [128], hardware vendors realized that errors in designs can cause major losses and hence started to use model checking and other formal methods to analyze their systems. Also, software companies such as Microsoft are beginning to use model checking for the verification of crucial parts of their software.

Industries such as the automotive industry are interested in using model checking for the analysis of software for microcontrollers. In the development of software for microcontrollers, however, model checkers are far from being well established. Besides the state-explosion problem, one of the limiting factors is that most of the available model checkers use custom input models. Consequently, developers have to remodel specifications and implementations to feed them into the model checker. They have to do this every time the system is changed. This is usually not considered to be worth the effort as it is time-consuming and error-prone.

Existing model checkers are not able to handle all constructs needed to check microcontroller programs out-of-the-box [101, 104]. Due to short market cycles in many microcontroller software projects, developers often do not have enough time and training to create models of their software for model checking. Furthermore, many model checkers are not intuitively usable and differ from development tools usually applied by developers. Therefore, model checking is not widely used in microcontroller software projects.

1.1 Objectives

Our objective is to develop an approach for verifying microcontroller software. We want to implement this approach in a model checker, which should be usable by developers of microcontroller software. The approach should work on source code out-of-the-box. That is, the user should be able to check the source code without preprocessing or manually annotating the code or creating a model in a custom language.

To achieve our goals, we first have to decide whether to model check C or assembly code. Furthermore, we need to find out whether we can implement our approach by extending an existing model checker or if it is beneficial to develop a new model checker. We have to apply techniques to mitigate the state-explosion problem, which

model checking can suffer from. For this purpose, we want to utilize static analysis and employ different abstraction techniques. Furthermore, we want to find out whether accurately modeling particular microcontrollers enables us to develop more specific abstraction techniques. Finally, we have to develop a formal model to prove the validity of such abstraction techniques.

Since developers of microcontroller software should apply the model checker themselves, we must hide its internal details from them and provide a GUI that is similar to their other tools.

1.2 Contributions

The main contributions of this thesis supporting our objectives are as follows.

- We have developed an approach to model check microcontroller assembly programs and implemented this approach within our model checker [MC]SQUARE. [MC]SQUARE uses a special *simulator* for microcontroller assembly code to build the state space. Within this simulator, we have accurately modeled the microcontroller to support reasoning about microcontroller features such as registers, I/O registers, and the values of variables. To tackle the state-explosion problem, we have implemented various *abstraction techniques* in the simulator.

- We have developed a new abstraction technique called *delayed nondeterminism*, which reduces the size of the state space. It introduces *lazy states* into our model checker. A lazy state is a state that consists of explicit and symbolic parts and hence represents a set of states. As our model checking algorithms are explicit, the symbolic parts, which are induced by nondeterminism, are lazily resolved only when the model checker accesses them. Using this technique, we have combined explicit and symbolic techniques in [MC]SQUARE.

- We have adapted two abstraction techniques, namely *dead variable reduction* and *path reduction*, to be applicable in model checking of microcontroller assembly code. These two techniques use *static analyses*, which were initially not directly applicable to microcontroller assembly code. We have adapted these static analyses and improved their accuracy by using *abstract interpretation*.

- Within the simulator, we have realized two abstractions techniques that exploit the specifics of the microcontroller. One technique called *lazy stack evaluation* deals with the contents of the stack. The other technique called *lazy interrupt evaluation* addresses the invocation of interrupt handlers.

- We have developed a *formal model* of our microcontroller simulator, which can be used to prove the validity of abstraction techniques implemented in [MC]SQUARE. This formal model can also be used as an intermediate representation for the implementation of simulators for other microcontrollers.
- Using our formal model, we have proven that the abstraction technique called delayed nondeterminism preserves a simulation relation between the concrete and the abstract state space.
- We have conducted two case studies showing that model checking can indeed be used to analyze microcontroller assembly programs of a certain, reasonable size.

1.3 Outline

Chapter 2 describes preliminaries used throughout the thesis. The next chapter discusses the choice of whether to model check C or assembly code and provides requirements for a new model checker for microcontroller assembly code.

Chapter 4 gives an overview of our assembly code model checker [MC]SQUARE and depicts its architecture. The subsequent three chapters detail the four main packages of [MC]SQUARE. First, the *Simulator* package, which is used to create states, is described. Then, the *Static Analyzer* package, which conducts the static analyses used by some abstraction techniques, is presented. Finally, in Chapter 7, the *Model Checker* package, which performs the actual model checking, and the *State Space* package, which manages the states, are detailed.

Chapter 8 presents two case studies, which demonstrate the usage of [MC]SQUARE. The first case study presents the effects of the different abstraction techniques implemented in [MC]SQUARE. The second case study details the way the user applies [MC]SQUARE for model checking. Chapter 9 concludes the thesis and gives directions for future improvements. We cover related work within each chapter.

1.4 Bibliographic Notes

Some parts of this thesis are based on work that we described in earlier publications. The survey of the C code model checkers, which is described in Sect. 3.1, is summarized by us in two papers [101, 104]. We described a preliminary version of the architecture of [MC]SQUARE presented in Sect. 4.2 elsewhere [103]. The idea to use a simulator to build the state space for model checking is sketched in two publications [102, 103]. We outlined the handling of nondeterminism detailed in Sect. 5.6 in a paper [60]. The formal model presented in Sect. 5.7 is described by us in two publications [60, 88]. The static analyses detailed in Chap. 6 are summarized

in a paper [107]. We also applied the programs used in the case study shown in Sect. 8.1 elsewhere [60, 88, 101–104, 107]. The case study presented in Sect. 8.2 is published in a paper [106]. The individual chapters of this thesis detail related work.

1.5 Notation

In this thesis, we use the following notations. Names of tools are typeset in a capitalized font like ExampleTool. Names of packages and classes are set in a slanted font, for example, *ExampleClass*. Source code is typeset in a sans-serif font, for instance, exampleMethod(). Ideas and important terms are emphasized such as *example idea*.

2 Preliminaries

This chapter presents some preliminaries, which are used throughout this thesis. The first section describes the computation tree logic. Section 2.2 presents the basic idea of model checking, gives a basic classification of different model checking algorithms, and explains the terms counterexample and witness. The last section gives an introduction into static analysis.

2.1 Computation Tree Logic

Propositional logic only allows to reason about states. In reactive systems we are also interested in describing the sequence of states. Temporal logic [93] extends propositional logic to allow the reasoning about the sequence of states. Examples for a temporal logic are *Linear Temporal Logic* (*LTL*) [92, 93] and *Computation Tree Logic* (*CTL*) [13, 31, 42]. LTL formulas describe properties of the set of all paths. In contrast, CTL formulas describe properties of computation trees. That is, these formulas can describe properties of single paths within these trees. In the following we describe CTL in detail. This section is primarily based on the book by Clarke et al. [33].

CTL is a sublogic of CTL* [31, 32, 43]. In CTL*, there are path quantifiers and temporal operators. Path quantifiers include **A** and **E**. **A** means that from a certain state on all paths satisfy some property. **E** means that at least one path leaving the current states satisfies some property. The temporal operators describe the properties of a path through the tree:

- Next (**X**) means that the property has to hold in the next state.
- Finally (**F**) requires that the property holds in one state on the path.
- Globally (**G**) means that the property holds in all states on the path.
- Until (**U**) requires that the first property holds until the second property becomes true.

In CTL* there is no restriction on how to combine temporal operators and path quantifiers. In CTL, a temporal operator always must be preceded by a path quantifier. In CTL there are two kinds of formulas: state formulas and path formulas. Given a set AP of atomic propositions, the syntax of CTL is defined as follows:

- If $p \in AP$, then p is a state formula.
- If f_1 and f_2 are state formulas, then $\neg f_1$, $f_1 \vee f_2$, and $f_1 \wedge f_2$ are state formulas.
- If f_1 is a path formula, then $\mathbf{E}\, f_1$ and $\mathbf{A}\, f_1$ are state formulas.
- If f_1 and f_2 are state formulas, then $\mathbf{X}\, f_1$, $\mathbf{F}\, f_1$, $\mathbf{G}\, f_1$, and $f_1\, \mathbf{U}\, f_2$ are path formulas.

The semantic of CTL can be defined with respect to a Kripke structure. A Kripke structure is triple $\langle S, R, L \rangle$, where:

- S is the set of states,
- $R \subseteq S \times S$ is the total transition relation and
- $L : S \rightarrow 2^{AP}$ is a function that labels each state with a set of atomic propositions.

Details about the semantics of CTL formulas are given elsewhere [5, 15, 33, 57].

In this thesis, we need the terms subformulas, proper subformulas, length of a formula, and the subformula evaluation order. These terms are defined in the following. All these definitions are taken from Heljanko [57].

Definition 1 (Subformulas)**.** Let f be a CTL formula. The set of *subformulas* $\mathrm{sub}(f)$ of f is recursively defined as follows:

1. If f is an atomic proposition $p \in AP$, then $\mathrm{sub}(f) = \{f\}$.
2. If f is of the form $\neg f_1$, $\mathbf{EX}\, f_1$, $\mathbf{EF}\, f_1$, $\mathbf{EG}\, f_1$, $\mathbf{AX}\, f_1$, $\mathbf{AF}\, f_1$, or $\mathbf{AG}\, f_1$, then $\mathrm{sub}(f) = \{f\} \cup \mathrm{sub}(f_1)$.
3. If f is of the form $f_1 \vee f_2$, $f_1 \wedge f_2$, $\mathbf{E}[f_1\, \mathbf{U}\, f_2]$, or $\mathbf{A}[f_1\, \mathbf{U}\, f_2]$, then $\mathrm{sub}(f) = \{f\} \cup \mathrm{sub}(f_1) \cup \mathrm{sub}(f_2)$.

Definition 2 (Proper Subformulas)**.** Let f be a CTL formula. The set of *proper subformulas* $\mathrm{psub}(f)$ of f is defined by:

$$\mathrm{psub}(f) = \mathrm{sub}(f) \backslash \{f\}.$$

Definition 3 (Length of a Formula)**.** Let f be a CTL formula. The *length of a formula* f, $\mathrm{length}(f)$ is defined by:

$$\mathrm{length}(f) = |\mathrm{sub}(f)|,$$

where $|\mathrm{sub}(F)|$ means the cardinality of the set $\mathrm{sub}(f)$.

Definition 4 (Subformula Evaluation Order)**.** Let f be a CTL formula. We define the partial order $<_s$, which is called *subformula evaluation order*, as follows. For all CTL formulas f' and $f'' \in \text{sub}(f)$, $(f', f'') \in <_s$ (also written as $f' <_s f''$), iff $f' \in \text{psub}(f'')$ (f' belongs to the set of proper subformulas of f'').

In this thesis, we often refer to a logic called ACTL [38, 41, 52]. ACTL is a sublogic of CTL in which only the **A** path quantifier is allowed and negations are only feasible in atomic propositions. This fragment is called the universal fragment of CTL. Another sublogic of CTL is ECTL [38, 41, 52]. In ECTL formulas only the **E** path quantifier is permitted and negations are only allowed in atomic propositions. This fragment of CTL is called existential fragment of CTL.

Furthermore, we often refer to invariants. An invariant is a formula of the form $\mathbf{AG}\, f$, where f is only using operators from propositional logic and not using path operators or temporal operators.

2.2 Model Checking

Model checking [5, 15, 33, 78, 108] is a formal method for the automatic verification of systems. It uses an exhaustive search over all reachable states of the system to check whether the system (model) satisfies a given property (specification). The method was independently proposed by Clarke and Emerson [31] and by Queille and Sifakis [95]. Given a model M and a formula f, it checks whether M is a model of f ($M \models f$). That is, it checks if the system satisfies the given property.

The method is depicted in Fig. 2.1. In a first step, which is called modeling, the system description is transformed into the system model. A system description is, for example, a program written in C, Java or Assembly language. A system model is, for example, a Kripke structure, a labeled transition system, or a finite automaton. The modeling can be done manually or automatically. A manual modeling is error-prone and time-consuming, but it can help to find errors in the system description or in the specification because it complies with an intensive review of the system. When dealing with large descriptions of real systems, an automatic approach is preferable.

The requirements have to be manually formalized because they are mostly given in natural language. The result of this formalization is the formal specification given as formulas in a temporal logic such as CTL*, CTL, or LTL. The model and the specification are inputs given to the model checker.

The model checker uses an exhaustive search over all reachable states of the model to check whether the model satisfies the formula. In the end, it returns a result. The result may be that the model satisfies the formula together with a witness or that the model does not satisfy the formula together with a counterexample (see Sect. 2.2.3). Due to the state-explosion problem, it may happen that the model checker runs out of memory and does not return a result.

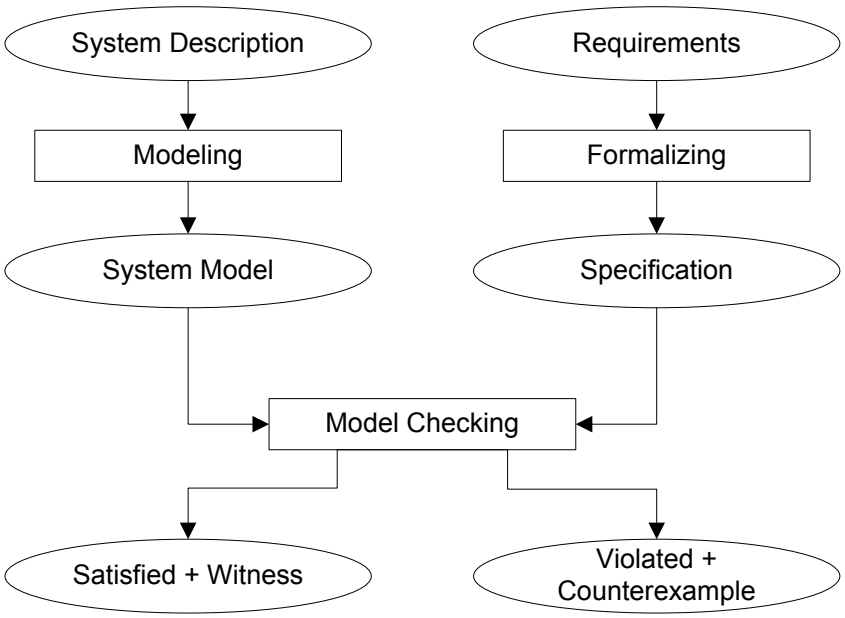

Figure 2.1: Model checking process (see Baier and Katoen [5] and Manna and Pnueli [78]).

There are different ways to conduct model checking. In the following, we describe and compare some of them. A thorough classification of different model checking techniques is given by Visser [123]. At the end of this section, we detail the differences between a counterexample and a witness.

2.2.1 Explicit vs. Symbolic Model Checking

In *explicit model checking* the state and the transitions are represented explicitly. That is, each state is stored as it is. Explicit model checking algorithms have to visit single states and work on them. In *symbolic model checking* states are not represented explicitly but implicitly. Symbolic model checking algorithms work on sets of states instead of single states. The sets of visited states and transitions is often encoded in Binary Decision Diagrams or Ordered Binary Decision Diagrams. Depending on the structure of the transition system, these representations can be very compact. However, there are cases where these representations are not effective. Hu et al. [65] show that each of these two methods has domains where it outperforms its counterpart.

2.2.2 Global vs. Local Model Checking

In *global model checking*, the complete state space is built before model checking because a global model checking algorithm evaluates the truth values of all subformulas for all states. In contrast, only the truth values of certain subformulas in certain states are evaluated in *local model checking*. A local model checking algorithm only needs to evaluate the subformulas and visit the states that are needed to evaluate the truth value of the formula in the initial states of the model. Therefore, it is possible generate the state space on-the-fly during model checking when using a local model checking algorithm.

2.2.3 Counterexamples & Witnesses

As we use CTL model checking, we define counterexamples and witnesses in reference to model checking of CTL formulas. These definitions are based on the books by Baier and Katoen [5] and Clarke et al. [33].

Definition 5 (Counterexample)**.** A *counterexample* is a part of the state space indicating why a universally quantified path formula (e.g., $\mathbf{AG}\, f$) is refuted.

Definition 6 (Witness)**.** A *witness* is a part of the state space that indicates why an existential quantified path formula (e.g., $\mathbf{EF}\, f$) is satisfied.

If a universally quantified formula is false, the model checker finds a witness for the negation of this formula. For instance, if the formula $\mathbf{AG}f$ is false, the model checker returns a witness for the formula $\mathbf{EF}\neg f$, which is a counterexample for the formula $\mathbf{AG}f$. Hence, the terms witness and counterexample can be used alternatively. Therefore, we only use the term counterexample in this thesis whenever we mean counterexample or witness.

2.3 Static Analysis

Static analysis, which is also called *program analysis*, offers techniques to statically determine at compile time an approximation of the values or behaviors observed during runtime. In static analysis, the program is not executed, but the analysis are conducted on the source code of the program. This has the advantage that static analysis can be applied to very large programs. The disadvantage is that sometimes the approximation of the values and behaviors calculated by the static analysis can be too coarse. That is, no definite conclusions can be drawn from the results of the static analysis. This section, which is mainly based on the book by Nielson et al. [87], introduces the basic principles of static analysis.

Here, we sketch two different types of static analyses: *control flow analysis* (*CFA*) and *data flow analysis* (*DFA*). CFA is used to determine how control evolves from one

program location to another program location. That is, it is used to determine the *control flow graph* of a program. The nodes in this graph are program locations, and the edges connect two program locations if it is possible for the control to get from one location to the other. In the DFA, this graph is used to determine different data flow properties. Example for DFAs are: *live variables analysis*, *reaching definitions analysis*, and *available expressions analysis*.

DFA can be conducted intraprocedural or interprocedural. In an intraprocedural analysis, the analysis is conducted for each function alone. Interactions between functions are not accounted. In an interprocedural analysis, the analysis additionally considers interactions between different functions. That is, it considers the effect of function calls and interrupts. DFAs can be expressed as so-called *data flow equations*. One way to solve these equations is the *worklist algorithm* [87] (see Sect. 6.3.2).

In *abstract interpretation* [36] a program is interpreted using an abstract semantics instead of the concrete semantics of the program. Using a sufficiently abstract semantics enables analyses that are otherwise not computable. In abstract interpretation, the abstract semantics has to be a superset (over-approximation) of the concrete semantics. Due to the abstract semantics, the program can exhibit behavior that is not possible using the concrete semantics. Thus, false alarms are possible, but if no errors are found in the abstract semantics, it is guaranteed that no errors are present in the concrete semantics.

3 Model Checking Software for Microcontrollers

This chapter details why we use assembly code instead of C code for model checking software for microcontrollers. The first section gives an overview of existing C code model checkers and explains why none of them are currently able to model check microcontroller C code out-of-the-box. After that, we explain why we decided not to check C code. The next section presents the advantages and disadvantages of using assembly code for model checking and evaluates them. The last section gives a list of requirements that a model checker for microcontroller assembly code has to fulfill to be applicable. A comparison of existing assembly code model checkers is given in Sect. 5.8.

3.1 Model Checking C Code

In almost every microcontroller software project there is a development phase in which the software exists as C code. Hence, one solution to model check software for microcontrollers is to model check the corresponding C code.

We conducted a case study [101, 104] in which we tried to model check microcontroller C programs using existing C code model checkers. Table 3.1 gives a survey of all model checkers considered in the case study. The table presents for each model checker a short description of the techniques utilized. We composed the information shown in the table during the case study. Therefore, it is possible that some techniques may have changed and some newer model checkers are not mentioned at all. We provide more details about the case study in the corresponding papers [101, 104].

All model checkers shown in the table except for StEAM are used to verify hardware-independent ANSI C software such as protocols or high-level drivers. Most of them check sequential programs, but MAGIC, KISS, and Zing are also able to handle concurrent programs. StEAM is utilized to check parallel C++ programs. Some of the model checkers shown in the table support almost full ANSI C and C++ respectively, while others restrict the set of allowed constructs due to the techniques applied in the model checking process.

Table 3.1: List of C code model checkers.

Model Checker	Institute	Model	Techniques used
BLAST [59]	UC Berkeley	C	C intermediate language (CIL) [86], control flow automaton, predicate abstraction [35, 51], counterexample-guided abstraction refinement (CEGAR) [34], theorem prover
BOOP [127]	IST Graz	C	Boolean program [8], predicate abstraction, CEGAR, theorem prover, model checking with MOPED [110]
CBMC [28]	CMU	C/C++	bounded model checking using SAT solver
FeaVer [64]	Bell Labs	C	translation into Promela, model checking with Spin [62]
FocusCheck [69]	Iowa State University	C	CIL, translation into pushdown system, constraint solver, model checking of pushdown system
F-Soft [67]	NEC	C	CIL, CFG, predicate abstraction, SAT solver, model checking with VeriSol (DiVer) [48]
MAGIC [23, 24]	CMU	C	CIL, modular verification, control flow automaton, predicate abstraction, CEGAR, theorem prover
MOPS [25]	UC Davis, UC Berkeley	C	CFG, translation into pushdown automaton, model checking of pushdown automaton
SatAbs [29, 30]	CMU	C/C++	Boolean program, predicate abstraction, CEGAR, SAT solver, model checking with SMV [79]
SLAM [9, 10]	Microsoft Research	C	Boolean program, predicate abstraction, CEGAR, theorem prover, model checking with Bebop [8]
StEAM [73, 80, 81]	University Dortmund	C++	translation into machine code, state space generation with Internet C++ Virtual Machine
KISS [94]	Microsoft Research	-	extension to SLAM to check concurrent programs
Zing [1, 2]	Microsoft Research	C	translation into Zing model, model checking with Zing model checker

Model checkers that use theorem provers suffer from three restrictions due to the underlying general-purpose theorem provers. First, only a limited number of C constructs are supported. Second, pointer arithmetic is only supported in a restricted manner. Third, possible arithmetic overflows are neglected. Model checkers utilizing SAT solvers require the model to be finite. Consequentially, recursion and dynamic memory allocations are not handled.

Approaches that translate the C code into another model or language have to cope with two problems. First, the target language has to support all constructs present in the source language, which is sometimes not possible. Second, the translation process often leads to growth of the model to be checked and hence, the state spaces created during model checking tend to be bigger. Besides, the bigger state spaces lead to longer error traces, which are more complex and hence, more difficult to understand.

Beside ANSI C language features, microcontroller C programs also comprise extra language features such as compiler-specific constructs, hardware-dependent constructs, and embedded assembly language statements. All these features are not handled by the model checkers shown in Tab. 3.1. Furthermore, microcontroller C programs often access the memory directly as certain operations of the microcontroller are controlled by special registers, which are located at fixed memory addresses (e.g., I/O registers used to read input from the environment). In contrast, direct memory accesses are reported as errors by most C code model checkers because direct accesses to memory can lead to errors in an environment where dynamic linking and loading is supported.

Listing 3.1 shows an example for a microcontroller C program that controls an automotive window lift. The program is one of the programs used in case studies described in Chap. 8, which are used to determine the overall performance of [MC]-SQUARE and compare effects of different techniques applied. At first sight, it looks like an ANSI C program. It contains function calls, assignments, if clauses, and while loops. Most variables are read and written by the program, while a couple of the variables, such as TCCR1B, are only written. These variables are used to control the microcontroller. Some of the model checkers shown in Tab. 3.1 remove or ignore variables that are only written and thus disregard important parts of microcontroller C programs.

Listing 3.2 displays the program shown in List. 3.1 after being preprocessed. This listing evidences that the window lift program is not an ANSI C program, but it is a typical microcontroller C program. The program contains embedded assembly language statements, direct memory accesses, and accesses to certain microcontroller functions. None of the model checkers described above can handle these constructs out of the box. Some produce warnings, while others just ignore these features. In case of direct memory accesses, a couple of the C code model checkers throw error messages because direct memory accesses are considered as errors in ANSI C

Listing 3.1: Part of window lift program before preprocessing.

```
...
int main (void) {
  init();                          // call initialization
  sei();
  while(1) {
    inputs = PINA & 0x0F;
    cli();
    if (direction != 5) {
      if (inputs & (1 << 1)) {// down
        if (direction != 1 && direction != 2) {
          TCCR1B = 0x00;
          TIFR = 0xFF;
          TCNT1 = 0x00;
          TIMSK = (1<<OCIE1A);
          TCCR1B = 0x05;
          direction = 1;
        }
...
```

programs running on general-purpose computers that support dynamic linking and loading. Using direct memory accesses in conjunction with dynamic linking and binding is error-prone because wrong parts of the memory can be accessed, which is an issue for code and stack safety, for example.

As these C code model checkers only support a subset of the needed constructs, it is not possible to use them out-of-the-box to model check microcontroller C programs. We tried to extend one of these C code model checkers to support all needed constructs, but we found out that the necessary changes were too costly [101, 104]. Therefore, we propose another solution.

3.2 Model Checking Assembly Code

Since we could not use existing C code model checkers to model check microcontroller C programs, we decided to use the assembly code for model checking. Assembly code is the artifact that is deployed to the microcontroller and not an intermediate representation such as C code. Therefore, model checking of assembly code has various advantages compared to model checking of C code [6, 73, 80, 82, 102, 105]:

- As the assembly code is the result at the end of the development, all errors introduced during the complete development process can be found. These

Listing 3.2: Part of window lift program after preprocessing.

```
...
int main (void) {
  init();
  __asm__ __volatile__ ("sei" ::);
  while(1) {
    inputs = (*(volatile uint8_t *)((0x19) + 0x20)) & 0x0F;
    __asm__ __volatile__ ("cli" ::);
    if (direction != 5) {
      if (inputs & (1 << 1)) {
        if (direction != 1 && direction != 2) {
          (*(volatile uint8_t *)((0x2E) + 0x20)) = 0x00;
          (*(volatile uint8_t *)((0x38) + 0x20)) = 0xFF;
          (*(volatile uint16_t *)((0x2C) + 0x20)) = 0x00;
          (*(volatile uint8_t *)((0x39) + 0x20)) = (1<<4);
          (*(volatile uint8_t *)((0x2E) + 0x20)) = 0x05;
          direction = 1;
        }
...
```

errors include:

  - compiler errors, that is, errors introduced by compiler behavior or optimizations,
  - errors introduced during post-compilation steps (e.g., insertion of instrumentation code or optimizations),
  - errors in microcontroller usage (e.g., write access to reserved registers),
  - errors that are not visible in intermediate representations such as C code (e.g., reentrance problems), and
  - hardware-dependent errors such as stack overflows and underflows.

- Programs consisting of components written in different programming languages can be verified. When model checking the source code, only single components can be verified, and for each programming language a specific model checker has to be utilized.
- Source code of the software is not required. Hence, programs that use libraries, which are not available in source code, can be analyzed.
- The model checker does not need to exploit the compiler and optimizer behavior.

- Hardware-dependent constructs can be handled.
- Assembly language statements that are embedded into the source code are considered appropriately by the model checker (i.e., not ignored as done by most C code model checkers).
- Assembly language has a clean and well documented semantics. Microcontroller vendors provide documentations describing the semantics of the different assembly language constructs.
- Assembly language statements are easier to handle than certain C constructs (e.g., pointer arithmetic or function calls via pointers).

Beside these various advantages, model checking assembly code has some disadvantages:

- A model checker for microcontroller assembly code is hardware dependent. That is, the model checker has to be adapted for each new microcontroller that should be supported.
- Since assembly code has more lines of code and involves more details than C code, the state spaces created during model checking of assembly code tend to be larger than the state spaces created during model checking of C code. This growth of the state spaces may also lead to longer counterexamples.

The advantages and disadvantages of model checking assembly code for microcontrollers lead to requirements that have to be fulfilled by a model checker to be applicable in industry. To the best of our knowledge, there was no model checker available that was able to model check microcontroller assembly code without manual preparation when we started the development of our model checker. Section 5.8 gives a detailed description of related work regarding assembly code model checking.

3.3 Requirements for an Assembly Code Model Checker

This section describes the requirements that our model checker for microcontroller assembly code has to satisfy. Table 3.2 presents the non-functional requirements (qualities)[1], and Tab. 3.3 shows the functional requirements[1].

The aim of our development is a model checker that serves two purposes. First, it is usable by developers working in industry, which are not familiar with the application of formal methods. Second, the model checker serves as a research tool that can be utilized to evaluate new algorithms and abstraction techniques. These

[1] We use the terms functional requirement and quality as defined by Bass et al. [12].

two purposes are reflected by qualities Q1 and Q2. These two qualities implicate other qualities and requirements. Extendability (Q3–Q6) in different directions is an implication of both qualities. The model checker is extendable to support new microcontrollers to be applicable in industry. Moreover, it is possible to integrate existing simulators into the model checker. To use the model checker as a research tool in academia, it is extendable to use other model checking algorithms and abstraction techniques. Since there are different operating systems used in industry and academia, the tool is executable on the three major operating systems (Q7).

Table 3.2: List of non-functional requirements.

Number	**Requirement**
Q1	The model checker is usable by developers working in industry, which are not familiar with formal methods.
Q2	The model checker is usable as a research tool in academia to evaluate new algorithms in different areas (e.g., model checking, static analysis, simulation of systems).
Q3	The model checker is extendable to support new microcontrollers.
Q4	It is possible to use external simulators to build the state space.
Q5	The model checker is extendable to support new model checking algorithms.
Q6	The model checker is extendable to support new abstraction techniques.
Q7	The model checker works on the following operating systems: Microsoft Windows, Mac OS, and Linux.

Q1 and Q2 are the driving qualities (see Bass et al. [12]) of our model checker's development. These two qualities implicate most of the functional requirements shown in Tab. 3.3. To support users of the model checker, arbitrary assembly programs (F1) for certain microcontrollers (F2 and F3) are supported without the need to manually prepare them. Debug information in Dwarf [40] or Stabs [46] format is used to map information from assembly code to C code (F4). Specifications are given in CTL (F5), and it is possible to use atomic propositions about registers, I/O registers, and variables (F6). For every program under verification, the specifications is stored in a list (F7). To minimize the size of the state space and to evaluate abstraction techniques, the model checker implements different abstraction techniques (F8). The tool provides a GUI (F9). Within this GUI, users are able to choose between various abstraction techniques, model checking algorithms, and state compression techniques (F10). Counterexamples are presented in source code (assembly and C), as a state space graph, and in the CFG of the assembly code (F11). Within all these representations, users are able to step through the counterexample and to survey in each state the values of registers, I/O registers, variables, and formulas (F12).

Table 3.3: List of functional requirements.

Number	Requirement
F1	The model checker accepts arbitrary microcontroller assembly code files given in ELF file format as input.
F2	The model checker first supports the ATMEL ATmega16 microcontroller. Other microcontrollers, such as the ATMEL ATmega128 and Infineon XC167, are added later.
F3	The model checker supports all constructs found in assembly code for the supported microcontrollers.
F4	The model checker handles debug information given in Dwarf or Stabs format to map information between assembly and C code.
F5	The model checker accepts specifications given in CTL.
F6	The model checker allows to use propositions about registers, I/O registers, and variables within the specification.
F7	For every program under verification, users are able to store a list of specifications.
F8	The model checker implements different abstraction techniques (e.g., dead variable reduction and path reduction).
F9	The model checker provides a GUI.
F10	User are able to choose between different abstraction techniques, model checking algorithms, and state compression techniques.
F11	Counterexamples (witnesses) are presented in the source code (assembly and C), as a state space graph, and in the CFG of the assembly code.
F12	Users are able to step through the counterexample and to analyze the values of registers, I/O registers, variables, and formulas.

The requirements presented in this section are the basis for the development of our model checker. The next chapter describes the model checker [MC]SQUARE, which is the result of this development.

4 [mc]square

[MC]SQUARE stands for Model Checking MicroControllers. It is a discrete CTL model checker for microcontroller assembly code. We have developed [MC]SQUARE at the embedded software laboratory during the last four years. It is written in Java as one requirement is that the model checker is usable on any of the three major operating systems. Currently, we are using Java SE 6.

This chapter gives an overview of [MC]SQUARE[1]. Section 4.1 presents the features of [MC]SQUARE. The subsequent section details the architecture of [MC]SQUARE. In the end, we evaluate the chosen architecture in Sect. 4.3.

We published a preliminary version of this architecture in a paper [103].

4.1 Features

[MC]SQUARE is a discrete CTL model checker for microcontroller assembly code. Following, a list of the main features of [MC]SQUARE.

- Supported microcontrollers
  - ATMEL ATmega16
  - ATMEL ATmega128
  - Infineon XC167
- Inputs
  - Assembly program given as an ELF file
  - Specification given as a CTL formula
- Stepwise simulation of the program
- Model checking by means of three different algorithms
  - Global
  - Local
  - Invariant

[1] We use revision 2233 of [MC]SQUARE in this thesis unless otherwise quoted.

- Abstraction techniques to tackle the state-explosion problem
  - Dead variable reduction
  - Delayed nondeterminism
  - Lazy interrupt evaluation
  - Lazy stack evaluation
  - Path reduction
- Minimization of memory requirements via
  - Compression algorithms such as run-length encoding and ZIP compression
  - Storage of states on hard disk
  - Incremental storage of states
- Presentation of Counterexamples (witnesses) in
  - Assembly code
  - Control flow graph of the assembly code
  - C code
  - State space graph

At present, [MC]SQUARE handles assembly code of the following microcontrollers: ATMEL ATmega16, ATMEL ATmega128 and Infineon XC167. It handles programs given in Executable and Linking Format (ELF) [116] and the specifications given as CTL formulas. The specification may contain propositions about registers, I/O registers, and variables (including C variables). Additionally, [MC]SQUARE checks for stack overflows, stack underflows, and non-intended use of microcontroller features such as write access to reserved registers. Furthermore, users can use [MC]SQUARE to stepwise simulate the assembly program.

We have implemented three explicit model checking algorithms in [MC]SQUARE: a global CTL model checking algorithm presented by Clarke et al. [33], a local CTL algorithm first introduced by Vergauwen and Lewi [121] and later adapted by Heljanko [57], and an algorithm used to verify invariants. Section 7.1 describes details of the applied model checking algorithms.

During state space creation, [MC]SQUARE uses various abstraction techniques to lower the size of the state space. These abstraction techniques include, for example, dead variable reduction, path reduction, and delayed nondeterminism. Chapters 5–7 give detailed descriptions of the implemented abstraction techniques.

The users can choose the abstraction techniques that are applied during model checking. Thereby, they are able to adjust the granularity of the abstraction used and hence, influence the size of the resulting state space. Furthermore, they can

select other options to lower memory consumption such as different compression levels and storage of states on hard disk.

[MC]SQUARE presents counterexamples, which are created during model checking, in the assembly code, in the control flow graph of the assembly code, in the C code, and as a state space graph. Hence, users can pick the representation that suites their needs best. In each of these representations, they can analyze the values of registers, I/O registers, and variables and the truth values of the formulas.

4.2 Architecture

This section describes the current architecture of [MC]SQUARE, which was developed to reflect the requirements shown in Sect. 3.3. Figure 4.1 shows the layered architecture of [MC]SQUARE as a *UML package diagram* [12, 17, 61, 113, 119] consisting of the six basic packages: *Parser*, *Static Analyzer*, *Model Checker*, *State Space*, *Simulator*, and *Graphical User Interface* (*GUI*). For clarity, we left out support packages such as *Util*, *Error*, and *Exception*, which execute miscellaneous supporting functions such as array operations, compression algorithms, or hard disk operations. The relations between the packages shown in the figure are *access* relations. In the implementation, the packages shown in Fig. 4.1 are directly mapped to Java packages.

Every step of the model checking process is implemented in a separate package. Additionally, the *Simulator* and the *State Space* are separated into single packages. The communication between packages is conducted by means of well-defined interfaces. Therefore, single parts of the process can be exchanged independently. By separating the *Model Checker* and the *Simulator*, model checking is conducted independently from the underlying microcontroller.

In the following, a short description of each of the six packages is given. Chapters 5–7 detail the three important packages: *Simulator*, *Static Analyzer*, and *Model Checker*.

GUI The *GUI* serves two purposes. First, it handles the interaction with the user. That is, it presents data to the user and gets input from the user. The user can, for example, load files, edit formulas, simulate programs, model check programs, and inspect counterexamples. The second purpose of the *GUI* is to control the different processes used in [MC]SQUARE. That is, it starts, for instance, static analysis, simulation, and model checking. The different processes are implemented as threads within the corresponding packages. The *GUI* uses the observer pattern [47] to collect the data that is presented to the user. Furthermore, we implemented the model-view-controller pattern to separate the data (model) from its representation (view).

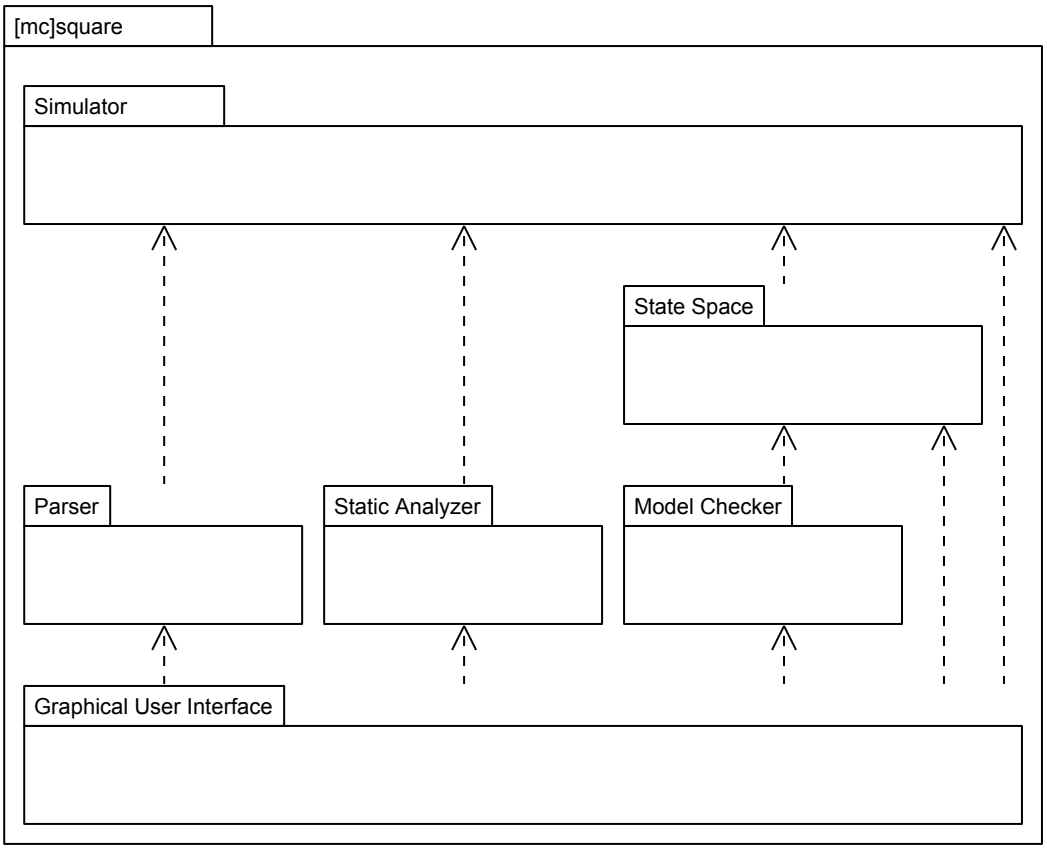

Figure 4.1: The layered architecture of [MC]SQUARE.

