DIPLOMA THESIS
for an academic degree
"Master of Science in Engineering"

Model Checking and Static Analysis of Intel MCS-51 Assembly Code

by Thomas Reinbacher, BSc
A-2020 Kleinstelzendorf, Weidenweg 44

1. Examiner: FH-Prof. Dipl.-Ing. Dr. Martin Horauer
2. Examiner: Ing. Dipl.-Ing. Michael Kramer
Vienna, June 3rd, 2009

Written at the University of Applied Sciences Technikum Wien
Master Degree Programme Embedded Systems
Affidavit

„I hereby declare by oath that I have written this paper myself. Any ideas and concepts taken from other sources either directly or indirectly have been referred to as such. The paper has neither in the same nor similar form been handed in to an examination board, nor has it been published. “

Place, Date

Signature
Abstract

Verification of embedded systems software is crucial for providing flawless functionality of nowadays intelligent computer systems found in automobiles, elevators, aircrafts, medical devices, robots, etc. The common approach most widely used in industry relies on testing of defined corner cases. Although everyone is aware of the fact that only a very limited set of the test space can be covered in this way, no other more complete approaches have been widely adopted so far.

Formal verification methods such as model checking complemented with various techniques to reduce state spaces has recently gained some momentum in this regard. Nevertheless, formal verification of embedded systems still played a minor role in the past. Practical restrictions of this approach are (i) due to the problem to (manually) create a model of the system beforehand and (ii) due to the resulting large state spaces.

This master thesis focuses on model checking and static analysis of Intel MCS-51 assembly code with the [mc]square framework.

In the presented approach, issue (i) is solved by using a dedicated target CPU simulator. In order to tackle (ii) existing abstraction techniques are adapted for the Intel MCS-51 target architecture. A novel state space reduction technique termed Delayed Nondeterminism with Look Ahead is introduced. The presented abstraction technique centers around the coherence among boolean operators with particular regard to the 3-valued microcontroller memory model.

Besides, the Intel MCS-51 CPU simulator is integrated into the existing static analysis framework of [mc]square. A novel data-flow analysis termed Register Bank Analysis is described in order to handle register bank swapping. Register bank swapping is a particular feature of some embedded microcontrollers such as the Intel MCS-51. This approach allows narrowing and refining the subsequent data-flow analyses, leading to more precise analysis results. The additional precision in turn contributes to a reduction of state spaces during model checking.

In order to evaluate the benefits and to show the applicability of the introduced concepts, a real world case study is conducted. The case study source code is taken from an industrial application. The microcontroller software is model checked with [mc]square by taking advantage of the presented state space abstractions and static analysis techniques.

Keywords: Assembly code model checking, static analysis of assembly code, abstraction techniques, case study, [mc]square
Kurzfassung


Formale Verifikationsmethoden wie Model Checking sind vielversprechende Ansätze um die Fehlerfreiheit von Software zu zeigen. Im Kontext von Embedded Systems spielten diese formalen Ansätze in der Vergangenheit nur eine untergeordnete Rolle. In der Praxis zeigen sich Schwierigkeiten durch (i) die manuell durchgeführte Modellierung des Systems und (ii) die unhandbar großen Zustandsräume.


Acknowledgements

Not because it is customary, but because it is appropriate: I would like to thank my advisor, FH-Prof. Dr. Martin Horauer, for his excellent guidance and for giving me the opportunity to join one of his research projects within the Department of Embedded Systems at the University of Applied Sciences FH Technikum Wien. He allowed me a great degree of freedom in my work and kindly helped me to gain ground in academic work. Most valuable to me were his numerous tips, his pragmatic approach of doing things, and our fruitful discussions both related and unrelated to work. I highly enjoyed the time working together.

Next, I want to thank Dr. Bastian Schlich and his team from the Embedded Software Laboratory at the RWTH Aachen University. Even though we were most time geographically separated, he greatly contributed to set up a smooth and rich collaboration. He was always willing to listen to my problems and gave me plenty of support to get started with model checking and \texttt{[mc]SQUARE}.

Last – but definitely not least – I thank my family and friends. They supported me in everything I did and greatly helped me to make my way.

\textit{Thomas Reinbacher}

Vienna, June 2009
# Contents

1 Motivation and Introduction ............................................. 1

2 Contribution ........................................................................ 3
   2.1 Status Quo ................................................................. 3
   2.2 Thesis Contribution ..................................................... 3
   2.3 Long-Term Vision ....................................................... 4

3 Background ........................................................................... 5
   3.1 A World Where Nothing Works and Nobody Knows Why ........ 5
   3.2 Formal Verification ..................................................... 6
   3.3 The Verification Problem ............................................. 7
   3.4 Model Checking .......................................................... 8
      3.4.1 The Model Checking Problem ................................. 9
      3.4.2 The Kripke Structure ........................................... 9
      3.4.3 The Temporal Logic CTL ..................................... 9
      3.4.4 The Model Checking Workflow ............................... 11
      3.4.5 Coffee Vending Machine Example ........................... 12
      3.4.6 Local vs. Global Model Checking Algorithms .......... 13
      3.4.7 The Pros and Cons of Model Checking .................... 13
   3.5 Assembly Code Model Checking and [mc]square ................. 14
   3.6 C51Simulator – Intel MCS-51 Simulator Component .......... 16
      3.6.1 The Intel MCS-51 Microcontroller ........................... 16
      3.6.2 The Big Picture .................................................. 18
      3.6.3 Test and Verification of the C51Simulator Component .... 18
      3.6.4 The Software Architecture of the C51Simulator ........ 19
   3.7 Related Work ............................................................ 21
      3.7.1 The Assembly Code Model Checking Approach .......... 21
      3.7.2 3-valued Abstraction Techniques ............................. 22
      3.7.3 Static Analysis ................................................... 23
      3.7.4 Simulators for [mc]SQUARE .................................... 23

4 Abstraction Techniques .......................................................... 25
   4.1 Abstraction in Model Checking ...................................... 25
      4.1.1 Reducing System Complexity through Abstraction ........ 25
      4.1.2 Turing’s Halting Problem and Why Model Checking Works Anyway 26
      4.1.3 Nondeterministic Behavior in Assembly Code Model Checking .... 28
   4.2 Implementation – Abstraction Techniques for the C51Simulator .... 30
      4.2.1 Delayed Nondeterminism ....................................... 30
      4.2.2 Delayed Nondeterminism with Look Ahead .................. 31
      4.2.3 Nondeterministic Program Status Word ..................... 38
1 Motivation and Introduction

It is fair to state, that in this digital era correct systems for information processing are more valuable than gold.

(Henk Barendregt)

Embedded Systems are becoming ubiquitous. Most existing intelligent computer systems do not even have a screen or input devices. They are embedded and therefore hidden in various kinds of objects: automobiles, elevators, aircrafts, medical devices, industrial robots etc. The demand of efficiency and flexibility in information processing, leads to a movement from manual, mechanical, and hydraulic systems towards highly integrated embedded solutions.

Each day, we are putting an increasing trust in these software and hardware systems. Software is the main enabler for innovative features and new application areas and most times the elaborate part of the system. However, a fact that is often overseen is the natural imperfection of the design team involved in the software implementation process. The ever increasing system complexity is another contributor to the vulnerability of state of the art embedded systems [1].

The assembly code that was written for the first moon landing in 1969 was the estimated equivalent of 7500 lines of C code. The code had to fit into the few kByte of program memory featured by the mission computer [2]. Nowadays embedded solutions can scale up easily to an amount of a few million lines of code. A lot of things may have changed since then, but one decisive question remains: How to guarantee and prove that the software is working correctly, without any flaws?

Formal verification methods such as model checking, theorem proving, and abstract interpretation have gained some momentum in verifying those systems. Indeed, almost all notable software companies [3, 4, 5] have developed and deployed model checking tools to ensure design correctness. In 2008, the achievements of model checking were greatly honored when the Association for Computing Machinery (ACM) awarded the prestigious Turing Award – the Nobel Prize in computer science – to the pioneers in this field: Edmund Clarke, Allen Emerson, and Joseph Sifakis.

As of today, model based software development and formal verification is well established in most of today’s software engineering processes. Nevertheless, formal verification has played a minor role in the context of embedded systems in the past. The reasons are manifold, e.g., past model checking tools were only capable of handling small designs with a few hundred lines of machine code and generating the required behavioral models is most times tedious, challenging, and error-prone. This is especially true for the area of embedded systems. Software written for embedded systems is always linked to a certain application and a target hardware platform. Microcontroller specific programming language extensions are used to access particular hardware features that cannot be enabled by high level
programming language syntax. Formal verification of the high level application code is often not sufficient to master the verification challenges of high-reliable and safety critical applications. Target platform peculiarities make formal verification of existing embedded software a tough job.

Recently, model checking of assembly code became the focus of research projects [6, 7, 8]. It has some remarkable advantages compared to model checking programs written in high level programming languages. The code that is deployed to the hardware is checked and not just an intermediate representation, thus, any errors introduced during the development process can be found (e.g., compiler errors, toolchain errors, wrong periphery setup, and errors not visible in the C code at all).

First tools, such as [mc]SQUARE (Model Checking for Micro Controllers) [9] from the Technical University of Aachen emerged and proved their feasibility in research and academia. Although this approach seems promising to formally verify embedded software, further abstraction techniques are needed to mitigate the prevalent state-explosion problem.
2 Contribution

2.1 Status Quo

In 2004, the RWTH Aachen University started off research incentives towards a model checker for microcontroller assembly code. A first architecture was proposed in [6], and a tool named [mc]SQUARE was developed. While early versions of the tool focused exclusively on model checking, static source code analysis gradually took over a major part in [mc]SQUARE. The initial target microcontroller supported was the ATMEL ATmega family. In 2007, the Department of Embedded Systems of the University of Applied Sciences Wien established a research cooperation [10] with the RWTH Aachen University. Henceforth, the Department of Embedded Systems was actively involved in assembly code model checking research as well as in the further development of [mc]SQUARE. One of the first tasks was to extend [mc]SQUARE by an Intel MCS-51 simulator component, thus, allowing a wider area of application for the toolchain. Consequently, the Intel MCS-51 simulator integration brought along significant know-how for microcontroller families that might be included in future versions of [mc]SQUARE. First research results were presented to the scientific community in the paper Challenges in Embedded Model Checking – a Simulator for the [mc]SQUARE Model Checker [11] presented at the Symposium on Industrial Embedded Systems (SIES) 2008. More details and a first example code verified by [mc]SQUARE using the Intel MCS-51 simulator were published in [12].

2.2 Thesis Contribution

The main contributions of the present master thesis are (i) the further development of the C51Simulator component as well as (ii) matured abstraction techniques for state space reduction. Furthermore, the C51Simulator component is (iii) integrated into the existing static analysis framework where the main focus lies on mastering architectural features of the Intel MCS-51 target. Finally, the feasibility of the [mc]SQUARE approach to assembly code model checking is demonstrated by (iv) formally verifying a real life industry application. The application source code is provided by an external company, which uses the source code in one of their products.

The present master thesis introduces a novel and powerful abstraction technique termed Delayed Nondeterminism with Look Ahead [13] for state space reduction during model checking. Furthermore, a new data-flow analysis termed Register Bank Analysis [14] is presented to narrow and refine static analysis results for the Intel MCS-51 target. Moreover, limits and limitations of the [mc]SQUARE approach are pointed out and possible solutions to overcome existing shortcomings are discussed.
2.3 Long-Term Vision

Assembly code model checking – as every formal verification method – aims at solving one of the biggest challenges in nowadays software development: obtaining flawless and specification compliant source code, thus, guaranteeing applications and products working seamless within state-of-the-art, safety-critical, and highly reliable applications enabling all the comforts and services in our modern society.

Thus, the further development of [MC]SQUARE towards an industrial applicable tool can be seen as a contribution to leverage the verification problem in nowadays embedded software development processes\footnote{However, it would be foolhardy to state that tools such as [MC]SQUARE will ever become the holy grail of program verification. Nevertheless, without fail, they are a step in the right direction.}. 

\footnote{However, it would be foolhardy to state that tools such as [MC]SQUARE will ever become the holy grail of program verification. Nevertheless, without fail, they are a step in the right direction.}
3 Background

If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.

(Gerald Weinberg’s Second Law)

This chapter presents (theoretical) background related to formal verification and model checking. In what follows, the need of formal verification in the embedded systems domain is motivated by examples of famous software bugs. Next, a classification of formal verification methods is given and the term verification problem is defined. Later, the foundations of model checking are presented and the temporal logic CTL is covered. Then, advantages and disadvantages of model checking are discussed. Later on, the Intel MCS-51 simulator component of the [mc]SQUARE model checker is described. The chapter concludes with a summary of related work.

3.1 A World Where Nothing Works and Nobody Knows Why

Reliability is a major concern in nowadays software engineering processes. As system complexity is continually rising, traditional testing methods fail to cope the verification challenge. Software development, even for embedded systems, has become a global task where developers of various branch offices are involved. Geographically separated engineers are producing thousands or even million lines of code and perhaps have never seen each other. Quality control of modern software production processes has become significantly difficult [15]. On the other hand, malfunction of software is costly in terms of failure of the application itself but also due to the resulting consequences, such as fatal accidents, loss of money, shutting down of vital systems, reputation loss, and repayments.

With this in mind, a few selected, famous software failures are presented and briefly discussed.

Explosion of the Ariane 5 launcher on its maiden flight (1996). The maiden flight of the Ariane 5 launcher in June 1996 failed because of a malfunction in the control software. An untreated software trap caused the self-destruction of the rocket only 37 seconds after the launch. A failed data conversion from 64 bit long floating point to 16 bit long signed integer is arguable one of the most expensive software bugs in the aerospace industry (cf. [16]).

Loss of the NASA Mars Climate Orbiter (1999). The spacecraft was intended to enter the Mars orbit at an altitude of 140-150 km above the surface. The navigation software failed, and caused the spacecraft to reach an altitude as low as 57 km. The spacecraft was destroyed by atmospheric stresses and friction at this low altitude. The
root cause for the loss of the spacecraft was the failure to use imperial units instead of metric units, leading to an erroneous trajectory computed using this incorrect data (cf. [17]).

US-Northeast blackout (2003). A massive power outage on August 14th, 2003, affected over 50 million people in northeastern USA and eastern Canada. A previously unknown software flaw in a widely-deployed energy management system contributed to the devastating scope of the blackout. The software flaw caused alarm systems to stall because of a race condition (cf. [18]).

Toyota Prius software causes stopping and stalling on highways (2005). A software bug in the Electronic Control Module (ECM) causes Toyota Prius gas-electric hybrid cars to stall or shut down while driving at highway speeds. Approximately 75,000 vehicles were affected by this software bug (cf. [19]).

Microsoft Excel multiplication bug (2007). Any multiplication evaluating to 65,535 will deliver incorrect results in early version of Microsoft Excel 2007. For instance, the multiplication of 850 by 77.1 results in 100,000 instead of the correct value of 65,535 (cf. [20]).

A1 mobile network breakdown (2008). A software problem was responsible for the breakdown of the mobile network service in October 2008, affecting nearly 500,000 customers in Lower Austria and Vienna (cf. [21]).

ÖBB train ticketing machine selling single fare tickets for 3720.8 € (2008). A single fare ticket for the domestic railway line between Hollabrunn (Lower Austria) and Handelskai (Vienna) is normally sold for 6.8 €. However, in some rare cases the fully automatic ticket machine at the platform charges the passenger 3720.8 €. That happens only in case the language is changed from German to English before the ticket buying process is initiated (speaking from the author’s own experience).

Even when strictly abiding software programming rules and design guidelines, software is man-made and, therefore, may never be perfect. The development and use of methods attempting to remove man-made errors in software engineering is crucial to pave the way for further advances in software engineering. This is the ultimate goal of formal software verification. Hence, the formal approach of software verification may be seen as a major contributor to software correctness, reliability, and safety of present and future applications.

3.2 Formal Verification

Over the past decade we have learnt that software programs and hardware designs in general – even after intensive testing efforts – are containing bugs (see Section 3.1). More than half of the development time for modern embedded designs is spent on testing and debugging in order to approach a reliable design. While the software industry is rather supporting the development of improved testing methods, computer scientists tend to find alternative approaches to close the predominant verification gap in modern designs. Numerous research endeavors propose formal verification as the answer to some verification issues. Formal verification has been one of the hot topics in computer science research for
more than four decades [22]. Figure 3.1 gives a rough classification of formal verification methods.

The main concept behind formal verification relies on the observation that computer programs can be seen as mathematical objects with well-determined behavior. Mathematical logic is used to describe the desired behavior of the computer program which is subject to verification. The process of formally verifying a program is now to give a mathematical proof to show that the program works as specified.

![Formal verification methods classification diagram]

Figure 3.1: Formal verification methods classification.

Basically, literature distinguishes two main areas of formal software verification approaches. The first one is a rather mathematical related one, called theorem proving. In theorem proving, a proof of correctness is achieved through the derivation of a theorem. A short overview of theorem proving is given in [23]. However, software verification can also be achieved without explicitly establishing mathematical proofs. The more popular approach to formal verification is called model checking and is very well received in modern-day software development processes.