Parser The *Parser* package provides capabilities to parse different file formats, such as ELF, and logics, such as CTL, and to transfer them into the corresponding internal representations used throughout [MC]SQUARE. It is, for example, used to process programs and formulas provided by the user.

Static Analyzer The *Static Analyzer* conducts different static analyses and abstraction techniques and annotates the program. It is used before model checking. The *Simulator* uses the annotations to limit the size of the state space during creation. Furthermore, the *Static Analyzer* creates a control flow graph of the assembly code, which is, for instance, used to present the counterexample. Chapter 6 details the *Static Analyzer*.

State Space The *State Space* package is used to store the states. It uses the *Simulator* to create successor states on demand. The *State Space* contains different state space implementations. Some use the main memory, whereas others use the hard disk to store the states. The interface to access the *State Space* is independent from the method used to store the states. Hence, model checking is implemented independent from the method used to store the states. Chapter 7 describes details of the *State Space*.

Simulator The *Simulator* package serves different purposes. First, it is used by the *State Space* package to create successors of given states. When creating successor states, the *Simulator* natively handles nondeterminism and creates an over-approximation of the behavior exhibited by the corresponding microcontroller. This is important to preserve the validity of the formulas checked. Most of the abstraction techniques, which are used to limit the size of the state space, are implemented within the *Simulator*. The second purpose of the *Simulator* is to allow other packages to get information about the microcontroller and the program without the need to implement specifics of the different microcontrollers and programs. We implemented three different microcontrollers within the *Simulator*. Chapter 5 presents the internals of the *Simulator*.

Model Checker The *Model Checker* conducts the actual model checking. In this package, the different model checking algorithms are implemented. Currently, it includes a global and a local model checking algorithm and an algorithm to check invariants. Depending on the chosen algorithm, states are created before model checking or on-the-fly during model checking. The *Model Checker* does not manage the creation of states. It only requests states from the *State Space*, which creates states using the *Simulator* if required. Moreover, the *Model Checker* is responsible for the creation and processing of counterexamples. Chapter 7 details the *Model Checker*.

4.3 Evaluation

Extendability and maintainability are the requirements that notably influenced the architecture. These two requirements are achieved by the chosen division of packages. Each step of the model checking process, that is, parsing, static analysis, state space building, and model checking, is conducted within a single package. Communication between these packages is conducted by means of well-defined interfaces. Thereby, parts of these processes can be exchanged without the need to change the complete application. For example, model checking is conducted independently of the microcontroller used, and hence, adding new microcontrollers can be done without changing the model checking.

To show that [MC]SQUARE is extendable, we already extended it in several directions. The first version of [MC]SQUARE used a patched version of a simulator called AVRORA [117, 118] to build the state space. AVRORA is a cycle-accurate simulator for microcontroller assembly programs written for the ATMEL ATmega16, 32, and 128. We had to change AVRORA because it is cycle-accurate and does not natively support nondeterminism. Preserving cycle-accuracy would lead to real-time model checking [14, 71, 72], which suffers even more from the state-explosion problem.

Using the patched version of AVRORA, we found out that [MC]SQUARE spent 85% to 95% of the processing time building the state space. Therefore, we decided to concentrate on improving the creation of state spaces instead of the model checking. Changing AVRORA was rather involved because it was built for cycle-accurate simulation of microcontroller programs and not for the creation of state spaces for model checking. Many parts of AVRORA were automatically created and not written by hand and thus, these parts were not easy to understand and difficult to change.

Every time a new version of AVRORA was published, we had to apply the same changes. In the second version of [MC]SQUARE, we exchanged AVRORA by our own simulator, which adopted some parts of AVRORA, to avoid these changes and to ease the adaption of the state space creation. Our simulator natively handles nondeterminism and builds a safe over-approximation of the behavior shown by the microcontroller. The simulator is implemented within the *Simulator* package.

Later, we added an additional ATMEL ATmega microcontroller and the Infineon XC167 microcontroller to the *Simulator* package [100, 103]. In the *Model Checker* package, we added a local model checking algorithm and an algorithm used to check invariants [103]. The *State Space* package was extended to store states on hard disk [103, 106]. Furthermore, we added the capability to conduct different static analyses to [MC]SQUARE [107]. This was obtained by adding the *Static Analyzer* package and adapting the procedures using the static analyses. During these extensions, we adapted the architecture of [MC]SQUARE to better reflect the requirements of extendability and maintainability. This makes it easier to extend [MC]SQUARE in

the future.

However, extendability has its limits. At present, based on the classification given by Visser [123], all model checking algorithms implemented within [MC]SQUARE are explicit, graph-based, structural model checking algorithms. Changing to an automata-based algorithm or to symbolic model checking algorithms is more involved than just changing single parts of a procedure. Schommer [109] tried to extend [MC]SQUARE to support symbolic model checking. This try was very involved, but it was not successful in the end. We finally removed it from [MC]SQUARE. It failed because known abstractions, such as modulo and interval abstractions, did not work as expected, and no abstractions that solve this problem could be found within this work. It did not fail due to the architecture of [MC]SQUARE. Schommer [109] gives details in his thesis. Although we were not successful in adding symbolic model checking to [MC]SQUARE, we combined explicit and symbolic model checking techniques within [MC]SQUARE. Chapter 5 explains the details about the combination of explicit and symbolic techniques.

Our aim is to use [MC]SQUARE as a research platform and as a tool that can be applied in industry. Extendability is important because new model checking algorithms, new microcontrollers, and new abstraction techniques have to be added to [MC]SQUARE. As described above, [MC]SQUARE satisfies the requirement of extendability and maintainability. Additionally, the modular structure enables students to work on parts of [MC]SQUARE that they are specialized and interested in. For example, a student interested in static analysis does not have to bother with model checking. Beside extendability, usability is also important for our aim. The architecture supports usability by hiding all internals except for the CTL specifications from the user via the *GUI*.

5 State Space Building in [mc]square

It turned out that the main focus of this thesis is the domain-specific creation of state spaces for model checking microcontroller assembly code. This includes the application of domain-specific abstraction techniques. In [MC]SQUARE, the *Simulator* package builds the state space. It uses a similar process to build the state space as is used in typical simulators, which simulate the behavior of microcontroller programs. That is, it simulates the effect of instructions on the model of the microcontroller. Our *Simulator* differs in two important respects from other simulators. First, it natively supports nondeterminism and second, it creates an over-approximation of the real behavior of the microcontroller to preserve validity of the model checking results. We integrated abstraction techniques into the *Simulator* to limit the size of the state space already during creation.

Another purpose of the *Simulator* is to hide microcontroller peculiarities from the other packages of [MC]SQUARE. Thus, the other packages can access the microcontroller or the states of the microcontroller without considering the peculiarities of the respective microcontroller. Hence, the other packages of [MC]SQUARE are implemented hardware independently.

This chapter describes the modeling of the ATMEL ATmega16 microcontroller within the *Simulator*. The general structure and functionality of the *Simulator* is similar for all microcontrollers. Different microcontroller models only vary in hardware-dependent details such as memory, external devices, instructions, and interrupts.

The first section gives an overview of the *Simulator* package. Section 5.2 describes the states that are used within the *Simulator*. The subsequent sections detail the four important parts of the *Simulator* package. First, the model of the microcontroller (Sect. 5.3) and the model of the program (Sect. 5.4) are described. These two are the representation of the real microcontroller and the real program within [MC]SQUARE. Then, the creation of successor states, which is done by the *InstructionSimulator*, is explained in detail in Sect. 5.5. Section 5.6 details the *Determinizer*, which handles nondeterminism within the *Simulator*. Section 5.7 describes the formal model of the *Simulator*. Then, this formal model is used to establish the correctness of the delayed nondeterminism abstraction technique. Section 5.8 presents related work regarding model checking of assembly code and delayed nondeterminism.

We published the general idea of using a simulator to build a state space in several papers [102, 103]. Furthermore, we outlined the ideas presented in Sect. 5.6

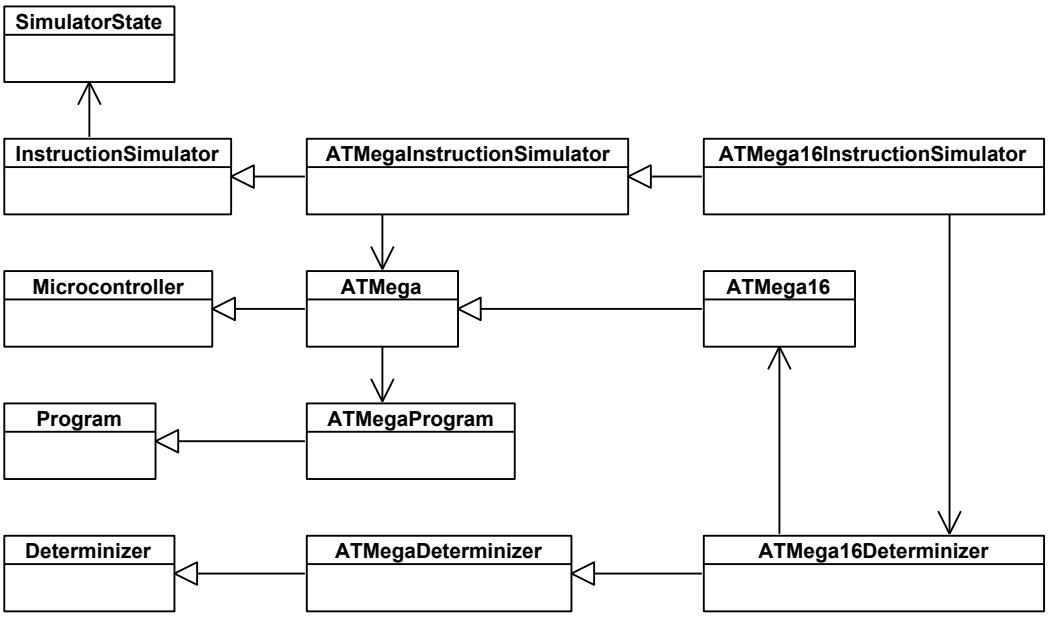

Figure 5.1: Classes of the *Simulator* package related to the ATMEL ATmega16 microcontroller.

in another paper [60] and published a summary of the formal model described in Sect. 5.7 elsewhere [60, 88].

5.1 Simulator Overview

The *Simulator* package generates states for model checking by simulating the effects of instruction executions on the model of the microcontroller. Additionally, the *Simulator* package provides facilities for the other packages to access the microcontroller model through well-defined interfaces. Thus, other packages do not have to deal with the peculiarities of the different microcontrollers.

Figure 5.1 gives an overview of the *Simulator* package. It is a *UML class diagram* [12, 17, 61, 113, 119] depicting the topmost classes of the *Simulator* package. Minor classes are omitted for clarity. On the left side of the figure, the superclasses are shown, and on the right side of the figure, the specialized subclasses are shown. We follow this guideline in all UML class diagrams shown in this chapter.

The *Microcontroller* class and its subclasses represent microcontrollers and their different features within [MC]SQUARE. In these classes, for instance, the memory, the registers, and the external devices are modeled. Section 5.3 provides details on the modeling of these different features. The program run on the microcontroller is

modeled within the *Program* class and its subclasses. Details are given in Sect. 5.4.

The *InstructionSimulator* class and its subclasses control the generation of states. The semantics of the different instructions are modeled within these classes. They use the *Microcontroller* and *Determinizer* classes to execute the instructions and to simulate their effects on the microcontroller model. Classes outside the *Simulator* package use the *InstructionSimulator* class to create states and access the microcontroller. The *InstructionSimulator* uses states of type *SimulatorState* to pass the created states to the other packages such as the *State Space*. Section 5.2 describes the *SimulatorState* class, and Sect. 5.5 details the *InstructionSimulator*.

The *Determinizer* class is used to handle and hence resolve the nondeterminism. The *Determinizer* and its subclasses are described in Sect. 5.6. The handling of nondeterminism is also detailed in conjunction with the formal model of our *Simulator* in Sect. 5.7.

In this chapter, we are especially interested in all *ATMega16* classes as we focus on the modeling of the ATmega16, but we explain features modeled in the superclasses whenever they are not refined in the *ATMega16* classes. Some of the following sections refine the *Simulator* UML class diagram shown in Fig. 5.1.

5.2 Simulator State

The *SimulatorState* class, which represents the state of the simulator, is used by the *InstructionSimulator* to exchange states with other packages such as the *State Space*. It consists of three parts:

- the name of the state,
- the truth values of the atomic propositions used within the formula, and
- data representing the state of the microcontroller (*microcontroller state*).

The name of a state is determined by a hash function applied to the microcontroller state. If a hash collision occurs, it is resolved via a quadratic probing algorithm. As names map one-to-one to states, they are also used for states stored in the *State Space*.

The truth values of the atomic propositions are evaluated by the *InstructionSimulator*, which is located within the *Simulator* package, and stored in the *SimulatorState*. The *Model Checker* uses these pre-evaluated atomic propositions during model checking. Thus, it does not have to handle the peculiarities of the microcontroller.

The microcontroller state comprises all details representing the state of the microcontroller. It contains the complete memory of the microcontroller including general-purpose registers, I/O registers, SRAM, flash memory, and EEPROM. Additionally, it stores fields that are not stored in the memory of the microcontroller

such as the program counter, sleep mode bits, and boot lock bits. Microcontroller states are stored as a byte array. As they are rather big, up to 2 kB in case of the ATmega16, and not all parts are necessarily used, they are compressed using techniques such as run-length encoding and ZIP compression. This considerably reduces the size of the microcontroller states.

A microcontroller state can contain nondeterministic values. Nondeterministic values are introduced by certain I/O registers or by specific abstraction techniques such as lazy stack evaluation (see Sect. 5.3.3) or delayed nondeterminism (see Sect. 5.6.3). Through nondeterministic values, the state does not only represent a single state but a set of states. The state representation thus contains both explicit and symbolic parts. The simulator only works on the explicit parts. Whenever the *Simulator* accesses a symbolic part, that is, a nondeterministic value, the nondeterminism of this value is resolved lazily. Therefore, we call such a state a *lazy state*. The resolution of nondeterminism is detailed in Sect. 5.6. Using this technique, we have combined explicit and symbolic techniques within [MC]SQUARE. To implement lazy states, the model of the microcontroller memory has to store nondeterministic values. Section 5.3.2 describes the modeling of the microcontroller memory and the way nondeterministic values are stored within it.

5.3 Microcontroller

We have modeled different microcontrollers within the *Simulator* package. This section presents the ATmega16 microcontroller model, which is the focus of this chapter. The technical information about the ATmega16, given in this section, is taken from the ATMEL documentation [3, 4].

We had to model the microcontroller as accurately as needed to verify the properties we were interested in. Modeling the microcontroller too accurately contributes to the state-explosion problem during model checking. We had to find the right level of abstraction. Our central idea was to abstract from time and to sacrifice cycle-accuracy because we observed the state-explosion problem when using the cycle-accurate simulator AVRORA. That is, we do not simulate the number of clock cycles, but we do simulate whether the timer is running or not.

The ATmega16 is an 8 bit microcontroller featuring:

- 32 general-purpose working registers,
- 32 general-purpose I/O lines,
- 1 kB SRAM,
- 512 byte EEPROM,

- 16 kB in-system programmable flash memory,
- a JTAG interface,
- three timer/counters (8 and 16 bit),
- 21 internal and external interrupts,
- a two-wire serial interface,
- 10 bit analog to digital converter (ADC),
- a watchdog timer,
- a Serial Peripheral Interface (SPI) serial port, and
- six power saving modes.

A complete list of all features of the ATmega16 can be found in the ATMEL documentation [3, 4]. We have modeled the ATmega16 within the *ATMega16* class, which extends the *ATMega* class (see Fig. 5.1). The features that are present on all ATmega microcontrollers such as the registers are modeled within the *ATMega*. The features that are specific to the ATmega16 such as the I/O registers are modeled in the *ATMega16* class. There are other features that are modeled in their own classes, for example, I/O ports and timers.

In the following, we explain the modeling of some of these features. A detailed description precedes the modeling of each feature. Some of the ATmega16 features such as the core, the registers, and the memory exhibit a deterministic behavior. Other features introduce nondeterminism due to their nature, for example the I/O ports and ADC, or their modeling, for instance timers. This section first describes the modeling of some deterministic features of the ATmega16, namely, the core, the memories, and the stack. After that, three Atmega16 features that introduce nondeterminism are detailed, namely, interrupts, I/O ports, and timers.

5.3.1 Core

The core of the ATmega16 features 131 different instructions. 32 general-purpose working registers are directly connected to the arithmetic logic unit, allowing two registers to be accessed within a single instruction. Six of the registers can be used as 16 bit indirect address register pointers. The core is responsible for fetching the next instruction, executing instructions, setting bits in the status register, etc. These tasks are not modeled within the *Microcontroller*, but they are modeled within the *InstructionSimulator*, which is described in Sect. 5.5.

The core has some own state variables, such as the *program counter* (*PC*), which are not stored in the SRAM data memory. To store these values within the microcontroller state, we added an extra array to the model of the ATmega16.

5.3.2 Memories

The ATmega16 uses a Harvard architecture as every AVR microcontroller. In a Harvard architecture, the memories and buses for program and data are separated. Hence, the memory of the ATmega16 microcontroller is divided into three memory spaces:

- data memory space,
- EEPROM memory space, and
- program memory space.

The ATmega16 SRAM memory comprises 1120 bytes. Figure 5.2 shows the SRAM data memory space of the ATmega16. The first 96 bytes contain the register file and the I/O registers. The registers are used by the core to conduct calculations and to store values. The I/O registers are used to access certain features and devices of the ATmega16 microcontroller such as timers, I/O ports, or interrupts. The internal data SRAM is accessed through the last 1024 bytes. It is used to store variable values and the stack.

The EEPROM memory is nonvolatile and can be used by the developer to store values permanently. Its size is 512 bytes, and it is 8 bit wide.

The program memory stores the program. Its size is 16 kB, and it is 16 bit wide.

SRAM Data Memory

The SRAM data memory space is modeled as two byte arrays, each 8 bit wide. The first array stores the actual value of each memory location, for example, register, I/O register, or variable. The second array records whether a memory location contains a deterministic or a nondeterministic value. It is used to realize lazy states (see also Sect. 5.2). A byte in the second array is called the *to be determinized mask* (*TBDM*) of a memory location. It records that nondeterminism, introduced by abstraction techniques, has to be resolved for the corresponding memory location. The TBDM is not used to model the nondeterminism generated by the I/O registers. Whether an I/O register generates a nondeterministic value or not is implemented within special *Register* classes (see Sect. 5.3.5 and 5.3.6).

The TBDM records whether each bit is nondeterministic or not. Some bits of a memory location can be nondeterministic while others are deterministic. When a bit in the TBDM is zero, its value is deterministic, and hence, the corresponding

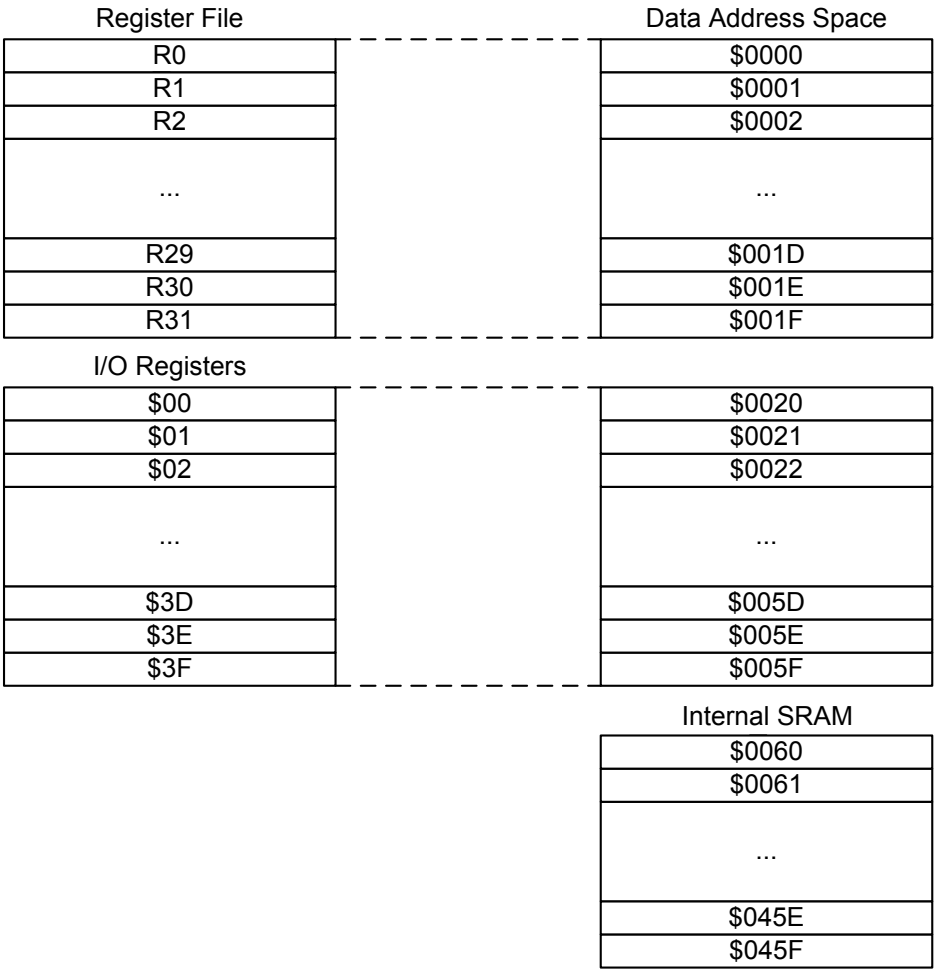

Figure 5.2: ATMEL ATmega16 data memory map [4].

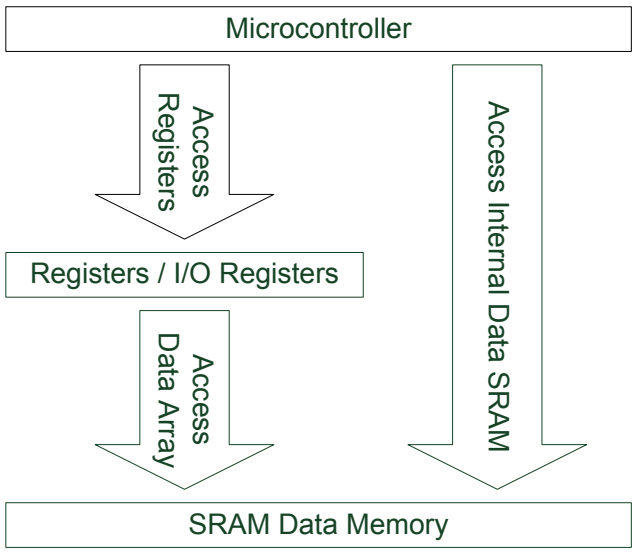

Figure 5.3: Different accesses to the SRAM data memory.

value stored within the first array is used. When a bit is one in the TBDM, it is nondeterministic. Its value in the first array is not used and is set to zero. Therefore, large parts of both arrays hold zeros. In fact, some parts of the second array are zero all the time. This seems to double the memory consumption to 2 kB for the data memory space, but these two arrays can be compressed very efficiently via one of the compression methods implemented within [MC]SQUARE.

As described before, the SRAM data memory space contains three memory regions (see Fig. 5.2):

- register file,
- I/O memory (I/O registers), and
- internal data SRAM.

Accesses to these three memory regions are handled differently. Figure 5.3 shows the different accesses to the SRAM data memory space. The parts representing the internal data SRAM are accessed directly because they do not require special treatment. The parts storing the values of registers and I/O registers are accessed through specific *Register* classes. These classes are needed because some registers have access restrictions, involve side-effects, or generate nondeterminism. An example for registers with access restrictions are reserved registers. I/O registers often involve side-effects, for instance, activation of interrupts or timers. This may lead to nondeterminism in other I/O registers. The special *Register* classes are

used to observe restrictions, to conduct changes including side-effects, and to handle nondeterminism.

There are different types of registers. There are *Register* classes representing general-purpose working registers, registers that cannot store nondeterministic values, registers having reserved bits, and special purpose I/O registers. These different classes encapsulate the logic behind the respective registers. That is, if a write access to a register occurs, the corresponding handles the memory addresses that have to be written and the side-effects that have to be executed. The same is done when registers are read. For instance, certain I/O registers can return nondeterministic values depending on the values of other I/O registers. We describe details of I/O registers representing I/O ports in Sect. 5.3.5 and timers in Sect. 5.3.6. These special *Register* classes do not store values. They are only used to access the arrays storing the values. Microcontroller states can be saved and restored by copying the underlying byte arrays.

EEPROM Data Memory

We have modeled the EEPROM data memory as a byte array that cannot be accessed directly. All accesses to it are made through certain I/O registers. These I/O registers are modeled by special classes, which effect the actual changes. The EEPROM data memory does not store nondeterministic values. Hence, a second array is not needed. As the EEPROM does not usually change much during runtime, only changes made relative to its initial values are stored in the microcontroller state.

Flash Program Memory

The flash program memory is modeled as a 16 bit array because ATmega instructions are 16 or 32 bits wide. The flash memory can be read directly, but when writing to it, certain I/O registers are used to determine what is to be done, for example, delete page, write page, or write into temporary buffer. The flash memory does not store nondeterministic values because it contains the program. As it can be written during runtime, the program may change itself. Hence, we had to implement an on-the-fly disassembler. Since an on-the-fly disassembler is rather slow, we implemented a two-staged approach. When the state space creation is started, the initial flash memory is disassembled into an assembly program (see Sect. 5.4). The instructions of this assembly program are used in all states where the page in program memory holding the current instruction was not changed. Whenever an instruction is to be executed that resides in a page in program memory that is changed, the on-the-fly disassembler is used to disassemble the current instruction. This combines the performance of a static approach with the ability to handle self-modifying code.

Like the EEPROM data memory, the program memory is not stored in every single microcontroller state completely, but it is stored as differences to the initial flash memory.

5.3.3 Stack

The stack of the ATmega16 is located within the internal data SRAM of the microcontroller. It starts at the end of the internal data SRAM and grows towards the start, that is, newer values are written to lower memory addresses. The maximum size of the stack is equal to the size of the internal data SRAM. Due to this configuration, it is possible that the stack overflows and collides with values stored within the internal data SRAM. [MC]SQUARE checks for such stack overflows and returns warnings if a stack overflow is found. The stack pointer is implemented as two registers in the I/O space: Stack Pointer Low and Stack Pointer High.

To model the behavior of the stack, we only had to model the stack operations PUSH and POP because the contents of the stack and the stack pointer are already stored due to the fact that [MC]SQUARE stores the complete memory of the ATmega16. We have modeled the two operations the same way as they are executed on the microcontroller. The PUSH operation first pushes the value onto the stack and then decrements the stack pointer. The POP operation first increments the stack pointer and then returns the value. The value returned remains within the internal data SRAM, it is not reset to 0. Hence, after control returns from a function, the data that was pushed and popped by the function is still in place. Thereby, functions that are originally independent become dependent because they share their temporary stack contents. Each function influences parts of the microcontroller state, that is, the memory region where the stack is located, which are also accessed by the other functions.

Initially, we modeled the stack exactly, but we observed that the resulting state spaces sometimes became too large. One possible idea for a solution is to reset values to zero after they have been popped. However, because functions can access these memory locations directly, this can lead to wrong results. Instead, we have implemented an abstraction technique we call *lazy stack evaluation*, which sets values to nondeterministic using the TBDM after they have been popped, that is, the actual value is reset and the TBDM is set. In the rare case a memory location that was part of the stack is read, all 256 possible values are returned.

As the memory locations residing in the internal data SRAM are initialized to zero, resetting the value of a memory location to zero and marking it as nondeterministic creates additional states. To avoid these additional states, [MC]SQUARE initializes the internal data SRAM to nondeterministic when using lazy stack evaluation. That is, it sets the value to zero and marks the memory locations as nondeterministic. This is a valid over-approximation and usually, all memory locations are initialized or written

before they are read. Using lazy stack evaluation produces an over-approximation and reduces the size of the state space significantly.

Usually, lazy stack evaluation does not add behavior because a program typically does not read parts of the stack after use, and values are written or initialized before they are accessed. However, as the validity of a formula can depend on the real behavior exhibited by the microcontroller, the user can deactivate this option in [MC]SQUARE. Without using this option, many programs cannot be model checked as the resulting state spaces become too large.

5.3.4 Interrupts

The ATmega16 has 21 different interrupts. The interrupt vectors are located in the program memory space. The lowest address is occupied by the Reset Vector. The other interrupt vectors are located at higher addresses. All interrupts have fixed priorities: the lower the address, the higher the priority. Thus, the Reset Vector has the highest priority. The other interrupts include external interrupts, timer interrupts, and ADC interrupts.

There are two kinds of interrupts. The first type is triggered by an event that sets the corresponding flag. It is handled whenever the corresponding flag is set and the interrupt is enabled. If an event occurs and the corresponding interrupt is not enabled, it will be handled when the interrupt is enabled later. That is, flags are remembered until they are manually reset or until the related interrupt is enabled and hence, can be handled. Examples of this type of interrupt are: timer interrupts, external interrupts, and ADC interrupts. The second type of interrupt is only triggered as long as the interrupt condition is present. If an event occurs and the respective interrupt is not enabled, this event is not remembered. This type of interrupt does not necessarily have an interrupt flag. External interrupts used in level mode and EEPROM interrupts are examples for this type.

Every interrupt has its own interrupt enable bit. This bit has to be set together with the Global Interrupt (I) Enable bit, which is located in the Status Register (SREG), in order to enable the corresponding interrupt. Interrupts are handled before each instruction is executed. They are checked one-by-one from highest priority to lowest. When an interrupt is found that is enabled and has occurred, checking is aborted and the corresponding interrupt handler (IH) is called by first pushing the current PC onto the stack and then setting the PC to the address of the IH. When an IH is called, the I bit is cleared and all interrupts are deactivated. To use nested interrupts, users have to manually activate interrupts within IHs. At the end of the IH, the RETI instruction is executed. It changes the PC to the return address located on top of the stack and sets the I bit to enable interrupts. After an interrupt handler is left, at least one normal instruction is executed.

We have modeled the handling of the interrupts in the same way. The method that

checks whether an interrupt is both enabled and triggered is located in the *ATMega16* class because the interrupt priorities differ for the different microcontrollers of the ATmega family. If an enabled and triggered interrupt is found, the *ATMega16* calls the corresponding IH as described above. If no interrupt is enabled and triggered, the method simply returns.

The method that checks for the occurrence of interrupts is called by the *InstructionSimulator* class. This class, which is described in Sect. 5.5, controls the creation of states. The triggering of nondeterministic interrupts is done by the *Determinizer*, which is described in Sect. 5.6.

The execution of at least one instruction after an IH is left negatively influences the size of the state space because it introduces a new state variable. This variable is of type Boolean and stores whether an IH has just been left or not. In the worst case, this variable leads to a doubling of the number of states and thus to a doubling of the state space size. To ease this problem, we have added an abstraction technique that removes this restriction. After an IH is finished, the next IH can be called without the need to execute a normal instruction in between. This option is called *lazy interrupt evaluation*. It is an over-approximation that may lead to behavior not observed on the real microcontroller, but it helps to significantly reduce the size of the state space.

Nested interrupts are disabled by default on the ATmega16 microcontroller, but users can activate them in IHs. When model checking, nested interrupts are a problem as they can cause stack overflows because every time an IH is entered, another interrupt can occur. The new interrupt is handled immediately, and thus, the stack holding the return addresses grows until a stack overflow occurs. As a model checker has to visit all possible states, it detects this case even though it might not be possible in a real system where interrupts can occur less frequently. A model checker cannot exclude occurrences of interrupts as this can cause an under-approximation and hence unsound results. To help users in case nested interrupts are enabled in the program, we added a feature allowing users to limit the number of simultaneous interrupt occurrences. However, the user has to keep in mind that this could mask possible errors. Gückel [49] provides details in his thesis.

5.3.5 I/O Ports

The ATmega16 features four I/O ports. Each of these ports has three distinct I/O memory address locations: one for the data register PORTx, one for the data direction register DDRx, and a third for the port input pins PINx. The lowercase x can be replaced with A to D representing Port A to Port D. Beside the general digital I/O functionality, each port has alternate functions, which interfere with the general digital I/O functionality.

Figure 5.4 is a UML class diagram depicting the classes of the *Simulator* package

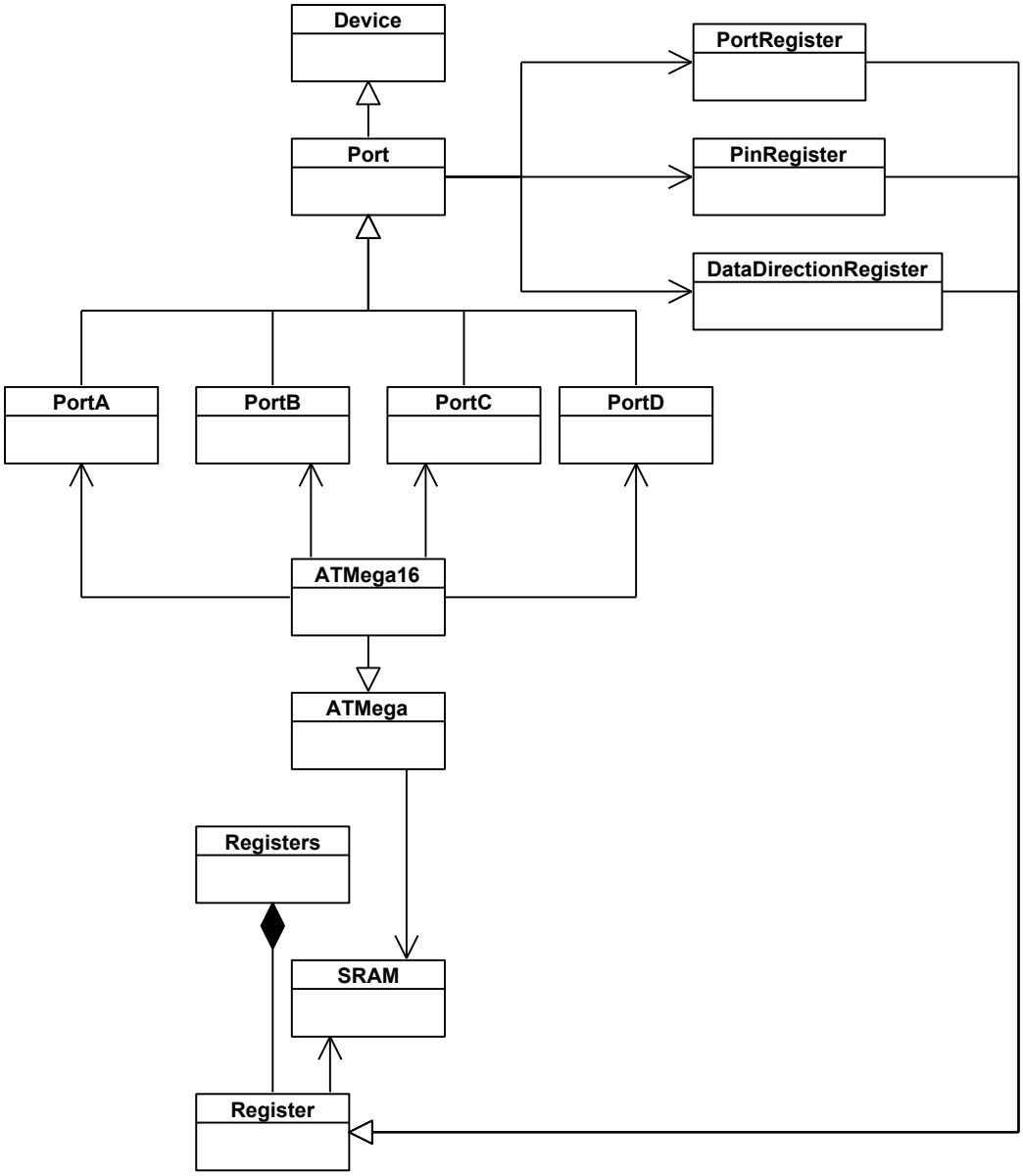

Figure 5.4: Classes of the *Simulator* package related to I/O ports.

that model the I/O ports of the ATmega16. The ATmega16 is represented by the *ATMega16* class. The four I/O ports are represented by the four *Port* classes: *PortA*, *PortB*, *PortC*, and *PortD*. The three I/O registers used to operate the ports are modeled by the three *Register* classes: *PortRegister*, *DataDirectionRegister*, and *PinRegister*. The alternate port functions of the specific ports are modeled within the specialized *Port* classes or in external devices, which are represented by classes of type *Device*. These external devices use the *Register* classes to communicate with the corresponding *Port*. The properties of the I/O ports are modeled within the refined *Port* classes. Each *Port* has specific methods and fields to model the behavior of the corresponding port. The I/O registers are modeled in the same way. They have, for example, an address and a value. The addresses of the I/O registers are stored within the corresponding *Register* classes, but the values of the I/O registers are not stored within the *Register* classes. These values are stored in the array representing the SRAM. The *Register* classes are used by other classes such as the *Port* to access this array (see also Sect. 5.3.2).

In the following, we first detail the modeling of the three I/O registers used to operate the I/O ports. Afterwards, we explain the modeling of an alternate port function represented within a *Port*.

Port x Data Direction Register (DDRx) selects the direction of the pins of an I/O port. If a bit of this register is set to one, the corresponding pin is used as an output pin. If a bit of this register is set to zero, the respective pin is used as an input pin. That is, it determines which pins of this port are used for output and which pins are used for input. This register influences the behavior of the other two registers of this port.

The *DataDirectionRegister* class models the DDRx registers. The value of the *DataDirectionRegister* is deterministic. It always returns the value assigned to it. The *DataDirectionRegister* influences the value of the *PinRegister* and the function of the *PortRegister*.

Port x Data Register (PORTx) serves two purposes. If the port is used as an input port, PORTx activates and deactivates the pull-up resistors of the pins. A one activates the pull-up resistors. A zero deactivates them. When the port is used as an output port, the value of PORTx is used as the output. Writing a one drives the pin high. Writing a zero drives it low.

The *PortRegister* class models the PORTx registers. Like the *DataDirectionRegister*, it is deterministic and always returns the value assigned to it. We did not model the pull-up resistors because our simulation does not involve real hardware. If the *Port* is used for output, which is determined by the *DataDirectionRegister*, the value of *PortRegister* is used as output.

Port x Input Pins Address(PINx) is used to read the actual value of the port. If a pin is used as an input pin, the value externally applied to the pin is read. If a pin is used as output, the value written to the corresponding bit in PORTx is read.

The *PinRegister* class models the PINx registers. The value of the *PinRegister* is dependent on the value of the *DataDirectionRegister* and the *PortRegister*. The bits set to one in the *DataDirectionRegister* are deterministic and return the value of the *PortRegister* because they are used for output. The bits set to zero in the *DataDirectionRegister* are nondeterministic as they are used as input. For example, if the *DataDirectionRegister* is set to 0xff, the *PinRegister* returns the value of the *PortRegister*. If the *DataDirectionRegister* is set to 0x00, reading the *PinRegister* returns all 256 possible values. If the *DataDirectionRegister* is set to 0x0f, the four topmost bits are nondeterministic and the four lower bits are deterministic. In this case, reading the *PinRegister* returns 16 different values.

We did not use the TBDM for the values of these three *Register* classes because abstraction techniques are not used for *Register* classes representing I/O ports, and hence, nondeterministic values are not assigned to them. The value of the *PinRegister* is only nondeterministic if the corresponding *DataDirectionRegister* is configured as input.