### 3.3 The Verification Problem

The verification problem can be simply stated as [22]: *Given a program \( M \) and its specification \( \varphi \) determine whether or not the behavior of \( M \) meets the specification, i.e., does \( M \models \varphi \) hold?*

Alan Turing [24] formulated the problem in terms of Turing Machines. Given a Turing Machine \( T \) and a specification \( \varphi \) decide whether \( T \) will eventually halt, e.g., on a blank input tape. That leads to the halting problem which is proven to be algorithmically unsolvable. Although the halting problem is unsolvable, practical formal verification proves its strength in closing the verification gap since one usually focuses on finite state systems rather than reasoning about infinite behavior (cf. Section 4.1.2).

State of the art embedded designs are reaching an unprecedented level of complexity and
the observed shift from stand-alone to real ubiquitous, pervasive, and networked safety-critical applications calls for effective methods to formally prove the correct behavior of a design. Until now, the advances in formal verification helped to successfully verify simple programs of moderate size that are used in safety critical applications. As recently pronounced by Hoare and Misra the forthcoming challenge in the field of formal verification is seen as the process of merging the elaborated theoretical understanding of computer programs as well as existing tools in order to enable fully automatic verification of real life, large scale, and complex embedded designs.

In [25], Hoare and Misra proclaim the verification grand challenge as an international project to construct a program verifier that would use logical proof to give an automatic check of the correctness of programs submitted to it. What sounds for a moment out of touch with reality is, based on their assumptions, within the reach of the next 20 years. In their vision the verification grand challenge will lead to a tool that can be seen as the “swiss army knife” of formal verification, solving the verification challenge for future hardware and software designs. Hoare and Misra estimated more than a thousand person-years of effort to accomplish this project. To get an idea of the complexity of such a project: the Linux Kernel v2.6 – one of the world’s largest software projects – started its development back in 1991 and since then the development effort has gained an accumulated number of five thousand person-years [26].

The verification grand challenge is undoubted an ambitious and catchy project, nevertheless, if it succeeds it will revolutionize the way how we develop (safety-critical) software and it will make essential contributions to reliability, safety, and trustworthiness of future software developments. In 2002, the US Department of Commerce estimated annual costs to the US economy of about 60 billion US dollars due to avoidable software errors [27]. Thus, producing error-free software is not only safe for people using those systems it is even highly economical advantageous.

It is a long and steep way to the fully automatic, formal software verification and contributions made in order to achieve this ambitious goal come piece by piece. Hence, the work put into this thesis can be seen as a small step towards Hoare and Misra’s vision of a fully automatic software verification.

3.4 Model Checking

Model checking [28, 29] is an automatic, model-based, property verification approach with the aim to automatically verify finite state systems. The main concept behind model checking is basically a straightforward brute force exploration of the states of a given system to check whether the given system model satisfies a certain property (specification). Model checking was introduced in the early 1980’s and pioneered independently by Clarke and Emerson [30] in the US and by Quelle and Sifakis [31] in France.

Comprehensive research on (i) efficient search algorithms in order to ensure minimal effort when traversing system states and (ii) abstraction techniques to combat the state-explosion problem was a major contributor to shift model checking from research applications to industry practice. With the ever increasing available computational power it is now possible to check systems ranging close to real life industry applications.
3.4.1 The Model Checking Problem

The model checking problem is an instance of the verification problem (cf. Section 3.3). Model checking provides an automated method for verifying concurrent (nominally) finite state systems that uses an efficient and flexible graph search, to determine whether or not the ongoing behavior described by a temporal property holds of the system’s state graph. The method is algorithmic and often efficient because the system is finite state, despite reasoning about infinite behavior [32].

3.4.2 The Kripke Structure

In model checking, finite (nondeterministic) state machines are used to represent the behavior of the system. A special type of these state machines are Kripke structures. Every single state is labeled with Atomic Propositions (\(\mathcal{AP}\)) which are boolean variables and the evaluations of expressions in that state. These expressions correlate to the particular system properties, e.g., boolean expressions over variables or registers. A Kripke structure \(\mathcal{M}\) is represented as an ordered sequence of four objects:

\[
\mathcal{M} = (\mathcal{S}, s_0, \mathcal{R}, \mathcal{L})
\]

- \(\mathcal{S}\): finite set of states
- \(s_0\): initial state \(s_0 \subseteq \mathcal{S}\)
- \(\mathcal{R}\): transition relation \(\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}\)
- \(\mathcal{L}\): interpretation function \(\mathcal{L} : \rightarrow 2^{\mathcal{AP}}\)

The transition relation \(\mathcal{R}\) specifies for each state whether and which successor states are possible, i.e., for each state \(s \subseteq \mathcal{S}\) there is a successor state \(s' \subseteq \mathcal{S}\). The interpretation function \(\mathcal{L}\) labels each state with the set of \(\mathcal{AP}\) that are true in that state. A path \(\pi\) in the Kripke structure \(\mathcal{M}\) from a state \(s\) is a sequence of states \(\pi = s_0s_1s_2...\) such that \(s_0 = s\) and \(\mathcal{R}(s_i, s_{i+1})\) holds for all \(i \geq 0\) [28].

3.4.3 The Temporal Logic CTL

Computational Tree Logic (CTL) is a combination of a linear temporal logic and a branching-time logic and was proposed by Clarke and Emerson in 1980 [33]. The model of time is a tree-like structure in which the future is not determined. In model checking, temporal logic is used to express the systems specification, i.e., the property \(\varphi\). In a linear temporal logic, various operators are provided to describe events along a single computation path. In contrary, a branching-time logic provides operators to quantify over a set of states that are successors of a given (the current) state. CTL combines these two kinds of operators and properties are therefore constructed from path quantifiers and temporal operators.

**CTL Path Quantifiers**

- **A** – for All paths from a certain state on
- **E** – there Exists at least one single path leaving from a certain state
CTL Temporal Operators

- **X ϕ** – ϕ holds neXt time
- **F ϕ** – ϕ holds sometime in the Future
- **G ϕ** – ϕ holds Globally in the future
- **p U ϕ** – p holds Until ϕ holds

In CTL, a temporal operator always must be preceded by a path quantifier. The suggestion of using temporal logic for reasoning about ongoing concurrent programs (reactive systems) goes back to Pnueli in 1977 [34]. This thesis focuses exclusively on CTL model checking. A survey on other temporal logics is given in [35]. A few examples of common CTL expressions are given in Figure 3.2. Another well received temporal logic is Linear Temporal Logic (LTL). Whereas CTL considers the whole computation tree, LTL does only consider individual runs of the automata. Thus, CTL allows to reason about the branching behavior, considering multiple possible runs at once. However, CTL and LTL have a large overlap, thus, a considerable number of properties are expressible in both temporal logics. Although they have a common superset, namely Computational Tree Logic* (CTL*), not all properties can be expressed in both logics. For instance, a property commonly known as *resetability* is expressed in CTL as

- **AG (EF ϕ)** – from any state there is always a path where eventually ϕ holds

and cannot be expressed in LTL. Consequently, some LTL properties such as **A (FG ϕ)** – along every path, there is some state from which ϕ will hold forever – and fairness constraints (cf. Section 6.4.5), cannot be expressed in CTL either. More on the expressiveness of CTL*, CTL, and LTL is given in [36, 29].

![Figure 3.2: CTL examples and intuitions.](image-url)
3.4.4 The Model Checking Workflow

In practice, the system model $M$ is described by a semantical model, i.e., a Kripke structure and the specification (property) $\varphi$ is described by a formula given in temporal logic.

![Model Checking Workflow Diagram](image)

Figure 3.3: The model checking workflow.

Proving a certain property is performed by determining the truth of formulas in certain system states. In order to apply model checking, one needs a modeling language in which the system is described as well as a notation for the formulation of properties and algorithms to step through the state space. As shown in Figure 3.3, a typical model checking workflow is composed of three major steps:

**Define a formal model of the system** that is subject to verification by creating a model of the system in a language that fits the model checker’s input language. Those modeling languages are usually tight coupled to the model checker itself, such as Process or Protocol Meta Language (PROMELA) used by the SPIN model checker [37]. System modeling usually involves the process of abstraction (see Section 4.1), i.e., simplifying the original system. System modeling focuses on the main properties in order to better manage the system complexity.

**Provide a particular system property** that should be proved. In other words, a question

1. $\text{AG} (\text{event U abort})$
2. $\text{EF} \text{ event} > 100$
3. $\text{EF} \text{ event} = 20$

$M |\models \varphi$?  

Notification | yes | no | Counterexample
about the system behavior is formulated that should be answered by the model checker. The system property is usually derived from the specification and given in a temporal logic.

Invoke the model checking tool and receive a notification whether the given system property was fulfilled or not. In case the system property could not be verified, a counterexample is generated to finger-point to the source of error in the system model.

3.4.5 Coffee Vending Machine Example

A simple model of a coffee vending machine is introduced to exemplify the use of Kripke structures and CTL. Its textual specification reads as follows:

- After inserting a coin, the user can choose her/his favorite coffee.
- A coffee is only brewed after a valid selection is made.
- The user is able to abort the procedure at any time.

Figure 3.4 shows the resulting Kripke structure, with all the states, transitions, and state variables. Each state is labeled with the atomic propositions \( \mathcal{AP} \) that are true or false in the state. The labels given on the transitions are not part of the Kripke structure itself. The coffee vending machine can be formally written as:

\[
\mathcal{M} = (S, s_0, R, L)
\]

\[
S := \{S_1, S_2, S_3\}
\]

\[
s_0 := \{S_1\}
\]

\[
R := \{(S_1, S_2), (S_2, S_1), (S_2, S_3), (S_3, S_1), (S_3, S_3)\}
\]

\[
L(s_1) = \{\neg\text{coin}, \neg\text{brew}, \neg\text{selection}\}
\]

\[
L(s_2) = \{\text{coin}, \neg\text{brew}, \neg\text{selection}\}
\]

\[
L(s_3) = \{\text{coin}, \text{brew}, \text{selection}\}
\]

As noted in Section 3.4.1, model checking is based on a graph search, therefore, the transition system is transformed to computation paths. This is done by unwinding the Kripke structure to obtain a computation tree, as shown in Figure 3.4(b).

Most model checkers expect system properties given in some temporal logic. For the coffee vending machine meaningful system properties might be:

- Coffee is brewed after a selection was made.
- Coffee is brewed sometime.

These properties can be written in CTL as:

- \( AG[\text{selection } \Rightarrow \text{brew}] \) Whenever a selection is made coffee is brewed for sure.
- \( EF[\text{brew}] \) There is a state where coffee is brewed.

1\( \Rightarrow \) represents implication (first order logic).
3.4 Model Checking

3.4.6 Local vs. Global Model Checking Algorithms

In literature, there are two different approaches of exploring the state space of a given system, i.e., local and global model checking algorithms. A global model checking algorithm first builds the whole state space. Search and labeling algorithms are applied afterwards to find particular states in which the system property cannot be proven. In global model checking the state space is traversed backwards to find counterexamples. As the whole state space is available, a global model checking algorithm is able to present all possible counterexamples. It may compare the length of the counterexamples and only present the shortest to the user. A major drawback of global model checking is the generation of states that are not relevant to prove the given formula, thus, making the state-explosion problem even worse.

Consequently, in local or on-the-fly model checking only states are visited that are needed to prove the truth value of the formula in a given state. Hence, on-the-fly state space building is possible when using local model checking algorithms. It is obvious that a local model checking algorithm can hardly find the shortest counterexample. Nevertheless, local model checking is a first step to alleviate the state-explosion problem.

[mc]SQUARE implements a local model checking algorithm as described by Heljanko in [38]. A comparison of local and global model checking is elaborated in [39].

3.4.7 The Pros and Cons of Model Checking

Compared to traditional approaches, such as simulation and testing, model checking offers two major advantages:

- Model checking is a fully automatic approach. It does neither require user guidance nor does it claim for user expertise in the fields of mathematics, logic, or theorem proving. Anyone who uses design and simulation tools is able to apply model checking, since modern tools aim to offer a push-button solution. Model checkers are integrated within existing design tool chains.

- Counterexample generation. Whenever the model checker reveals that a given property failed to hold, the process of model checking allows to produce a counterexample/witness. A counterexample finger-points the user to the root cause of the problem, by demonstrating a behavior that falsifies the property. For the process of
debugging, such an error trace is profoundly advantageous, since the counterexample
gives a complete insight into the system’s behavior.

Nevertheless, all that glitters is not gold. The broad application of model checking in
industry is taking place quite slowly, mainly because of its three major disadvantages:

- The state-explosion problem. The main challenge in model checking is to cope the
  problem of state-explosion. In general, a model checker aims to enumerate and ana-
  lyze the set of states a system may ever reach. The overall number of system states,
  even when dealing with small systems, is often too large to be handled with reasonable
  computing resources. Peled [40] summarizes effective strategies for fighting against
  state-explosion and proposes a combination of Binary Decision Diagrams (BDD)\(^2\),
  Partial Order Reduction (POR), and Symmetry. More details are also given by
  Clarke et al. in [28].

- Reported errors may be false negatives [40]. Model checking requires, as the name
  implies, modeling of the system. In order to alleviate the state-explosion problem,
  abstraction is needed (cf. Section 4.1). Thus, the program that is verified may not
  be the original one and consequently, if model checking reports a property violation
  in the abstracted model of the system, one has to make sure that the error is indeed
  a real one, i.e., it can be reconstructed on the real target platform. The process of
  checking the counterexample on the real system is often carried out manually. False
  negatives arise from the differences between an actual system’s behavior and the
  behavior represented by the abstracted model. Manually ruling out false negatives is
  time intensive and an error prone task itself. Therefore, a major future challenge for
  the model checking community may be the automated elimination of false negatives.
  A more detailed discussion on how to overcome the problem of false negatives is
  carried out in Section 7.1.

- Model checking can only verify a given specification. Thus, an important point is the
  completeness of the specification. It is challenging to make sure that the specification
  covers all properties that the system should satisfy and to establish a one to one match
  of a given textual specification and the derived formal specification.

3.5 Assembly Code Model Checking and [mc]square

An important point in embedded software verification is the mismatch between what gets
verified during system verification and the actual version of the application running on
the embedded target processor. In other words, there may be a mismatch between the
system model and the actual version which is deployed in the field. As widely known and
discussed in [41], embedded processors do not execute the high level representation of the
software application directly, e.g., C or C++ code, they can only execute mnemonics that
are part of the instruction set.

Especially for the embedded systems domain, custom-designed microcontrollers are in
use. Most of the specific microcontroller features cannot be directly invoked through
the high level programming language. Therefore, compiler and toolchain provider extend

---
\(^2\)A BDD or a Propositional Directed Acyclic Graph (PDAG) is a data structure that is used to represent
a boolean function. It can be seen as a compressed representation of sets.
standardized programming languages with so called microcontroller specific extensions. These additional language features allow the engineer to enable/disable interrupts, read and write data to/from peripheral units, use special data types, invoke additional hardware blocks, etc.

Not surprisingly, model checking of high level descriptions often fails to meet the needs to verify embedded systems code. Fortunately, model checking and static analysis of assembly code gained the attention of recent research projects [42, 7, 8].

Formal verification based on assembly code has some tremendous advantages over model checking of high level system models. The code that is deployed to the hardware is verified and not just an intermediate representation. A compiler, which is a highly complex piece of software itself, translates the high level code to microcontroller instructions.

In most approaches to embedded code verification, a high level behavior of the system is analyzed, but there is a lack of a cross-check verifying whether the behavior of the model remains unchanged after code compilation. Thus, when using model checking of assembly code one can detect any errors introduced during the whole development process, including compiler errors, toolchain errors, wrong periphery setup, errors not visible in the C code at all, etc.

\[
\begin{align*}
0101010001001000 \\
0100111101001101 \\
0100000101010011
\end{align*}
\]

Figure 3.5: The model checking workflow of the [MC]SQUARE approach (cf. Figure 3.3).

With [MC]SQUARE (Model Checking for Micro Controllers), the Department of Computer Science XI of the Technical University of Aachen developed a model checker that is precisely tailored for formal verification in the context of microcontrollers. [MC]SQUARE is an explicit, timeless, CTL based, assembly code model checker and features model check-
ing and static source code analysis of software written for embedded targets. Supported
target platforms are the ATMELE ATMega series [9], the Intel MCS-51 [11], the Infineon
XC16x [43], and Programmable Logic Controllers (PLCs) [44].

The [MC]SQUARE model checker uses an accurate and customized Central Processing
Unit (CPU) simulator to automatically derive the system model out of an implementation. Thus, the manual and often error-prone process of model creation can be shifted from the
test engineer towards the implementation of the verification tool. This leads to the revised
model checking workflow as shown in Figure 3.5.

In the following, a high level introduction to the C51Simulator component of
[MC]SQUARE is given and only those parts of the model checker are discussed that are
relevant for the elaboration of this thesis. More details on assembly code model checking
and the tool [MC]SQUARE are given by Schlich in [9].

3.6 C51Simulator – Intel MCS-51 Simulator Component

[MC]SQUARE uses a customized microcontroller simulator component for state space build-
ing. Hence, in order to support new target platforms a microcontroller simulator has to
be created. The following section describes this process for the Intel MCS-51 simulator
component. More details on the actual implementation can be found in [45, 11].