Beside this general I/O functionality, every port has alternate functions. Port A is used as the analog input for the ADC. Port B is used as input and output for the SPI, as input for the analog comparator, to connect External Interrupt 2, as input for Timer/Counter0 and 1, and to connect the external clock for USART. Port C is used for connecting external timer oscillators, as input and output for the JTAG interface, and as input and output for the two-wire serial interface. Port D is used for input and output of Timer/Counter1 and Timer/Counter2, as input and output for USART, and to connect External Interrupt 0 and External Interrupt 1.

As an example for the modeling of an alternate port function, we explain the modeling of External Interrupt 0. Pin 2 of Port D is used as the input for External Interrupt 0. If External Interrupt 0 is enabled, this pin triggers the interrupt independently of the mode of the pin. If the pin is used for input, the interrupt is triggered from outside. If the pin is used for output, the interrupt is triggered by the program. That is, it is triggered when a one is written to PORTx. The flag in the General Interrupt Flag Register (GIFR) is set even if the interrupt is not enabled.

External Interrupt 0 is modeled within *PortD*. If this alternate port function is enabled, *PortD* triggers the corresponding bit in the representation of the GIFR. It triggers this bit nondeterministically if the *DataDirectionRegister* is configured as input (see also Sect. 5.6). If the *DataDirectionRegister* is configured as output, *PortD* triggers the bit deterministically whenever values are written to the *PortRegister*.

Modeling the devices accurately helps to minimize the size of the state space and to reflect the behavior of the microcontroller. If the I/O ports are not modeled accurately, reading a PINx register always returns all 256 possible values even if the port is used for output. This is an over-approximation of the behavior of the port that is too coarse to verify interesting properties involving the port. Hence, it is important to accurately model the devices of the microcontroller as far as possible.

5.3.6 Timers

The ATmega16 has three timers: two 8 bit timers (Timer/Counter0 and 2) and one 16 bit timer (Timer/Counter1). In [MC]SQUARE, we abstract from time because its accurate modeling would, in our approach, lead to state spaces that are too large to be handled. Model checking of real-time models is, for example, described by Bengtsson et al. [14] and Larsen et al. [71, 72]. The model of a timer used within [MC]SQUARE over-approximates the real behavior of a timer. We only distinguish two timer states: the timer is running or not running. Depending on whether a timer is running or not, the registers accessing the timer have different values. This section describes the modeling of Timer/Counter0 within [MC]SQUARE.

Figure 5.5 shows the UML class diagram depicting the classes that model the timers of the ATmega16. Timer/Counter0, which is represented by *Timer0*, is controlled by five I/O registers. The values of these I/O registers are stored within the array representing the SRAM data memory and not in the *Register* classes. The *Register* classes only store the addresses of the corresponding I/O registers (see also Sect. 5.3.2). In the following, these five registers and their modeling in [MC]SQUARE are described:

Timer/Counter Control Register 0 (TCCR0) is utilized to control Timer/Counter0. It is used to select the clock source, to choose the output compare mode, to choose the waveform generation mode, and to trigger the output compare event. Possible clock sources are, for example, none, internal, and external.

The instance of the *ControlRegister* in *Timer0* models the TCCR0 register. Its value is deterministic and independent of the state of *Timer0*. This instance controls *Timer0* determining whether it is running or not. If a clock source is selected, *Timer0* is running. If no clock source is selected, *Timer0* is not running. Depending on the *ControlRegister*, the values of certain register representations are nondeterministic in [MC]SQUARE because we abstract from time.

Timer/Counter Register 0 (TCNT0) is the register that stores the current value of Timer/Counter0. It can be read and written, that is, users can read and change the current timer/counter value. Doing so can introduce side-effects,

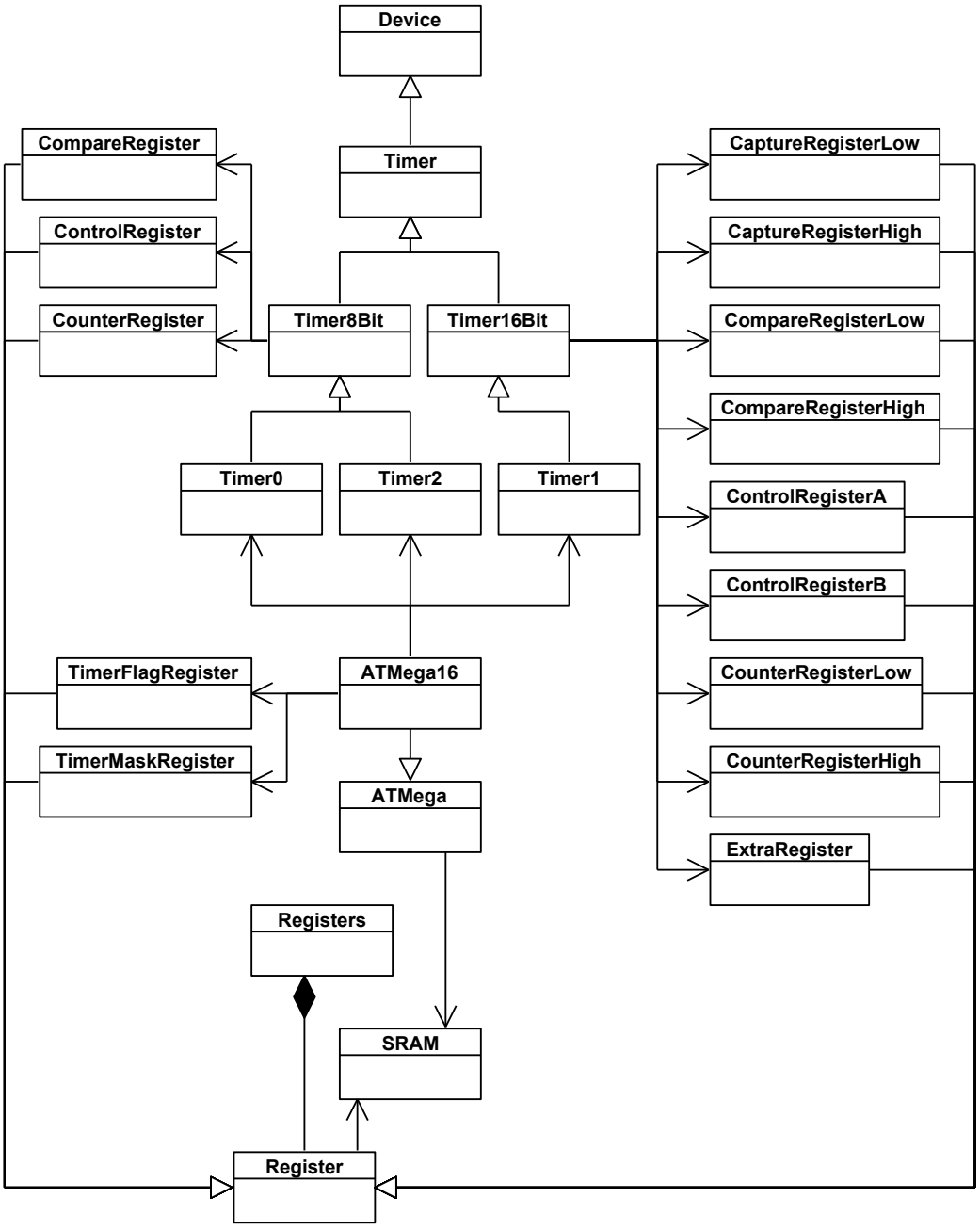

Figure 5.5: Classes of the *Simulator* package related to timers.

for example, events can be missed. In case the register overflows, an overflow event is raised by setting the corresponding bit in the TIFR register. If the overflow interrupt is enabled, it is triggered by the overflow event.

The instance of the *CounterRegister* in *Timer0* models the TCNT0 in [MC]-SQUARE. The value of the *CounterRegister* depends on the state of *Timer0*, that is, whether it is running or not, which is determined by the *Control-Register*. If *Timer0* is not running and was not running before, the value of the *CounterRegister* is deterministic. If *Timer0* is running, the value of the *CounterRegister* is nondeterministic. If *Timer0* was running before and is now deactivated, the value of this *Register* is nondeterministic too. This is remembered within the TBDM of the *CounterRegister*. After the nondeterminism of the value is resolved, the value is deterministic again because the TBDM is reset.

Output Compare Register 0 (OCR0) stores the output compare value for Timer-/Counter0, which is continuously compared with the TCNT0 register. Every time both values are equal, a compare match is signaled by setting the corresponding bit within the TIFR. If the compare match interrupt is enabled, it is triggered.

The instance of *CompareRegister* in *Timer0* models the OCR0 register. As it stores the compare value, it is completely deterministic regardless of the state of *Timer0*.

Timer Interrupt Flag Register (TIFR) is used by all three timers. Each of the timers has its own bits inside the TIFR. Timer/Counter0 uses two bits of this register: Output Compare Flag 0 and Timer/Counter0 Overflow Flag. The former is utilized to store whether an output compare event occurred and the latter is used to store whether an overflow event occurred. This register is set by the hardware and can be read by the program. Bits of this register are reset when either a one is written to them, or the associated interrupt handler is entered. These flags can be used in two different ways. First, they can be used by utilizing the corresponding interrupts. Second, users can poll these flags in their programs and reset them manually after handling the events.

TimerFlagRegister models the TIFR register. As this register is used by all timers, it is not modeled in the *Timer* classes but rather in the *ATMega16* class. *Timer0* uses two bits of the *TimerFlagRegister*. The values of these bits are dependent on the state of *Timer0*. If *Timer0* is running, the values of these two bits are nondeterministic, but this nondeterminism can be restricted. If the nondeterminism of the value of one of these bits was resolved and instantiated to one due to an access by the *Simulator*, it remains one until it is reset. Otherwise, if it was instantiated to zero, the value remains nondeterministic

in the next state. If *Timer0* is not running and was not running before, the values of both bits are zero. If *Timer0* is not running, but was running before, the values of these two bits are nondeterministic and have to be instantiated once to become deterministic again. This is recorded in the TBDM of the *TimerFlagRegister* because the actual source of the nondeterminism, that is, timer is running, is no longer enabled. The resolution of the nondeterminism is not done within this class but rather within the *Determinizer*.

Timer Interrupt Mask Register (TIMSK) determines which timer/counter interrupts are enabled. This register is used by all three timers. Timer/Counter0 uses two bits of this register: Timer/Counter0 Output Compare Match Interrupt Enable and Timer/Counter0 Overflow Interrupt Enable. The first bit determines whether the compare match interrupt is enabled and the second bit determines whether the overflow interrupt is enabled. Writing a one to a bit activates the respective interrupt. Writing a zero to a bit deactivates the interrupt. The interrupts are only enabled if the I bit in the SREG is also set.

The TIMSK register is modeled by the *TimerMaskRegister*. It is implemented within the *ATMega16* class because it is used by all three timers. The value of this register in deterministic and independent of the state of *Timer0*. It is, for example, used by the *Determinizer* to decide which timer interrupts are enabled.

Modeling the devices accurately helps to minimize the size of the state space. Without modeling the dependencies between the different timer registers, the values of both the TCNTx and the TIFR would be nondeterministic regardless of the state of the timer. Timer interrupts could then occur at every program location. Another reduction of nondeterminism that could not be used without modeling the dependencies of the timers is the abstraction we implemented for the TIFR, which avoids paths where the values of the interrupt flags are first one and then, in the next state, zero without being reset. On the real microcontroller such paths are not possible, but if these dependencies were not modeled, the value of the TIFR would always be nondeterministic making such paths possible. This would increase the size of the state space and lead to more false alarms.

The modeling of Timer/Counter2 is similar to the modeling of Timer/Counter0. As Timer/Counter1 is a 16 bit timer, its modeling is more complex. Timer/Counter1 has eleven registers in contrast to the five registers of Timer/Counter0 and Timer/Counter2. Additionally, Timer/Counter1 has features that are not present in either Timer/Counter0 and Timer/Counter2.

5.4 Program

Before a program can be executed on the ATmega16, it has to be deployed to the microcontroller. The program that is deployed to the microcontroller is usually given as an ELF file. In the deployment process, this ELF file is written to the flash program memory of the ATmega16.

[MC]SQUARE uses the same ELF file as input for model checking. Before the program can be used within [MC]SQUARE, is has to be parsed and transformed into the internal program representation, which is modeled in the *Program* class. This is done by an ELF parser implemented within the *Parser* package. The parser disassembles and transforms the ELF file into an instance of the *Program* class. As the program can change the flash program memory during execution, the program can change itself. Hence, whenever an instruction is executed that resides in a page of the flash program memory that was changed by the program, [MC]SQUARE has to disassemble the instruction on-the-fly (see also Sect. 5.3.2). In this case, [MC]SQUARE cannot execute the instruction located in the static *Program* representation. The on-the-fly disassembler is only used for instructions that reside on altered pages in the flash memory. All other instructions are taken from the static *Program* representation because it is faster. Abstraction techniques that are based on static analysis can only be applied to the static *Program*, that is, results of the static analysis are only used if the program does not change itself.

Figure 5.6 shows a UML class diagram of the classes related to the *Program* representation. The *ATMegaProgram* consists of *ATMegaInstructions*. There are different types of *ATMegaInstructions* such as *None*, which means no operands, *Reg*, which means one operand is a register, and *Imm*, which means one operand is immediate. In this figure three different instructions are shown: *ADC*, *ADD*, and *BRCC*. *ADC* and *ADD* are of type *RegReg*, which means that they have two operands of type register. The instruction *BRCC* is of type *Imm*. These are only three of the 131 instructions present on ATmega microcontrollers.

The semantics of the instructions are not implemented within the *Instruction* classes, but separately within the *InstructionSimulator* and the *Static Analyzer* classes because each uses different semantics. Section 5.5 presents the semantics used within the *InstructionSimulator*, and Chap. 6 details the semantics used within the different static analyses. The *Instruction* classes store the properties of the different instructions and provide methods that, for example, support the resolution of nondeterminism for the instructions. These methods define whether the nondeterminism of values accessed by the corresponding instruction has to be resolved or can be copied (see also 5.6).

The *ATMegaProgram* stores debug information collected from the C code by the compiler, which is used to map features from the assembly code to the C code. [MC]SQUARE uses the debug information, for example, to present the counterexample

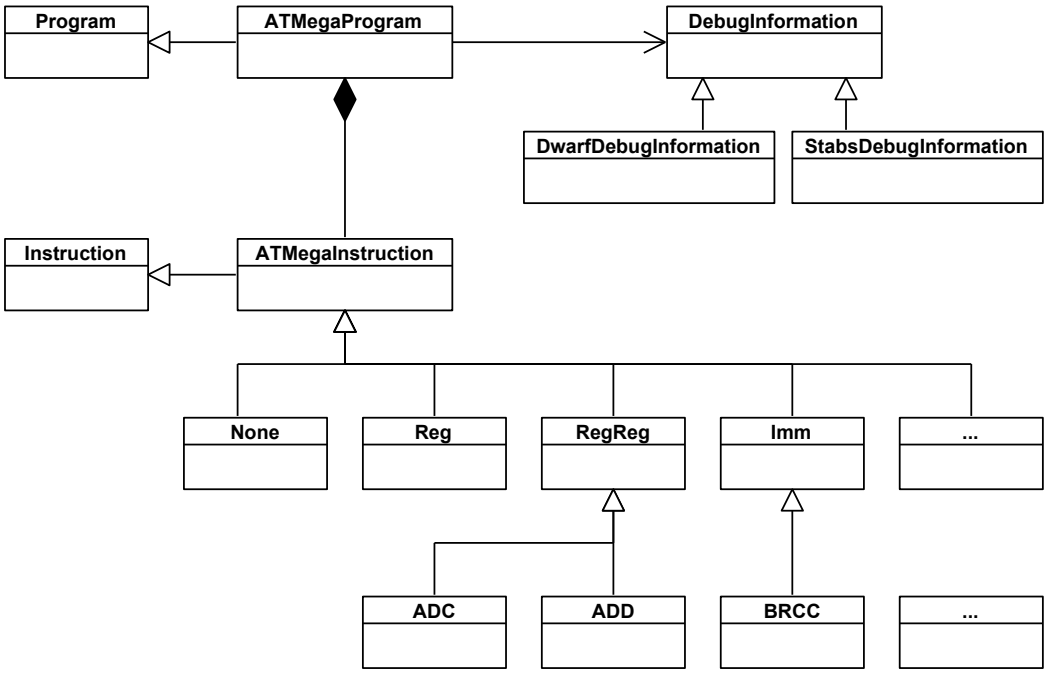

Figure 5.6: Classes of the *Simulator* package related to the program representation.

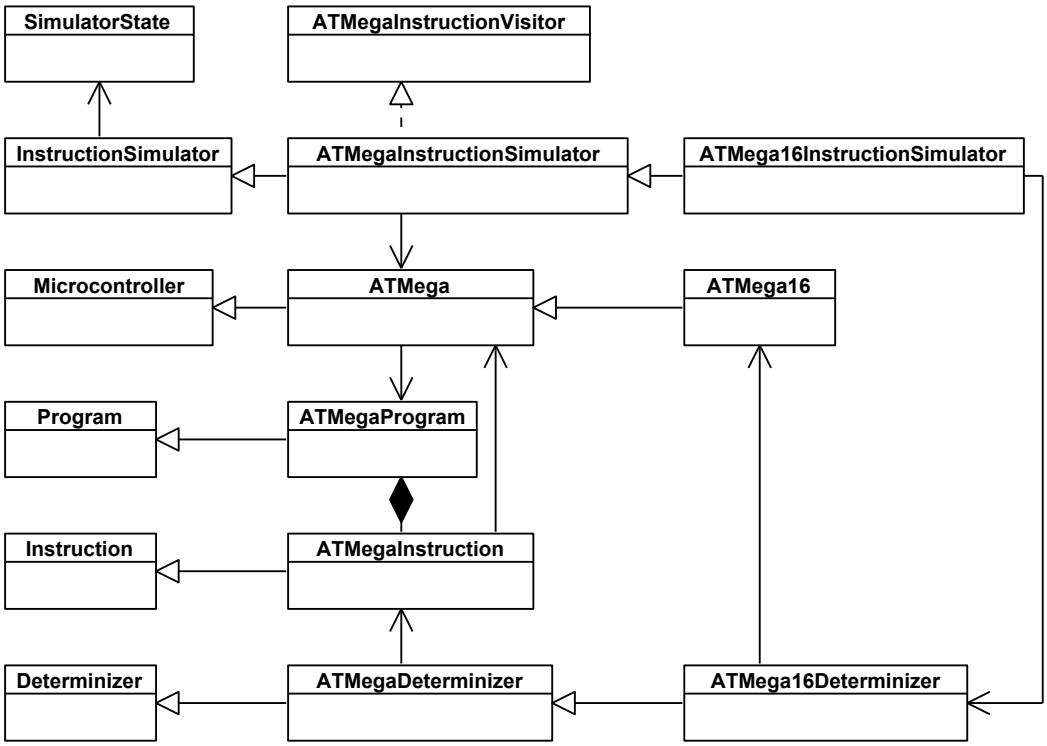

Figure 5.7: Classes of the *Simulator* package related to the *InstructionSimulator*.

in the C code. The debug information can be given in Stabs [46] or Dwarf [40] format. The *Program* has a textual representation that is human-readable in contrast to the ELF file, which is not human-readable. This textual representation is, for instance, used for the presentation of counterexamples and for user-controlled simulations.

5.5 Instruction Simulator

The *InstructionSimulator*, which is, for example, used by the *State Space* package, controls the creation of successor states within the *Simulator* package. Figure 5.7 shows a UML class diagram depicting the classes of the *Simulator* package related to the *InstructionSimulator*. As mentioned before, the semantics of the instructions are not defined within the *Instruction* classes because the *Simulator* and the *Static Analyzer* use different semantics. To make the handling of the different semantics easier, we use the *visitor design pattern* [47] in [MC]SQUARE as it is used in AVRORA.

The *ATMegaInstructionVisitor* interface represents the set of 131 instructions supported by the ATmega microcontrollers. Every implementation of the *ATMegaIn-*

structionVisitor has to implement a visit() method for each of the instructions. The visit() method implements the semantics of the respective instruction inside the current context, that is, simulation or static analysis.

The *ATMegaInstructionSimulator* realizes the *ATMegaInstructionVisitor* interface. It implements the semantics of the instructions used for simulation, that is, state space building. The *ATMega16InstructionSimulator* extends the *ATMegaInstructionSimulator* and adds the specifics of the *ATMega16*.

In the following, we detail the process that is used for the creation of successor states. Figure 5.8 shows a *UML activity diagram* [12, 17, 61, 113, 119] representing this process. The creation of successor states is initiated by the *State Space* package. The *State Space* calls the *InstructionSimulator*, here the *ATMega16InstructionSimulator*, and passes the microcontroller state for which all successor states should be created. Then, the *ATMega16InstructionSimulator* loads the microcontroller state into the *Microcontroller*, in this case, the *ATMega16*. After that, it retrieves the next instruction, which is represented by the *ATMegaInstruction* class, from the *ATMega16* and creates an *ATMega16Determinizer*, which is used to resolve possibly existing nondeterminism. The next computation steps, required to build the successors, are executed inside a loop. In each loop cycle, one successor state is created.

Within this loop, the following tasks are accomplished. First, the *ATMega16Determinizer* resolves the nondeterminism by adapting the *ATMega16* if required. This is done by setting bits or bytes of certain memory locations. This can cause the triggering of interrupts or the assignment of values to memory locations addressed within the current instruction. Section 5.6 gives details about the resolution of nondeterminism.

The *ATMega16InstructionSimulator* then calls a method located in the *ATMega16* class to check whether an interrupt has been triggered. In this case, the *ATMega16* calls the corresponding interrupt handler. The calling of an IH is explained in Sect. 5.3.4. If no interrupt has been triggered, the *ATMega16InstructionSimulator* executes the actual instruction by calling the corresponding visit() method implemented within the *ATMega16InstructionSimulator*. In this visit() method, the effect of the instruction, which is represented by the *ATMegaInstruction* class, on the *ATMega16* is simulated. That is, the PC is incremented, bits within the SREG are set if needed, and involved memory locations are changed.

Then, the atomic propositions are checked by the *InstructionSimulator* because the *Model Checker* cannot test them as they are hardware independent. Atomic propositions are hardware dependent because they contain statements about the different microcontroller features such as registers, I/O registers, and variables.

In the last step of the loop, the resulting microcontroller state is read from the *ATMega16* and, together with the truth values of the atomic propositions, written into the array of successor states, which are represented by instances of the

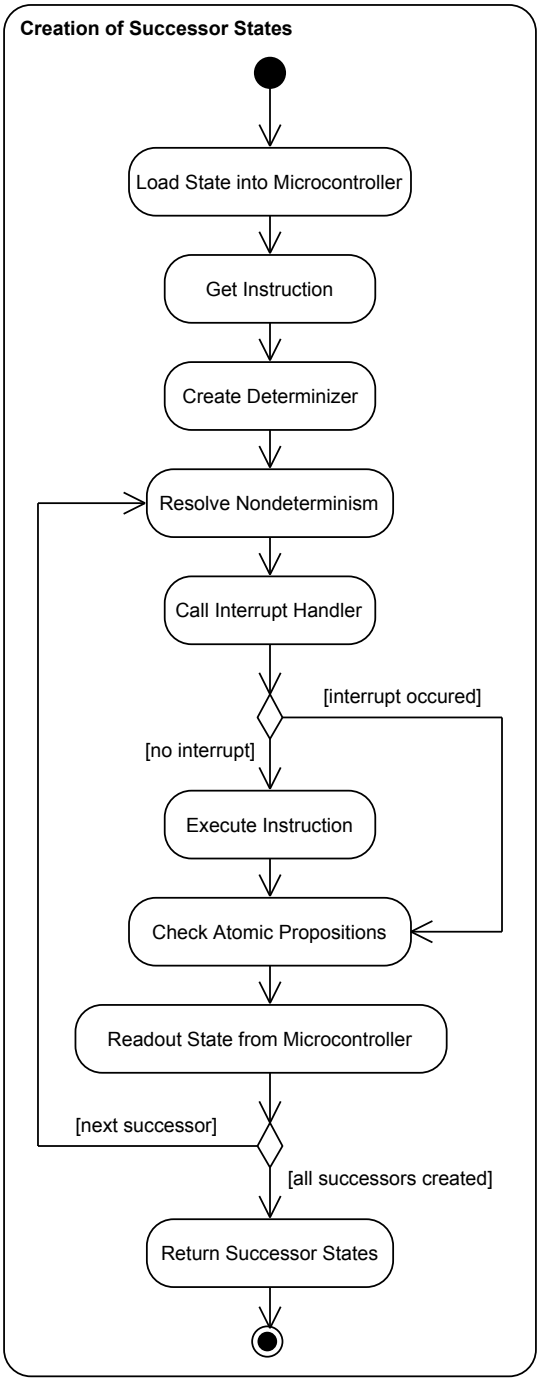

Figure 5.8: Process used to create successor states.

SimulaterState class. Then, the source state is loaded back into the *ATMega16* and the next loop cycle is executed. This loop continues until all possible successor states of the source state have been created. Finally, when all successor states have been created, the array of successor states, including the truth values of the atomic propositions, is returned by the *ATMega16InstructionSimulator* to the *State Space*.

As an example, consider the instruction ADD R1 R2. An ADD instruction adds the values of two general-purpose working registers, in this case R1 and R2. Here, it is assumed that nondeterminism is not involved. To execute the instruction, first the PC is incremented by two (size of the ADD instruction). Then, the registers R1 and R2 are read and some calculations are executed. After that, the bits in the SREG are set accordingly and the result of the operation is written into R1. Examples for instructions with nondeterminism are given in Sect. 5.6.

5.6 Determinizer

The *Determinizer* resolves nondeterminism, which is incorporated in many classes described in the preceding sections. It is introduced by the environment or the modeling of devices of the microcontroller including abstractions. In the previous sections, examples of both sources of nondeterminism are presented. I/O ports, for instance, introduce nondeterminism as they can be used to read input from the environment. Timers, on the other hand, introduce nondeterminism due to their modeling in [MC]SQUARE.

To close the system under verification [70], [MC]SQUARE has to resolve the nondeterminism. I/O registers can continuously generate nondeterministic values while the nondeterminism of other memory locations, which is induced by abstraction techniques and not by the nature of the memory locations, has to be resolved only once. After the resolution, the values of these memory locations are deterministic again. As mentioned before, this kind of nondeterminism is stored within the TBDM of the memory locations. Whether the value of an I/O register is nondeterministic or not is usually not stored within the TBDM, but determined by the corresponding *Register* class.

The *Determinizer* resolves the nondeterminism by instantiating the nondeterministic values in a process called *instantiation*. Section 5.6.1 gives an overview of the instantiation process applied in [MC]SQUARE. The subsequent two sections describe the two different instantiations implemented within [MC]SQUARE. The first is called immediate instantiation and the second delayed nondeterminism.

5.6.1 Determinizer Overview

The *Determinizer* controls the instantiation of nondeterminism, which is done by replacing abstract, nondeterministic values by all possible, concrete values. It is

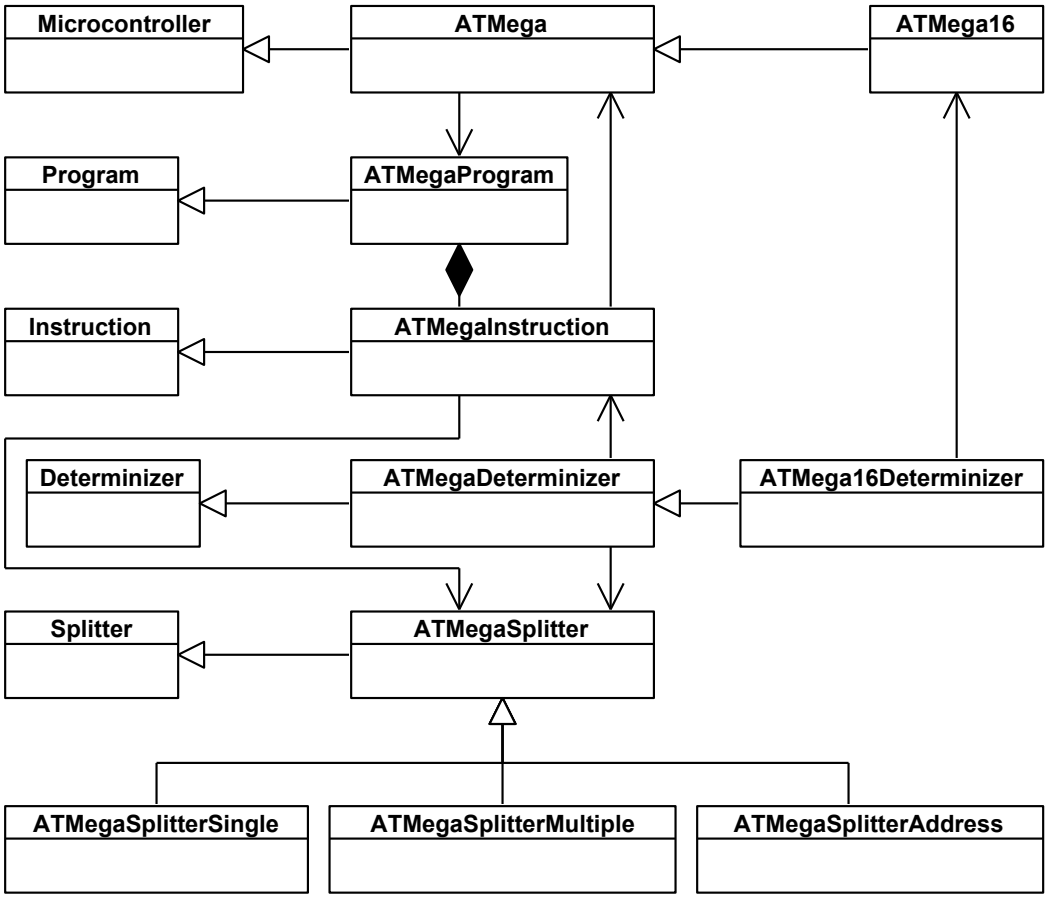

Figure 5.9: Classes of the *Simulator* package related to the *Determinizer*.

used by the *InstructionSimulator* to resolve the nondeterminism if required. In the instantiation process, we distinguish two cases, namely, the instantiation of interrupts and the instantiation of values used within the current instruction.

Figure 5.9 shows the UML class diagram of classes related to the *Determinizer*, for clarity, we omitted some details. The instantiation is conducted within the loop of the *InstructionSimulator*, in which successor states are created. The *InstructionSimulator* uses the *Determinizer* at the beginning of each loop cycle before a new successor state is created to adapt the *Microcontroller* accordingly. Each time the *Determinizer* is called, it writes a different combination of values. First, it instantiates possible interrupts and afterwards, it instantiates the values used within the current instruction.

Here, we detail this process for the ATmega16. The *ATMega16Determinizer* is

used by the *ATMega16InstructionSimulator* to instantiate nondeterministic values within the *ATMega16*.

Instantiation of interrupts works as follows. The *ATMega16Determinizer* first checks which interrupts are nondeterministic. To do so, it checks which of the enabled interrupts have an active source and which of the enabled interrupts are marked via the TBDM to be instantiated. Depending on the way instantiation is done, the number of possible interrupts differs. [MC]SQUARE currently supports immediate instantiation (see Sect. 5.6.2) and delayed instantiation, which is called delayed nondeterminism (see Sect. 5.6.3).

After that, the *ATMega16Determinizer* determines whether a deterministic interrupt, that is, a software interrupt, occurred. If so, only nondeterministic interrupts that have a higher priority than the deterministic interrupt can occur. If no deterministic interrupt occurred, all possible nondeterministic interrupts are triggered.

Every time the *ATMega16Determinizer* is called, it triggers another interrupt by setting the corresponding interrupt flag. To set an interrupt flag, the *ATMega16Determinizer* not only sets a single bit but a value combination, which ensures that no other interrupt is triggered. After one interrupt is triggered, the *ATMega16InstructionSimulator* executes the rest of the loop to create the state and then calls the *ATMega16Determinizer* again, which triggers the next interrupt. This continues until all nondeterministic interrupts are triggered.

After the instantiation of interrupts, if there is no deterministic interrupt, the *ATMega16Determinizer* starts instantiating the values used within the actual instruction. If there is a deterministic interrupt, the instruction is not executed in the current state, and so, the *ATMega16Determinizer* exits without instantiating the values used within the instruction.

To instantiate the values used within the actual instruction, the *ATMega16Determinizer* calls a method located in the *ATMegaInstruction* class to determine whether nondeterminism has to be resolved or not, which depends on the instruction and the kind of instantiation used. If nondeterministic values need to be instantiated, the *ATMega16Determinizer* gets an *ATMegaSplitter* instance from the *ATMegaInstruction*. The *ATMegaSplitter* is used to instantiate the values of the affected memory locations. The *ATMegaInstruction* passes the memory locations and their bits to the *ATMegaSplitter* for instantiation. Depending on this information, one of the different types of *ATMegaSplitters* is chosen, either, *ATMegaSplitterSingle*, *ATMegaSplitterMultiple*, or *ATMegaSplitterAddress*. The chosen *ATMegaSplitter* is used to assign all possible value combinations to the involved memory locations. Each time the *ATMega16Determinizer* is called by the *ATMega16InstructionSimulator*, it assigns another value combination until all possible combinations are assigned. If no values need to be instantiated, the *InstructionSimulator* just executes the current instruction once.

In the following two sections, we detail the two different ways to instantiate

Table 5.1: Four interrupts exemplifying instantiation of nondeterminism.

	i0	i1	i2	i3
source active	+	+	+	+
interrupt enabled	+	+	-	+
interrupt flag	∗	∗	∗	∗

nondeterminism in [MC]SQUARE, that is, immediate instantiation and delayed nondeterminism.

5.6.2 Immediate Instantiation

When using immediate instantiation, instantiation is conducted whenever a memory location holding a nondeterministic value is accessed, for example, by reading an I/O port or during handling of interrupts. During immediate instantiation all possible values are assigned to a memory location even if the value of the memory location is not used afterwards. The TBDM of memory locations is never written when using immediate instantiation because nondeterminism has to be resolved immediately, and the TBDM is solely used to delay instantiation. In the following, we first explain the immediate instantiation of interrupts, and then, we detail the immediate instantiation of values.

Immediate Instantiation of Interrupts

The *Determinizer* checks which sources of interrupts are active. If a source of an interrupt is active, the corresponding flag is nondeterministic. During the handling of interrupts, all flags belonging to an active interrupt source are instantiated. That is, all possible combinations of values for these flags are assigned. Thus, the nondeterminism involved in handling interrupts is resolved completely.

However, there are many value combinations that are not needed because in some value combinations lower-priority interrupts are blocked by higher-priority interrupts, and some interrupts may not be enabled at all. Thus, this kind of instantiation creates many unnecessary successor states, in fact, it creates 2^s different states, where s is the number of active interrupt sources.

Table 5.1 shows an example with four interrupts (i0–i3). The sources of all four interrupts are active, but only three of these interrupts are enabled. As the sources of all four interrupts are active, the flags of all interrupts are nondeterministic, which is encoded as $*$. Thus, immediate instantiation creates all 16 possible combinations as shown in Tab. 5.2. The value of the flag register is different in all cases, but in many cases the same interrupt handler is called. That is, when handling the interrupts by calling the IH of the highest-priority interrupt, only the following four

Table 5.2: Immediate instantiation of four interrupts (see also Tab. 5.1).

i0	i1	i2	i3	
0	0	0	0	→ no interrupt
0	0	0	1	→ interrupt 3
0	0	1	0	→ no interrupt
0	0	1	1	→ interrupt 3
0	1	0	0	→ interrupt 1
0	1	0	1	→ interrupt 1
0	1	1	0	→ interrupt 1
...	...	...	...	...
1	1	1	0	→ interrupt 0
1	1	1	1	→ interrupt 0

cases are distinguishable: either interrupt 0, interrupt 1, interrupt 3, or no interrupt occurs.

Immediate Instantiation of Values

Values of memory locations are instantiated similarly. Whenever a memory location holding a nondeterministic value is accessed, this value is directly instantiated. If a nondeterministic value is read from an I/O port and written to a register, 256 different states are created. This is done even if the value is not required, or only parts of the value are needed.

The program shown in List. 5.1 reads two values from PINA and PINB. Both ports are nondeterministic in this example. It then tests whether bit 2 in R18 is cleared. If the bit is cleared, the next instruction is skipped. Otherwise, the program resets the Watchdog Timer via WDR. Then it checks whether bit 3 in R19 is cleared. If the bit is cleared, it skips the next instruction and resets the Watchdog Timer. Using immediate instantiation, 256 different successor states are created at line 2. At line 3, these 256 different states give rise to 65,536 successors, which advance through the rest of the program. Only 4 of these 65,536 traces are relevant for this program fragment because only bit 2 of R18 and bit 3 of R19 are used. All other bits are unused and hence are not relevant for this program fragment assuming that R18 and R19 are not utilized in a formula provided by the user.

The example shows that immediate instantiation can cause an exponential blowup of the state space. Instantiating an eight bit value yields 256 successors states. If this is done in two consecutive lines, already 65,536 different states are created. The instantiation of interrupts causes a similar overhead. Our aim is to avoid the overhead created by immediate instantiation. That is, instantiation should only be

Listing 5.1: Example program for instantiation of values.

```
1 ...
2 IN R18 PINA
3 IN R19 PINB
4 SBRC R18 2
5 WDR
6 SBRC R19 3
7 WDR
8 ...
```

executed if and when the concrete value is required.

5.6.3 Delayed Nondeterminism

Delayed nondeterminism (*DND*) is an abstraction technique that tries to avoid the overhead caused by immediate instantiation by delaying the instantiation of nondeterministic values. Delayed nondeterminism tries to delay instantiation of nondeterministic values for as long as possible and to instantiate only the required parts. Hence, this abstraction technique has two aspects. First, it instantiates only those bits (bytes) that are needed by the current computation and hence, all other bits (bytes) remain nondeterministic. Second, it defers the instantiation of nondeterministic values until they are really needed, and thus, successor states are created at a later moment. Both aspects help to lower the size of the resulting state space. To implement this abstraction technique, [MC]SQUARE uses the TBDM to record which parts of a memory location need to be instantiated.

DND introduces lazy states into [MC]SQUARE, that is, a state no longer represents a single state but a set of states. Some parts of a lazy state are explicit, other parts are symbolic. The symbolic parts, which are induced by nondeterministic values, are instantiated lazily, that is, whenever they are accessed by the *Simulator*. Model checking is still conducted using explicit algorithms. Each time a concrete value is needed by the *Simulator* for simulation or evaluation of atomic propositions, nondeterministic values are instantiated. The *Model Checker* only accesses the atomic propositions, which are evaluated by the *Simulator*. Section 5.7 describes the formal model of the simulator implemented in [MC]SQUARE and uses it to prove correctness of this abstraction technique. In the following, we detail delayed nondeterminism for interrupts first and for values afterwards.

Delayed Nondeterminism of Interrupts

When using immediate instantiation, all bits (flags) used for triggering interrupts are instantiated the moment interrupt handling is performed. This is even done when the corresponding interrupts are not enabled because the nondeterminism is resolved for the flag registers, and the flag registers only depend on the sources of the respective interrupts.