3.6.1 The Intel MCS-51 Microcontroller

The Intel MCS-51 success story started back in 1980, when Intel started to ship its brand
new microcontroller family, widely known as 8051, which later on became one of the
most popular and successful microcontrollers ever. Nowadays, almost every well-known
Integrated Circuit (IC) manufacturer\(^3\) has the Intel MCS-51 in its product portfolio or
they even made their own instruction set compatible derivatives.

Moreover, open-source synthesizable Intel MCS-51 Intellectual Property (IP) cores are
available in Register Transfer Level (RTL) code such as Very High Speed Integrated Cir-
cuit Hardware Description Language (VHDL) or Verilog, ready to be used within Field
Programmable Gate Array (FPGA) and Application Specific Integrated Circuit (ASIC)
based designs. The original Intel MCS-51 design directly influenced a remarkable number
of recent microcontroller architectures.

Basically, it is an 8 bit Complex Instruction Set Computer (CISC) microcontroller orga-
nized as Harvard Architecture. Code and data memory are strictly separated and instruc-
tions differ in their length.

Main Features [46, 47]:

- 128 bytes of Internal Random Access Memory (IRAM)
- 4096 bytes of internal Program Read Only Memory (ROM)
- 32 byte of bitaddressable memory block
- Four 8 bit wide general purpose I/O ports

\(^3\) An estimated number of over fifty companies worldwide.
- Two 16 bit timer units
- Full-Duplex Universal Asynchronous Receiver Transmitter (UART)
- Five different interrupt sources and two levels of interrupt priorities
- 256 different instructions
- Five different addressing modes
- The majority of instructions are executed within 12 system clock cycles

Registers as well as I/O ports are memory mapped, therefore, accessed like any other memory location. The stack is located within the IRAM area and grows to higher data memory addresses. A particular and powerful architecture feature is the bit-manipulating capability of the CPU. Single bits can be set, cleared, or involved in other logical calculations. Four separate register banks are located at the bottom of the IRAM occupying the first 32 bytes of data memory. Register banks are altered by modifying two dedicated register bank selection bits within the Program Status Word (PSW). 21 Special Function Registers (SFRs) allow the configuration of peripherals. A few of them are bitaddressable, some are only byteaddressable and some can be accessed in either mode.

**Instruction Set**

The instruction set covers 256 different instructions, hence, resulting in 8 bit wide opcodes. Caused by the CISC architecture, instructions are either one, two, or three byte long. They can be separated into five groups: logical, arithmetic, program branching, data transfer, and boolean operations.

**Supported Addressing Modes**

Data and program memory are accessed by one of the five available addressing modes:

**Immediate addressing** is used whenever the source operand is a constant value rather than a variable. The constant value can be either included as a single byte into the instruction, or be derived from the opcode itself.

**Direct addressing** is used for accessing any IRAM location including SFRs.

**Indirect addressing** uses the registers R0 or R1 from the active register bank as base registers. The value stored into these registers indicates an address in IRAM where data should be read from or written to. Any pointer makes use of indirect addressing.

**Extended direct addressing** is basically the same as direct addressing but it is rather used to access additional external memory locations than IRAM locations.

**Indirect from program memory** enables reading from program memory.

The interested reader is referred to the Intel MCS-51 datasheet [46] for more details on the architecture and the instruction core.
3.6.2 The Big Picture

[mc]SQUARE uses a well-defined and slim interface to communicate and control the C51Simulator. The main task of the C51Simulator is to generate possible successor states for a given Program Counter (PC) location. In order to do so, the C51Simulator has to model and implement the whole instruction set, data memory management as well as peripheral units of the real target microcontroller. However, a few requirements (cf. [11]) for the simulator forbid the use of existing CPU simulators. [mc]SQUARE abstracts from time, hence, the use of an existing and off-the-shelf CPU simulator is not suitable for the [mc]SQUARE approach to assembly code model checking. Almost all Commercial Off The Shelf (COTS) microcontroller simulator engines are based on a cycle-accurate approach. Thus, without applying further modifications it is nearly infeasible to use conventional cycle-accurate simulator engines to build [mc]SQUARE conform state spaces. Moreover, some abstraction techniques are applied on-the-fly, i.e., during the state space generation, requiring extra behavior not found in standard CPU simulators.

As shown in Figure 3.6 the [mc]SQUARE framework provides a full CTL model checker, a counterexample generator, a comfortable Graphical User Interface (GUI), and an assembly code static analyzer. Whenever [mc]SQUARE needs data generated by the hardware the respective simulator component is invoked. A nice side effect of the simulator based approach is a full CPU simulator, allowing the user to analyze and debug the code prior to model checking. It is notable that [mc]SQUARE offers a new way of analyzing microcontroller programs, which is quite different to standard COTS tools, since the simulation covers the whole state space of the application.

3.6.3 Test and Verification of the C51Simulator Component

A major point of criticism on tool-based model checking, is the justifiable question regarding the verification of the tool itself. How to make sure that the tool doesn’t contain software bugs by itself, leading to false outputs during the model checking process? Hence, special
care must be taken at verifying the implementation of the simulator component. It is achieved by verifying the actual implementation against commercial available Intel MCS-51 simulators such as the Keil μVision debugger or μCSim which is included in the Small Device C Compiler (SDCC) [48] toolchain. The conceptional test approach is shown in Figure 3.7. A test pattern file is loaded into both simulators and each instruction is independently executed by the two simulators. After the execution, the whole memory area of both simulators is dumped into separated files and these files are compared against each other. More on the applied test and verification strategy is given in [45].

![Figure 3.7: C51Simulator verification process.](image)

### 3.6.4 The Software Architecture of the C51Simulator

The core task of the C51Simulator is, of course, the emulation of the Intel MCS-51 instruction set and peripheral units. At this particular point the simulator behaves as any other CPU simulator. Instructions are fetched and decoded from the program memory, involved memory locations are read, modified, and written back to their origin.

A simplified architectural overview is given in Figure 3.8, showing that the C51Simulator is build around five main building blocks. In the remainder of this section these building blocks are discussed in brief.
3 Background

**Instruction Set Core**

A basic, straightforward implementation of the semantics of the opcodes supported by the microcontroller as defined in the corresponding datasheet [46].

**Memory Model**

The memory model acts as a representation of the Intel MCS-51 data and program memory. As described in [49], [mc]squared uses abstraction techniques that center around the idea of a 3-valued memory representation. Such a ternary memory representation allows certain memory locations to be marked as unknown in order to avoid the creation of unneeded successor paths. For this reason, the memory model requires shadow memory to indicate whether the actual value is known.

Consequently, the simulator manages two blocks of memory. As shown in Table 3.1, every byte of memory is represented by its actual value and a second byte, serving as mask indicating whether or not a certain bit is deterministic (Those bits with value Nondeterministic (ND) are indicated by a *).

<table>
<thead>
<tr>
<th>Location</th>
<th>Binary value</th>
<th>ND-mask</th>
<th>Ternary value</th>
</tr>
</thead>
<tbody>
<tr>
<td>@ 0x0A</td>
<td>b 1110000</td>
<td>b 0001100</td>
<td>1111**00</td>
</tr>
<tr>
<td>@ 0x0B</td>
<td>b 0000111</td>
<td>b 1110000</td>
<td>****1111</td>
</tr>
<tr>
<td>@ 0x0C</td>
<td>b 10101010</td>
<td>b 01010101</td>
<td>1<em>1</em>1<em>1</em></td>
</tr>
<tr>
<td>@ 0x0D</td>
<td>b 00000000</td>
<td>b 01100110</td>
<td>0<strong>00</strong>0</td>
</tr>
<tr>
<td>@ 0x0E</td>
<td>b 00110011</td>
<td>b 00000000</td>
<td>00110011</td>
</tr>
<tr>
<td>@ 0x0F</td>
<td>b 01010101</td>
<td>b 11111111</td>
<td>********</td>
</tr>
</tbody>
</table>

Table 3.1: Memory representation in [mc]squared.

More on the benefits of this 3-valued memory representation is given in [13, 9] and in Section 4.1.3.
3.7 Related Work

Splitter

At certain points in the model checking flow it is necessary to predicate over memory location in order to prove a given specification. Thus, in the case a memory location involved in the formula is marked as ND, there must be a mechanism to strip down ND memory locations to every possible value combination resulting out of the ND. That is exactly what the Splitter is used for. The actual implementation of the Splitter can become quite tricky and complex, one of the main reasons are the various addressing modes supported by the respective target hardware. A few straightforward examples are given in Table 3.2.

<table>
<thead>
<tr>
<th>Location</th>
<th>Ternary value</th>
<th>Value combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>@ 0x0A</td>
<td>1111**00</td>
<td>$2^2 = 4$</td>
</tr>
<tr>
<td>@ 0x0B</td>
<td>****1111</td>
<td>$2^4 = 16$</td>
</tr>
<tr>
<td>@ 0x0F</td>
<td>********</td>
<td>$2^8 = 256$</td>
</tr>
</tbody>
</table>

Table 3.2: ND memory representations and resulting value combinations.

Determinizer

The Determinizer is, in principle, the decision making part acting whenever the C51Simulator has to resolve nondeterministic behavior. For any given state in the state space it is capable of generating all possible successor states. Further on, the Determinizer takes over the proper handling of interrupts and branches to Interrupt Service Routines (ISRs).

Interface

A slim interface connects the C51Simulator to the [mc]square model checker as well as to the GUI.

3.7 Related Work

There has been extensive research into the topic of formal verification in the past. This section divides the existing related work into four main areas.

3.7.1 The Assembly Code Model Checking Approach

Several model checkers such as SLAM [50], BLAST [51], MAGIC [52], MOPS [53], OPEN/CAESAR [54], or SOCKETMC [55] work on C code. These tools, however, are not applicable to embedded systems because of the special nature of programs targeted for microcontrollers [56]. A model checker for embedded systems has to support special features, for instance, direct memory access, interrupt handling, inline assembly instructions, usage of timers, or communication interfaces. Hence, model checking of machine code seems mandatory when trying to automate the process of model construction.

Related model checkers that work on the machine code level are StEAM [57], MCESS [42], and Estes [8], however, only the latter two are targeting embedded systems.

21
Estes model checks assembly code for the 68HC11 microcontroller constructing the state space either with a simulator or real hardware using the GNU debugger. In practice, this approach is only feasible for small programs. Constructing the state space for the model via the hardware takes time (unless dedicated hardware support is provided). Furthermore, using an out-of-the-box simulator/debugger to construct the model, on the other hand, restricts optimizations in order to minimize the state space. MCESS, in contrast, translates the assembly code of ATMEL ATmega 16 microcontrollers into hardware-independent byte-code for a specific virtual machine that is able to check properties given in LTL. However, due to this approach most hardware issues are abstracted rather coarse eventually removing essential information that may invalidate the entire verification process.

Unlike these approaches, [mc]square constructs the model with special, tailored simulators for microcontrollers.

### 3.7.2 3-valued Abstraction Techniques

3-valued logic was initially defined by Kleene [58]. 3-valued logic is used in many research areas connected with verification. There are model checking algorithms that directly work with 3-valued logic. Bruns and Godefroid [59] describe a 3-valued CTL model checking algorithm. Another approach is described in a paper written by Yahav [60]. In this paper, 3-valued logic is used to verify safety properties of concurrent Java programs. In contrast to these approaches, the model checking algorithms used by the [mc]square approach work with Boolean logic. 3-valued logic is only utilized in the memory representation that is used by the simulator, which builds the state space. All memory locations that are accessed by the model checking algorithms use Boolean logic.

Symbolic or X-valued simulation [61] is another technique that is related to 3-valued logic. Here, symbolic values are used in place of explicit values. In our approach parts of the states used 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. In [62] a symbolic simulation scheme is used to verify embedded array systems such as memory management units of high performance microcontrollers. Symbolic Trajectory Evaluation (STE) [63] is a lattice-based model checking technology that uses a form of symbolic simulation for hardware circuit verification.

In [61], a symbolic simulator is used to verify hardware systems. Similar [64] combines a linear-time logic model checking algorithm with lightweight theorem proving in higher-order logic. Whenever an X (denoted by ND in our approach) is accessed and a value is needed, new symbolic variables are added and simulation has to be repeated. In our method a dynamic refinement is conducted. There are some approaches combining explicit and symbolic executions (cf. [65, 66]), but these approaches do explicit execution and symbolic execution in parallel.

There are also some approaches using 3-valued logic in static analysis. Reps et al. [67] describe an approach to use 3-valued logic in abstract interpretation. In another paper, Sagiv et al. [68] present a way to use 3-valued logic for shape analysis. Both analyses are special purpose analyses. In our approach, we use the 3-valued logic in a memory model utilized within model checking, which is a dynamic analysis that is more general.
3.7 Related Work

3.7.3 Static Analysis

Typical static analyzers for C are not capable of dealing with features specific to embedded hardware due to the lack of a precise hardware model. This can be integrated though, as described by Fehnker et al. [69]. In their work, a static analyzer for C/C++ code called Goanna was extended to detect misuse of hardware features of the ATMEGA ATmega16.

Regehr and Reid [70] describe a system specifically suited for embedded software, which automatically generates abstractions using the specification of the microcontroller. An approach to automatic generation of transfer functions for data-flow analyses is described by Regehr and Duongsaa [71]. Their approach is to automatically derive abstractions and transfer functions from a specification, while our approach involves modeling such abstractions by hand. An earlier approach by Bergeron et al. [72] transforms the assembly code into a higher-level representation, on which static analysis is performed, but they do not consider interrupts, which makes this approach unsuitable for interrupt-driven software frequently found in embedded systems. Brylow et al. [73] describe static analysis for interrupt-driven software, but their approach supports only immediate values written into status registers. In practice, values written into status register are often stored and manipulated in registers.

Martin et al. [74] have described a loop analysis algorithm for cache prediction. In this approach, loop bodies are transformed into separate functions and interprocedural analysis algorithms are applied to perform a precise analysis of loops, which is similar to a context-sensitive analysis. A stack analysis using a context-sensitive abstract interpretation is described by Regehr et al. [75]. This analysis is used for a worst-case prediction of stack sizes. While interrupts are considered, recursion is unrolled only until a fixed bound specified by the user. A thorough description of challenges during static analysis of microcontroller assembly code is included. Another approach to stack analysis of x86 assembly programs is described by Linn et al. [76], which is not suitable in presence of interrupts. An intraprocedural static slicing algorithm for assembly code is described by Cifuentes and Fraboulet [77], but stack variables are not supported at all.

The occurrence of interrupts in embedded software can be seen as a restricted form of multi-threading. Numerous approaches for static analysis of concurrent programs have been developed. An approach by Lal and Reps [78] adapts static analyses for sequential programs and extends them to work in a concurrent setting. Other approaches, such as the work by Qadeer and Rehof [79] or Lal et al. [80], tackle state-explosion due to thread interleavings by imposing an upper bound on the number of context switches.

In contrast, our approach of Register Bank Analysis aims at refining assembly code static analysis for the Intel MCS-51 microcontroller by proposing a tailored analysis to cope with the architectural feature of register bank swapping.

3.7.4 Simulators for [mc]square

Other simulators for [MC]SQUARE were previously implemented by Schlich [9] (ATMega family), Schuer [43] (Infineon Xc167), and Wernerus [44] (PLC).
3 Background
4 Abstraction Techniques

All the world is an abstract interpretation (of all the world).
(David Schmidt)

In this chapter the concept of abstraction is introduced and the need of abstraction in model checking is emphasized. First, the terms over-approximation and under-approximation are explained. Next, a thought experiment is conducted, showing the exponential connection between the state space size and the amount of data memory of a microcontroller. Then, nondeterministic behavior in assembly code model checking is addressed and a 3-valued memory model is presented. Finally, three state space abstraction techniques and their actual implementation into the Intel MCS-51 simulator component are described.

4.1 Abstraction in Model Checking

Abstraction refers to the progress of obtaining a simpler version of the checked system, by reducing the number of details that need to be taken care of. Abstraction is performed in order to retain only information that is relevant for a particular purpose.

4.1.1 Reducing System Complexity through Abstraction

Abstraction is quite natural and human, e.g., the human ear is able to recognize frequencies in a narrow bandwidth only. The bandwidth typically stretches from about 16 Hz up to 20 kHz. Thus, all other frequencies are neglected, or in other words abstracted, since they are out of the relevant range. Ever since the early beginnings of model checking in the late 1970’s research teams [32] are facing a problem generally known as the state-explosion problem, describing the limitation set by available computation power and the resulting states that can be stored, examined, and verified against the user-stated claims. Abstraction or simplification of the analyzed model towards manageable versions of the analyzed system is crucial for the application of formal methods and a key concept to mitigate the state-explosion problem. Nevertheless, abstraction introduces new verification challenges among the original system and the simplified one [40]:

- Proving that the essential properties are preserved between the original system and its simpler version (Bisimulation relation\(^1\)).

- Proving the correctness of the simplified version. This task may be achievable after the simplification through model checking.

\(^1\)Bisimulation refers to a relation between state transition systems, associating systems which behave in the same way in the sense that one system simulates the other and vice-versa.
Abstraction is usually based on using additional human knowledge through manual or semiautomatic tools. Applying abstraction is challenging and usually a walk on a thin line between sound results and a miss of important properties in the abstracted model. Literature defines the terms over-approximation for system models containing more information as needed and as a counterpart the term under-approximation for system models lacking important system properties one is interested in (cf. Figure 4.1).