In delayed instantiation, a bit triggering an interrupt is only instantiated if both the interrupt and its source are enabled, or if the interrupt is enabled and marked to be nondeterministic by means of the TBDM. Additionally, it is checked whether the corresponding interrupt is not blocked by higher-priority interrupts. If the interrupt is not enabled or a low-priority interrupt is blocked by a high-priority interrupt, the bits of this interrupt are not instantiated and remain nondeterministic. That is, DND for interrupts only instantiates flags of interrupts that are actually handled. Thus, in contrast to 2^s, where s is the number of active interrupt sources, only $a+1$, where $a \leq s$ is the number of active interrupts, different states are created.

DND for interrupts is possible because handling of interrupts is done in the same way on the real microcontroller (see also Sect. 5.3.4). The bits that stay nondeterministic when using DND for interrupts are not considered by the microcontroller either. Handling of interrupts is done by the microcontroller as follows. Starting with the highest-priority interrupt, the handler checks whether this interrupt has been triggered and is enabled. If it has been triggered, the test is aborted, and the flag values of the lower-priority interrupts are not of interest and can remain nondeterministic in our model, denoted by $*$. If it has not been triggered, an interrupt with a lower priority is checked. This continues until either a triggered interrupt is found, or it is detected that no interrupt has occurred. The *Determinizer* applies the same procedure when instantiating interrupts using delayed instantiation.

Whenever a source of nondeterminism is deactivated, the respective registers have to record that their values are still nondeterministic because they usually determine this by querying their source, which is now deactivated. Hence, they need to remember that their values have to be instantiated once. Each register has a TBDM, which is used to record whether its value is nondeterministic and has to be instantiated. When a source of nondeterminism is deactivated, the *Microcontroller* sets the corresponding flags in the TBDMs of the respective registers.

For example, if interrupt 0 and interrupt 1 occur at the same time, the flag of interrupt 0 is instantiated first in delayed instantiation. If the IH of interrupt 0 now deactivates the source of interrupt 1, occurrence of interrupt 1 would no longer be possible. This would be erroneous. To handle this situation correctly, the flag of interrupt 1 is marked in the TBDM of the corresponding interrupt flag register when its source is deactivated. Thus, it is guaranteed that this value will be instantiated later. After the IH of interrupt 0 is left, interrupt 1 can occur once more. If the IH

Table 5.3: Delayed instantiation of four interrupts (see also Tab. 5.1).

i0	i1	i2	i3	
1	∗	∗	∗	→ interrupt 0
0	1	∗	∗	→ interrupt 1
0	0	∗	1	→ interrupt 3
0	0	∗	0	→ no interrupt

of interrupt 0 deactivates interrupt 1 and not its source, interrupt 1 cannot occur afterwards. This is identical to the behavior observed on the microcontroller.

[MC]SQUARE cannot handle nondeterminism in registers that are used within the formula because it uses explicit model checking algorithms. Therefore, DND is forbidden for registers contained within the formula being checked. As DND for interrupts cannot be deactivated in contrast to DND for values, propositions about interrupt flag registers are not allowed within formulas.

Given the interrupt configuration shown in Tab. 5.1, the *Determinizer* only creates the four combinations shown in Tab. 5.3 when using delayed instantiation. Compared with immediate instantiation, the same IHs are called when using delayed instantiation of interrupts. However, the values of the flag registers are different as in immediate instantiation all combinations are created, and in delayed instantiation some values remain nondeterministic.

Summarizing, DND for interrupts handles interrupts as the microcontroller does. Nondeterministic interrupts are only instantiated if the corresponding interrupt is enabled and can be triggered. Hence, the overhead of unneeded instantiations is avoided. In Sect. 5.7, we prove that DND preserves a simulation relation using a formal model of the simulator. Section 7.3 details, for each abstraction technique, which logic remains valid.

Delayed Nondeterminism of Values

In immediate instantiation, all nondeterministic values are instantiated immediately the moment they are accessed. When using delayed nondeterminism, instantiation is delayed until the values are actually needed for a computation, and instantiation is only performed for the required parts of the values. That is, the complete byte, a single bit, or nothing is instantiated as needed. The *Instruction* class determines whether values have to be instantiated and which parts of the values have to be instantiated.

Whether values have to be instantiated and which parts of the values have to be instantiated depends on the respective instruction and the memory locations accessed by the instruction. There are instructions for which nondeterministic

values *may* be instantiated such as IN, LD, or MOV and instructions for which nondeterministic values *must* be instantiated, for example, ADD, SUB, or SBIS.

In the *may* case, instantiation is only performed if nondeterminism is copied into memory locations where it is not allowed, such as I/O registers. Otherwise, instantiation is not performed. When DND copies nondeterminism into a memory location, it sets the value of the memory location to zero and adapts the TBDM of the memory location accordingly. In this way the memory location records that its value is nondeterministic.

In the *must* case, nondeterministic values are always instantiated because they are needed for the execution of the instruction. For example, an ADD instruction adds up the values of two registers and writes the result to one of the involved registers. During this operation, it sets the bits of the SREG, for which nondeterministic values are forbidden, accordingly. These bits can only be set if the result of the operation is explicit. When such an instruction is executed, the nondeterminism in affected memory locations is instantiated by creating all required combinations of values.

Delayed instantiation is forbidden for some memory locations. It is not allowed for I/O registers because these registers control the behavior of the *Microcontroller*. A nondeterministic value in an I/O registers could introduce nondeterminism in the control flow. Some I/O registers contain nondeterministic values or are the source of nondeterminism in other I/O registers, but writing nondeterministic values into these registers is not allowed. Furthermore, DND is forbidden for memory locations used within formulas because [MC]SQUARE uses explicit model checking algorithms, and these algorithms cannot handle nondeterministic values.

As for DND for interrupts, if a source for nondeterminism is deactivated, some registers have to record that their value is nondeterministic. This is done in DND for values as it is done in DND for interrupts. The TBDM of the respective memory location is set accordingly. For example, flag registers record the nondeterminism because the values of these registers are remembered on the real microcontroller too. On the other hand, PINx registers do not record the nondeterminism because they only hold nondeterministic values while the corresponding ports are used for input. Once an I/O port is changed to output, the value of PINx is no longer nondeterministic.

The program shown in List. 5.1 reads two input values, and depending on these input values, it resets the Watchdog Timer. When using immediate instantiation, 65,536 different traces leave this program fragment, when only one trace enters it. We mentioned in the previous section that only four of these traces are relevant for this program fragment. Delayed instantiation only creates these four traces. It works as follows. In line 2, input from PINA is read. As the concrete value is not needed yet, the nondeterminism is copied from PINA to R18. In the next line, the same is done with PINB and R19. After line 3, just one trace exists. In line 4, the

concrete value of bit 2 of R18 is needed. Hence, before executing this statement, [MC]SQUARE instantiates this bit. This results in two successor states. One state in which bit 2 is zero, and the other state where bit 2 is one. In line 6, the same is done with bit 3 of R19. As this line is entered by two traces, it is left by four traces. That is, using DND only 4 traces leave the program fragment instead of 65,536.

The results are different if one of the registers is used within the formula. If, for example, R18 is used within the formula, 256 successor states are created in line 2. These 256 traces then run through the rest of the program. The delayed instantiation of R19 is not influenced in this case.

In delayed instantiation of interrupts, the instantiation is delayed and only performed for required parts. It has an important effect on the size of the state space. The user cannot deactivate it as the state space of most programs using more than one interrupt would be too large to be handled by [MC]SQUARE.

In delayed instantiation of values, the instantiation is delayed, only performed for required parts, and additionally, nondeterminism is copied into other locations. That is, it is delayed even longer. Therefore, it also has a significant effect on the size of the state space, but it can introduce extra behavior through the copying of nondeterminism into other memory locations. Hence, we implemented it as an option that can be deactivated by the user if the over-approximation is too coarse. A coarser over-approximation increases the probability of false alarms.

Section 5.7 sketches a proof that shows that DND preserves a simulation relation. Section 7.3 details, for each abstraction technique, which logic remains valid. Chapter 8 demonstrates the effect of DND via two case studies.

5.7 Formal Model of the Simulator

This section introduces our formal modeling approach for microcontroller systems, consisting of hardware, software, and environment. The motivation of this development is twofold. First, it enables us to formally establish the correctness of our abstraction techniques. In this section, we prove that *delayed nondeterminism* preserves a simulation relation (see Sect. 5.7.4). That is, delayed nondeterminism yields an over-approximation: every possible behavior of the original system is also represented in the abstract system (the abstract system simulates the concrete system).

Second, the formal model allows us to formalize a large number of existing microcontroller system. Thus, it can be used as a kind of intermediate specification, supporting the rapid development of model checking tools for embedded systems. We published the formal model elsewhere [60, 88].

5.7.1 Handlers and Guarded Assignments

In our approach, the state of a (microcontroller) system is decomposed into a control state and a data state. We assume that the data space is organized as a global memory with linear byte addresses. The latter are denoted by A and are assumed to have a length of m bytes (in our application, $m = 2$). Thus, $A \triangleq \mathbb{C}^m$ where $\mathbb{C} \triangleq \mathbb{B}^8$ and $\mathbb{B} \triangleq \{0, 1\}$. Here, the bth bit of a byte $c \in \mathbb{C}$ is denoted by $c[b]$.

In order to incorporate nondeterminism, we extend this definition by introducing a *nondeterministic bit value* $*$, and let $\mathbb{B}_* \triangleq \mathbb{B} \cup \{*\}$ and $\mathbb{C}_* \triangleq \mathbb{B}_*^8$. Moreover we distinguish a set of *deterministic addresses* $D \subseteq A$ in which only deterministic values are allowed to be stored. These will later be used for certain I/O registers and for the (symbolic) addresses occurring in the formula to be verified. Thus, *memory states* can be represented by mappings from the set $V := \{v \mid v : A \to \mathbb{C}_*\}$ where $v(a) \in \mathbb{C}$ for every $a \in D$.

The behavior of a system is determined by its current *control location*, which is the program counter in our case. It is represented by a finite set Q. Thus, the set of *(system) states* is given by $S \triangleq Q \times V$. State changes are specified by three so-called *handlers*:

- a *nondeterminism handler* of the form $g_1; \ldots; g_k$ where $k \geq 0$, which introduces nondeterministic values where necessary,
- an *interrupt handler* of the form $h_1 : q_1 > \ldots > h_l : q_l$ where $l \geq 0$ and $q_1, \ldots, q_l \in Q$, which specifies the system's reaction to events such as interrupts, and
- for each control location $q \in Q$, an *instruction handler* of the form $q : h'_1 : q'_1 > \ldots > h'_m : q'_m$ where $m \geq 1$ and $q'_1, \ldots, q'_m \in Q$, which defines the normal execution of machine instructions.

Each g_i, h_i, h'_i is a *guarded assignment* of the form $e_0 \to x_1 := e_1, \ldots, x_n := e_n$ where $n \geq 0$, $e_0, \ldots, e_n$ are value expressions, and $x_1, \ldots, x_n$ are (disjoint) address expressions (see below). A guarded assignment is called *enabled* if its *guard* e_0 evaluates to 1 in the current memory state. Its execution yields a new memory state in which, for every $1 \leq i \leq n$, the value stored at x_i is determined by e_i. The guard e_0 can be omitted if it is the constant 1.

Given a current state $(q, v) \in S$, the next state is determined by

1. executing every enabled guarded assignment g_i in the nondeterminism handler in the given order, followed by
2. an application of the first enabled guarded assignment h_i in the interrupt handler, stopping at the corresponding control location q_i. If no such assignment exists, then

3. again the complete nondeterminism handler is executed, and finally,
4. the first enabled guarded assignment h'_j in the instruction handler for q is applied, stopping at q'_j.

Formally, given handlers of the above form, the successor state $(q', v') \in S$ is defined as follows:

$$(q', v') \triangleq \begin{cases} (q_i, [\![h_i]\!](v_1)) & \text{if } I \neq \emptyset \text{ and } i = \min I \\ (q'_j, [\![h'_j]\!](v_2)) & \text{if } I = \emptyset, J \neq \emptyset, \text{ and } j = \min J \\ (q, v_2) & \text{if } I = J = \emptyset \end{cases}$$

where

$$\begin{aligned} v_1 &\triangleq [\![g_k]\!](\ldots([\![g_1]\!](v))\ldots), \\ v_2 &\triangleq [\![g_k]\!](\ldots([\![g_1]\!](v_1))\ldots), \\ I &\triangleq \{i \in \{1, \ldots, l\} \mid h_i \text{ enabled in } v_1\}, \text{ and} \\ J &\triangleq \{j \in \{1, \ldots, m\} \mid h'_j \text{ enabled in } v_2\}. \end{aligned}$$

Here, $[\![g]\!] : V \to V$ denotes the meaning of a guarded assignment g as a mapping on memory states; it will be defined in Section 5.7.3.

In our application, the nondeterminism handler is employed to deal with nondeterminism: it checks which I/O registers or which parts of I/O registers are nondeterministic and writes $*$ values into the corresponding nondeterministic I/O registers. These I/O registers include, for example, input registers and interrupt flag registers. The interrupt handler conducts the actual processing of interrupts. It first tests, in the order of descending interrupt priority, whether an interrupt is raised, and if so jumps to the corresponding interrupt handler. If no interrupt was handled (i.e., $I = \emptyset$), the nondeterminism handler is run again and finally, the actual machine instruction at the current location is executed by applying the corresponding instruction handler.

Here, the repeated call of the nondeterminism handler is required since after the interrupt handler has ignored a cleared interrupt flag, the latter could be set by an external event before the machine instruction is executed. For example, if during the handling of interrupts the flag storing a timer overflow was not set, it can still be set before the execution of the actual instruction as time elapses during the handling of interrupts. Moreover, it is important to observe that the nondeterminism handler performs every enabled guarded assignment while the execution of the interrupt and the instruction handler stops after applying the first enabled guarded assignment. Instantiation of nondeterministic values is conducted during the handling of interrupts and the executing of instructions. Section 5.7.3 provides details on the instantiation of nondeterministic values.

As mentioned earlier, each guarded assignment is of the form $e_0 \to x_1 := e_1, \ldots, x_n := e_n$ with value expressions e_i and address expressions x_j. *Address*

expressions are of the form a or $a{\downarrow}+d$ or $a[b]$ where $a \in A$, $b \in \{0,\ldots,7\}$, and $d \in \mathbb{Z}$. The first two are byte addresses, either given directly or indirectly by dereferencing the address stored at a and adding displacement d. The expression $a[b]$ refers to the bth bit of the byte which is stored at a.

Value expressions are of the form $op(y_1,\ldots,y_k)$ where op is an operation of type $op : T_1 \times \ldots \times T_k \to T_0$ such that, for every $1 \leq j \leq k$, $T_j \in \{\mathbb{C}, \mathbb{C}_*, \mathbb{B}, \mathbb{B}_*\}$ and y_j is an address expression. Here, we always assume that operations respect memory sizes, that is, that $T_j \in \{\mathbb{C}, \mathbb{C}_*\}$ ($T_j \in \{\mathbb{B}, \mathbb{B}_*\}$) whenever y_j denotes a byte (bit) address. A similar restriction applies to the result type T_0 and to the corresponding left-hand side address x_i. We require the result type of the guard e_0 to be $\mathbb{B}$.

The *semantics of an address expression* α depends on the current memory state $v \in V$, and is denoted by $[\![\alpha]\!]_v$. For byte address expressions, we let $[\![a]\!]_v \stackrel{\Delta}{=} a \in A$ and $[\![a{\downarrow}+d]\!]_v \stackrel{\Delta}{=} a' + d \in A$ if $a' = v(a)\ldots v(a+m-1) \in A$. Thus, in the second case, the result is the address a' which is stored at a, adding displacement d. The semantics is undefined if a' is not a valid address, that is, contains a nondeterministic bit value $*$. For bit address expressions, we let $[\![a[b]]\!]_v \stackrel{\Delta}{=} (a,b) \in A \times \{0,\ldots,7\}$.

To determine the *semantics of a value expression*, we have to apply the corresponding operation to the argument values: if $op : T_1 \times \ldots \times T_k \to T_0$ and $[\![y_j]\!]_v \in T_j$ for every $1 \leq j \leq k$, then $[\![op(y_1,\ldots,y_n)]\!]_v \stackrel{\Delta}{=} op([\![y_1]\!]_v,\ldots,[\![y_k]\!]_v)$. Otherwise, the result is undefined.

Note that the admissible types of operations in value expressions support nondeterministic bit values as both arguments and results. Thus, it is possible, for example, to describe a simple copy instruction by choosing the identity on $\mathbb{C}_*$ or $\mathbb{B}_*$ as the operation. On the other hand, nondeterministic values in argument addresses can be excluded by choosing the argument type $\mathbb{C}$ or $\mathbb{B}$. In such cases, access to an address containing a nondeterministic values requires instantiation; see Sect. 5.7.3 for details. Moreover, it is possible to mix deterministic and nondeterministic values; if, for example, a $*$ bit is multiplied by 0, the operation can still be evaluated as 0.

The next section shows how the microcontroller system under consideration can be represented by our formal model. Afterwards, we continue with formally defining the meaning of guarded assignments.

5.7.2 Modeling the ATMEL ATmega16

In order to model the execution of machine code on the ATMEL ATmega16 microcontroller, the general framework developed in the previous section is instantiated as follows:

- Since the machine code is stored in flash memory, control locations in Q correspond to flash memory addresses.

- Each address comprises $m \triangleq 2$ bytes.
- The following distinguished addresses are denoted by symbolic names:
  - general-purpose registers R0, ..., R31
  - indirect addressing registers X = R27:R26, Y = R29:R28, Z = R31:R30
  - I/O registers such as
    - status register SREG with flag bits C (= 0), Z (= 1), N (= 2), ..., I (= 7)
    - timer registers such as TIMSK, TIFR and TCCR0,
    - interrupt registers such as GICR and GIFR,
    - stack pointers (SPL, SPH),
    - data direction registers (DDRA, ...),
    - port registers (PORTA, ...) and
    - port input registers (PINA, ...),

    and single bit positions within these (CS00, ...),
  - C variables used in the application program.
- The deterministic addresses in D comprise the addresses which are referenced in the formula to be verified and certain I/O registers such as the DDR, PORT, and TCCR registers.
- The nondeterminism handler, which is repeatedly executed before the interrupt and the instruction handler, writes $*$ values (nondeterminism) into the nondeterministic I/O registers. This is done, for example, for input registers and interrupt flag registers. Here, we consider an input register and two interrupt flag registers (timer and external interrupt); other registers are handled similarly:

  $$\begin{aligned}&\mathsf{PINA} := (\mathsf{DDRA} \wedge \mathsf{PORTA}) \vee (\neg\mathsf{DDRA} \wedge *^8);\\ &\mathsf{TCCR0}[\mathsf{CS02}] = 1 \vee \mathsf{TCCR0}[\mathsf{CS01}] = 1 \vee \mathsf{TCCR0}[\mathsf{CS00}] = 1\\ &\rightarrow \mathsf{TIFR}[\mathsf{TOV0}] := nd(\mathsf{TIFR}[\mathsf{TOV0}]);\\ &\mathsf{DDRB}[\mathsf{DDB2}] = 0 \rightarrow \mathsf{GIFR}[\mathsf{INTF2}] := nd(\mathsf{GIFR}[\mathsf{INTF2}]); \ldots\end{aligned}$$

  where $nd : \mathbb{B}_* \rightarrow \mathbb{B}_*$ is defined by $nd(*) := *$, $nd(0) := *$, and $nd(1) := 1$. This function is used for interrupt flags because a 1 in an interrupt flag remains unchanged even if the source of the interrupt is enabled. A one is never set to $*$.

- The interrupt handler is specified as follows (again considering timer and external interrupts):

$$\mathsf{SREG[I]} = 1 \wedge \mathsf{TIMSK[TOIE0]} = 1 \wedge \mathsf{TIFR[TOV0]} = 1 \rightarrow : 18\downarrow \; > \\ \mathsf{SREG[I]} = 1 \wedge \mathsf{GICR[INT2]} = 1 \wedge \mathsf{GIFR[INTF2]} = 1 \rightarrow : 36\downarrow > \ldots$$

- Every machine instruction, which is stored at some location $q \in Q$, gives rise to an instruction handler. Some exemplary instructions:
  - **ADD Ri,Rj:** $q : \mathsf{R}i := \mathsf{R}i + \mathsf{R}j, \mathsf{SREG[Z]} := (\mathsf{R}i + \mathsf{R}j = 0), \mathsf{SREG[C]} := \ldots, \ldots : q + 2$
  - **RJMP** k**:** $q :: q + k + 1$
  - **IJMP:** $q :: \mathsf{Z}\downarrow$
  - **JMP** k**:** $q :: k$
  - **SBRC Ri,b:** $q : \mathsf{R}i[b] = 0 \rightarrow : q + 2 > \mathsf{R}i[b] = 1 \rightarrow : q + 3$
  - **BREQ** k**:** $q : \mathsf{SREG[Z]} = 1 \rightarrow : q + k + 1 > \mathsf{SREG[Z]} = 0 \rightarrow : q + 2$
  - **MOV Ri,Rj:** $q : \mathsf{R}i := \mathsf{R}j : q + 2$
  - **LD Ri,X+:** $q : \mathsf{R}i := \mathsf{X}\downarrow, \mathsf{X} := \mathsf{X} + 1 : q + 2$
  - **LD Ri,-X:** $q : \mathsf{R}i := \mathsf{X}\downarrow - 1, \mathsf{X} := \mathsf{X} - 1 : q + 2$
  - **LDD Ri,X+d:** $q : \mathsf{R}i := \mathsf{X}\downarrow + d : q + 2$
  - **IN Ri,A:** $q : \mathsf{R}i := \mathsf{PINA} : q + 2$

5.7.3 Coping with Nondeterminism

In our formal model, nondeterministic bit values can arise due to the application of an operation $op : T_1 \times \ldots \times T_k \rightarrow T_0$ with result type $T_0 \in \{\mathbb{C}_*, \mathbb{B}_*\}$. In the simplest case, op is just the identity, that is, the assignment is of the form $x \stackrel{\Delta}{=} y$ with $[\![y]\!]_v = a \in A \setminus D$ and $v(a) \notin \mathbb{C} \cup \mathbb{B}$. An example is the IN instruction. The IN instruction reads input from an I/O port, and hence, it may introduce nondeterministic values.

In the standard implementation, this situation is handled by *immediate instantiation* (see Sect. 5.6.2), meaning that each assignment of nondeterministic bit values is resolved by considering all possible assignments of concrete values. It is clear that this involves an exponential blowup, for example the assignment of byte value $*^8$ gives rise to 256 different successor states.

Our goal is to avoid this overhead by *delaying nondeterminism* (see Sect. 5.6.3), that is, by replacing nondeterministic by concrete values only if and when it is required by the following computation. As mentioned before, "if" and "when" refer to two different aspects of this optimization, which both lead to a reduction of the

number of states created. First, delayed nondeterminism only instantiates those bits (bytes) that are used by some instruction, and hence, all other bits (bytes) may remain nondeterministic. This lowers the number of successors which have to be created. Second, delayed nondeterminism defers the splitting of nondeterministic values until they are really needed. Hence, successors are created at a later point in time. Both aspects help to minimize the number of created states, while still preserving a safe over-approximation (shown in Sect. 5.7.4).

In order to formally develop this abstraction technique, we introduce a partial order $\sqsubseteq \subseteq \mathbb{B}_* \times \mathbb{B}_*$, given by $0 \sqsubseteq *$ and $1 \sqsubseteq *$, and lift it to bytes and memory states by pointwise extension: $c[7] \dots c[0] \sqsubseteq c'[7] \dots c'[0]$ iff $c[b] \sqsubseteq c'[b]$ for every $b \in \{0, \dots, 7\}$, and $v \sqsubseteq v'$ iff $v(a) \sqsubseteq v'(a)$ for every $a \in A$. Thus $v \sqsubseteq v'$ if v' is "more general" than v.

Immediate Instantiation

Immediate instantiation is explained in Sect. 5.6.2. It follows the principle that in the course of the computation only deterministic values may be stored. (Nevertheless, it is still possible, due to the initial choice of the memory state, that $v(a) \in \mathbb{C}_* \setminus \mathbb{C}$ for specific addresses $a \in A \setminus D$, such as $a = \mathsf{PINA}$.)

We say that a guarded assignment of the form $e_0 \to x_1 := e_1, \dots, x_n := e_n$ is *enabled* in memory state $v \in V$ if $[\![e_0]\!]_v = 1$. Its execution nondeterministically yields every $v' \in V$, which is obtained by first evaluating every right-hand side expression e_i, by taking every possible instantiation of nondeterministic bit values, and by updating v accordingly. Formally, $v' \stackrel{\Delta}{=} v[[\![x_i]\!]_v \mapsto c_i; 1 \le i \le n]$ such that $c_i \in \mathbb{C} \cup \mathbb{B}$ with $c_i \sqsubseteq [\![e_i]\!]_v$ for every $1 \le i \le n$. Here $v[a \mapsto c]$ denotes the modification of v at address a by storing the new value c.

Composing the effects of the nondeterminism handler, interrupt handler, and instruction handler for the current control location $q \in Q$ as described in Section 5.7.1, we obtain a *concrete transition* $(q, v) \xrightarrow{h} (q', v')$ where h is the first enabled guarded assignment of the interrupt or the instruction handler, and q' is the corresponding successor location. Given an initial system state $s_0 \in S$, this yields a *concrete transition system* $T^c = (S, \bigcup_{h \in G} \xrightarrow{h}, s_0)$ where the set G collects all guarded assignments in the interrupt and instruction handlers.

Delayed Nondeterminism

As described in Sect. 5.6.3, the goal of delayed nondeterminism is to instantiate nondeterministic bit values as late as possible, that is, not necessarily when they are computed, but only if and when they are required by a subsequent computation step (i.e., during handling of interrupts or execution of instructions). More concretely, the

instantiation of bit address $(a, b) \in A \times \{0, \ldots, 7\}$ with $v(a)[b] = *$ in state $(q, v) \in S$ is required for a guarded assignment g of the form $e_0 \to x_1 := e_1, \ldots, x_n := e_n$ if

- (a, b) is referred by the guard e_0 (*guard* case), or
- g is enabled and some e_i refers to (a, b) in an operation argument position which does not allow nondeterministic bit values (*argument*), or
- g is enabled and some x_i dereferences a (*indirection*), or
- g is enabled and for some $1 \leq i \leq n$, the evaluation of e_i yields a nondeterministic value which cannot be stored at address x_i since it is in D (*target*).

We say that a bit address (a, b) is *referred to* by a value expression $op(y_1, \ldots, y_k)$ if it is referred to by some address expression y_j, which is the case if $y_j = a[b]$, $y_j = a$, $y_j = a{\downarrow} + d$, or $y_j = a'{\downarrow} + d$ for some $a' \in A$ such that $[\![a'{\downarrow} + d]\!]_v = a$.

We can formalize the above distinction of cases by the following incremental instantiation procedure: a guarded assignment g of the form $e_0 \to x_1 := e_1, \ldots, x_n := e_n$ yields $v' \in V$ if there exist intermediate memory states $v_1, v_2, v_3, v_4 \in V$ such that

1. $v_1 \sqsubseteq v$ with $v_1(a, b) \neq v(a, b)$ iff $v(a, b) = *$ and (a, b) is referred by e_0 (*guard*), and
2. g is enabled in (q, v_1), i.e., $[\![e_0]\!]_{v_1} = 1$, and
3. $v_2 \sqsubseteq v_1$ with $v_2(a, b) \neq v_1(a, b)$ iff $v_1(a, b) = *$, some e_i is of the form $op(y_1, \ldots, y_n)$ with $op : T_1 \times \ldots \times T_n \to T_0$, and (a, b) is referred by some y_j where $T_j \in \{\mathbb{C}, \mathbb{B}\}$ (*argument*), and
4. $v_3 \sqsubseteq v_2$ with $v_3(a, b) \neq v_2(a, b)$ iff $v_2(a, b) = *$, some x_i is of the form $a{\downarrow} + d$, and $b \in \{0, \ldots, 7\}$ (*indirection*), and
5. $v_4 \stackrel{\Delta}{=} v_3[[\![x_i]\!]_{v_3} \mapsto [\![e_i]\!]_{v_3}; 1 \leq i \leq n]$, and
6. $v' \leq v_4$ with $v'(a, b) \neq v_4(a, b)$ iff $v_4(a, b) = *$, $[\![x_i]\!]_{v_4} \in \{a, (a, b)\}$ for some $1 \leq i \leq n$, and $a \in D$ (*target*).

Similarly to the previous section, the composition of the nondeterminism handler, interrupt handler, and instruction handler yields *abstract transitions* of the form $(q, v) \stackrel{h}{\Longrightarrow} (q', v')$ where h and q' are again determined by the first enabled guard in the interrupt handler or the instruction handler for q. Together with an initial system state $s_0 \in S$, this induces an *abstract transition system* $T^a = (S, \bigcup_{h \in G} \stackrel{h}{\Longrightarrow}, s_0)$.

5.7.4 Establishing Correctness

Our goal is to verify the correctness of the program under consideration by model checking it with respect to a specification. The latter is given by a temporal formula over a set P of bit value expressions, which act as the atomic propositions. These propositions yield an extension of both the concrete and the abstract transition systems to *labeled transition systems (LTSs)*. The concrete LTS is of the form $L^c = (S, \bigcup_{h \in G} \xrightarrow{h}, s_0, \lambda)$ where $\lambda : S \to 2^P : (q, v) \mapsto \{p \in P \mid [\![p]\!]_v = 1\}$ labels each state by the set of all propositions valid in that state. As all addresses in the formula belong, by definition, to the set of deterministic addresses D, it is guaranteed that $[\![p]\!]_v$ is always defined. Analogously, $L^a = (S, \bigcup_{h \in G} \xRightarrow{h}, s_0, \lambda)$ is obtained in the abstract case.

The idea is to model check the abstract LTS rather than the concrete one. As we now show, every computation in L^c corresponds to a computation in L^a. This excludes false positives: whenever every abstract computation satisfies the given specification, this also applies to every concrete computation. The converse, however, is not true: copying nondeterministic bit values may have the effect that the same nondeterministic value is replaced by different concrete values. Thus, L^a is an over-approximation of L^c. This may lead to false negatives, that is, spurious computations in L^a that violate the specification.

Formally the connection between L^c and L^a is given by a *simulation* [84], which is a binary relation $\rho \subseteq S \times S$ such that $s_0 \rho s_0$ and, whenever $s_1 \rho s_2$,

- $\lambda(s_1) = \lambda(s_2)$ and
- for every transition $s_1 \xrightarrow{h} s_1'$, there exists $s_2' \in S$ such that $s_2 \xRightarrow{h} s_2'$ and $s_1' \rho s_2'$.

Indeed, it can be shown that L^a simulates L^c, that is, that every sequence of guarded assignments with immediate instantiation can be reproduced using delayed instantiation. More concretely, the simulation relation is given by the partial order on bit values: $(q_1, v_1) \rho (q_2, v_2)$ iff $q_1 = q_2$ and $v_1 \sqsubseteq v_2$. In summary, this means that our delayed nondeterminism abstraction is sound with respect to path-universal logics such as ACTL and LTL.

5.8 Related Work

This section presents related work regarding assembly code model checking, focusing on state space construction, and related work regarding the delayed nondeterminism abstraction technique. We present related work in this section because the model checkers mentioned mostly differ in the way they build the state space and the way they apply abstraction techniques.

5.8.1 Model Checking Machine Code

Other model checkers that model check machine code or code similar to machine code are: CodeSurfer/x86 and WPDS++ [7], Estes [82], Java PathFinder [122], MCESS [98, 105], and StEAM [73, 80, 81]. All these model checkers, except the tool set CodeSurfer/x86 and WPDS++, are explicit model checkers like [mc]square.

The model checking tool set consisting of CodeSurfer/x86, Path Inspector, and WPDS++ uses a translation approach to model check x86 executables. The translation process is different from the other model checkers described in this section. CodeSurfer/x86 extracts a model in the form of a weighted pushdown automaton from the x86 executable. This model is the intraprocedural CFG of the x86 executable. Then, the WPDS++ library performs the model checking. It implements a symbolic reachability algorithm. The Path Inspector serves as a user interface for automating certain safety queries. The properties that are checked with this tool set are intraprocedural data flow properties. These properties are usually checked by static analyzers. Another tool that conducts static analyses with the help of model checking is Goanna [44, 45]. Goanna checks C and C++ code rather than machine code.

The approach used in [mc]square is different. [mc]square creates the state space directly from the machine code and does not use a translation. Furthermore, the semantic model used by [mc]square is richer. The CodeSurfer/x86 tool set only checks the CFG of the program, which is a compile time model. In contrast, [mc]square creates a runtime model of the program. Therefore, the state spaces created by [mc]square are generally larger, but contain more information. Hence, [mc]square can be used to verify arbitrary properties of the program (runtime) and not just properties of the CFG of the program (compile time). The advantage of the CodeSurfer/x86 tool set and Goanna is that they can check larger programs.

The Estes model checker checks code written for the Motorola 68hc11 and the Hitachi H8/300 processors. The state space is built using the real hardware or a back-end simulator through the GNU debugger. The advantage of using the GNU debugger is that Estes does not have to deal with the semantics of the instructions. The disadvantage is that the creation of the state space cannot be influenced by, for example, different abstraction techniques or variable modelings. To model check a program, users have to provide an environment written in C++. This environment is primarily a set of locations (instruction addresses) where an environment response is needed or where property invariants need to be checked. Users have to compile the environment together with the source code of the model checker. Additionally, users have to specify the parts of the memory to be stored in the states, which is the memory abstraction. Estes can check for invariants, stack overflows, and read-only violations. Since Estes uses a notion of discrete time the state space explosion is

bigger than when model checking without time.

Despite the different processors used, [MC]SQUARE and ESTES differ in other ways. [MC]SQUARE abstracts from time, preserving an over-approximation, and therefore, the generated state spaces tend to be smaller than in ESTES. Model checking with time leads to real-time model checking [14, 71, 72]. The properties that can be checked with [MC]SQUARE are not restricted to invariants but can be arbitrary CTL formulas (for restrictions see Sect. 7.3). Furthermore, [MC]SQUARE conducts the same checks as ESTES does, for example, checks for stack overflows and unintended uses of microcontroller features. Moreover, users do not have to provide an environment that deals with nondeterminism. In our approach, we concentrate on the creation of the state space, that is, we focus on the domain-specific abstractions implemented within the simulator. We do not want to use existing simulators as we think that significant savings in space and time can be achieved by a tailored implementation.

JAVA PATHFINDER accepts Java bytecode and employs collapsing techniques for efficiently storing states. Our experiments have shown that such methods do not pay off in the case of [MC]SQUARE since its states are less complex. Another difference is that JAVA PATHFINDER has to deal with parallel processes and therefore applies abstraction techniques such as partial order reduction, which cannot be applied in [MC]SQUARE. Moreover, the memory model used within JAVA PATHFINDER makes it possible to apply symmetry reduction techniques. Again, this is not possible in [MC]SQUARE because the order of data within memory is important.

MCESS was developed in a diploma thesis in cooperation between the CWI and our institute. It works on assembly code for the ATMEL ATmega16 like [MC]SQUARE. MCESS does not model check the assembly code, but translates it into the bytecode used in the NIPS virtual machine (VM) [125, 126], which is used to build the state space and to perform model checking. Besides the different approach, MCESS model checks LTL formulas and [MC]SQUARE model checks CTL formulas. As MCESS uses a translation process, it cannot handle some constructs such as recursion and other constructs such as Two Wire Serial Interface not handled accurately enough.

STEAM model checks machine code for the Internet C Virtual Machine (ICVM). The machine code is compiled from C++ source code. STEAM focuses on model checking parallel and hardware-independent C++ programs. It builds the state space by monitoring a modified version of the ICVM, which simulates the program at the assembly code level. STEAM uses a modified version of the GNU C Compiler (GCC) to compile C++ code to the assembly code that is used within ICVM. If a new version of GCC is used, the ICVM has to be readapted. In contrast to STEAM, our approach aims at the verification of assembly code written for a specific microcontroller. As mentioned before, our main focus is the customization of the simulator used to build the state space. Thereby, we can influence the building of

the state space and adapt the degree of over-approximation with various abstraction techniques.

There are other approaches to the verification of machine or object code than model checking. Peleska and Haxthausen [89] describe an approach that checks whether an assembly program is a correct implementation of a SystemC model. The assembly program and the SystemC model are mapped to behavioral models. Then, it is proven that the behavioral models are equivalent by applying transformations. This approach works for certain SystemC models that are automatically generated from higher-level models.

Subramanian and Cook [114] detail a different approach that uses the theorem prover NQTHM to automatically verify a small subset of MC68020 object code against C code. Wahab [124] presents an approach that verifies whether a program given in object code satisfies its specification. The object code of the program is translated into an abstract language. Then, an abstraction of the program is constructed, and it is proven that the abstraction establishes all properties required by the specification.

5.8.2 Delayed Nondeterminism

The abstraction technique delayed nondeterminism is dynamically applied at runtime. It introduces lazy states into [MC]SQUARE. Lazy states are states containing explicit and symbolic parts. Thus, a lazy state no longer represents a single state, but a set of states. As [MC]SQUARE uses an explicit model checking algorithm, the symbolic parts of the state have to be instantiated whenever they are accessed. Thus, this method combines explicit and symbolic methods. To the best of our knowledge, no comparable approach has been developed so far to control the effect of nondeterminism in modeling embedded systems.

In *X-valued simulation* [19], beside the usual Boolean values zero and one, a third value called X is used. X represents the unknown value, which is similar to our $*$ value. This value is used whenever the exact value is not known. Some Boolean functions can still be decided on X values. In other cases, a complete new simulation with more details has to be started. In [MC]SQUARE, we dynamically refine (instantiate) in cases where we need more details. Bingham and Hu [16] and Regehr and Reid [97] also use an abstraction similar to the X-valued abstraction.

Symbolic simulation [20, 21] is similar to the technique applied in [MC]SQUARE. In symbolic simulation, symbolic variables are used in place of explicit ones. These symbolic variables can have the values zero, one, and X as in X-valued simulation. Whenever more information is needed due to an X, the simulation is started again with more symbolic variables. In our approach parts of the states can be symbolic, but whenever the simulator or the model checker needs to access symbolic parts of a state, these parts are instantiated, and hence, become explicit. All parts of a state

that are not accessed remain symbolic. These refinements are done dynamically, and hence, our approach avoids the overhead of a completely new simulation. There are some approaches combining *explicit* and *symbolic executions* [50, 111], but they do explicit and symbolic execution in parallel.

Another verification method for concurrent systems, which is based on a similar idea, is called *narrowing* [83]. Here, the states and transitions of the system are symbolically represented respectively by terms and rewriting steps. Terms can contain variables that abstract from details of the system state which are currently not interesting, but which can later be expanded by substitution steps if necessary. Thus, in some sense, variables correspond to the nondeterministic values in our approach.

Another direction of work, which is worth mentioning, is the consideration of nondeterminism in connection with functional programming languages. The paper by Clark [27] studies the implementation of nondeterministic choice in this setting and refers to the problem of copying nondeterministic values, which is also the reason for over-approximation in our model.

6 Static Analysis in [mc]square

This chapter describes the static analyses that are implemented in [mc]square. [mc]square uses static analysis to support the application of abstraction techniques during model checking. Other tools such as Astree [37], CodeSurfer/x86 and WPDS++ [7], Coverity Prevent, FlexeLint, Goanna [44, 45], Klocwork K7, Metal [54], PolySpace Verifier, PREfix/PREfast [22], and RT-Tester [90, 91] use static analysis itself to analyze programs. As static analysis can be applied to very large programs in contrast to model checking, we can use static analysis to annotate the program with information that is used during state space creation to limit the size of the resulting state spaces.

Static analysis is conducted by the *Static Analyzer* package. The static analysis is not only used to support the application of abstraction techniques, but also to create the control flow graph, which is, for example, used to present the counterexample to the user.

The first section presents the challenges of applying static analysis to microcontroller assembly code. Section 6.2 gives an overview of the *Static Analyzer* package. The next section describes the different static analyses used in [mc]square. These analyses include control flow analysis, live variables analysis, reaching definitions analysis, global interrupt flag analysis, and stack analysis. These static analyses are used by the abstraction techniques utilized in [mc]square. Section 6.4 details two of these abstraction techniques, namely dead variable reduction and path reduction. In the end, we present related work regarding static analyses and the applied abstraction techniques.

We summarized parts of this chapter in a paper [107]. Löll [77] presents more details about some of the abstraction techniques.

6.1 Challenges in Static Analysis of Assembly Code

Different challenges arise when abstraction techniques that use static analysis are applied to microcontroller assembly code. The abstraction techniques presented in this chapter are already implemented in other model checkers (see Sect. 6.5) using intraprocedural static analysis. We could not transfer these techniques one-to-one to [mc]square because the assembly code found on microcontrollers contains features that cannot be handled using intraprocedural static analysis. Furthermore, the other model checkers obtain the information needed to apply the abstraction techniques

completely statically. In contrast, [MC]SQUARE obtains some information statically and others dynamically because some constructs present in microcontroller assembly code cannot be handled statically.

In the following, we describe features that make the static analysis of microcontroller assembly code more difficult. Microcontroller assembly code is hardware dependent, that is, every microcontroller has its own instruction set and its own semantics. Moreover, microcontrollers have registers such as I/O registers influencing the behavior of the microcontroller. Writing such a register can, for example, activate an interrupt, deactivate a timer, or output a value to the environment. This behavior has to be considered inside the static analysis. Another challenge is the handling of nondeterminism, which cannot be handled completely statically as it is dependent on the values of I/O registers.

In the microcontroller assembly code, functions are not explicitly declared in header files. Every location of an assembly program can be reached via a call statement. All program fragments reachable by a call statement are defined to be functions. As recursion is used in microcontroller assembly programs, functions could not be handled by inline expansion (copy body of a function to every place where it is called) or bounded call-strings [87]. Handling functions by assuming that they change all variables is also not appropriate as this over-approximation would be too coarse to obtain usable results. Additionally, all memory locations (e.g., registers, I/O registers, and variables) in the assembly code are globally accessible at every program location. Hence, interprocedural analyses have to be used to preserve soundness of the results obtained.

The microcontroller assembly code does not contain parallel processes, but it contains interrupts. Interrupts can intercept the main process at any location whenever they are enabled. The main process is only continued when control is returned from the interrupt handler. Thereby, interrupts can communicate with the main process at every program location and change all memory locations. This makes static analysis even more difficult. Interrupts are handled the same way as functions in [MC]SQUARE, but an additional analysis using abstract interpretation is applied to obtain a more accurate set of locations where interrupts are enabled.

Furthermore, the assembly code contains pointer operations such as indirect loads, stores, calls, and jumps. Whenever a pointer is used, it is assumed that the pointer can access all memory locations in case of a load or store instruction and all program locations in case of a call or jump instruction. This is done because the value of the pointer is not statically known. Handling indirect loads and stores this way is not a problem. Handling indirect calls and jumps like this is a problem because the control flow graph would be useless. To avoid this, [MC]SQUARE does not add call edges in this case, and hence, the control flow graph created is incomplete. If the control flow graph is incomplete, the results from data flow analyses are not valid over-approximations. Therefore, [MC]SQUARE does not use abstraction techniques

that rely on data flow analyses if an indirect call or an indirect jump is found in the program. Dead variable reduction is an abstraction technique that relies on data flow analyses and hence, cannot be used in this case. Path reduction does not rely on these analyses, and therefore, it can be used even if indirect control is present within the program. As indirect control is not often used in microcontroller assembly code but sometimes found in library functions, these abstraction techniques can be used for most programs. Indirect loads and stores do not have such a negative influence. They make the results less accurate as they increase the degree of over-approximation, but the results of the analyses are still valid.

6.2 Static Analyzer Overview

The *Static Analyzer* package implements static analyses and abstraction techniques that use static analyses. Figure 6.1 shows the UML class diagram of important classes of the *Static Analyzer* package. For clarity, we omit some details.

The *StaticAnalyzerController* controls the different static analyses and abstraction techniques. It implements the workflows needed to conduct the different abstraction techniques. The *CFGBuilder* creates the *StaticAnalyzerProgram*, which is the program representation used within the *Static Analyzer*, from the microcontroller program, which is implemented by the *Program* class. The other analyses and abstraction techniques all work on the *StaticAnalyzerProgram*. The live variables analysis, for example, is implemented within the *LVABuilder*. The *ATMegaLVABuilder* implements the parts that are specific to the ATmega microcontroller. All ATmega-specific classes shown in the diagram implement the *ATMegaInstructionVisitor*. That is, they implement the semantics of the instructions used within the different static analyses and abstraction techniques. The next sections describe these static analyses and abstraction techniques in detail.

The model of the microcontroller used within the *Static Analyzer* is implemented within the *ATMegaDependencyMap*. This model is used by all static analyses to observe the dependencies between different memory locations. For example, writing an I/O register can read or write another I/O register. Without these dependencies, the results of the static analysis would be wrong. It is important to model these dependencies as accurate as possible for the static analysis. Some dependencies are dynamic and hence these dependencies have to be over-approximated. The model of the microcontroller used in the *Static Analyzer* is simpler than the model used within the *Simulator*, but the structure is similar. Therefore, we do not detail the model here.

The *StaticAnalyzerController* executes the different static analyses and abstraction techniques in a specific sequence because they depend on each other. Figure 6.2 shows a UML activity diagram depicting the sequence of the static analyses and

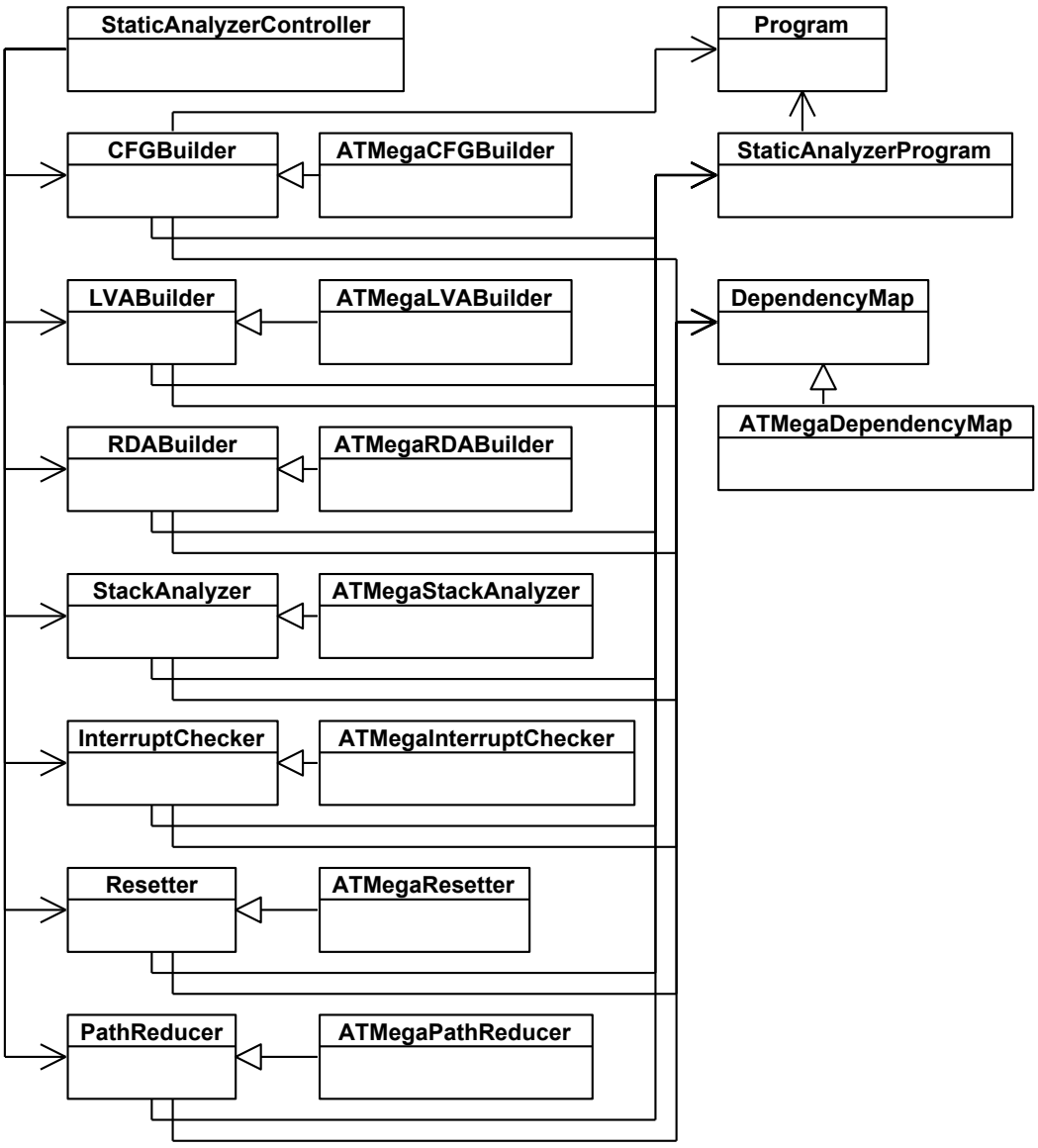

Figure 6.1: The *Static Analyzer* package.

abstraction techniques. Whenever a step of this sequence is successfully executed, the next step is conducted. If one of the analyses fails, the *StaticAnalyzerController* jumps to the last step of the sequence. In the sequence, the control flow graph, which is used by all other analyses, is built first. Afterwards, the stack analysis and the first step of the reaching definitions analysis are conducted. The results of these two analyses are used by the global interrupt flag analysis, which is conducted subsequently. Then, the live variables analysis and the remaining steps of the reaching definitions analysis are carried out. In the end, the specific analysis for the corresponding abstraction technique and the abstraction technique itself are executed.

6.3 Static Analyses

This section details the five static analyses, which are used by the abstraction techniques presented at the end of this chapter. The static analyses described include one control flow analysis and four data flow analyses, namely: live variables analysis, reaching definitions analysis, stack analysis, and global interrupt flag analysis.

6.3.1 Control Flow Analysis

The goal of the *control flow analysis* is the creation of the *control flow graph* (*CFG*). The algorithm used to create the CFG visits each location of the program once. For each location, it executes a method that adds the according vertices and edges to the CFG. The algorithm uses the visitor design pattern introduced in Sect. 5.5. The *InstructionVisitor* class defines the set of instructions. The *CFGBuilder* implements this visitor. For each of the instructions defined in the *InstructionVisitor*, it implements the visit() method. Within the visit() methods the algorithm creates the corresponding vertices and edges.

When visiting a certain location, depending on the kind of the instruction, a vertex and one or more edges are created. For arithmetic, logic, data transfer, bit, and bit-test instructions the algorithm adds a new vertex with the address of the instruction and an edge to the successor location. For branch instructions it depends on the kind of the branch instruction. For generic branches a vertex and two edges are added, that is, one edge to each of the possible successors. For jump instructions the algorithm adds one vertex and one edge. For call instructions, the algorithm adds one vertex and two edges. One of the edges is a usual edge pointing to the direct successor of the location, and the second edge is a call edge that points to the start of the function. Additionally, the target of the call edge is added to the list of function headers. Interrupts are handled similarly, but no edges are added for interrupts. If the algorithm encounters an indirect jump or an indirect call,

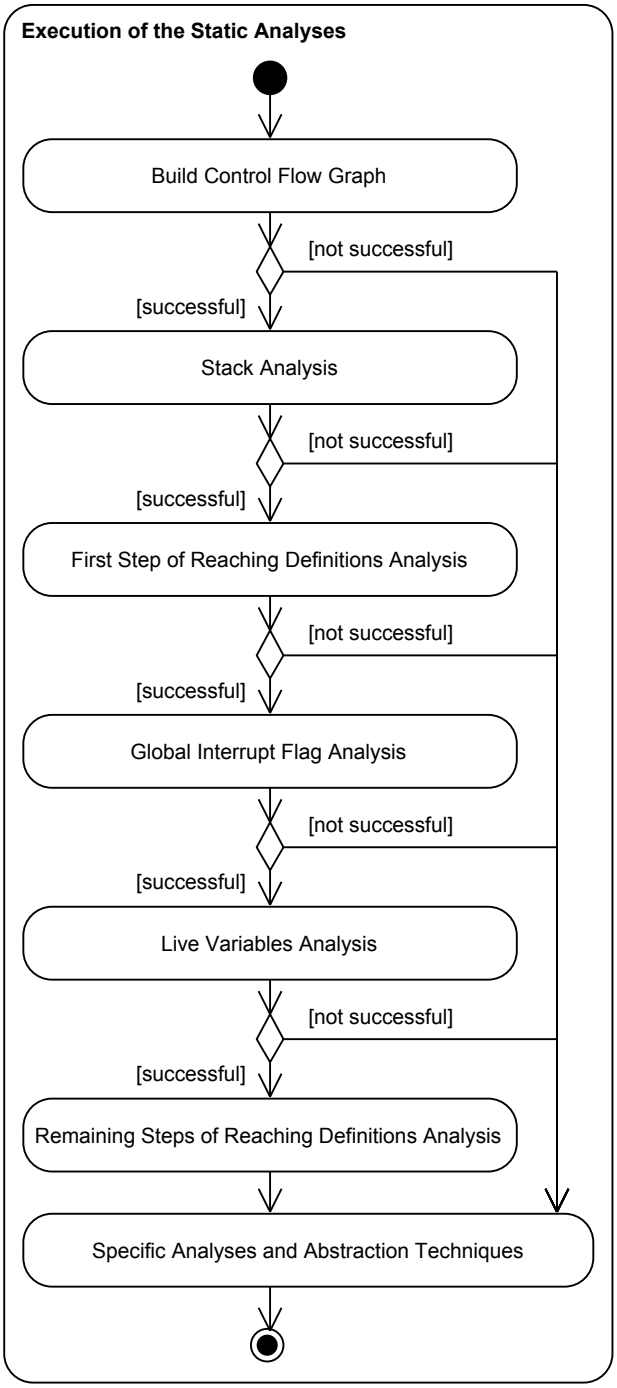

Figure 6.2: Sequence of the static analyses and abstraction techniques.

the algorithm aborts because the CFG cannot be created in this case as the target addresses are not known at compile time.

There are two kinds of CFGs: the CFG of the complete program and the CFGs of the single functions. To generate the CFG of the complete program, the algorithm starts from the first location and traverses all vertices by following all edges including call edges. To create a CFG of a single function, the algorithm starts traversing from the head of the function and follows all edges except for call edges.

6.3.2 Live Variables Analysis

Live variables analysis (*LVA*) is a data flow analysis that determines for each program location the set of variables alive at that location. As defined by Nielson et al. [87], a variable v is alive at the exit of a program location l if there exists a path π from l to a location l' where v is used, and if π does not contain a location l'' where v is redefined. That is, a variable is alive if it is potentially read before it is written again. In our setup, we do not do this for variables but for general memory locations. Variables are contained in these memory locations. The LVA is a backward analysis. That is, it processes the CFG backwards.

The data flow equations for a live variable analysis can be solved by the *worklist algorithm* [87]. Nielson et al. [87] use this worklist algorithm for an intraprocedural data flow analysis. Due to the presence of function calls and interrupts in the microcontroller assembly code, this intraprocedural algorithm has to be enhanced to deal with function calls and interrupts. As explained earlier, the standard approaches, such as expanding, call-strings, and assuming that all variables are manipulated by a function, are not appropriate in our context. Here, we describe how we transformed the intraprocedural data flow analysis into an interprocedural analysis using the worklist algorithm.

As aforementioned, functions are all program fragments which can be reached via call statements. Additionally, all interrupt handlers are also handled like functions. In the assembly language used, functions do not have formal parameter values. Communication between functions is done via global variables, registers, the stack, or memory locations indicated by pointers. The latter case is seldom used and leads to an over-approximation in our approach as indirect loads and stores possibly access all memory locations. The most common case is the use of global variables and registers.

To handle functions and interrupt handlers, [MC]SQUARE determines the static behavior of the functions (context insensitive). The *static behavior* of a function f regarding the LVA, denoted by $Behavior_{\mathbf{LV}}(f)$, is a set containing all memory locations that are read by f. In the worst case, this approach leads to an over-approximation of the behavior of a function assuming that all memory locations are read, but in most cases just a few memory locations are accessed. The LVA benefits

from the stack analysis and the global interrupt flag analysis, which are described later. Without theses analyses the results obtained during the LVA would be too inaccurate.

The LVA works using three steps. In each of the three steps the worklist algorithm is applied. The three steps are as follows:

1. Analyze each function alone to determine its static behavior. (intraprocedural)
2. Analyze each function, this time, using the static behavior of called functions at call sites. If the static behavior of a function is changed, these changes are propagated. (interprocedural)
3. Propagate information from call sites to called functions. This does not change the static behavior of functions, and hence, functions are not analyzed again.

In the following, we explain these three steps in detail.

Step 1

In the first step, every function including the main function and all interrupt handlers are analyzed alone and call instructions including occurrences of interrupts are ignored within the functions. That is, the worklist algorithm is executed intraprocedurally for each function and each interrupt handler alone.

The worklist algorithm for the LVA works in reverse order. That is, the edges are processed from their target to the source. First, all edges of the CFG of the function are added to the worklist. The entry set and the exit set of each location are empty in the beginning. The entry set is the set of information that is given by the predecessor location. The exit set is the set of information that is given to the successor location. Our notation is similar to the notation used by Nielson et al. [87]. The only difference is that Nielson et al. [87] define that a labeled location can contain more than one instruction. In the assembly code every labeled location only contains a single instruction. The two sets are defined as follows for the LVA:

$$\mathbf{LV}_{exit}(l) = \begin{cases} \emptyset & l \text{ has no successor,} \\ \bigcup\{\mathbf{LV}_{entry}(l') | l' \text{ successor of } l\} & \text{otherwise.} \end{cases}$$

$$\mathbf{LV}_{entry}(l) = (\mathbf{LV}_{exit}(l) \setminus \mathit{kill}_{\mathbf{LV}}(l)) \cup \mathit{gen}_{\mathbf{LV}}(l)$$

The functions for the entry set and the exit set are dependent on the analysis applied. The functions $\mathit{kill}_{\mathbf{LV}}(l)$ and $\mathit{gen}_{\mathbf{LV}}(l)$ are dependent on the instruction utilized in the corresponding location l. To create the entry set of a location for the LVA, the algorithm takes the first edge from the worklist and applies the respective function $\mathbf{LV}_{entry}(l)$.

The function $kill_{\mathbf{LV}}(l)$ determines for each location the set of memory locations defined at this location. The function $gen_{\mathbf{LV}}(l)$ computes for each location the set of memory locations read at this location. Here are some examples for possible $kill_{\mathbf{LV}}(l)$ and $gen_{\mathbf{LV}}(l)$ functions of instructions used on the ATMEL ATmega16:

- An arithmetic instruction such as ADC Rd Rr reads both registers and writes Rd. Therefore, it yields the following functions:

$$kill_{\mathbf{LV}}(l) = \{\mathsf{Rd}\},$$
$$gen_{\mathbf{LV}}(l) = \{\mathsf{Rd}, \mathsf{Rr}\}.$$

- An I/O instruction such as IN Rd A writes Rd and reads the I/O register addressed by A. This yields the following functions:

$$kill_{\mathbf{LV}}(l) = \{\mathsf{Rd}\},$$
$$gen_{\mathbf{LV}}(l) = \{\mathsf{A}\}.$$

- A load instruction such as LDI Rd K writes constant K into register Rd. Hence, it yields the following functions:

$$kill_{\mathbf{LV}}(l) = \{\mathsf{Rd}\},$$
$$gen_{\mathbf{LV}}(l) = \emptyset.$$

Additionally, most of these functions also influence the SREG and dependent I/O register. For the sake of brevity, we left out these definitions in this overview.

After the application of the function $\mathbf{LV}_{entry}(l)$ on the exit set of l, the algorithm checks whether the exit set of l', which is the entry set of l, has changed. If this is the case, it adds all edges pointing to l' to the worklist. That is, all edges from locations l'', which are predecessors of l', are added. This algorithms continues until the worklist is empty and no set is further changed. Now, the set of memory locations read is known for each location. The *initial static behavior* of a function is given by the behavior of the start location of the function. From this set, we can remove the memory locations that are identified as local variables by the stack analysis (see 6.3.4). The result of this step is for every function the set of memory locations read by the function or the interrupt handler.

Step 2

In the first step of the algorithm, we ignored function calls and interrupts. They are handled within the second step of our algorithm. If a function f calls a function g, the behavior of function g is added to the behavior of the call sites within function

f. This may change the behavior of function f. If the behavior of f is changed, all functions which call f are analyzed again to propagate the behavior. This is done via the worklist algorithm used in step 1 of our algorithm until a fixed point is reached. The worklist algorithm is again executed for each function and each interrupt handler, but this time function calls and occurrences of interrupts are not ignored. The following functions are used to propagate the behavior for locations l where a function g is called or an interrupt occurs:

$$kill_{\mathbf{LV}}(l) = \emptyset,$$
$$gen_{\mathbf{LV}}(l) = Behavior_{\mathbf{LV}}(p).$$

This is an over-approximation because the called function could also write and hence kill some variables. We have to do this because interrupts can occur but do not have to occur. Therefore, the algorithm cannot be sure that the variables are written. If after the termination of the worklist algorithm the behavior of a function f has changed, our algorithm reruns the worklist algorithm for all functions g that call f. This is done until the behavior of no function changes, that is, a fixed point is reached. The worst case is that all functions access all memory locations.

During the second step, results obtained from the global interrupt flag analysis (described in 6.3.5) are used to properly identify program locations where interrupts can occur. At all these locations, interrupts are handled as calls to the corresponding interrupt handler, that is, behavior is added at these locations. At locations where the global interrupt flag analysis determined that interrupts are globally deactivated, the algorithm does not add the behavior obtained from interrupt handlers.

Step 3

In the last step, our algorithm propagates the set of memory locations alive at the call site of a function g to all locations in g to set them alive for complete g. This is needed because these memory locations have to be alive after g is handled. That is, function g is not allowed to reset these memory locations that are possible read afterwards. In this step, we again use the worklist algorithm described before. This last step does not change the behavior of a function and therefore, in this step, the algorithm does not propagate changes to calling functions. The result of this analysis is for each program location a set of memory locations alive at that location.

6.3.3 Reaching Definitions Analysis

The *reaching definitions analysis* (*RDA*) is another data flow analysis. It determines for each program location and all variables in which program location each of the variables was possibly written the last time. Again, we work with general memory

locations instead of variables. RDA is a forward data flow analysis that works in the usual order on the CFG.

The data flow equations for a reaching definitions analysis can also be solved by the worklist algorithm [87]. We use the same algorithm to lift the worklist algorithm to an interprocedural algorithm as we did for the LVA (see Sect. 6.3.2). This algorithm uses the same three steps. It only differs in the definition of the behavior, the function that is applied on the entry set to get the exit set, and the way the behavior is propagated in the second step of the algorithm.

The static behavior of a function f in the context of the RDA, denoted by $Behavior_{\mathbf{RD}}(f)$, is the set of memory locations that are written by the function and the program locations where they were written. The behavior of a function is determined by the exit set of the last location of the function. In contrast, the behavior of a function regarding the LVA is determined by the first location of the function.

The entry set and the exit set of a location l in the RDA are determined by the following functions:

$$\mathbf{RD}_{entry}(l) = \begin{cases} \emptyset & l \text{ has no predecessor,} \\ \bigcup\{\mathbf{RD}_{exit}(l') |\ l' \text{ predecessor of } l\} & \text{otherwise.} \end{cases}$$

$$\mathbf{RD}_{exit}(l)) = (\mathbf{RD}_{entry}(l) \setminus kill_{\mathbf{RD}}(l)) \cup gen_{\mathbf{RD}}(l)$$

Here are some examples for $kill_{\mathbf{RD}}(l)$ and $gen_{\mathbf{RD}}(l)$ functions used in the RDA for the ATMEL ATmega16:

- An arithmetic instruction such as ADC Rd Rr reads both registers and writes Rd. Therefore, it yields the following functions:

$$kill_{\mathbf{RD}}(l) = \{\mathsf{Rd}\},$$
$$gen_{\mathbf{RD}}(l) = \{\mathsf{Rd}\}.$$

- An I/O instruction such as IN Rd A writes Rd and reads the I/O register addressed by A. This yields the following functions:

$$kill_{\mathbf{RD}}(l) = \{\mathsf{Rd}\},$$
$$gen_{\mathbf{RD}}(l) = \{\mathsf{Rd}\}.$$

- A load instruction such as LDI Rd K writes the constant into the register. Hence, it generates the following functions:

$$kill_{\mathbf{RD}}(l) = \{\mathsf{Rd}\},$$
$$gen_{\mathbf{RD}}(l) = \{\mathsf{Rd}\}.$$

Listing 6.1: Use of a working register.

```
1 PUSH R1
2  ...
3 IN R1 PINA
4 OUT PORTB R1
5  ...
6 POP R1
```

Additionally, each of these functions changes the program location associated with the definition, that is, the program location where the memory location was written last. Most of these functions also influence the SREG and dependent I/O register. For brevity reasons, we omit these definitions in this overview.

The algorithm determines the static behavior for each function and the interrupt handlers as in the LVA (step 1) using the function for the exit sets of the locations given above. In step 2 this behavior is propagated to other functions and interrupt handlers. The only difference in the propagation is that the behavior of the function is not given to the predecessor of the call as in the LVA but to the successor of a call. In the last step (step 3) the behavior from the call location is propagated into the function as it is done in the LVA. The result of the RDA is for each program location an over-approximation of the definitions reaching this location. That is, a mapping that describes for each variable where it was possibly written the last time.

During the RDA, our algorithm does not only determine the locations where the variables were written last, but it also determines the set of possible values of these variables. These values are used within other analyses such as the global interrupt flag analysis.

6.3.4 Stack Analysis

In assembly code, the stack is used to temporarily save the contents of working registers used within a function. At the beginning of a function, the contents of the working registers are put onto the stack, and at the end of the function, the contents of these registers are taken back from stack and written into the corresponding registers. Hence, for a data flow analysis it looks as if the function reads and writes the corresponding registers, although the function does not use the values of these registers. Listing 6.1 shows an example. R1 is used as a working register within the shown function. A general static dataflow analysis determines that this function reads and writes R1.

[MC]SQUARE uses the *stack analysis* algorithm to find out whether registers are actually read or written by a function. This check supports the data flow analysis,

for example, to identify working registers. It also supports the global interrupt flag analysis described in the next section.

To find out which variables are actually read and written, we have to identify the contents of the stack. Due to the dynamic nature of the stack, the size and contents of the stack at a specific program location can only be determined during runtime. To solve this problem, and hence, to get a more accurate data flow analysis, we use an abstract interpretation to determine for each program location the static stack contents.

This abstract interpretation observes all accesses to the stack by means of PUSH and POP operations, changes of the stack pointer, and write accesses to the memory area of the stack. This analysis is carried out in a depth first search (DFS) manner starting at the first location of the function. For every location where the stack is accessed the corresponding action is executed on the stack. For example, if in a location a value is pushed onto the stack, the algorithm adds an entry to the representation of the stack contents that this value was pushed in the current location. When the algorithms reaches the last location of the function, it checks whether the size of the stack is zero. If the size of the stack is zero, the stack analysis is successful. If the size of the stack in the last location is not zero, the stack analysis of this function fails because the function exchanges data with other functions via the stack.

If during the algorithm a location is visited a second time (e.g., due to a loop), the algorithms checks if the new stack contents for this location is the same as the old stack contents. If this is not the case, the stack was changed (e.g., in a loop) and hence, we no longer know the static contents of the stack. Thus, the stack analysis of this function fails too. That is, the algorithm cannot determine the working registers of this function.

If the stack analysis fails for a function, the algorithm assumes that the function uses all registers that are accessed inside the function including the working registers. If the stack analysis is successful, the algorithms checks whether the values are written into the original registers when popped. That is, that, for example, the value of register 1 is written back into register 1. If this check is successful, we know the set of working registers of this function.

The stack analysis is done for all functions including interrupt handlers. If the analysis of a function f fails, the analysis of every function g calling f also fails. This is the same for interrupts. As [MC]SQUARE does not yet know where interrupts are active or inactive during this analysis, the stack analysis of all functions fails if the analysis of an interrupt fails.

The stack analysis algorithm of [MC]SQUARE correctly recognizes that in the example shown in List. 6.1 the original value of R1 is restored at location POP R1. This result is then used within the data flow analyses to remove R1 from the reading and writing behavior of this function.

6.3.5 Global Interrupt Flag Analysis

The *global interrupt flag analysis* (*GIFA*) determines for each program location the status of the Global Interrupt Enable bit, which is located in the SREG (see also Sect. 5.3.4). The I bit defines whether interrupts are enabled or not. Without an analysis that determines the value of the I bit, [MC]SQUARE has to assume that interrupts are active at every program location. This would make the results of the other data flow analyses very inaccurate. For example, if an interrupt handler reads a certain register and the corresponding interrupt is active at any program location, this register can never be reset by the dead variable reduction. To improve the precision of the results, we implemented a dataflow analysis that tries to determine the status of the global interrupt flag for each program location.

The GIFA is a forward data flow analysis. The data flow equations for the GIFA are solved using the worklist algorithm presented in Sect. 6.3.2 and Sect. 6.3.3. It uses the same three steps and only differs in the definition of the behavior of a function and the way the entry and the exit sets are defined. The static behavior of a function f regarding the GIFA, denoted by $Behavior_{\mathbf{GIF}}(f)$, is the status of the global interrupt flag of the last location of the function. That is, the behavior determines whether the function enables, disables, or does not change the state of the interrupts.

The entry set and the exit set of a location l in the GIFA are determined by the following functions:

$$\mathbf{GIF}_{entry}(l) = \begin{cases} \{ir_enabled\} & \exists l' \text{predecessor of } l . \mathbf{GIF}_{exit}(l') \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathbf{GIF}_{exit}(l)) = (\mathbf{GIF}_{entry}(l) \setminus kill_{\mathbf{GIF}}(l)) \cup gen_{\mathbf{GIF}}(l)$$

Here are some examples for possible $kill_{\mathbf{GIF}}(l)$ and $gen_{\mathbf{GIF}}(l)$ functions for the ATmega16:

- An usual arithmetic instruction such as ADC Rd Rr does not influence the global interrupt flag. Hence, $kill_{\mathbf{GIF}}(l)$ and $gen_{\mathbf{GIF}}(l)$ do not change anything.

$$kill_{\mathbf{GIF}}(l) = \emptyset,$$
$$gen_{\mathbf{GIF}}(l) = \emptyset.$$

- The instruction SEI enables interrupts by setting the global interrupt flag. Thus, it does not kill, but it generates $ir_enabled$:

$$kill_{\mathbf{GIF}}(l) = \emptyset,$$
$$gen_{\mathbf{GIF}}(l) = \{ir_enabled\}.$$

- The instruction CLI does the opposite of SEI. It disables interrupts by clearing the global interrupt flag:

$$kill_{\mathbf{GIF}}(l) = \{ir_enabled\},$$
$$gen_{\mathbf{GIF}}(l) = \emptyset.$$

- The instruction OUT A Rr is more complex. Depending on the register addressed by A, it influences or does not influence the global interrupt flag. If A is the address of the SREG (I bit is located within the SREG), the value of Rr determines whether interrupts are enabled or disabled. The address A is a constant, but the value of Rr is not a constant. The value of Rr is determined by the RDA before the GIFA is executed. If the RDA fails, the algorithm has to assume that interrupts are enabled to ensure a safe over-approximation. This leads to the following functions:

$$kill_{\mathbf{GIF}}(l) = \begin{cases} \{ir_enabled\} & \text{A address of SREG}\wedge \\ & \text{8th bit of Rr} = 0 \\ \emptyset & \text{otherwise} \end{cases}$$

$$gen_{\mathbf{GIF}}(l) = \begin{cases} \{ir_enabled\} & \text{A address of SREG}\wedge \\ & (\text{8th bit of Rr} = 1\vee \\ & \text{value of Rr} = \text{unknown}) \\ \emptyset & \text{otherwise} \end{cases}$$

The algorithm is executed in the same way as for the LVA and the RDA. The result of the algorithm is an over-approximation of the value of the global interrupt flag for every program location. That is, interrupts are disabled if the algorithm is sure that the interrupts are disabled and otherwise, interrupts are enabled. This analysis is important for the other data flow analyses and abstraction techniques as it improves the accuracy of the analyses and hence the effectiveness of the abstraction techniques. The GIFA depends on results of other analyses.

The example shown in List. 6.2 demonstrates the way the contents of the SREG is saved in the beginning of an interrupt handler and restored in the end of the interrupt handler. Without the information from the RDA and the stack analysis, the GIFA could not determine the value that is assigned to the SREG in the last line. Hence, the algorithm would assume that interrupts are enabled in the last line. This would allow cascading interrupts in this interrupt handler and lead to unlimited recursion.

For the shown interrupt handler, the stack analysis correctly determines that, at the end, the contents of R0 is taken from the stack. The algorithm also knows

Listing 6.2: Restoring the value of the status register.

```
1 IN R0 SREG
2 PUSH R0
3 ...
4 POP R0
5 OUT SREG R0
```

that R0 holds the contents of SREG saved in the beginning of the interrupt handler. Hence, the analysis finds out that the I bit has the same value in the end as in the beginning. That is, interrupts are not enabled in the interrupt handler. Similar patterns are not only used in interrupt handlers, but also in usual functions.

6.4 Abstraction Techniques

This section describes two abstraction techniques implemented in [MC]SQUARE, which use the static analyses that were described in the previous sections. The results that can be obtained by applying these abstraction techniques are detailed in Chap. 8. In the following, we first describe dead variable reduction and then we describe path reduction.

6.4.1 Dead Variable Reduction

Dead variable reduction (*DVR*) [62, 63, 129] is an abstraction technique that copes with the state-explosion problem. It tries to reduce the number of states generated during state space construction. The idea behind dead variable reduction is that states only differing in dead variables are equivalent and hence can be merged into a single state. A dead variable is a variable that is not in the set of live variables (see Sect. 6.3.2). In the microcontroller assembly code, memory locations are the variables we are interested in.

To apply DVR, [MC]SQUARE first uses the LVA to determine for each location the set of live variables. After it determines the sets of live variables, it identifies the set D of dying variables for each program location. To do this, the algorithm successively compares the sets of live variables of two consecutive program locations l and l'. The variables that are alive at l and are no longer alive at l', die after the execution of the statement in l. After that, the variables contained within the CTL formula are removed from each set D. Then, every program location is annotated with its corresponding set D indicating the variables which can be reset by the *Simulator*. [MC]SQUARE assigns 0 to a variable when it dies. Hence, states which

differ only in the value of a dead variable are automatically merged. To preserve the validity of the model checking results, variables used within the CTL formula (specification) and the values of I/O registers are not reset. The I/O registers are not reset because they induce side-effects in the microcontroller.

DVR reduces the size of the state space created by resetting dead variables. In our case studies, we found out that this especially helps in presence of functions. In functions, local variables are used. These local variables are represented by general-purpose registers in the assembly code. In two different functions, different local variables may be represented by the same register due to compiler optimizations. Thus, these two functions usually being independent are now dependent on each other. That is, the interleaving between the functions is now accounted by the model checker. This drastically increases the size of the state space. DVR is able to find the local variables and to reset them in the end of the function. Thereby, the two functions stay independent. Chapter 8 gives details about the effect of DVR.

6.4.2 Path Reduction

Path reduction (*PR*) [129] is an abstraction technique that is used to compress single successor paths, which are computational paths consisting of states having only a single successor, into a single step. That is, only the first and the last state of these paths are stored to reduce memory consumption. The disadvantage of PR is that it only preserves CTL*-X. That is, validity of the next operator is not preserved. We give details about the logics preserved by the different abstraction techniques in Sect. 7.3.

Path reduction does not rely on other data flow analysis such as dead variable reduction. Hence, PR can be used even if LVA and RDA fail. Thus, PR can be applied to all possible assembly programs including programs that feature indirect control. PR works as follows.

The PR algorithm determines so-called *breaking points* (*BPs*) [129]. BPs are locations that have to be stored. During the creation of the state space only states generated from breaking points are stored. States generated from non-breaking points are not stored. In the original approach, all breaking points can be determined statically. In our environment, this is not possible due to the microcontroller assembly code (e.g., indirect control, indirect data accesses, and nondeterminism). [MC]SQUARE determines some of the breaking points statically, others are determined dynamically during state space creation.

Yorav and Grumberg [129] define the following locations l to be breaking for the WHILE language:

1. l is the initial or terminating program location,

2. l is associated with the program location of an assignment changing a variable used within the formula,
3. l i associated with the program location of a non-deterministic assignment,
4. l is the head of a while statement,
5. l is labeled by a procedure call, or is the statement immediately following a procedure call and
6. l is labeled by a communication statement (send or receive), or is the statement following a communication.

All these BPs are determined statically for the While language. This is different for microcontroller assembly code. Some of the conditions for BPs are different, and some of the conditions cannot be checked statically. In the following, we explain for each of the aforementioned BPs how they are handled in [mc]square.

[mc]square checks the first and the second condition statically, too. However, the third condition cannot be checked statically because in contrast to the While language, nondeterminism is not indicated by statements in the assembly code. Varying memory locations can introduce nondeterminism and are accessed via different statements. A memory location can change back and forth between nondeterministic and deterministic behavior (e.g., input port is switched to output or a timer is activated). Hence, the over-approximation of a static approach is too coarse. Therefore, [mc]square checks the third condition dynamically during state space creation. That is, every state having more than one successor is stored.

The fourth condition needs some special treatment because in the assembly language used there exists no while statements, but there exist loops. The fourth condition is needed to guarantee termination during state space building. If there is a loop only consisting of single successors, the PR algorithm cannot detect revisits without this condition, and hence, the state space creation does not terminate. To avoid this, one location of each loop has to be breaking. In [mc]square all locations are defined to be breaking that have more than one predecessor in the CFG because a location that closes a loop has more than one predecessor and all loops have at least one such location. A location has more than one successor in the CFG if it is the target of a jump or branch instruction. The algorithm finds most of these targets statically. Only the targets of indirect jumps cannot be found statically, and thus, [mc]square identifies them dynamically during state space creation. Thus, in every loop there is at least one location that is breaking. As not every one of these target locations is part of a loop, some locations are unnecessarily breaking, but the runtime overhead needed to detect only real loops would be considerably higher.