![Over- and under-approximation in abstraction](image_url)

**Figure 4.1:** Over- and under-approximation in abstraction [81].

### 4.1.2 Turing’s Halting Problem and Why Model Checking Works Anyway

Alan Turing first proved that there is no way of deciding once a computer has started a calculation whether that calculation will terminate. In other words, it is not decidable whether a Turing Machine [24, 40] will come to a halt given a particular program input. The problem is known as the Halting Problem for Turing Machines and was first discussed in 1936 [24].

For the field of software verification the halting problem means that it is in general not possible to write a program that automatically checks another program given as input parameter. Thus, the halting problem is the foundation for the mathematical fact that in general verification of a program is undecidable. More on limitations on what can be decided by an algorithm is defined by the theory of computability [82].

A legitimate question that now arises is, why formal program verification is still gaining tremendous attention in recent research [32] and even commercial tools are celebrating great achievements in the field of automatic program verification when Alan Turing back in 1936 already proved that all those problems are in general undecidable.

Computers that we are using today are not comparable to Turing Machines. A Turing Machine is a mathematical model, which uses a linear tape as a storage device. The tape is divided into cells and each cell is labeled by a symbol from a given alphabet. The tape has a fixed left end, and is infinite on the right. A single cell on the tape corresponds to a register in main memory within modern-day computers. Whereas, the storage device on a Turing Machine has infinite capacity (due to the infinite tape), memory is always limited in conventional computers, especially for embedded systems.

It follows, that a Turing Machine can reside in an infinite number of distinct system-states. This is not true for conventional computers. Since physical memory is always limited, the number of system states is limited to a finite number of states. Therefore, program code that runs on conventional computers can be described by a Finite State Machine (FSM). A FSM has a finite number of states and a finite number of transitions
between those states. The upper limit of possible states is defined by all possible register and memory configurations. The transitions are depending on the underlying hardware architecture. It is even possible to generate a finite state graph for all possible programs that may run on the computer. Each program would have a different entry node in the state graph. Depending on the current instruction of the program – that represents the transitions – it is possible to follow the graph in order to observe the intended behavior by the program. As one can imagine, those (complete) state graphs are huge, even though their generation is theoretically possible.

Summarized, the undecidable Halting Problem for Turing Machines is reduced for real life computer systems with limited memory to a decidable one since the focus lies on:

- model checking of finite state machines, i.e., finite state reactive systems
- propositional temporal logics to describe properties of the FSM model

Nevertheless, model checking of assembly code remains a tough job, mainly caused by the state-explosion problem. To illustrate the state-explosion problem, a thought experiment is conducted. Imagine an ordinary microcontroller, featuring a read-only program memory and a read-write data memory. Each memory location is 8 bit wide. Table 4.1 shows the relation between the number of data memory bytes and the resulting states the system may reside in. It is evident that resulting state space is exponential in the number of the data memory size.

<table>
<thead>
<tr>
<th>Data memory size</th>
<th>Resulting system states (state space size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte</td>
<td>(256^1 = 256)</td>
</tr>
<tr>
<td>2 byte</td>
<td>(256^2 = 65536)</td>
</tr>
<tr>
<td>3 byte</td>
<td>(256^3 = 16777216)</td>
</tr>
<tr>
<td>4 byte</td>
<td>(256^4 = 4294967296)</td>
</tr>
<tr>
<td>5 byte</td>
<td>(256^5 = 1099511627776)</td>
</tr>
<tr>
<td>6 byte</td>
<td>(256^6 = 281474976710656)</td>
</tr>
<tr>
<td>7 byte</td>
<td>(256^7 = 72057594037927936)</td>
</tr>
<tr>
<td>8 byte</td>
<td>(256^8 = 18446744073709551616)</td>
</tr>
<tr>
<td>16 byte</td>
<td>(256^{16} = 340282366920938463463374607431768211456)</td>
</tr>
</tbody>
</table>

Table 4.1: Data memory size and resulting system states.

In fact, for the Intel MCS-51 target, the IRAM is compiled out of 256 bytes of memory, leading to an approximate of \(256^{256}\) possible system configurations\(^2\). Under the spell of Moore’s Law – the number of transistors that can be placed inexpensively on an integrated circuit doubles every two years [83] – the number of possible system configurations increases tremendously with every new microcontroller family.

As the presented examples make clear, even for tiny systems with only a few bytes of memory the number of possible system states is tremendous, thus, claiming the use of abstraction in order to alleviate the state-explosion problem.

\(^2\)256\(^{256}\) equals to a number with 616 decimal places.
4.1.3 Nondeterministic Behavior in Assembly Code Model Checking

One of the biggest challenges in explicit (in our case assembly code) model checking is dealing with nondeterministic behavior. Nondeterminism is introduced by the environment in which the microcontroller is operating in, e.g., by unknown values of I/O ports. This uncertainty requires a dedicated treatment by the model checker. It leads to the creation of multiple successor states by instantiation of nondeterministic values with every possible value combination, which in turn means a further expansion of the overall state space.

Recapitulating the working scheme of [mc]SQUARE, a model of the particular microcontroller is responsible for state space building. In general, the microcontroller software model faces nondeterminism by either performing communication with the external environment, e.g., reading a value over the I/O lines, undertaking serial communication, or by interrupts that are likely to occur at every system state as long as the corresponding interrupt source is enabled. For the Intel MCS-51 target sources of nondeterminism are:

(i) The four I/O ports
(ii) The four timer registers
(iii) The serial communication receive register
(iv) The five interrupt flags
   • Serial interrupt flag
   • Timer 0/1 overflow flag
   • External event 0/1 flag

To make the issue of introducing nondeterministic values clear, the assembly code snippet in Listing 4.1 is investigated. This assembly code instructs the microcontroller to read a single byte from the 8 bit wide I/O ports P0 and P1 and stores the fetched values within the internal Random Access Memory (RAM) at locations 0x20 and 0x21, respectively.

```
1  MOV 0x20 , P0
2  MOV 0x21 , P1
```

Listing 4.1: Assembly code excerpt.

Following the idea of explicit state space generation reveals that the two assembler instructions shown in Listing 4.1 generate altogether $256 \times 256 = 65536$ successor states. The considerable number of successors is originated by the immediate instantiation of nondeterministic values contained in I/O ports. The two MOV instructions are stored successively in the program memory. The value of P0 is unknown, and therefore, the model checker creates 256 successor states to remove uncertainty concerning the actual value of the port. Afterwards, the second MOV instruction is executed. Each of the 256 successors then creates further 256 successors for the instantiation of P1, whose actual value is unknown too.

Environment information is not present, hence, all 65536 successors are created in order to cover all conceivable situations. Let us consider the fact of immediate successor creation from a different point of view. Suppose, that the stated claim that is subject to verification does not include statements over memory location 0x20 nor over 0x21. In this case, there is no need to create successors states. It is sufficient to find a mechanism to mark certain bit positions whose value is unknown and, thus, can be read as ND.
To that end, a 3-valued logic approach for modeling the microcontroller memory is used. Whereas binary logic is composed out of elements that are valued on the set \( \{0, 1\} \), i.e., each value obtains either true or false, 3-valued logic or ternary logic [58] is defined as follows in [84]: Ternary logic is a system \( \Delta \) whose elements called statements are valued in the set \( \{0, 1, 2\} \). If \( x \) is a statement\(^3\), the value of \( x \) can be interpreted as a mapping \( \nu : \Delta \to \{0, 1, 2\} \) such that:

\[
\nu(x) := \begin{cases} 
0; & \text{if } x \text{ is perhaps true, perhaps false} \\
1; & \text{if } x \text{ is true} \\
2; & \text{if } x \text{ is false} 
\end{cases} \quad (4.1)
\]

In the remainder of this thesis the term ND is used for the first line of the semantic representation stated in Equation 4.1.3. Ternary logic is well known in hardware description languages such as VHDL or Verilog to represent unknown values of, e.g., input circuit latches or uninitialized memory locations. Synthesis tools use this ND representation to reveal design errors, which the designer can correct before synthesis towards an actual circuit.

From the state space view, the 3-valued memory representation introduces a certain type of states, namely lazy states. A lazy state combines both explicit and symbolic parts of the state space\(^4\). Any state including memory locations marked as ND is called lazy state. Consequently, a single lazy state represents a set of explicit states. A lazy state and the corresponding nondeterministic state space representation are shown in Figure 4.2.

---

Note that in our approach a statement refers to a single bit location within the IRAM of the microcontroller.

\[^3\]mc SQUARE still uses explicit model checking algorithms.

---

Figure 4.2: Nondeterministic state space representation.
4.2 Implementation – Abstraction Techniques for the C51Simulator

As aforementioned, abstraction is the main concept to overcome the state-explosion problem. In what follows, the implemented abstraction techniques for the C51Simulator are presented. The different concepts have different effects on the achievable state space reductions as well as on the maintained expressiveness. The stronger the applied abstraction, the higher the over-approximation. The actual results strongly depend on the source code structure, the number of I/O accesses, the number of used interrupts, etc. A rough estimation, based on empirical knowledge, about the effects of the three introduced concepts are given in Table 4.2.