[mc]square checks the fifth condition statically as in the original approach. It marks all locations containing a call or an indirect call statement and the succeeding

locations to be breaking. The sixth condition is not directly applicable to the assembly language used because it does not contain parallel processes but interrupts, which show a similar behavior (described in Sect.5.3.4). There are no statements used for communication controlling the communication between the main process and the interrupt handlers. Whenever an interrupt is active and enabled, it can interrupt the main process and change all memory locations. Thereby, it can communicate with the main process. To represent this behavior in this analysis, each location where interrupts may be enabled has to be breaking. [MC]SQUARE checks the sixth condition not statically, but dynamically because whether an interrupt is active or not is not only determined by the global interrupt flag. Each interrupts has its own flags that determine whether it is active or not. Discovering all these bits statically leads to an over-approximation that is too coarse. Many locations would needlessly be labeled breaking.

After [MC]SQUARE has statically determined some of the breaking points, it starts creation of the state space. During the creation of the state space, the algorithm dynamically checks for the other breaking points and only stores states created from breaking points. The dynamic examination of breaking points increases the runtime, but it also increases the accuracy. We decided to check all conditions statically that did not yield too coarse over-approximations, all others conditions are checked dynamically.

Most microcontroller programs use interrupts extensively. As all states have to be stored that are created from breaking points and all program locations where interrupts are active are breaking points, PR seems to be inefficient when model checking microcontroller programs. But in these programs interrupts are not active at every program location and during the execution of interrupt handlers other interrupts are usually deactivated. Thus, interrupt handlers are generally long single successor chains and hence, can be compressed efficiently using PR. Chapter 8 details the effects of PR on the size of the state space.

6.5 Related Work

Motivated by the observation that usually memory is the limiting factor in the application of model checking, many approaches have been developed to combat the state-explosion problem (see Clarke et al. [33] for an overview). We adapted two static analysis methods, namely path reduction and dead variable reduction, which were first described by Yorav and Grumberg [129]. They describe DVR and PR for a parallel version of the WHILE language. This approach is implemented in the MURPHI model checker. The language used in MURPHI is similar to the parallel WHILE language. In the language used, every process has its own local variables. Global variables are not used. Communication is done by means of send and receive

statements. The language does not contain indirect control. Murphi handles functions by inline expansion (also called inlining) and hence, it uses intraprocedural static analysis. It determines the breaking points used for PR statically.

In his master's thesis, Quirós [96] adapts the approach described by Yorav and Grumberg [129] to a bytecode language used in the NIPS VM [125, 126], which is a virtual machine for state space generation. This bytecode language is similar to the parallel While language as it has no indirect control and the communication between processes is conducted by fixed instructions. The only important difference for static analysis is that the bytecode language exhibits local and global variables. Local and global variables are easily distinguishable in this language as different instructions are used to access global and local variables respectively. DVR is only applied to local variables because intraprocedural static analysis is conducted in this approach as it is done by Yorav and Grumberg [129]. The breaking points used for PR are determined statically, too.

Spin [63] uses both DVR and PR. It works on a language called *Promela*, which is similar to the two languages described before regarding these two reduction techniques. That means method calls are handled by inlining, communication is conducted by certain instructions, and indirect control is not present. Both analyses are done statically via an intraprocedural approach before model checking.

In contrast, [mc]square works on assembly code including indirect control, indirect data access, recursion, interrupts, and globally accessible memory. This makes intraprocedural approaches and inlining infeasible. Restricting DVR to local variables is not possible as all memory locations are globally accessible and communication can occur at every program location (due to interrupts). [mc]square cannot determine the breaking points completely statically as some constructs can only be handled dynamically during runtime (e.g., nondeterminism, indirect control, and interrupts).

Lewis and Jones [74] describe a different approach for DVR, which is used in the model checking tool Estes. PR is not used in the Estes model checker. DVR is done dynamically during state space creation to exploit runtime information. Due to the dynamic nature of the approach, the results are in certain situations more accurate, but increase the runtime. The user has to provide some information to use DVR (e.g., description of the behavior of the environment, addresses of main function, and interrupt handler start and end address). This is not needed in [mc]square as the user does not have to provide any information. [mc]square avoids some runtime overhead by conducting parts of the analysis statically.

Another abstraction technique implemented in many model checkers is *partial order reduction* [62], which is used to limit the number of different interleavings between parallel processes. This abstraction technique is not implemented in [mc]-square because there are no parallel processes in microcontroller assembly code. There exists only a pseudo-parallelism introduced by interrupts, but this parallelism

cannot be handled by static analyses.

Static slicing is implemented in some model checkers. It can be used to statically remove locations, which do not influence the formula checked, from the program under verification. Currently, this method cannot be used in [MC]SQUARE as the microcontroller assembly code includes constructs such as indirect memory access, recursion, and indirect control flow. These constructs preclude the creation of a complete CFG of the assembly program, which is needed to apply slicing.

7 Model Checking in [mc]square

The actual model checking in [MC]SQUARE is conducted by the *Model Checker*. This chapter details the *Model Checker* and the *State Space* package. For the *Model Checker* package, we describe the implemented model checking algorithms and an algorithm for the creation of counterexamples. Currently, there are three different algorithms implemented: a local CTL model checking algorithm, a global CTL model checking algorithm, and an algorithm for checking invariants.

For the *State Space* package, we describe details of the different state space implementations. The common implementation stores states in main memory whereas another one stores states on hard disk. The different abstraction techniques utilized in [MC]SQUARE influence the state space and hence the validity of model checking results. Some of them preserve the validity CTL while others only preserve the validity of sublogics of CTL, such as CTL-X or ACTL. We specify which abstraction technique preserves validity of which logic.

The first section details the *Model Checker*. The second section explains the *State Space*. After that, Sect. 7.3 lists for each implemented abstraction technique the logic that is preserved. In the end, we present related work regarding model checking.

7.1 Model Checker

The *Model Checker* package conducts the actual model checking and creates the counterexample and the witness respectively. [MC]SQUARE uses three different model checking algorithms: a local model checking algorithm, which was first described by Vergauwen and Lewi [121] and later adapted by Heljanko [57], a global model checking algorithm presented by Clarke et al. [33], and an algorithm to verify invariants. For all these three model checking algorithms, [MC]SQUARE has a specific algorithm to create corresponding counterexamples.

Figure 7.1 shows a UML class diagram of the *Model Checker* package. [MC]SQUARE conducts model checking within a thread. There is a thread for the global and another thread for the local model checking algorithm. The *GlobalModelCheckerThread* uses the *GlobalCTLModelChecker* and the corresponding counterexample generator. The *LocalModelCheckerThread* uses the *LocalCTLModelChecker* and the respective counterexample generator. Additionally, the *LocalModelCheckerThread* can use the

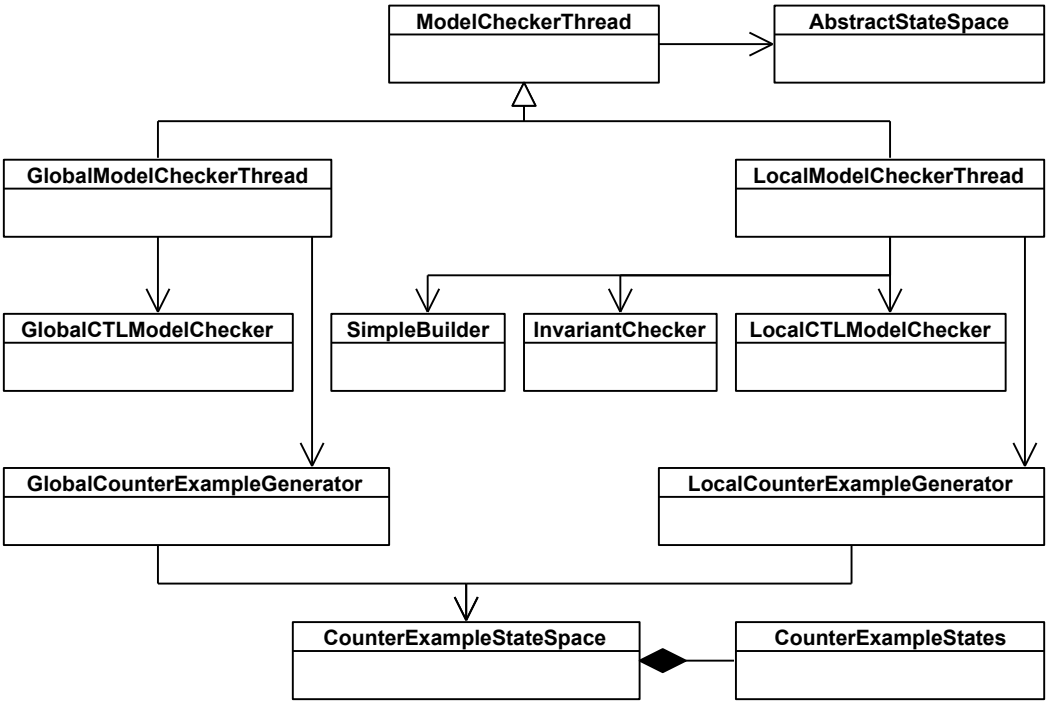

Figure 7.1: The *Model Checker* package.

SimpleBuilder, which only builds the state space, and the *InvariantChecker*, which checks for the validity of invariants.

The *Model Checker* does not create and manage states. States are managed and created by the *State Space* and the *Simulator*. The *Model Checker* just requests states from the *State Space* and changes truth values of states. The *State Space* stores the states and uses the *Simulator* to create successor states. As the atomic propositions of the formulas contain hardware-dependent statements, these propositions are already evaluated by the *Simulator*. Thus, the *Model Checker* conducts model checking hardware independently, and hence, we do not have to adapt the model checking algorithms to the peculiarities of the different microcontrollers.

This section first describes the three different model checking algorithms in detail. Then, the algorithm to create a counterexample that is used when conducting local model checking is detailed exemplarily.

7.1.1 Local Model Checking

A local model checking algorithm checks whether in a model M a given state s satisfies a given formula f ($M, s \models f$). In contrast to global model checking algorithms, a local model checking algorithm only needs to evaluate the subformulas and visit the states that are needed to evaluate the truth value of the formula in the given state s, which is in most cases the initial state of the model. Therefore, it is possible generate the state space on-the-fly during model checking when using a local model checking algorithm. A global model checking algorithm evaluates truth values of all subformulas in all states (see also Sect. 2.2.2). In the following, we first describe the process applied when using the local model checking algorithm. Then, we detail the local model checking algorithm.

Process

Figure 7.2 shows a UML activity diagram depicting the process that is used when conducting local model checking. First, the ELF file is loaded and transformed into a human-readable assembly program. Then, the provided CTL formula is parsed and transformed into a formula object. Thereafter, static analysis is conducted and the assembly program is annotated. This step is only accomplished if the user has chosen to use abstraction techniques requiring static analyses. Afterwards, model checking is started on the annotated program. States are created on-the-fly by means of the *State Space* and the *Simulator*. As aforementioned, the *Simulator* creates an over-approximation of the behavior shown by the microcontroller. Depending on the formula, [MC]SQUARE checks whether certain states satisfy specific formulas or subformulas respectively. After model checking is conducted, a counterexample is created if the user has chosen to create one. The process that is applied when using

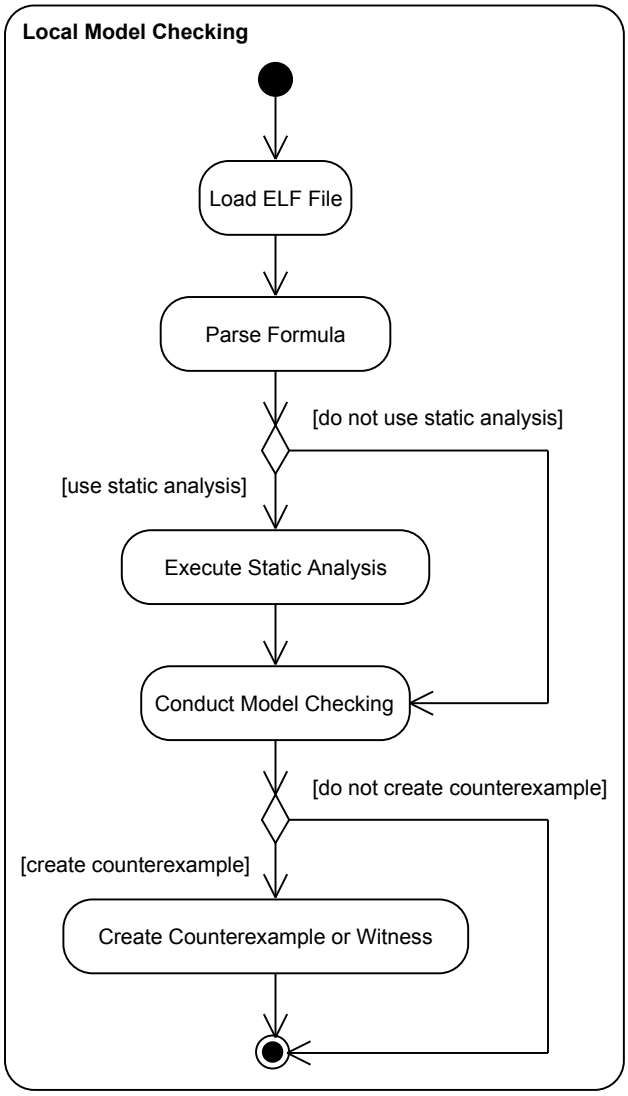

Figure 7.2: The Local model checking process.

the invariant checking algorithm (see Sect. 7.1.3) is similar to this process.

Algorithm

The local model checking algorithm implemented in [MC]SQUARE is an adaption of the ALMC algorithm, which was first presented by Vergauwen and Lewi [121] and later adapted by Heljanko [57]. The original ALMC algorithm uses very high-level data types and operations which are difficult to implement [57]. The algorithm presented by Heljanko [57] is easier to implement as it uses simple data types and operations. However, we had to change the algorithm to use stacks instead of recursive method calls because of the limited recursion depth of Java.

The ALMC algorithm handles atomic propositions p and the following operators: $\neg$, $\wedge$, **AX**, **AU**, and **EU**. All other operators (e.g., **EG**, **EF**, and **AF**) can be expressed in terms of these operators. The *Parser* converts formulas given by the user into the format needed by the local model checking algorithm. The user does not have to bother with these details.

Beside the data used within the *Simulator*, a state utilized for model checking needs to store the array of successor states and the truth values of all subformulas. The states used for model checking are represented by the *ModelCheckerState* class (see Sect. 7.2). The truth values of the subformulas are not stored as just true and false, but as unknown, marked, true, and false. Since in the ALMC algorithms no state is marked and the truth value of the corresponding subformula is known at the same time, this information can be stored in a single field, which saves memory.

The ALMC algorithm works goal-oriented in a top-down manner on the structure of the formula [121]. That is, it starts with evaluating the topmost formula in the given state. For evaluating this formula, truth values of formulas or subformulas in the current state or in successor states may be needed. Therefore, the original algorithm tests whether the required truth values are already known. If not, it recursively calls a method to evaluate the needed truth values. As we cannot use recursive method calls due to the limited recursion depth, we had to change this part of the algorithm to work with stacks.

In our version of the ALMC algorithm, we added a stack called *global stack*, which stores verification jobs. A *verification job* is a mapping between a state s and a formula f, where f has to be evaluated in s. Whenever a new truth value is needed, a new verification job with the corresponding mapping between the state and the formula is created and added to the global stack. Then, the algorithm returns form the current verification job and executes the new one. When all needed truth values are evaluated, the algorithm returns to the original verification job and evaluates it. Then, it removes it from the global stack.

In case of an **AU** or **EU** formula a verification job additionally contains the current maximum DFS number, a hash map, which maps states to DFS numbers,

and two *local stacks*. One of these local stacks is used to store names of successor states in which the current formula has to be checked before it can be evaluated in the current state. We call theses verification jobs *dependent verification jobs*. They are only used for dependent **AU** and **EU** verification jobs. They share the DFS numbers and the second local stack called *counterexample stack*, which is used to store potential counterexample and witness states respectively. However, new verification jobs for nested subformulas are put on the global stack as these jobs do not share data with the current verification job.

The main loop of the adapted algorithm works as follows. First, the algorithm initializes the global stack and pushes a new verification job containing the initial state and the topmost formula on the global stack. Then, in the main loop, it peeks at the verification job located on top of the global stack, takes the state and the formula referenced in this verification job, and calls the method used to check the operator employed within the formula. When checking an **AU** or **EU** formula, the corresponding method takes the state to be checked from the local stack, which is contained within the verification job taken from the global stack. Where required, new verification jobs are created and put on the respective stack. If a verification job is finished, it is removed from the stack. The main loop is executed until the global stack is empty, or a goal state is found.

In the following, it is sketched how the algorithm handles the different operators. Atomic propositions are already checked within the *Simulator* because they are hardware dependent and we implemented the *Model Checker* package hardware independently. Examples for atomic propositions are: $\mathsf{PORTA} = 0x55$, $\mathsf{R1} \leq 0xaf$, and $\mathsf{button_pressed} = 1$. These atomic propositions may include features, which are hardware dependent such as PORTA or R1.

Checking $\neg$, $\wedge$, and **AX** is intuitive. When checking $\neg f_1$, the algorithm tests whether f_1 is satisfied or not. If f_1 is satisfied, $\neg f_1$ is not satisfied and the other way round. To check for the validity of $f_1 \wedge f_2$, the algorithms tests whether f_1 and f_2 are satisfied. If they are both satisfied, $f_1 \wedge f_2$ is also satisfied. If one of the subformulas is not satisfied, $f_1 \wedge f_2$ is not satisfied. To check for the validity of $\mathbf{AX} f_1$, the algorithm tests whether f_1 is satisfied in all successors of the current state. If it is the case, $\mathbf{AX} f_1$ is true. In other cases, $\mathbf{AX} f_1$ is not satisfied in the current state.

To check whether the current state s satisfies the formula of the form $f = \mathbf{A}[f_1 \,\mathbf{U}\, f_2]$, the algorithm first checks whether s satisfies f_2. If it does, s satisfies f. If s does not satisfy f_2, the algorithm checks whether it satisfies f_1. If s does not saturate f_1, f is not satisfied in s either. If s saturates f_1, all successors of s have to be checked whether they satisfy f or not. If a loop is found, on which all states satisfy f_1, but no state saturates f_2, these states do not satisfy f.

The structure of the algorithm for checking a formula of the form $f = \mathbf{E}[f_1 \,\mathbf{U}\, f_2]$ is similar, but the algorithm is more involved. To check whether the current state s

saturates f, the algorithm first checks whether s satisfies f_2. If this is the case, s saturates f and hence it is a so-called *goal state*. When a goal state is found, the algorithm has to label all states that can reach this goal state via states satisfying f_1. This is done using a DFS, which is described later. If s does not saturate f_2, the algorithm checks whether it satisfies f_1. If f_1 is not satisfied, s can not saturate f. In case f_1 is satisfied, the algorithm checks whether one of the successors satisfies f. If a loop is found consisting of states all satisfying f_1, but none satisfying f_2, the algorithm cannot conclude that all these states do not saturate f (when checking $\mathbf{A}[f_1 \,\mathbf{U}\, f_2]$ this can be done). All these state remain on the counterexample stack.

When a goal state is found, a search for maximal strongly connected components (SCCs) using a modified version of the algorithm by Tarjan [115] is started. This algorithm starts in the initial state and conducts a DFS using only states that were visited before and satisfy f_1. In case a maximal SCC is found before the goal state is reached, this maximal SCC is removed from the witness stack because it can not reach the goal state. Hence, all states of this maximal SCC do not saturate f. Then, the search in continued until the goal state is reached. When the goal state is reached, all states that are still on the counterexample stack satisfy f and are labeled accordingly.

The algorithm for checking **EU** described by Heljanko [57] contains an error. In case a goal state is found, the algorithm forgets to mark this goal state. This leads to omission of labeling of states saturating the **EU** formula. To correct this error, we added instructions marking the found goal states. More details of this algorithm such as algorithms in pseudo code are presented in the corresponding papers [57, 121].

7.1.2 Global Model Checking

A global model checking algorithm determines for a model M and the set of all states S the set of states that satisfy the formula f ($\{s \in S \mid M, s \models f\}$). In contrast to a local model checking algorithm, a global model checking algorithms determines the truth values of the formula in all states while the local model checking algorithm only checks whether a given state satisfies the formula. Therefore, a local model checking algorithm is able to build the state space on-the-fly during model checking while a global model checking algorithm has to build the state space before model checking. In case the state space is too big to fit into memory, the global algorithm cannot determine anything whereas the local algorithm can still find counterexamples or witnesses in the part of the state space that fits into memory.

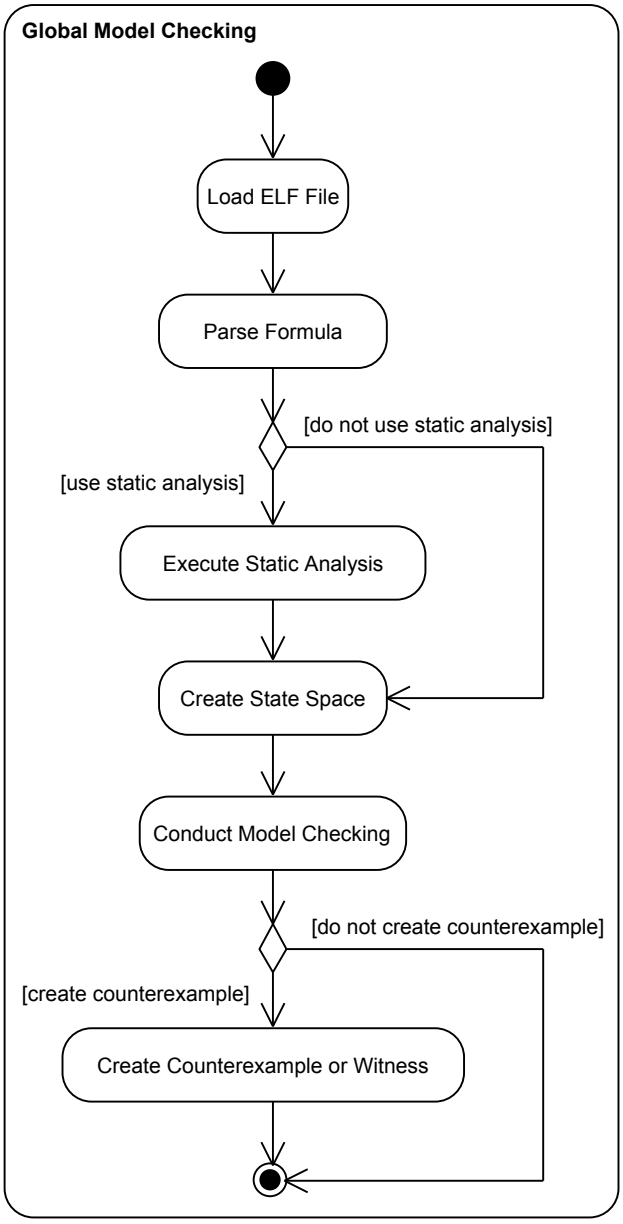

Figure 7.3: The global model checking process.

Process

Figure 7.3 shows an UML activity diagram of the process that is used when conducting global model checking. The first steps, that is, load ELF file, parse CTL formula, and conduct static analyses are the same as when doing local model checking (see also Sect. 7.1.1). After these steps, the complete state space is built using the *State Space* and the *Simulator*. After the state space is built, model checking is conducted by the *Model Checker*. In the end, a counterexample or a witness is created.

Algorithm

The global model checking algorithm implemented in [MC]SQUARE is taken from Clarke et al. [33]. Beside atomic propositions, this algorithms handles the operators $\neg$, $\vee$, **EX**, **EU**, and **EG**. All other operators can be expressed in terms of these five operators [33]. The *Parser* converts formulas using other operators into equivalent formulas using only these five operators.

Beside the fields used within the *Simulator*, a state contains the following fields: array of successors, array of predecessors, array of formula truth values (true and false), and a visited flag. Temporarily, the DFS number and the number of the SCC that contains this state are stored within the state.

This algorithm works bottom-up on the structure of the formula in contrast to the ALMC algorithms, which works top-down on the structure of the formula. Bottom-up means that it starts with evaluating the leafs of the subformulas' tree and works its way up to the topmost subformula, which is the complete formula. Since in [MC]SQUARE the atomic propositions, which are the leafs of the subformulas tree, are already evaluated within the *Simulator*, the algorithms starts evaluating the subformulas just above the atomic propositions. The order in which subformulas are evaluated is described by Def. 4 on page 9, which describes a partial order over the subformulas. The global algorithm starts with the bottom element and works its way up. That is, during the ith stage, subformulas with $i-1$ nested subformulas are processed.

In the main loop this algorithm picks a subformula to be checked and then starts checking it for all states $s \in S$. This algorithm does not have to bother with unevaluated subformulas of the current formula because it is known that these subformulas were evaluated before. When this subformula is evaluated, the next subformula is chosen. The algorithm terminates when the topmost formula is evaluated in all states. In the following it is described how the different operators are handled.

Checking $\neg$, $\vee$, and **EX** is trivial. When checking $\neg f_1$, all states that do not satisfy f_1 are labeled satisfying $\neg f_1$. Checking $f_1 \vee f_2$ is similar. All states that satisfy f_1, f_2, or both satisfy $f_1 \vee f_2$ and are labeled accordingly. The algorithm

checks $\mathbf{EX}f_1$ by checking whether one of the successor states of the current state satisfies f_1. If it is the case, the current state satisfies $\mathbf{EX}f_1$.

Checking $\mathbf{E}[f_1\,\mathbf{U}\,f_2]$ is more involved. To handle a formula of the form $f = \mathbf{E}[f_1\,\mathbf{U}\,f_2]$, the algorithm first finds all states that satisfy f_2. Then starting from these states, it searches backwards for all states that can reach these states using only states that satisfy f_1. All these states satisfy f and are labeled correspondingly.

Checking a formula of the form $f = \mathbf{EG}f_1$ is slightly more complicated. From the set of states S, the algorithm removes all states that do not satisfy f_1. The resulting set is called S'. The successors and predecessors of each state are adapted accordingly. Then S' is partitioned into SCCs using the algorithm by Tarjan [115]. From each state that is part of a SCC, the algorithm searches backwards using only states satisfying f_1. All these states satisfy f. More details about the global model checking algorithm such as algorithms in pseudo code can be found in the book by Clarke et al. [33].

This model checking algorithm was the first one implemented in [MC]SQUARE. It is in most cases slower than the local model checking algorithm because the complete state space has to be built before model checking is started. The local model checking algorithms builds in many cases only a part of the state space. Even in the worst case for the local model checking algorithm, in which it has to build the complete state space, the global model checking algorithm is not faster because, when model checking programs with [MC]SQUARE, most of the time is spent building the state space, and only a small amount of the time is spent checking the states. The memory requirements of the global model checking algorithm are significantly higher because it has to store an array of predecessor states, which is not needed in the local model checking algorithm. Hence, the user should use the local model checking algorithm or the algorithm used for checking invariants.

7.1.3 Invariant Checking

Many interesting properties can be expressed as invariants. Checking an invariant is easier than checking general CTL formulas. An invariant is given by conditions on a state, which have to hold in all reachable states. That is, given a propositional logic formula f, an invariant can be written as $\mathbf{AG}f$.

The algorithm used to verify invariants is similar to the local model checking algorithm described in Sect. 7.1.1. It uses the same process and the same type of states. The atomic propositions used within the propositional logic formula are already evaluated in the *Simulator* package. Therefore, this algorithm only has to handle two operators: $\neg$ and $\wedge$. All other propositional logic operators can be expressed using these two operators. The *Parser* package converts them.

The algorithms works as follows. It starts checking the invariant in the first state. Then, using a DFS algorithm the complete state space is built. During the creation

of the state space, the invariant is checked for each state. If a state is found that does not satisfy the invariant, this algorithm terminates and returns the error trace that leads to the state not satisfying the invariant.

For checking the invariant, the algorithms needs to check the two supported operators. To check for the validity of a formula $f = \neg f_1$, it is checked whether the state s satisfies f_1. If it satisfies f_1, s does not satisfy f and the other way round. The algorithm checks a formula of the form $f = f_1 \wedge f_2$ by first checking f_1 and f_2. If one of these two subformulas is not satisfied, the formula f is not satisfied. If both subformulas are saturated, the formula f is also saturated.

In this algorithms, [MC]SQUARE uses recursive method calls to check for the validity of subformulas. It does not use a stack of verification jobs as used in the two model checking algorithms described before. In this algorithm, recursion can be used because the maximal occurring recursion depth when checking an invariant is the number of subformulas. When checking an invariant, no dependent verification jobs for successor states need to be created.

This algorithm uses less space and is faster than the local and the global model checking algorithms described before. The counterexample, which is created in case the invariant is violated, is easier to understand than counterexamples that are created when checking generic CTL formulas. Therefore, the user should use this algorithm whenever it is possible to express the desired property as an invariant.

7.1.4 Counterexample Generation

This section describes the way [MC]SQUARE creates a counterexample. As introduced in Sect. 2.2.3, we only use the term counterexample and it refers to both counterexamples and witnesses. Whenever a counterexample has to be generated, for instance, if a formula of the form $\mathbf{AG}f$ is not satisfied, [MC]SQUARE starts the generation of the counterexample. If a formula containing existentially and universally quantified subformulas is checked, counterexamples are created for each subformula and combined into one counterexample. This combined counterexample shows why the complete formula including its subformulas is satisfied or violated. The counterexample is presented as a graph, in the assembly code, in the CFG of the assembly code, and in the C code if it is present.

This section describes the generation of counterexamples when using the local CTL model checking algorithm, which is described in Sect. 7.1.1. The generation of counterexamples depends on the model checking algorithm used because it has to handle the same set of operators. The generation of counterexamples is similar in the invariant checking algorithm and in the local model checking algorithm. The only difference is that the generation of counterexamples for the invariant checker does not have to handle the constructs that are only used in the local model checking algorithm such as **EG**, **AU**, and **AX**. The generation of counterexamples for the

global model checking algorithms is different. It is based on the same idea, but it has to handle different operators.

The algorithm that creates the counterexample works in a top-down manner on the structure of the formula. That is, the algorithm starts with the topmost formula and works its way down to the leafs of the subformula tree. The leafs are the atomic propositions of the formula. First, the truth value of the topmost formula in the initial state is considered, and the initial state is added to the counterexample, which is represented by the *CounterExampleStateSpace* class. Depending on this truth value and the formula, the algorithm has to add other states to the counterexample. To add other states to the counterexample, the algorithm uses a stack of *counterexample jobs*. A counterexample job stores the name of the current state, the current subformula to be refuted or witnessed, and the name of the predecessor state. The creation of the counterexample terminates when the stack of jobs is empty. The result is a counterexample containing all states needed to indicate the truth values of the formula and its subformulas.

While the local model checking algorithm processes the successor states from left to right, the algorithm to produce the counterexample handles the successor states from right to left. In case a state was found indicating that the formula is not satisfied, the state space was not built completely (the same applies for a witness). As the model checker builds the state space from left to right, this state is reached earlier if the algorithm searches from right to left. Using this approach, the counterexamples tend to be smaller.

In the following, it is described how the algorithms handles the different operators. If in a state s the truth values of an atomic proposition p affects the counterexample, s is added to the counterexample. That is, s either proves a formula (witness) or disproves a formula (counterexample). Whether it proves or disproves a formula does not depend on this formula but on the enclosing path formula. For instance, in the formula $f = \mathbf{AG}\, p$, f determines whether p affects the counterexample or not. To handle a formula of the form $\neg f_1$, a new counterexample job with the current state s, the formula f_1, and the predecessor of s is created and put on the stack. When handling a formula of the form $f_1 \wedge f_2$, two new counterexample jobs are created: one for the formula f_1 and another one for f_2.

Handling a formula of the form $f = \mathbf{AX} f_1$ differs from handling the before described formulas. In case f is true, this counterexample job is ignored because the current state is not part of a counterexample for f. If f is not satisfied in the current state, it is added to the counterexample, and a new counterexample job for the successor state in which f_1 does not hold is created and added to the stack. To find the successor state that does not satisfy f, the algorithm searches the list of successors from right to left and tests whether the corresponding state saturates f.

A counterexample for a formula $f = \mathbf{A} f_1 \,\mathbf{U}\, f_2$ is a path that has a loop consisting of states all satisfying f_1 and none satisfying f_2 or a path on which a state exists

that does not satisfy f_1 and f_2. To find such a counterexample, the algorithm first checks whether the current state s satisfies f. If this is the case, the job is ignored because s does not disprove f and hence, does not belong to the counterexample for f. If s does not saturate f, it is added to the counterexample and it is tested if a loop is closed.

In case a loop with the afore described properties is closed, the counterexample is completed and the current counterexample job is finished. If no loop is closed, the algorithms checks whether s saturates f_1. If f_1 is not satisfied, s is the so-called goal state and hence, a new counterexample job for the subformula f_1 is added to the stack and the current counterexample job is finished. In case f_1 is satisfied in s, the algorithm performs two things. First, it adds a new counterexample job with formula f_2 and s to the stack to show why f_2 is not saturated. Second, the current counterexample job is changed to point to a successor state s' that does not satisfy f. To find s', the list of successor states of s is searched from right to left. Then, s' is handled. This process continues until a state that does not saturate f_1 or a loop is found.

Generating a witness for a formula of the form $f = \mathbf{E} f_1 \, \mathbf{U} \, f_2$ works as follows. A witness for f is a path that only contains states saturating f_1 and is ended by a state satisfying f_2. To find such a witness, the algorithm first checks whether the current state s saturates f. If it does not saturate f, this counterexample job is ignored because s does not witness f. If s saturates f, s is added to the witness, and it is checked whether s saturates f_2. In case it satisfies f_2, a new counterexample job with f_2 and s is added to the stack, and the current counterexample job is finished. In case s does not saturate f_2, a new counterexample job with f_1 and s is put onto the stack to show that f_1 is satisfied in s, and the current counterexample job is changed to point to a successor state s' that saturates f. Then, this s' is handled. If the algorithm finds a loop consisting of states satisfying f_1 but not f_2, it backtracks and uses the next successor state s'. This process continues until a state satisfying f_2 is found, which is the goal state.

7.2 State Space

The *State Space* package is used to store and manage the states. The different state space implementations use different methods to store the states. Some use the main memory while others use the hard disk. The *State Space* package also contains different types of states, which are used for different purposes such as state space building or model checking.

Figure 7.4 shows a UML class diagram of the *State Space* package. The *Model Checker* uses the *AbstractStateSpace* interface to access the state space during model checking. The interface, for example, provides methods to access states, the data

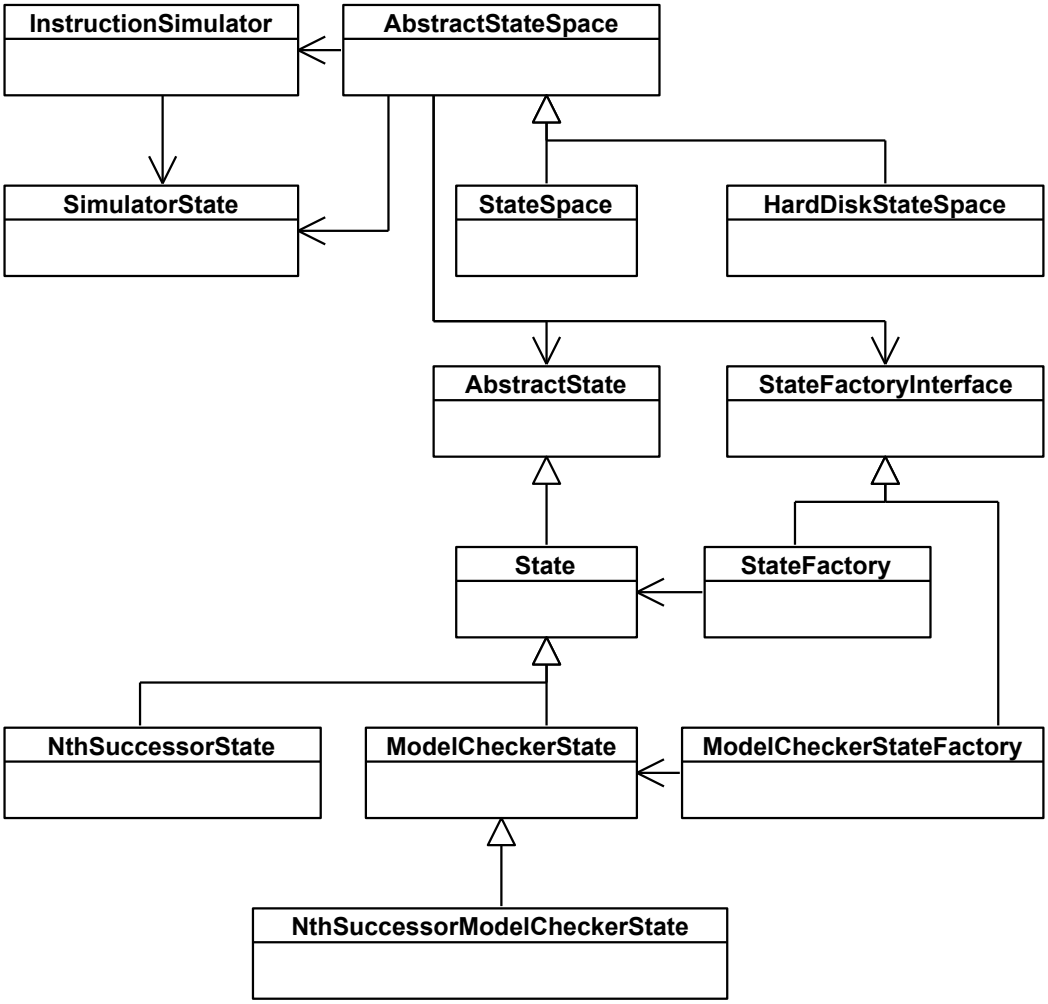

Figure 7.4: The *State Space* package.

of states, and the truth values of subformulas. The *Model Checker* does not know which state space implementation it actually uses. That is, it does not know how the states are stored. The state space implementations use the *InstructionSimulator*, which is not part of the *State Space* package, to create successor states on-the-fly. To exchange states with the *InstructionSimulator*, the *SimulatorState* class is used.

There are two implementations of the *AbstractStateSpace*: the *StateSpace*, which stores the states in main memory, and the *HardDiskStateSpace*, which stores the states on hard disk. Both implementations of the *AbstractStateSpace* use a hash map. The difference between the two implementations is the contents of the hash map. The *StateSpace* stores the complete states in the hash map. As keys, it uses the names of the states. This can be done because the *Simulator* ensures that every name is only assigned once. The *HardDiskStateSpace* stores only an index to the file where the corresponding states are stored in the hash map. As keys, it uses the names of the states as the *StateSpace* does. We decided to store the hash map containing the indices in main memory. Storing this hash map on hard disk would reduce the performance drastically as this hash map is accessed every time a state is accessed. The disadvantage is that main memory is still the limiting factor for the the state space size even though the hard disk is used.

The *HardDiskStateSpace* uses more than one file to store the states on the hard disk. To determine the file where a state is stored, a modulo function is applied to the names of the states, which are integer numbers. The process to get a state from the hard disk works as follows. First, the *HardDiskStateSpace* looks in the hash map whether the state exists or not. If the state exists, the *HardDiskStateSpace* gets the index of this state from the hash map. Then, the *HardDiskStateSpace* applies the modulo function to the name of the state to determine the file that stores this state. Following, the *HardDiskStateSpace* reads a byte array from the file at the given index position and creates the actual state from the contents of the byte array.

The *HardDiskStateSpace* does not read a state from disk every time a state is requested by the *Model Checker*, and it does not write a state to disk every time a new state is created or the contents of a state has changed. It uses a read and a write cache to keep the number of accesses to the hard disk as small as possible to improve the overall performance. Additionally, as [MC]SQUARE is written in Java, it takes advantage of the hard disk cache operated by the Java Virtual Machine.

Both implementations of the *AbstractStateSpace* create the proper state from the *SimulatorState*, which is the state that they get from the *Simulator*. To make it easier to implement new types of states in [MC]SQUARE, we decided to hide the type of the states from the state space implementations. The different types of states are needed for the different algorithms applied. As the state space implementations have to create states, we used the *factory design pattern* [47] to decouple the creation of the states to a so-called *factory*. The state space

implementations use an implementation of the *StateFactoryInterface* to create the proper states. There are two implementations of the *StateFactoryInterface*: the *StateFactory*, which creates states of type *State* and *NthSuccessorState*, and the *ModelCheckerStateFactory*, which creates states of type *ModelCheckerState* and *NthSuccessorModelCheckerState*.

Every state space implementation accesses states via the *AbstractState* class. The *AbstractState* defines the fields and methods a state has to implement. There are four different implementations of the *AbstractState* class. The *State* is a basic implementation that only stores the name of the state, its data, and the successors of the state.

The *NthSuccessorState* is a special kind of a *State*. It is used to incrementally store states. An *NthSuccessorState* does not store the byte array representing the microcontroller state, but it stores from which state the data of this state can be created and how many steps have to be executed to create the data of this state. That is, every *nth* step a state is stored completely with its data, and from this state the data of successor states is created by loading the source state and executing a certain number of steps. This reduces the space needed to store the states. The performance is not decreased significantly if n is not chosen too large. Whenever there is a state with more than one successor, the successor states have to be stored completely including their data.

States of type *ModelCheckerState* are used when model checking. Beside, the data that is stored by the *State* class, the *ModelCheckerState* class additionally stores an array of type *FormulaStatus*. This array stores the status of the different subformulas in the current state, that is, it stores whether these subformulas are satisfied or not. An entry in this array can have the value true, false, unknown, and marked (see also Sect. 7.1). The *NthSuccessorModelCheckerState* stores the same information as the *ModelCheckerState*, but it stores it incrementally like the *NthSuccessorState*.

7.3 Influence on Validity of Formulas

We implemented two CTL model checking algorithms in [MC]SQUARE. They check whether a model, in this case the state space of an assembly program, satisfies a CTL formula. Furthermore, we implemented several abstraction techniques within [MC]SQUARE. These abstraction techniques influence the validity of the formulas. The abstraction techniques are not implemented within the *Model Checker* package, but within the *Simulator* and *Static Analyzer* package. The users can enable and disable most of them, but they have to be aware of the influence on the validity of the formulas.

In the following, we list for each abstraction technique currently implemented

in [MC]SQUARE whether it preserves CTL, ACTL (universal fragment of CTL), or another sublogic of CTL (see also Sect. 2.1):

Dead Variable Reduction Dead variable reduction does not influence the validity of formulas because it is not applied for variables that are used within the formula (see also Yorav and Grumberg [129]). Hence, it preserves validity of CTL formulas.

Delayed Nondeterminism Delayed nondeterminism preserves the validity of ACTL formulas. Section 5.7.4 sketches the proof that shows that DND preserves a simulation relation between the concrete and the abstract LTS.

Lazy Interrupt Evaluation Lazy interrupt evaluation introduces additional behavior and hence, yields an over-approximation. Thus, it preserves validity of ACTL formulas.

Lazy Stack Evaluation Lazy stack evaluation implies an over-approximation as lazy interrupt evaluation does and thus, it preserves the validity of ACTL formulas.

Path Reduction Path reduction preserves CTL*-X as shown by Yorav and Grumberg [129]. That is, it preserves CTL-X formulas in our case. In our opinion, the loss of the next operator is not critical as we recommend not to use it. We recommend not to use the X operator because [MC]SQUARE conducts model checking on the assembly program, but the developer usually works on the C code. Thus, the semantics of the next operator may not be obvious to the developer.

If users apply all abstraction techniques implemented in [MC]SQUARE, [MC]SQUARE preserves the validity of ACTL-X formulas. If they want to use the next operator, they have to disable path reduction. Preserving ACTL means that if [MC]SQUARE determines that an ACTL formula is true, this is also true for the concrete system. The same holds if [MC]SQUARE finds out that an ECTL formula is false. If [MC]-SQUARE finds out that an ACTL formula is false or than an ECTL formula is true, the users have to check if this also holds for the concrete system. That is, they have to check whether the provided counterexample is possible in the concrete system.

7.4 Related Work

[MC]SQUARE uses explicit model checking algorithms as, for example, COSPAN [56], EMC, MURPHI [39], PROD [120], and SPIN [62] do. We took the global model checking algorithm used in [MC]SQUARE from [33] and the local algorithm from [57, 121]. Beside removing some errors and eliminating recursion from the local algorithm, we did not change the algorithms as we are mainly interested in the

domain-dependent generation of state spaces. In contrast to the aforementioned explicit model checkers, [MC]SQUARE does not use a proprietary input language, but works on assembly code for certain microcontrollers.

Although [MC]SQUARE uses states that are partly symbolic, the model checking algorithms utilized do not have to deal with symbolic states as they are handled by the *Simulator*. Symbolic model checking is conducted by, for example, BLAST [59], BOOP [127], MAGIC [23, 24], NuSMV [26], SLAM [9, 10], and SMV [79].

As aforementioned, we abstract from time as the state explosion observed was too big when modeling time. Estes [82] preserves discrete time, but it faces the state-explosion problem when checking bigger programs. There are model checkers that do real-time model checking such as HyTech [58], KRONOS [130], and Uppaal [14, 71, 72].

[MC]SQUARE is able to use the hard disk for storing states created during model checking. This is also called hard disk model checking [11, 55, 112]. The algorithms by Bao and Jones [11], Hammer and Weber [55], and Stern and Dill [112] try to influence the search order such that hard disk accesses are avoided. These approaches use the hard disk for state space building and for checking invariants only. As we use an on-the-fly CTL model checking algorithm, we cannot delay accesses to the hard disk. Whenever a state is created, the algorithm has to know whether it is new or not. It cannot delay this decision because it has to check the current subformula in exactly this state. In contrast to the other approaches, our approach conducts model checking independent of the method used to store the states. That is, the model checker is not aware that the states are stored on a hard disk.

We use a hash map similar to the hash map presented by Hammer and Weber [55] to quickly check whether a state is new or not. If the hash map does not contain the hash value of the state, the state is new. If the hash map contains the hash value, a hard disk lookup can no longer be avoided. Hammer and Weber [55] use an additional data structure that can be used in this case. Only if this data structure does not give a conclusive answer either, a hard disk lookup is needed. Unlike them, we implemented an additional write and read cache, and we use the operating system and the JVM hard disk cache facilities. As in our approach states may change during model checking due to new successors or new subformula values, we determine the maximum size of a state beforehand and reserve enough space for the state on the hard disk. Thus, states do not have to be moved on the hard disk and hence, files do not get fragmented.

8 Case Studies

This chapter describes two case studies conducted with [MC]SQUARE. The first case study shows the effect of abstraction techniques on the sizes of state spaces of different programs. In this case study, we were only interested in the differences of the sizes of state spaces that were created using different abstraction techniques. We were not interested in the results of the model checking.

The second case study details the use of [MC]SQUARE checking bigger programs. In this case study, we were interested in actual formulas, the use of [MC]SQUARE, and the model checking results. We explain how the user can find and identify errors using [MC]SQUARE. We also used the programs utilized in the first case study in other case studies [60, 88, 101–104, 107]. We published the second case study in a paper [106].

8.1 Effects of Abstraction Techniques on Different Programs

In this section, we present a case study that demonstrates the effects of the abstraction techniques implemented in [MC]SQUARE using five different programs. The five programs chosen for this case study were all written by students in lab courses, during diploma theses, or in exercises. None of these programs was intentionally written to be model checked afterwards. All these programs were used on the ATMEL ATmega16 microcontroller.

In the following, we first detail the five programs and explain the effects of the different abstraction techniques. After that, we evaluate the results of this case study.

8.1.1 Execution

The case study was conducted on a laptop equipped with a Intel Core Duo CPU at 2.33 GHz, 4 GB main memory, and a hard disk with a capacity of 100 GB. We used revision 2233 of [MC]SQUARE for this case study.

Table 8.1 shows the results of the case study. The first column gives the name of the program. The second column indicates the options that were used while checking the program. The third column presents the number of states that were

stored, and the fourth column presents the number of states that were actually created including revisits. The fifth column gives the size of the state space in main memory. The sixth column shows the time in seconds needed for model checking including creation of the state space and all preparatory steps such as preprocessing, parsing, and static analysis. And the last column shows the reduction of the number of states stored in percent.

The second column indicates the options that were used for model checking the program. In this column, there are six lines for each program. Every line represents another combination of options used for model checking the program. First, every program was model checked using the standard configuration. That is, no additional options were used in this configuration. It is important to notice that the standard configuration already includes some improvements and options that cannot be disabled. The improvements that are already used in the standard configuration include, for example, the improved modeling of the microcontroller and delayed nondeterminism for interrupts. For some programs, we additionally provide the size of the state space that was created with an old version of [MC]SQUARE that did not include these optimizations.

Second, we checked each program using delayed nondeterminism for values. The third run for each program was conducted using dead variable reduction. The fourth line represents the model checking run using path reduction. The fifth run was conducted using path reduction and dead variable reduction, and for the last run all three abstraction techniques were used together.

As we were not interested in the result of the model checking, but in the size of the state space, we used the formula **AG** true for model checking all programs. This formula does not influence the effect of the abstraction techniques and therefore, enables a fair comparison. We present two case studies interested in the outcome of the model checking result using actual formulas elsewhere [102, 103]. We used a power window lift program in one of the case studies [103] and a program of a fictive plant and a light switch in the other case study [102].

Light Switch

The first program is called light switch. It is used to demonstrate basic microcontroller functions (e.g., input/output from/to the environment). The program consists of 162 lines of assembly code (72 lines of C code), uses two timers, and does not employ interrupts. The program reads input from an I/O port (8 bit) and then checks whether a certain bit within the input is set or not. All other bits of the input value are not exerted for evaluation.

When using DND, [MC]SQUARE only splits up bits that are used by the program. Whenever input from the port is read in this program, [MC]SQUARE creates only two successors instead of 256 successors. These two successors are not created

Table 8.1: Effect of the abstraction techniques on different programs.

	Options used	**States stored**	**States created**	**Size [MB]**	**Time [s]**	**Reduction %**
Light Switch	standard	6,282	8,979	1.42	0.13	-
	DND	352	380	0.08	< 0.01	94.4%
	DVR	4,762	6,388	0.76	0.07	24.2%
	PR	1,670	18,926	0.43	0.34	73.42%
	DVR & PR	960	11,749	0.24	0.27	84.72%
	all	45	262	0.07	< 0.01	99.28%
Plant	standard	175,944	183,325	44	2.81	-
	DND	175,944	183,325	44	2.81	0%
	DVR	175,944	183,325	44	2.83	0%
	PR	9,844	218,022	2.47	2.82	94.41%
	DVR & PR	9,844	218,022	2.47	2.82	94.41%
	all	9,844	218,022	2.47	2.82	94.41%
Reentrance	standard	107,649	110,961	25	1.68	-
	DND	107,649	110,961	25	1.68	0%
	DVR	107,649	110,961	25	1.68	0%
	PR	6,631	122,999	1.53	1.54	93.84%
	DVR & PR	6,631	122,999	1.53	1.54	93.84%
	all	6,631	122,999	1.53	1.54	93.84%
Traffic Light	standard	9,998	10,514	2.32	0.20	-
	DND	9,998	10,514	2.32	0.20	0%
	DVR	9,998	10,514	2.32	0.20	0%
	PR	522	12,812	0.12	0.20	94.78%
	DVR & PR	522	12,812	0.12	0.20	94.78%
	all	522	12,812	0.12	0.20	94.78%
Window Lift	standard	2,342,564	2,589,665	591	62	-
	DND	316,334	442,055	80	6.04	86.5%
	DVR	303,148	331,456	75	5.23	87.06%
	PR	123,989	3,884,075	32	48	94.71%
	DVR & PR	15,687	506,237	3.93	6.23	99.33%
	all	5,232	217,577	1.30	2.62	99.78%

when the port is read, but they are created when the corresponding bit is evaluated (some instructions later). Hence, delayed nondeterminism influences the size of the state space of this program via two distinct effects. First, it lowers the number of successors that are created (two instead of 256), and second, it delays the split up until it is needed (bit is read for evaluation). It is important to mention that this is an over-approximation because all bits that are not used still have a nondeterministic value and could be split up whenever they are read.

The effect delayed nondeterminism had on the state space of this program was significant. The number of states stored dropped from 6,282 states to 352 states, which is a reduction by 94.4%. In this program, DVR lowered the size of the state space by 24.2%. DVR did not have a big influence because this program does not use many functions. DVR usually helps to reduce the dependencies between functions. PR lowered the number of states stored by 73.42%, but it significantly increased the number of states created. The number of states created increased because of revisits. As only the last state of single successor chain is stored, all states on this chain are revisited until [MC]SQUARE recognizes the revisit. In this small example, this had no effect on runtime. Using both analyses together, reduced the state space by 84.72%. The savings of both analyses did not add up completely, but the combination had a noticeable effect. The saving in number of states stored directly carried over to a reduction of the size of the state space in main memory.

When using all three options together, [MC]SQUARE stored only 45 states, which is a reduction by 99.28%. The effect of the abstraction techniques did not add up completely, but they support each other. For example, DVR helps to reduce the number of states created when using PR.

Plant

Plant is a program that controls a fictive chemical plant. It consists of 225 lines of assembly code (73 lines of C code) and uses one timer and two interrupts. Using standard options [MC]SQUARE created 175,944 states, which used 44 MB of main memory. An older version of [MC]SQUARE (revision 1582) stored 801,616 states for this program.

DND did not reduce the size of the state space because this program does not read values from the environment and hence, only DND for interrupts showed an effect in this program. As DND for interrupts cannot be deactivated, the effect of the reduction cannot be seen. The number of states stored by the older version of [MC]SQUARE gives an impression on the effect of DND for interrupts.

DVR did not have any effect on the size of the state space because the program does not use local variables. PR reduced the size of the state space by 94.41% to only 9,844 states, which yielded a memory reduction from 44 MB to 2.82 MB. PR had such a big influence because the program mainly works using interrupts, and

each interrupt handler is a long single successor chain. Using DVR and PR together did not change the effect, and using all three techniques together did not change the effect either. Taken the old value of 801,616 states, all techniques together achieved a reduction by 98.77%.

Reentrance

The program called reentrance is used to demonstrate the reentrance problem. A variable i is accessed both in the main process and in the interrupt handler leading to invalid values of i. It is a small program consisting of only 148 lines of assembly code (37 lines of C code) and using one interrupt.

Without using options, [MC]SQUARE created 107,649 states. As in the previous described program, DND and DVR had no influence due to the same causes. That is, the program uses only interrupts and does not read input from the environment. Again, PR had a significant influence and stinted the state space by 93.84%. The number of states created was only increased by 12,038 states due to fewer revisits and shorter single successor chains as in the other programs. Combining the different techniques did not have an additional effect because only PR influenced the size of the state space.

Traffic Light

Traffic light is a program which was developed by one of our students in a lab course. Its purpose is to control a traffic light. The program has 155 lines of assembly code (85 lines of C code) and uses one timer and two interrupts.

Using standard options, [MC]SQUARE created a state space comprising 9,998 states. Revision 1582 of [MC]SQUARE stored 35,613 states for this program. Using DND and DVR did not have any influence as in the previous two programs. Again, the causes are that DND for interrupts is already used when using the standard configuration, the program only works with interrupts, and the program does not read input from the environment. The effect of DND for interrupts can be estimated by comparing the 9,998 states created by the new version of [MC]SQUARE and the 35,613 states created by the old version.

Again, PR had a significant effect and reduced the size of the state space to just 522 states, which is a reduction by 94.78%. Compared to the old value of 35,613 states, this is a reduction by 98.53%. Combining the different options did not have any further effect as PR is the only abstraction technique that influenced the size of the state space.

Window Lift

The last program called window lift is an automotive task. Here, a controller for a power window lift used in a car was implemented. This solution consists of 289 lines of assembly code (115 lines of C code) and it uses two interrupts and one timer.

Using no options, [MC]SQUARE stored 2,342,564 states. Revision 1582 of [MC]-SQUARE stored 10,100,400 states when checking this program. This difference gives an impression of the effect of DND for interrupts, which is now integrated into the standard configuration of [MC]SQUARE, as it was not fully integrated in revision 1582. DND for values reduced the size of the state space to 316,334 states, which is a reduction by 86.5%. DVR reduced the state space to 303,148 states, which is slightly better than DND. The biggest reduction was again caused by PR, which reduced the state space to 123,989 states. Using DVR and PR reduced the size of the state space to 15,687 states. In this combination, DVR supported PR by reducing the number of states created. Combining all three abstraction techniques reduced the state space to 5,232 states, which is a reduction by 99.78%.

8.1.2 Evaluation

During this case study, we observed no combination of abstraction techniques that increased the number of states stored. That is, usually the user can activate all options and model check his programs. As we did not use other formulas than **AG** true, it is possible that another formula reduces the effect of some abstraction techniques because, for example, dead variable reduction cannot be used for variables utilized within the formula. The effect of the different abstraction techniques varied depending on the program checked.

Delayed nondeterminism did not show an effect for some of the programs. It is important to notice that this is the case because only DND for values can be activated and deactivated. DND for interrupts is activated all the time. The number of states stored by revision 1582 of [MC]SQUARE gives an impression about the effects of DND for interrupts. But even in this revision a part of DND for interrupts was already implemented. DND for values only has an influence in programs reading input from the environment. Two different criteria of DND lead to a reduction of the number of states created. First, DND only splits up bits that are used within an instruction, and hence, all other bits may remain nondeterministic. This lowers the number of successors which have to be created. Second, DND delays the split up of nondeterministic values until the value is really needed. Hence, successors are created at a later point in time. Both criteria help to minimize the number of created states, while still preserving a safe over-approximation. This was proven in Sect. 5.7.

Dead variable reduction worked for some of the five programs. In general, DVR

works for programs that contain many functions, use local variables, and do not use their global variables in all functions (no tight coupling between functions). Combining DVR and PR helped to decrease the number of states only created due to the application of PR. In this situation DVR supported PR. A problem of DVR is that it cannot be used for variables that are utilized within the formula.

Path reduction reduced the number of states stored significantly in all five programs, especially in the programs using interrupts. In some programs (e.g., window lift and plant) time needed was not reduced because of revisits. It is a trade-off between time and space. As space is the limiting factor, PR should be always used. Using DVR together with PR helped to reduce the number of states created. The loss of validity of the next operator (see also Yorav and Grumberg [129] and Sect. 7.3) is not a problem for us as we do not use the next operator in our specifications.

As none of the abstraction techniques shows a negative effect, the user of [MC]-SQUARE always should use them all. In the worst case, they show no improvement. Usually, they significantly reduce the size of the state space and the time needed for checking.

Transferring the results to real-world examples, we think that the window lift program reflects the behavior of real-world programs better than the other programs because it uses interrupts and it communicates with the environment (reading and writing to and from the environment) at the same time. Hence, we expect that the abstraction techniques used in this case study show the same behavior on real-world examples as they showed for the window lift program. The problem is that we cannot prove this as we cannot build the state spaces of bigger programs without using all options. That is, it is not possible to compare the results. Therefore, we conducted this case study using small programs. The case study described in the next section used bigger programs and actual formulas.

8.2 An Automotive Microcontroller Application

This section presents a case study about model checking an automotive real-time problem with [MC]SQUARE. In this case study, we checked three different programs all solving the same problem. The problem that is solved by the programs is the concurrent speed measurement of four different signals (channels). The solutions were created by students in a lab course. We provide details about this case study elsewhere [106].

In the following, we first explain the application. After that, we detail the execution of the case study. That is, we explain for three programs the way the specifications were encoded, the results of the model checking process, and the way errors were located using counterexamples. In the end, we evaluate the results of this case study.

8.2.1 Application

As aforementioned, we took the programs used in this case study from a lab course such as the one presented by Salewski et al. [99]. In this lab course, students had to solve an automotive real-time problem, namely a four channel speed measurement with a CAN bus interface (see Fig. 8.1). As the electronic control unit (ECU), an ATMEL ATmega16 microcontroller[1] in combination with a PHILIPS SJA1000 stand alone CAN controller was used. Our intention was to integrate this ECU into the experimental vehicle developed at our institute for research and educational purposes[2].

The application involves several challenges: Concurrent measurement of four speed signals (frequency measurement of rectangularly shaped signals), sensor data processing within the limited resources of the given microcontroller, and interfacing with the external CAN controller. All these tasks have strong real-time requirements. Table 8.2 shows the functions used for the calculation of the speed values depending on the current frequency. Only 8 bit are available to store each speed value making this conversion necessary. Additionally, there are requirements for accuracy and response depending on the actual frequency of each individual sensor signal. During movement of the vehicle at high velocity shorter response times are required than during movement at low velocity. Accordingly, sensor signals with high frequency demand short response times (<100ms) while sensor signals with low frequency demand longer measurement intervals to achieve the required accuracy (maximum failure 10%).

Each group, which participated in the lab course, developed their own strategy to achieve an optimized compromise between short response times and high accuracy for different situations (all signals with high frequency, all with low frequency, different signal input, changes in signal input, etc.) resulting in different implementations. Further on, all groups had to pass an elementary acceptance test at the end of the lab course to obtain a certain level of quality in all versions, as described by Salewski et al. [99].

8.2.2 Execution

We conducted the case study on a server equipped with an AMD Opteron processor with 1.8 GHz, 16 GB main memory, and a hard disk with a capacity of 120 GB. In this case study, we used revision 1583 of [MC]SQUARE and applied all abstraction techniques such as delayed nondeterminism, dead variable reduction, and path reduction.

[1] In these lab courses CPLDs/FPGAs were also used, but they are not relevant for this case study.
[2] see also: `http://www-i11.informatik.rwth-aachen.de/modelcar.html`

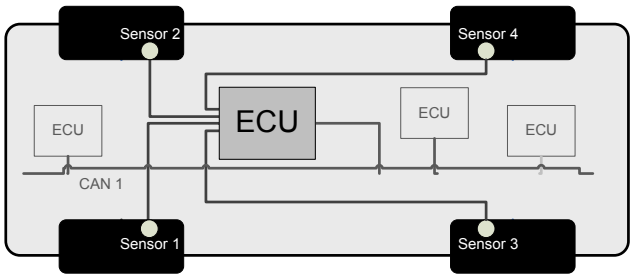

Figure 8.1: Application: 4-channel speed measurement with CAN bus interface.

Table 8.2: Functions to calculate the speed value.

Signal frequency f in Hz	**Speed value s**	**max. response in s**	**max. failure**
$f < 2$	$s = 0$	2	10%
$2 \leq f < 45$	$s = f * 0.0436 + 1$	1	10%
$45 \leq f < 550$	$s = f * 0.0436 + 1$	0.2	10%
$550 \leq f < 5733$	$s = f * 0.0436 + 1$	0.1	5%
$f \geq 5733$	$s = 251$	0.1	5%

We chose three out of the 23 different programs created in the lab course. [MC]-SQUARE could check some of the programs without any modification, while others had to be modified. To get consistent results, we decided to modify all programs in a uniform manner. As our focus was the calculation of the velocity and not the communication with the CAN bus (CAN bus was not modeled), we removed the sending of messages. Since the calculation is identical for all four wheels, we removed three of the four signals.

We checked the following properties on the programs: (see Tab. 8.2):

1. The program does not exhibit stack overflows (non-functional property).
2. The speed value s that is written to the CAN bus is always between 0 and 251 (functional property).
3. If the signal value f is greater than 5733 Hz, 251 is written to the CAN bus (functional property).

We did not encode the first property as a CTL formula because [MC]SQUARE automatically determines the size of the stack during the creation of the state space. Whenever a stack overflow occurs, [MC]SQUARE returns an error message. To check this property, we used the formula **AG** true. Using this formula, [MC]SQUARE only builds the state space and conducts its automatic checks.

Formula (8.1) represents the encoding of the second property as a CTL formula.

$$\mathbf{AG}\,(\mathsf{value} \geq 0 \wedge \mathsf{value} \leq 251) \tag{8.1}$$

This formula is an invariant and therefore, we used a special algorithm that is only able to check invariants. This algorithm is faster and uses less memory than the algorithm used for checking general CTL formulas.

We could not encode the third property for all programs using the same CTL formula because the third property depends on the structure and the data structures of the program under verification. Formula (8.2) demonstrates the general idea of a formula encoding the third property.

$$\mathbf{AG}\,(\mathsf{frequency} \geq 5733 \Rightarrow \mathbf{AF}\,\mathsf{velocity} = 251) \tag{8.2}$$

The problem is that not all programs internally measure the frequency in Hz and therefore, we had to determine the threshold for every single program alone. This formula is no longer an invariant but a general CTL formula and hence, we used the general CTL model checking algorithm.

The following sections describe the programs in more detail. Each section briefly presents the idea and the structure of a program. It describes the required modifications and the effort needed to change the respective program. Furthermore, it explains the errors found and presents solutions to these errors. Table 8.3 gives an overview about the results of this case study.

Program 1

The first program is structured as follows. The main function initializes all values and devices and then enters the `while(1){...}` loop. This loop is empty because all work is done by the different interrupt handlers. Each channel (wheel) has a separate interrupt assigned and the timer interrupt is used for generating time intervals. The communication with the CAN bus is done by three functions. To adapt this program, we deactivated the three interrupts used for the wheels not considered, and we commented out the contents of the function that sends the messages to the CAN bus. The time we needed to apply these changes was small.

The first property was not satisfied in this program. After checking 47,440,000 states (stored) [MC]SQUARE detected a stack overflow. The interrupt handler shown in List. 8.1 caused the stack overflow. The coder first deactivated interrupts (line 6) and at the end of the interrupt handler reactivated interrupts manually (line 34). This enabled nested interrupts. Thus, new interrupts could occur in the last lines of the handler and disrupt it. Thus, the stack grew because every time a new interrupt occurred, the calling context was pushed onto the stack. Eventually, a stack overflow occurred.

Listing 8.1: Part of program 1 exhibiting the error.

```
...
int freq0;
char s0;
...
SIGNAL (SIG_OVERFLOW0) {
  cli();
  outp(0x00,TCCR0);
  status++;
  if(status >= 8) {
    // calculate
    s0 = freq0 / 2;
...
    error_byte = 0;
    // limit value
    if (s0 >= 251) {
      s0 = 251;
      error_byte = error_byte + 1;
    }
...
    // send value
    if ((s0 == 0))
    {
      stat_low++;
      if (stat_low >= 5) {
        stat_low = 0;
        send();
      }
    }
    else send();
    freq0 = 0;
...
    status = 0;
    }
  sei();
  outp(0x00,TCNT0);
  outp(0x04,TCCR0);
}
...
```

Table 8.3: Results of the case study.

	Property	States stored	States created	Transitions	Time [s]	Satisfied	Stack size
1	1st	47,440,000	-	-	-	no	overflow
	2nd	8,507,012	192,523,046	13,749,892	4,753	yes*	22
	3rd	262,163	3,407,959	327,699	99	no	18
2	1st	85,297,279	936,881,099	115,860,688	29,120	yes	7
	2nd	4,070	46,234	4,071	1.7	no	7
	3rd	86,787,103	935,871,960	117,349,663	28,717	yes*	7
3	1st	7,012,375	179,830,878	14,024,727	5,067	yes	10
	2nd	7,012,375	179,830,878	14,024,727	5,620	yes	10
	3rd	7,012,375	179,830,878	14,024,727	5,651	yes	10

* after fixing the preceding error

To fix this error, we removed the call to cli() and sei() in this interrupt handler. We could do this because interrupts are automatically deactivated before an interrupt handler is entered and activated after it is handled. It seems that the coder of this program misunderstood this behavior. It is difficult to find this error by testing because many interrupts have to occur in a short time period. As it is possible, it is important to fix such errors. After applying the aforementioned changes, property one was satisfied. The maximum stack size detected in this program after fixing this error was 22.

Property two was satisfied in this program without any further modification. Property three was not satisfied in this program. We had to adapt the formula (see Form. (8.3)) as the frequency in this program was not measured in Hz.

$$\mathbf{AG}\,(\mathsf{freq0} \geq 502 \Rightarrow \mathbf{AF}\,\mathsf{s0} = 251) \tag{8.3}$$

The error, which led to the violation of the third property, occurred in the interrupt handler that is called periodically. In this interrupt handler, the variable used to store the value of the velocity overflowed. This interrupt handler is used to calculate the velocity and to pass the calculated value to the function performing the communication. The velocity (s0) is calculated by dividing the ticks (freq0) counted by the corresponding timer by two (see line 11). The coder of this program used a char (8bit) to represent the velocity and an int (16bit) to represent the number

of ticks. As the number of ticks can be greater than 512, it can happen that the variable used for the velocity overflows. Hence, if the number of ticks is greater or equal to 512, the velocity that is reported is greater or equal to zero instead of 251. We fixed this error by using an int value for the velocity instead of a char value.

Program 2

Program 2 is not structured at all. It is written in spaghetti code. The main function first initializes all values and devices, activates all interrupts, and then loops infinitely often. Each channel has its own interrupt assigned counting the number of signals. Furthermore, a timer interrupt is used to periodically calculate the velocity and send it to the CAN bus. The communication with the CAN bus is distributed over the complete program. The different sequences used to initialize the CAN bus, to specify the addresses, and to send the data are copied to all locations where they are needed.

Adapting this program was not as easy as it was for the first program. We deactivated the interrupts used for the wheels not considered as we did it for the first program. The removal of the communication with the CAN bus was time-consuming. We had to search for certain patterns and had to remove these patterns. Some parts of the communication were not commented out because the coder of this program conducted some calculation during the sending of the data, and we did not want to change the semantics of the program.

The first property was satisfied in this program. The maximum stack size measured for this program was 7. This is the smallest stack size of all programs used in this case study. This is not surprising because there are no functions calling other functions and interrupts are deactivated during all interrupt handlers in this program.

Property two was not satisfied in this program. Listing 8.2 shows the part of the program that caused the error. Line 4 checks whether counter1 is smaller than 251. After that, counter1 is incremented by one in line 5 and hence, the maximum value of counter1 is now 251. Before the value of the velocity is sent in line 13, it is again incremented by 1. Thus, the maximum value of the velocity sent is 252 instead of 251.

It would be easy to find this error by just inspecting the parts of the program that are shown in List. 8.2, but these parts are spread over the complete program which consists of 1,348 lines of C code. Therefore, this error was not found by the coder. To fix this error, we changed the boundary in line 4 from 251 to 250. Property three was satisfied after fixing the error encountered while checking the second property.

Listing 8.2: Part of program 2 showing the error.

```
...
SIGNAL (SIG_INTERRUPT0){
  help1++;
  if (help1 ==2  && counter1 < 251) {
    counter1++;
    help1 = 0;
  }
}
...
SIGNAL (SIG_OVERFLOW0) {
...
  asm volatile ("nop");
  outp (counter1 + 1 ,PORTA);
  outp (dat_wr_on,PORTB);
  outp (dat_wr_on,PORTB);
...
}
...
```

Program 3

Program 3 is structured similar to program 1. The main function first initializes all values and devices and then calculates the velocity and composes the messages within the while(1) loop using a state machine. Four interrupts are counting the signals generated by the different wheels. In contrast to program 1, the timer interrupt is counting the number of timer overflows that occurred. In program 1 the timer interrupt handler calculates the speed value. This program calculates the velocity in a separate function, which is called by the main function. The program handles the communication with the CAN bus by means of two other functions.

Adapting this program was rather easy as we only had to comment out the two functions dealing with the CAN communication and the three interrupt handlers counting the signals generated by the wheels not considered. Additionally, we had to make the variable representing the velocity globally accessible as propositions about local C variables are currently not possible in [MC]SQUARE. Property one was satisfied in this program. There were no stack overflows possible and the maximal observed stack size was 10. Both, property two and property three were also satisfied in this program without any changes.

8.2.3 Evaluation

In this case study, we analyzed three programs in detail, which we selected from 23 programs created by students in a lab course. The three programs we selected represent different ways the students solved the given problem. The problem features some interesting aspects. It is a real-time problem which also occurs this way, for example, in the automotive industry. Data is collected from sensors, calculations are conducted, and a result is sent via the CAN bus. It is the first time we used a data flow oriented program in a case study. The other case studies we did [103, 107] all concentrated on control flow oriented programs. Data flow oriented programs tend to have bigger state spaces and are more difficult to check using explicit model checking as less abstractions can be applied.

Our aim was to model check the given programs without any manual preparation (e.g., annotations and abstractions) as we did in other case studies [60, 88, 103, 107], but this was not possible for some of the 23 programs. We could check several programs without any modification but for others we had to apply modifications (e.g., comment out interrupts or remove communication with the CAN bus). Therefore, we decided to apply the same changes to all programs as we wanted to get comparable results. We think that these modifications have a scope that is small enough to be applicable. Programs that are well structured are easier to modify than programs without a clear structure. When modifying a program it is important to check whether the applied changes have an influence on the property that is checked. In this case study, it was easy to see that the applied changes did not influence the properties checked.

The differences in the sizes of the state spaces were only caused by the different implementations as all programs had to solve the same problem. Factors that influenced the sizes of the state spaces include: number of variables, use of global vs. use of local variables, variables storing the history of other variables, float divisions (not natively supported by the ATmega16), number of locations where interrupts are enabled, etc. Program 2 uses many global variables and many variables storing the history of other variables, and interrupts are enabled in almost all locations of this program. This resulted in the state space being the largest of all three programs. Program 1 contains more global variables than program 3, but not as many as program 2. As a consequence, the state space was smaller than the state space of program 2. Program 3 contains the least number of global variables. We even had to make one variable global in program 3 to check the third property. The results shown in Tab. 8.3 confirm these observations. Program 3 is the longest of the three programs, but it exhibited the smallest state space.

The size of the state space is not only influenced by data variables, but also by the values of registers and I/O registers. This influence is even stronger than the influence by data variables. As the three programs had to solve the same problem,

they used all the same microcontroller features. Hence, the differences in the sizes of the state spaces were not caused by the use of different features.

Other factors that usually influence the size of the state space could be automatically removed by the abstraction techniques implemented in [MC]SQUARE. For example, dead variable reduction removed the coupling of data variables between functions and interrupt handlers in some of the programs. In some programs DVR could not by used as DVR depends on the CFG, and the CFG could not be created for some programs due to indirect jumps or indirect calls. Delayed nondeterminism was used to alleviate the influence of the many different input values encountered in this application. This reduced, for example, the influence that delay loops have on the size of the state space.

The times needed for model checking shown in Tab. 8.3 range from 1 second up to 9 hours. The runs during which an error was found were obviously shorter than the runs that had to build the complete state space as it is usual for an on-the-fly model checking algorithms. The times needed for model checking these programs are that high because we used the hard disk model checking algorithm for all model checking runs done in this case study. As this algorithm stores parts of the states on hard disk, it needs more time than a model checking algorithm working in main memory only. The hard disk model checker was only needed for program 2, while the other two programs could be model checked without using the hard disk. As we wanted to get comparable results, we used this option for all programs. The time needed for model checking program 1 and 3 could be improved by a factor of approx. 10 by using the model checker that works in main memory only. When using the hard disk model checker, [MC]SQUARE can store up to 680,000,000 states on our server. When using the algorithm that works in memory only, [MC]SQUARE can store up to 69,000,000 states on our server. In both cases main memory is the limiting factor. The upper bounds given for the size of the state space refer to our server, which features 16 GB of main memory.

Summarizing, we think that this case study was successful. We only had to apply small changes to model check these data flow intensive programs. We could find some errors well known from real-world examples such as stack overflows and forgotten interrupt disabling in these implementations. These programs were written by students in a lab course and it was not planned to model check these programs. Hence, the students did not follow any guidelines that would make model checking easier. The length of the programs ranged from 300 to 500 lines of assembly code reachable during model checking. The original length of the programs varied from 1,000 to 5,000 lines of assembly code (400–1,300 lines of C code). The biggest resulting state space had 86,787,103 states (stored). As [MC]SQUARE can store up to 680,000,000 states on our server, there is still some performance reserve for more complex programs.

9 Conclusion

This chapter first summarizes the work described in this thesis and draws some conclusions. After that, we detail directions for future research and possible improvements.

9.1 Discussion

This thesis describes a new approach for model checking software for microcontrollers, which uses assembly code as input. Chapter 3 shows that model checking assembly code enables us to find errors that cannot be found in an intermediate representations such as C code. These errors include compiler errors, reentrance problems, stack overflows, and unintended use of microcontroller features. Model checking the assembly code has two problems. First, it makes the approach hardware dependent and second, due to the higher amount of details involved in assembly programs, the state spaces of assembly programs tend to be larger and thus aggravate the state-explosion problem. On the other hand, microcontroller programs are smaller than general-purpose computer programs and only feature a pseudo parallelism introduced by interrupts.

Chapter 4 describes an architecture that copes with hardware dependencies. Each step of the model checking process is put into a single package. Only the package creating the state space is implemented in a hardware dependent way. The state space is created using a simulator that simulates the effect of instructions on the microcontroller model. For this purpose, an existing, external simulator can be used, or a special-purpose simulator can be developed.

The modular architecture aids the management of hardware dependencies. Using an existing simulator, we observed the state-explosion problem as we could not influence the simulation, for example, by applying abstraction techniques. Therefore, we have developed a special-purpose simulator. By extending it to handle other microcontrollers without changing most packages, we have shown that the architecture is extendable. New static analyses or model checking algorithms can also be added without the need to adapt the complete tool. [MC]SQUARE is thus useful as a research tool for trying out new techniques. The GUI of [MC]SQUARE, which is operated like a simulator, enables users not familiar with formal methods to apply model checking.

The state-explosion problem is tackled in Chap. 5–7. Chapter 5 details our simulator, which creates the states for model checking. It natively handles nondeterminism and creates an over-approximation of the behavior of the microcontroller to preserve the validity of the model checking results. We tackle the state-explosion problem by accurately modeling important hardware features and abstracting hardware features that are not required. Furthermore, we apply abstraction techniques such as delayed nondeterminism, which introduces lazy states into [MC]SQUARE. A lazy state is a symbolic representation of not only a single state but a set of states. Thus, we have successfully combined explicit and symbolic model checking techniques within [MC]SQUARE. To show the validity of the delayed nondeterminism abstraction technique, we have developed a formal model of the simulator. Using this formal model, we have shown that the delayed nondeterminism abstraction technique preserves a simulation relation between the abstract and the concrete transition systems. That is, it preserved validity of ACTL and LTL formulas and of invariants.

The right degree of abstraction is mandatory for successfully applying model checking to microcontroller assembly code. If the abstraction is too coarse, interesting properties cannot be shown. If no abstractions are used, the state-explosion problem is likely to occur. Implementing the abstraction techniques within the simulator allows hardware-specific abstraction techniques to be applied. Combining symbolic and explicit model checking techniques enables the model checking of microcontroller programs that are otherwise unmanageable.

Chapter 6 details abstraction techniques that require static analysis. These abstraction techniques include dead variable reduction and path reduction. We have adapted these abstraction techniques and the underlying static analyses to be applicable to microcontroller assembly code. To improve the accuracy of the static analyses and, hence, to improve the results of these abstraction techniques, we have developed a model of the microcontroller, and we have implemented static analyses that determine information that is otherwise unavailable in static analyses such as the status of the Global Interrupt Enable bit.

Static analyses help to ease the state-explosion problem in many cases. They can be applied to large programs and still obtain results useful for model checking. However, microcontroller assembly code contains features that currently cause inaccurate results, such as indirect stores and loads, and lead to inefficient abstractions.

Chapter 7 presents the model checking algorithms applied, the algorithms employed to create the counterexamples, and the methods used to store states. The algorithms that create the counterexamples try to reduce their size. The counterexamples are presented in the assembly code, the CFG of the assembly code, the C code, and as a state space graph. In addition to the usual method of storing states in main memory, we have implemented a method to store states on hard disk. Furthermore, we have implemented a method that stores states incrementally. These techniques support the model checking of larger state spaces.

If a formula is not satisfied, a counterexample is very important as it allows users to locate the error within the program. Different representations of the counterexample enable users to choose the presentation that suits their needs best. The presentations within assembly and C code are similar to the traces of a simulator. The state spaces of real-world programs tend to be large despite the use of abstraction techniques. Therefore, it is important to store the states as efficiently as possible and to use techniques that enable the storage of large state spaces such as hard disk model checking. Whenever possible, time should be traded in for space.

Chapter 8 shows that our approach can successfully be used to model check microcontroller assembly programs. It describes two case studies that demonstrate the use of [MC]SQUARE on small programs without any manual preparation and on medium size programs with small manual adaptations. The second of these case studies shows that it is possible to use [MC]SQUARE to find and locate errors in real-world programs.

Summarizing, we think that applying model checking to microcontroller assembly code is practical for the programs considered in this thesis. The impact of the state-explosion problem is not as significant as when model checking personal computer programs. First, the accurate modeling of the microcontroller within the simulator and the use of hardware-dependent abstraction techniques such as delayed nondeterminism, path reduction, and dead variable reduction help to tackle the state-explosion problem. Second, in microcontroller programs there is no parallelism besides a pseudo parallelism introduced by interrupts. Furthermore, microcontroller programs are not as large as the programs that run on personal computers. The modular architecture makes our approach easy to extend. Using a simulator for the creation of the state space and hiding the internals behind the GUI enables users to use the tool like a generic simulator.

[MC]SQUARE can already be used in academia and education. It is able to find errors in real-world programs. To use [MC]SQUARE in industry projects, further research is needed to tackle the state-explosion problem. Moreover, the time to extend [MC]SQUARE to support new microcontrollers has to be further reduced. The next section discusses future research that helps to steer into this direction.

9.2 Future Work

Currently, there exists ongoing research to tackle the state-explosion problem. The diploma thesis by Gückel [49] describes applying user-defined models of the environment of a microcontroller. These environments limit the state space size and can be used to utilize [MC]SQUARE for testing. Furthermore, we are investigating whether we can extend the delayed nondeterminism abstraction technique to automatically remove further, unused details and the consequences of this extension on properties.

This research is focused on the abstraction of timers. Moreover, we are currently working on an abstraction technique that suppresses the occurrence of interrupts wherever possible without losing a safe over-approximation of the behavior of the microcontroller. This technique is related to partial order reduction [62], which is applied in SPIN.

As abstraction techniques are very important to this approach, future research should focus on the development of new hardware-dependent abstraction techniques and the adaptation of known abstraction techniques to microcontroller assembly code. We think that the combination of explicit and symbolic methods is promising. Therefore, it should be determined whether there are other symbolic techniques that could be integrated into our approach.

Further research into static analysis should concentrate on increasing the accuracy of the different static analyses. This can be achieved by, for example, improving the accuracy of the pointer analyses implemented in [MC]SQUARE. Currently, these analyses are inaccurate. Additionally, the accuracy of the microcontroller description used in the static analyses could be improved. It could be promising to reuse the simulator model of the microcontroller in the static analyses. Moreover, static analyses for finding errors independently of model checking should be added.

Explicit model checking is space- and time-consuming. We already have implemented an algorithm that uses the hard disk to store states. Additionally, the algorithms for state space building and model checking should be parallelized. Each single workstation of a cluster could then create states, exchange them over the network, and store them on the local hard disk.

To ease the creation of new simulators, research has to focus on techniques for automatically creating them from models given in hardware description languages. These simulators should natively support nondeterminism and create an over-approximation of the behavior shown by the corresponding microcontroller. Additionally, methods have to be developed to automatically generate the implementation of known abstraction techniques such as delayed nondeterminism.

Furthermore, we think that a transfer of this approach to other domains such as instruction list (IL) programs run on programmable logic controllers (PLCs) [75, 85] and abstract state machines [18, 53] would be interesting. Particularly, the transfer to IL programs on PLCs seems to be promising because IL programs are similar to assembly programs and PLCs are similar to microcontrollers. As PLCs are easier to handle than microcontrollers and involve less features, it could be possible to analyze larger programs.

Bibliography

[1] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Proc. 16th Int'l Conf. Computer Adided Verification (CAV 2004)*, volume 3114 of *Lecture Notes in Computer Science*, pages 484–487. Springer, 2004. ISBN 3-540-22342-8.

[2] T. Andrews, S. Qadeer, S. K. Rajamani, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *Proc. 15th Int'l Conf. Concurrency Theory (CONCUR 2004)*, volume 3170 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2004. ISBN 3-540-22940-X.

[3] Atmel Corporation. *8-bit AVR Instruction Set*, November 2005. URL `http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`. Visited: September 2007.

[4] Atmel Corporation. *Datasheet: ATmega16*, August 2007. URL `http://www.atmel.com/dyn/resources/prod_documents/doc2466.pdf`. Visited: September 2007.

[5] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008. ISBN 978-0262026499.

[6] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What you see is not what you execute. In *Proc. IFIP Working Conference on Verified Software: Theories, Tools, Experiments*, Zurich, Switzerland, 2005.

[7] G. Balakrishnan, T. W. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S.-H. Yong, C. H. Chen, and T. Teitelbaum. Model checking x86 executables with CodeSurfer/x86 and WPDS++. In *Proc. 17th Int'l Conf. Computer Aided Verification (CAV 2005)*, volume 3576 of *Lecture Notes in Computer Science*, pages 158–163. Springer, 2005. ISBN 3-540-27231-3. doi: http://dx.doi.org/10.1007/11513988_17.

[8] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proc. 7th Int'l SPIN Workshop SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes In Computer Science*, pages 113–130. Springer, 2000. ISBN 3-540-41030-9.

[9] T. Ball and S. K. Rajamani. The SLAM toolkit. In *Proc. 13th Int'l Conf. Computer Aided Verification (CAV '01)*, pages 260–264, London, UK, 2001. Springer. ISBN 3-540-42345-1.

[10] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. 8th Int'l SPIN Workshop Model checking of software (SPIN '01)*, pages 103–122, New York, NY, USA, 2001. Springer. ISBN 3-540-42124-6.

[11] T. Bao and M. Jones. Time-efficent model checking with magnetic disk. In *Proc. 11th Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 526–540. Springer, 2005. ISBN 3-540-25333-5.

[12] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. The SEI Series in Software Engineering. Pearson Education, second edition, 2003. ISBN 0-321-15495-9.

[13] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20(3):207–226, 1983.

[14] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. Workshop Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer, 1995.

[15] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie. *Systems and Software Verification: Model Checking Techniques and Tools*. Springer, Berlin, 2001. ISBN 3-540-41523-8.

[16] J. D. Bingham and A. J. Hu. Semi-formal bounded model checking. In *Proc. 14th Int'l Conf. Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 280–294, London, UK, 2002. Springer. ISBN 3-540-43997-8.

[17] M. Blaha and J. Rumbaugh. *Object-Oriented Modeling and Design with UML*. Pearson Education, second edition, 2005. ISBN 0-13-196859-9.

[18] E. Börger and R. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[19] R. E. Bryant. A switch-level model and simulator for MOS digital systems. *IEEE Transactions on Computers*, 33(2):160–177, 1984. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/TC.1984.1676408.

[20] R. E. Bryant. Symbolic simulation-techniques and applications. In *Proc. 27th ACM/IEEE Conf. Design automation (DAC '90)*, pages 517–521, New York, NY, USA, 1990. ACM. ISBN 0-89791-363-9. doi: http://doi.acm.org/10.1145/123186.128296.

[21] R. E. Bryant. A methodology for hardware verification based on logic simulation. *Journal of the ACM*, 38(2):299–328, 1991. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/103516.103519.

[22] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice & Experience*, 30(7): 775–802, 2000. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/(SICI)1097-024X(200006)30:7<775::AID-SPE309>3.0.CO;2-H.

[23] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design (FMSD)*, 25(2–3):129–166, 2004. ISSN 0925-9856. doi: http://dx.doi.org/10.1023/B:FORM.0000040026.56959.91.

[24] S. Chaki, E.M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6):388–402, 2004. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2004.22.

[25] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proc. 9th ACM Conf. Computer and Communications Security (CCS '02)*, pages 235–244, New York, NY, USA, 2002. ACM. ISBN 1-58113-612-9. doi: http://doi.acm.org/10.1145/586110.586142.

[26] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Proc. 14th Int'l Conf. Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364, London, UK, 2002. Springer. ISBN 3-540-43997-8.

[27] A. N. Clark. A lazy non-deterministic functional language. Visited: April 2007, 2000. URL `http://www.dcs.kcl.ac.uk/staff/tony/docs/LazyNonDetLanguage.ps`.

[28] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. ISBN 3-540-21299-X.

[29] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design (FMSD)*, 25(2–3):105–127, 2004. ISSN 0925-9856. doi: http://dx.doi.org/10.1023/B:FORM.0000040025.89719.f3.

[30] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer, 2005. ISBN 3-540-25333-5.

[31] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*. Springer, 1982. ISBN 3-540-11212-X. doi: 10.1007/BFb0025769.

[32] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/5397.5399.

[33] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT, Cambridge, Massachusetts, 1999. ISBN 0-262-03270-8.

[34] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. 12th Int'l Conf. Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000. ISBN 3-540-67770-4.

[35] M. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Proc. 10th Int'l Conf. Computer Aided Verification (CAV '98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer, 1998. ISBN 3-540-64608-6.

[36] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM SIGACT-SIGPLAN Symp. Principles of programming languages (POPL '77)*, pages 238–252, New York, NY, USA, 1977. ACM. doi: http://doi.acm.org/10.1145/512950.512973.

[37] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ analyzer. In *Proc. European Symp. Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005. ISBN 978-3-540-25435-5.

[38] D. Dams, O. Grumberg, and R. Gerth. Generation of reduced models for checking fragments of CTL. In *Proc. 5th Int'l Conf. Computer Aided Verification (CAV '93)*, pages 479–490, London, UK, 1993. Springer. ISBN 3-540-56922-7.

[39] D. L. Dill, A. J. Drexler, A. J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *Proc. 1991 IEEE Int'l Conf. Computer Design on VLSI in Computer & Processors (ICCD '92)*, pages 522–525, Washington, DC, USA, 1992. IEEE Computer Society. ISBN 0-8186-3110-4.

[40] DWARF Standards Committee. The DWARF debugging standard. `http://www.dwarfstd.org/`, 2007. Visited: January 2008.

[41] E. A. Emerson. Temporal and modal logics. In *Handbook of Theoretical Computer Science, vol. B*. Elsevier, 1990.

[42] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J. W. de Bakker and Jan van Leeuven, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 1980. ISBN 3-540-10003-2.

[43] E. A. Emerson and J. Y. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33 (1):151–178, 1986. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/4904.4999.

[44] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch. Goanna — a static model checker. In *Proc. 11th Int'l Workshop Formal Methods for Industrial Critical Systems (FMICS 2006)*, volume 4346 of *Lecture Notes in Computer Science*. Springer, 2006. ISBN 978-3-540-70951-0.

[45] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch. Model checking software at compile time. In *Proc. 1st Joint IEEE/IFIP Symp. Theoretical Aspects of Software Engineering (TASE '07)*, pages 45–56, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2856-2. doi: http://dx.doi.org/10.1109/TASE.2007.34.

[46] Free Software Foundation. STABS. `http://sourceware.org/gdb/current/onlinedocs/stabs.html`, 2006. Visited: January 2008.

[47] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, 1st edition, 1995. ISBN 978-0201633610.

[48] M. K. Ganai, A. Gupta, and P. Ashar. DiVer: SAT-based model checking platform for verifying large scale systems. In *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 575–580. Springer, 2005. ISBN 3-540-25333-5.

[49] D. Gückel. Extending the model checker [mc]square by user-defined environments. Diploma thesis, RWTH Aachen University, 2007.

[50] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. *ACM SIGPLAN Notices*, 40(6):213–223, 2005. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1064978.1065036.

[51] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th Int'l Conf. Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997. ISBN 3-540-63166-6.

[52] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):843–871, 1994. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/177492.177725.

[53] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[54] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. *ACM SIGPLAN Notices*, 37(5):69–82, 2002. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/543552.512539.

[55] M. Hammer and M. Weber. "To store or not to store" reloaded: Reclaiming memory on demand. In *Proc. 11th Int'l Workshop Formal Methods for Industrial Critical Systems (FMICS 2006)*, volume 4346 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2006. ISBN 978-3-540-70951-0.

[56] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In *Proc. 8th Int'l Conf. Computer Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 423–427. Springer, 1996. ISBN 978-3-540-61474-6.

[57] K. Heljanko. Model checking the branching time temporal logic CTL. Research Report A45, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, May 1997.

[58] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. *International Journal of Software Tools for Technology Transfer*, 1:110–122, 1997.

[59] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. *ACM SIGPLAN Notices*, 37(1):58–70, 2002. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/565816.503279.

[60] G. Herberich, T. Noll, B. Schlich, and C. Weise. Proving correctness of an efficient abstraction for interrupt handling. In *Proc. 3rd Int'l Workshop Systems Software Verification (SSV 08)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008. To appear.

[61] M. Hitz, G. Kappel, E. Kapsammer, and W. Retschitzegger. *UML@Work: Objektorientierte Modellierung mit UML2*. dpunkt.verlag, third edition, 2005. ISBN 3-89864-261-5.

[62] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2004. ISBN 9780321228628.

[63] G. J. Holzmann. The engineering of a model checker: The Gnu i-protocol case study revisited. In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors, *SPIN*, volume 1680 of *Lecture Notes in Computer Science*, pages 232–244, Berlin, 1999. Springer. ISBN 3-540-66499-8.

[64] G. J. Holzmann and M. H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, 2000.

[65] A. J. Hu, G. York, and D. L. Dill. New techniques for efficient verification with implicitly conjoined BDDs. In *Proc. 31st Annual Conf. Design Automation (DAC '94)*, pages 276–282, New York, NY, USA, 1994. ACM. ISBN 0-89791-653-0. doi: http://doi.acm.org/10.1145/196244.196377.

[66] International Electrotechnical Commission. Functional safety for electrical / electronic / programmable electronic safety-related systems. IEC61508, 1998.

[67] F. Ivanicic, I. Shlyakhter, A. Gupta, and M. K. Ganai. Model checking C programs using F-Soft. In *Proc. 2005 Int'l Conf. Computer Design (ICCD '05)*, pages 297–308. IEEE Computer Society, 2005. ISBN 0-7695-2451-6. doi: http://dx.doi.org/10.1109/ICCD.2005.77.

[68] M. Kanellos. Software glitch stalls some Toyota hybrids. `http://www.news.com/Software-glitch-stalls-some-Toyota-hybrids/2100-11389_3-5895574.html`, October 2005. Visited: January 2008.

[69] C. W. Keller, D. Saha, S. Basu, and S. A. Smolka. FocusCheck: A tool for model checking and debugging sequential C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 563 – 569. Springer, 2005.

[70] O. Kupferman and M. Y. Vardi. Module checking. In *Proc. 8th Int'l Conf. Computer Aided Verification (CAV 96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 75–86. Springer, 1996.

[71] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *Proc. 18th IEEE Real-Time Systems Symposium (RTSS '97)*, pages 14–24, Washington, DC, USA, 1997. IEEE. ISBN 0-8186-8268-X.

[72] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[73] P. Leven, T. Mehler, and S. Edelkamp. Directed error detection in C++ with the assembly-level model checker StEAM. In *Model Checking Software (SPIN)*, volume 2989 of *Lecture Notes in Computer Science*, pages 39–56. Springer, 2004. ISBN 3-540-21314-7.

[74] M. Lewis and M. Jones. A dead variable analysis for explicit model checking. In *Proc. 2006 ACM SIGPLAN Symp. Partial evaluation and semantics-based program manipulation (PEPM '06)*, pages 48–57, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-196-1. doi: http://doi.acm.org/10.1145/1111542.1111551.

[75] R. W. Lewis. *Programming Industrial Control Systems Using IEC 1131-3 (I E E Control Engineering Series)*. Institution of Electrical Engineers, Stevenage, UK, 1998. ISBN 0852969503.

[76] J. L. Lions. Ariane 5 flight 501 failure: Report of the inquiry board. Available at: `http://www.esa.int`, July 1996.

[77] J. Löll. Application of static analysis in the field of model checking software for embedded systems. Diploma thesis, RWTH Aachen University, 2007.

[78] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems*. Springer, New York, 1995. ISBN 0-387-94459-1.

[79] K. L. McMillan. *Symbolic model checking - an approach to the state explosion problem*. PhD thesis, SCS, Carnegie Mellon University, 1992.

[80] T. Mehler. *Challenges and Applications of Assembly-Level Software Model Checking*. PhD Dissertation, Universität Dortmund, 2005.

[81] T. Mehler and P. Leven. Introduction to StEAM - an assembly-level software model checker. Technical Report 193, University of Dortmund and University of Freiburg, September 2003.

[82] E. Mercer and M. Jones. Model checking machine code with the GNU debugger. In *Proc. 12th Int'l SPIN Workshop*, volume 3639 of *Lecture Notes in Computer Science*, pages 251–265. Springer, 2005. ISBN 978-3-540-28195-5.

[83] J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *ENTCS*, 117: 153–182, 2005.

[84] R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd Int'l Joint Conf. on Artificial Intelligence (IJCAI 1971)*, pages 481–489, 1971. ISBN 0-934613-34-6.

[85] P. D. Monari, F. Bonfatti, and U. Sampieri. *IEC 61131-3 Programming Methodology: Software Engineering Methods for Industrial Automated Systems*. ICS Triplex ISaGRAF, 2003. ISBN 978-0973467000.

[86] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. 11th Int'l Conf. Compiler Construction (CC '02)*, volume 2304 of *Lecture Notes In Computer Science*, pages 213–228. Springer, 2002. ISBN 3-540-43369-4.

[87] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, Berlin, corrected 2nd printing edition, 2005. ISBN 3-540-65410-0.

[88] T. Noll and B. Schlich. Delayed nondeterminism in model checking embedded systems assembly code. In *Proc. 3rd Int'l Haifa Verification Conf. (HVC 2007)*, volume 4899 of *Lecture Notes in Computer Science*, pages 185–201. Springer, 2008. ISBN 978-3-540-77964-3. doi: http://dx.doi.org/10.1007/978-3-540-77966-7_16.

[89] J. Peleska and A. E. Haxthausen. Object code verification for safety-critical railway control systems. In E. Schnieder and G. Tarnai, editors, *Proc. 6th Symp. Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007)*, pages 184–199. GZVB, Braunschweig, Germany, 2007. ISBN 978-3-937655-09-3.

[90] J. Peleska and H. Löding. Symbolic and abstract interpretation for C/C++ programs. In *Proc. 3rd Int'l Workshop Systems Software Verification (SSV 08)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008. To appear.

[91] J. Peleska, H. Löding, and T. Kotas. Test automation meets static analysis. In *Proc. Informatik 2007 — Informatik trifft Logistik*, volume P-110 of *Lecture Notes in Informatics*, pages 280–286, 2007. ISBN 978-3-88579-204-8.

[92] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Foundations of Computer Science (FOCS 1977)*, pages 46–57. IEEE Computer Society, 1977.

[93] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[94] S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *Proc. ACM SIGPLAN 2004 Conf. Programming language design and implementation (PLDI '04)*, pages 14–24. ACM, 2004. ISBN 1-58113-807-5. doi: http://doi.acm.org/10.1145/996841.996845.

[95] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, volume 137 of *Lecture Notes In Computer Science*, pages 337–351. Springer, 1982. ISBN 3-540-11494-7.

[96] G. Quirós. Static byte-code analysis for state space reduction. Master's thesis, RWTH Aachen University, March 2006.

[97] 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. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1037949.1024410.

[98] M. Rohrbach. An approach for model checking embedded systems software. Diploma thesis, RWTH Aachen University, 2006.

[99] F. Salewski, D. Wilking, and S. Kowalewski. Diverse hardware platforms in embedded systems lab courses: a way to teach the differences. *ACM SIGBED Review*, 2(4):70–74, 2005. ISSN 1551-3688. doi: http://doi.acm.org/10.1145/1121812.1121825.

[100] F. Scheuer. Extending the model checker [mc]square to handle the Infineon XC167 microcontroller. Diploma thesis, RWTH Aachen University, 2007.

[101] B. Schlich and S. Kowalewski. Model checking C source code for embedded systems. In T. Margaria, B. Steffen, and M. G. Hinchey, editors, *Proc. IEEE/-NASA Workshop Leveraging Applications of Formal Methods, Verification, and Validation (IEEE/NASA ISoLA 2005)*, pages 65–77. NASA, Maryland, USA, 2005. NASA/CP-2005-212788.

[102] B. Schlich and S. Kowalewski. [mc]square: A model checker for microcontroller code. In T. Margaria, A. Philippou, and B. Steffen, editors, *Proc. 2nd Int'l Symp. Leveraging Applications of Formal Methods, Verification and Validation*

(IEEE-ISoLA 2006), pages 466–473. IEEE Computer Society, 2006. ISBN 978-0-7695-3071-0. doi: http://dx.doi.org/10.1109/ISoLA.2006.62.

[103] B. Schlich and S. Kowalewski. An extendable architecture for model checking hardware-specific automotive microcontroller code. In E. Schnieder and G. Tarnai, editors, *Proc. 6th Symp. Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007)*, pages 202–212. GZVB, Braunschweig, Germany, 2007. ISBN 978-3-937655-09-3.

[104] B. Schlich and S. Kowalewski. Model checking C source code for embedded systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 2008. Accepted for publication.

[105] B. Schlich, M. Rohrbach, M. Weber, and S. Kowalewski. Model checking software for microcontrollers. Technical Report AIB-2006-11, RWTH Aachen University, August 2006. URL `http://aib.informatik.rwth-aachen.de/2006/2006-11.pdf`.

[106] B. Schlich, F. Salewski, and S. Kowalewski. Applying model checking to an automotive microcontroller application. In *Proc. IEEE 2nd Int'l Symp. Industrial Embedded Systems (SIES 2007)*, pages 209–216. IEEE, 2007. ISBN 1-4244-0840-7. doi: http://dx.doi.org/10.1109/SIES.2007.4297337.

[107] B. Schlich, J. Löll, and S. Kowalewski. Application of static analyses for state space reduction to microcontroller assembly code. In *Proc. 12th Int'l Workshop Formal Methods for Industrial Critical Systems (FMICS 2007)*, volume 4916 of *Lecture Notes in Computer Science*. Springer, 2008.

[108] K. Schneider. *Verification of Reactive Systems*. Texts in Theoretical Computer Science, EATCS Series. Springer, Berlin, 2004. ISBN 3-540-00296-0.

[109] J. F. Schommer. Symbolisches model-checking mit [mc]square. Diploma thesis, RWTH Aachen University, 2007.

[110] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, TU Munich, 2002.

[111] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. 10th European Software Engineering Conf. / 13th ACM SIGSOFT Int'l Symp. Foundations of Software Engineering (ESEC/FSE-13)*, pages 263–272. ACM, 2005. ISBN 1-59593-014-0. doi: http://doi.acm.org/10.1145/1081706.1081750.

[112] U. Stern and D. L. Dill. Using magnetic disk instead of main memory in the Murphi verifier. In *Proc. 10th Int'l Conf. Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 172–183. Springer, 1998. ISBN 3-540-64608-6.

[113] H. Störrle. *UML 2 für Studenten*. Pearson Studium, Munich, Germany, 2005. ISBN 3-8273-7143-0.

[114] S. Subramanian and J. Cook. Automatic verification of object code against source code. In *Proc. 11th Annual Conf. Computer Assurance (COMPASS'96)*, pages 46–55. IEEE Press, 1996. ISBN 9780780333901.

[115] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972. doi: 10.1137/0201010.

[116] TIS Committee. Tool interface standard executable and linking format specification version 1.2. `http://x86.ddj.com/ftp/manuals/tools/elf.pdf`, May 1995. Visited: September 2007.

[117] B. L. Titzer. Avrora: The AVR simulation and analysis framework. Master's thesis, University of California, Los Angeles, 2004.

[118] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proc. 4th Int'l Conf. Information Processing in Sensor Networks (IPSN'05)*, 2005.

[119] C. HK Tsang, C. SW Lau, and Y. K Leung. *Object-Oriented Technology: From Diagram to Code with Visual Paradigm for UML*. McGraw-Hill Education, 2005. ISBN 007-124046-2.

[120] K. Varpaaniemi, K. Heljanko, and J. Lilius. prod 3.2: An advanced tool for efficient reachability analysis. In *Proc. 9th Int'l Conf. Computer Aided Verification (CAV '97)*, pages 472–475, London, UK, 1997. Springer. ISBN 3-540-63166-6.

[121] B. Vergauwen and J. Lewi. A linear local model checking algorithm for CTL. In *Proc. 4th Int'l Conf. Concurrency Theory (CONCUR '93)*, volume 715 of *Lecture Notes in Computer Science*, pages 447–461, Berlin, 1993. Springer. ISBN 3-540-57208-2.

[122] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), 2003.

[123] W. C. Visser. *Efficient CTL* Model Checking using Games and Automata*. Phd thesis, University of Manchester, 1998.

[124] M. Wahab. *Object Code Verification*. Phd thesis, Department of Computer Science, University of Warwick, Coventry, UK, 1998.

[125] M. Weber. *Parallel Algorithms for Verification of Large Systems*. Phd dissertation, RWTH Aachen University, 2006. AIB-2006-02.

[126] M. Weber. An embeddable virtual machine for state space generation. In *14th Int'l SPIN Workshop Model Checking Software*, volume 4595 of *Lecture Notes in Computer Science*. Springer, 2007.

[127] G. Weißenbacher. An abstraction/refinement scheme for model checking C programs. Master's thesis, Institut für Softwaretechnologie der Technischen Universität Graz, 2003.

[128] A. Wolfe. Intel fixes a pentium FPU glitch. *Electronic Engineering Times*, 822:1, 1994.

[129] K. Yorav and O. Grumberg. Static analysis for state-space reductions preserving temporal logics. *Formal Methods in System Design*, 25(1):67–96, 2004. ISSN 0925-9856. doi: http://dx.doi.org/10.1023/B:FORM.0000033963.55470.9e.

[130] S. Yovine. Kronos: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, 1(1/2):123–133, 1997.

List of Abbreviations

ACTL	universal fragment of CTL
ADC	analog to digital converter
BP	breaking point
CAN	Controller Area Network
CEGAR	counterexample-guided abstraction refinement
CFA	control flow analysis
CFG	control flow graph
CIL	C intermediate language
CTL	Computation Tree Logic
DDRx	Port x Data Direction Register
DFA	data flow analysis
DFS	depth first search
DND	delayed nondeterminism
DVR	dead variable reduction
ECTL	existential fragment of CTL
ECU	electronic control unit
EEPROM	electrically erasable programmable read-only memory
ELF	Executable and Linking Format
GCC	GNU Compiler Collection
GIFA	global interrupt flag analysis
GIFR	Global Interrupt Flag Register
GUI	graphical user interface
I bit	Global Interrupt Enable bit
I/O	input/output
ICVM	Internet C Virtual Machine
IH	interrupt handler
JTAG	Joint Test Action Group
LTL	Linear Temporal Logic
LTS	labeled transition system
LVA	live variables analysis
OCR0	Output Compare Register 0
PC	program counter
PINx	Port Input Pins Address
PORTx	Port x Data Register

PR	path reduction
RDA	reaching definitions analysis
SAT	Boolean satisfiability problem
SCC	strongly connected component
SPI	Serial Peripheral Interface
SRAM	static random access memory
SREG	Status Register
TBDM	to be determinized mask
TCCR0	Timer/Counter Control Register 0
TCNT0	Timer/Counter Register 0
TIFR	Timer Interrupt Flag Register
TIMSK	Timer Interrupt Mask Register
UML	Unified Modeling Language
USART	Universal Synchronous and Asynchronous Receiver and Transmitter
VM	virtual machine

Index

$Behavior_{\mathbf{GIF}}(f)$, 88
$Behavior_{\mathbf{LV}}(f)$, 81
$Behavior_{\mathbf{RD}}(f)$, 85

abstract interpretation, 12, 76, 87
ACTL, 9
available expressions analysis, 12

breaking point, 91

control flow analysis, 11, 79–81
control flow graph, 12, 79
core, 33
counterexample, 11, 107
CTL, 7

data flow analysis, 11
data flow equation, 12
data memory, 34, 37
dead variable reduction, 90–91
delayed nondeterminism, 58–62, 68
determinizer, 53–62

ECTL, 9
EEPROM, 37

factory, 111
formal model, 62–70
formula
    length of, 8

global interrupt flag analysis, 88–90

I bit, 39, 88
I/O port, 40–44
initial static behavior, 83
instantiation, 53
    delayed, 58–62, 68
    immediate, 56–58, 68
instruction simulator, 50–53
interrupt, 39–40
interrupt handler, 39
invariant, 9, 106

job
    counterexample, 108
    verification, 101
      dependent, 102

Kripke structure, 8, 9

lazy interrupt evaluation, 40
lazy stack evaluation, 38
live variables analysis, 12, 81–84
LTL, 7
LTS, 9, 70

memory, 34–38
microcontroller, 32–47
model checker, 97–99
model checking, 9, 97–114
    explicit, 10
    global, 11, 103–106
    local, 11, 99–103
    symbolic, 10

nondeterminism, 53–62, 67–69

partial order reduction, 94, 134
path reduction, 91–93

program, 48–50
program analysis, 11
Program Counter, 34
program memory, 37–38

reaching definitions analysis, 12, 84–86

simulator, 29–62
SRAM, 34–37
SREG, 39, 88
stack, 38–39
    counterexample, 102
    global, 101
    local, 102
stack analysis, 86–87
state
    goal, 103
    lazy, 32, 34, 58
    microcontroller, 31
    simulator, 31–32
state space, 109–112
static analysis, 11, 75–95
static behavior, 81
static slicing, 95
subformula, 8
    evaluation order, 9
    proper, 8

TBDM, 34
timer, 44–47

witness, 11, 107
worklist algorithm, 12, 81

Aachener Informatik-Berichte

This list contains all technical reports published during the past five years. A complete list of reports dating back to 1987 is available from

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

To obtain copies consult the above URL or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2003-01 *	Jahresbericht 2002
2003-02	Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting
2003-03	Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations
2003-04	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs
2003-05	Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard
2003-06	Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates
2003-07	Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung
2003-08	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs
2004-01 *	Fachgruppe Informatik: Jahresbericht 2003
2004-02	Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic
2004-03	Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting
2004-04	Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming
2004-05	Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming

2004-06 Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming

2004-07 Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination

2004-08 Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information

2004-09 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity

2004-10 Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules

2005-01 * Fachgruppe Informatik: Jahresbericht 2004

2005-02 Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: "Aachen Summer School Applied IT Security"

2005-03 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions

2005-04 Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem

2005-05 Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots

2005-06 Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information

2005-07 Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks

2005-08 Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut

2005-09 Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures

2005-10 Benedikt Bollig: Automata and Logics for Message Sequence Charts

2005-11 Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture

2005-12 Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems

2005-13 Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments

2005-14 Felix C. Freiling, Sukumar Ghosh: Code Stabilization

2005-15 Uwe Naumann: The Complexity of Derivative Computation

2005-16 Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)

2005-17 Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)

2005-18 Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises "Features"

2005-19 Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers

2005-20 Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.

2005-21 Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited

2005-22 Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins

2005-23 Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves

2005-24 Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks

2006-01 * Fachgruppe Informatik: Jahresbericht 2005

2006-02 Michael Weber: Parallel Algorithms for Verification of Large Systems

2006-03 Michael Maier, Uwe Naumann: Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler

2006-04 Ebadollah Varnik, Uwe Naumann, Andrew Lyons: Toward Low Static Memory Jacobian Accumulation

2006-05 Uwe Naumann, Jean Utke, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunsch, Mike Fagan, Nathan Tallent, Michelle Strout: Adjoint Code by Source Transformation with OpenAD/F

2006-06 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Divide-and-Color

2006-07 Thomas Colcombet, Christof Löding: Transforming structures by set interpretations

2006-08 Uwe Naumann, Yuxiao Hu: Optimal Vertex Elimination in Single-Expression-Use Graphs

2006-09 Tingting Han, Joost-Pieter Katoen: Counterexamples in Probabilistic Model Checking

2006-10 Mesut Günes, Alexander Zimmermann, Martin Wenig, Jan Ritzerfeld, Ulrich Meis: From Simulations to Testbeds - Architecture of the Hybrid MCG-Mesh Testbed

2006-11 Bastian Schlich, Michael Rohrbach, Michael Weber, Stefan Kowalewski: Model Checking Software for Microcontrollers

2006-12 Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning

2006-13 Wong Karianto, Christof Löding: Unranked Tree Automata with Sibling Equalities and Disequalities

2006-14 Danilo Beuche, Andreas Birk, Heinrich Dreier, Andreas Fleischmann, Heidi Galle, Gerald Heller, Dirk Janzen, Isabel John, Ramin Tavakoli Kolagari, Thomas von der Maßen, Andreas Wolfram: Report of the GI Work Group "Requirements Management Tools for Product Line Engineering"

2006-15 Sebastian Ullrich, Jakob T. Valvoda, Torsten Kuhlen: Utilizing optical sensors from mice for new input devices

2006-16 Rafael Ballagas, Jan Borchers: Selexels: a Conceptual Framework for Pointing Devices with Low Expressiveness

2006-17 Eric Lee, Henning Kiel, Jan Borchers: Scrolling Through Time: Improving Interfaces for Searching and Navigating Continuous Audio Timelines

2007-01 * Fachgruppe Informatik: Jahresbericht 2006

2007-02 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl: SAT Solving for Termination Analysis with Polynomial Interpretations

2007-03 Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp: Proving Termination by Bounded Increase

2007-04 Jan Buchholz, Eric Lee, Jonathan Klein, and Jan Borchers: coJIVE: A System to Support Collaborative Jazz Improvisation

2007-05 Uwe Naumann: On Optimal DAG Reversal

2007-06 Joost-Pieter Katoen, Thomas Noll, and Stefan Rieger: Verifying Concurrent List-Manipulating Programs by LTL Model Checking

2007-07 Alexander Nyßen, Horst Lichter: MeDUSA - MethoD for UML2-based Design of Embedded Software Applications

2007-08 Falk Salewski and Stefan Kowalewski: Achieving Highly Reliable Embedded Software: An empirical evaluation of different approaches

2007-09 Tina Kraußer, Heiko Mantel, and Henning Sudbrock: A Probabilistic Justification of the Combining Calculus under the Uniform Scheduler Assumption

2007-10 Martin Neuhäußer, Joost-Pieter Katoen: Bisimulation and Logical Preservation for Continuous-Time Markov Decision Processes

2007-11 Klaus Wehrle (editor): 6. Fachgespräch Sensornetzwerke

2007-12 Uwe Naumann: An L-Attributed Grammar for Adjoint Code

2007-13 Uwe Naumann, Michael Maier, Jan Riehme, and Bruce Christianson: Second-Order Adjoints by Source Code Manipulation of Numerical Programs

2007-14 Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch: OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes

2007-15 Volker Stolz: Temporal assertions for sequential and concurrent programs

2007-16 Sadeq Ali Makram, Mesut Güneç, Martin Wenig, Alexander Zimmermann: Adaptive Channel Assignment to Support QoS and Load Balancing for Wireless Mesh Networks

2007-17 René Thiemann: The DP Framework for Proving Termination of Term Rewriting

2007-18 Uwe Naumann: Call Tree Reversal is NP-Complete

2007-19 Jan Riehme, Andrea Walther, Jörg Stiller, Uwe Naumann: Adjoints for Time-Dependent Optimal Control

2007-20 Joost-Pieter Katoen, Daniel Klink, Martin Leucker, and Verena Wolf: Three-Valued Abstraction for Probabilistic Systems

2007-21 Tingting Han, Joost-Pieter Katoen, and Alexandru Mereacre: Compositional Modeling and Minimization of Time-Inhomogeneous Markov Chains

2007-22 Heiner Ackermann, Paul W. Goldberg, Vahab S. Mirrokni, Heiko Röglin, and Berthold Vöcking: Uncoordinated Two-Sided Markets

2008-01 * Fachgruppe Informatik: Jahresbericht 2007

2008-02 Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing

2008-03 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René :Thiemann, Harald Zankl: Maximal Termination

2008-04 Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler

2008-05 Frank G. Radmacher: An Automata Theoretic Approach to the Theory of Rational Tree Relations

2008-06 Uwe Naumann, Laurent Hascoet, Chris Hill, Paul Hovland, Jan Riehme, Jean Utke: A Framework for Proving Correctness of Adjoint Message Passing Programs

2008-08 George B. Mertzios, Stavros D. Nikolopoulos: The λ-cluster Problem on Parameterized Interval Graphs

2008-09 George B. Mertzios, Walter Unger: An optimal algorithm for the k-fixed-endpoint path cover on proper interval graphs

2008-10 George B. Mertzios, Walter Unger: Preemptive Scheduling of Equal-Length Jobs in Polynomial Time

2008-11 George B. Mertzios: Fast Convergence of Routing Games with Splittable Flows

2008-12 Joost-Pieter Katoen, Daniel Klink, Martin Leucker, Verena Wolf: Abstraction for stochastic systems by Erlang's method of stages

2008-13 Beatriz Alarcón, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Raúl Gutierrez, Salvador Lucas, Peter Schneider-Kamp, René Thiemann: Improving Context-Sensitive Dependency Pairs

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.

Curriculum Vitae

Name	Bastian Schlich
Day of birth	28.03.1978
Place of birth	Essen/Ruhr

2004 –	Wissenschaftlicher Angestellter am Lehrstuhl Informatik 11 an der RWTH Aachen University
1998 – 2004	Studium der Informatik an der Universität Dortmund
1997 – 1998	Grundwehrdienst
1991 – 1997	Helmholtzschule in Essen
1988 – 1991	Otto-Pankok-Schule in Mülheim an der Ruhr
1984 – 1988	Städtische Gemeinschaftsgrundschule an der Hölterstraße in Mühlheim an der Ruhr