<table>
<thead>
<tr>
<th>Abstraction technique</th>
<th>State space reduction</th>
<th>Maintained expressiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>none</td>
<td>full</td>
</tr>
<tr>
<td>Delayed Nondeterminism</td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td>Delayed Nondeterminism with Look Ahead</td>
<td>medium</td>
<td>medium</td>
</tr>
<tr>
<td>Nondeterministic Program Status Word</td>
<td>high</td>
<td>low</td>
</tr>
</tbody>
</table>

Table 4.2: Comparison of abstraction techniques for the C51Simulator.

4.2.1 Delayed Nondeterminism

Delayed Nondeterminism was first presented in [49] and is an approach to state space reduction. As the name implies, resolving nondeterministic values by the Splitter (cf. Section 3.6.4) is postponed as long as possible. For that reason, [MC]SQUARE takes advantage of its 3-valued memory concept. Successor states are not necessarily produced when they are generated but only in case they are needed to prove a given system property or for a subsequent computation step. For example, a subsequent computation step is any conditional branch instruction, which requires the actual value of a nondeterministic memory location to solve the jump condition or to determine the target location where the branch leads to.

<table>
<thead>
<tr>
<th>Location</th>
<th>Binary value</th>
<th>ND-mask</th>
<th>Ternary value</th>
</tr>
</thead>
<tbody>
<tr>
<td>before executing MOV [0xA, 0xB]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>@ 0xA</td>
<td>b 11110000</td>
<td>b 00000001</td>
<td>1111000*</td>
</tr>
<tr>
<td>@ 0xB</td>
<td>b 00001111</td>
<td>b 11110000</td>
<td>****1111</td>
</tr>
<tr>
<td>after executing MOV [0xA, 0xB]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>@ 0xA</td>
<td>b 00001111</td>
<td>b 11110000</td>
<td>****1111</td>
</tr>
<tr>
<td>@ 0xB</td>
<td>b 00001111</td>
<td>b 11110000</td>
<td>****1111</td>
</tr>
</tbody>
</table>

Table 4.3: Memory contents before and after the MOV instruction.

To illustrate the concept of Delayed Nondeterminism, the instruction MOV [0xA, 0xB] is considered. With regard to the Delayed Nondeterminism approach, whenever the C51Simulator executes a MOV instruction, not only the value from 0xB is copied to 0xA – as one would expect when reading the defined instruction semantics in the datasheet –
it rather copies the corresponding ND-mask and the actual value. Hence, the generation of multiple successors is avoided by delaying the instantiation of the involved and perhaps nondeterministic memory location \(0xB\). This procedure is documented in Table 4.3 and illustrated in Figure 4.3.

![Figure 4.3: The Delayed Nondeterminism approach of handling the MOV [0xA, 0xB] instruction.](image)

A case study by Noll and Schlich [49] revealed the effect of Delayed Nondeterminism for different program configurations showing a possible state space reduction of 70% and above. Nevertheless, actual savings due to Delayed Nondeterminism depend on various factors, such as source code structure and the targeted hardware.

### 4.2.2 Delayed Nondeterminism with Look Ahead

Delayed Nondeterminism with Look Ahead carries the thought of Delayed Nondeterminism a bit further. First results were presented to the scientific community in [13].

Even though modern microcontrollers come along with a lot of different functionality, there is one thing they have all in common. A typical instruction set offers at least four different kinds of operations:

1. **Arithmetic operations** are used whenever the microcontroller has to perform arithmetic calculations such as \(\text{ADD}, \text{SUBB}, \text{INC}, \text{DEC}, \text{MUL}, \text{DIV}\), etc.

2. **Logical operations** are used whenever the microcontroller has to evaluate boolean equations. Typical examples are \(\text{ANL}, \text{ORL}, \text{XRL}, \text{CPL}, \text{RLC}, \text{RRC}\), etc.

3. **Data transfer operations** are utilized whenever data/program memory is copied from a given location, i.e., the source, to a destination location. Those operations are commonly referred to as \(\text{MOV}\) instructions.

4. **Program branching operations** are utilized whenever conditional or unconditional branches are needed in order to change the flow of execution when stepping through the microcontroller program. Program branching leads to a modification of the PC, thus, allowing subroutines, loops, and branches in general. Typical representatives are \(\text{SJMP}, \text{LCALL}, \text{JZ}, \text{CJNE}, \text{DJNZ}\), etc.

   Whereas Delayed Nondeterminism is only applicable for data transfer operations (3), the approach of Delayed Nondeterminism with Look Ahead focuses on logical operations (2), while still preserving all the advantages generated by Delayed Nondeterminism. Thus,
Delayed Nondeterminism with Look Ahead can be seen as an extension of Delayed Nondeterminism. Delayed Nondeterminism fails to prove its superiority when dealing with logic operations, since the straightforward approach of copying ND-masks, as described in Section 4.2.1, cannot be applied anymore.

The main idea of the Delayed Nondeterminism with Look Ahead approach to state space reduction is to take the semantic relations of the instructions into account. Delayed Nondeterminism with Look Ahead centers around the coherence among the boolean operators $\land$, $\lor$, and $\neg$ with particular regard to 3-valued logic. Relevant relations are summarized in Table 4.4.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A $\lor$ B</th>
<th>A $\land$ B</th>
<th>$\neg$ A</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>ND</td>
<td>true</td>
<td>ND</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>ND</td>
<td>ND</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>ND</td>
<td>true</td>
<td>true</td>
<td>ND</td>
<td>ND</td>
</tr>
<tr>
<td>ND</td>
<td>ND</td>
<td>ND</td>
<td>ND</td>
<td>ND</td>
</tr>
<tr>
<td>ND</td>
<td>false</td>
<td>false</td>
<td>ND</td>
<td>ND</td>
</tr>
</tbody>
</table>

Table 4.4: Truth table for 3-valued logic.

Embedded systems code is tightly coupled to the environment of the microcontroller. Analog and digital values are read from sensors, giving the application the possibility to react upon changes in the environment. Therefore, reading data over I/O ports of the microcontroller is essential for embedded applications. Since reading unknown values from the environment is one of the major contributors to the state-explosion problem, special care has to be taken to avoid the generation of needless successor states from the very beginning. Delayed Nondeterminism with Look Ahead tackles the problem right from the point where data is read from the I/O ports.

Reading values over the microcontroller I/O often involves bitwise operations performed by bit masks, since ports can be either accessed byte wise only, or the application is only interested in a certain number of bits, e.g., the lower nibble of an 8 bit wide port. Bit masking, or bit twiddling is a common way to individual operations on single bits. A summary of the most common usages of bitmasks is given in Table 4.5.

Compilers translate those bit-twiddling statements from the high level language towards logic operations supported by the microcontroller’s instruction set. In the following, a simple example is presented to explain the idea of Delayed Nondeterminism with Look Ahead.

Example

The C code in Listing 4.2 represents typical (low level) embedded code.

In what follows, this code excerpt is used to discuss the concept of Delayed Nondeterminism with Look Ahead. The code reads the value of the 8 bit wide I/O port, termed $Port1$, and uses a bitmask to extract the upper two bits out of the I/O port.
### 4.2 Implementation – Abstraction Techniques for the C51Simulator

<table>
<thead>
<tr>
<th>Operation</th>
<th>C code syntax</th>
<th>Operand</th>
<th>Mask</th>
<th>Op</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting bits to 0</td>
<td>y &amp;= ~(1 &lt;&lt; pos);</td>
<td>01101110</td>
<td>11110111</td>
<td>∨</td>
<td>01101110</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11101110</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01101110</td>
</tr>
<tr>
<td>Setting bits to 1</td>
<td>y</td>
<td>= (1 &lt;&lt; pos);</td>
<td>10010101</td>
<td>00001000</td>
<td>∨</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00001000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10011101</td>
</tr>
<tr>
<td>Toggling a bit</td>
<td>y ^= (1 &lt;&lt; pos);</td>
<td>10011101</td>
<td>00001000</td>
<td>⊕</td>
<td>10011101</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00001000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10010101</td>
</tr>
<tr>
<td>Testing a bit</td>
<td>y = x &amp; (1 &lt;&lt; pos);</td>
<td>00011101</td>
<td>00001000</td>
<td>∧</td>
<td>00011101</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00001000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00001000</td>
</tr>
<tr>
<td>Extract low nibble</td>
<td>y = x &amp; 0x0F;</td>
<td>10011101</td>
<td>00001111</td>
<td>∧</td>
<td>10011101</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00001111</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00001101</td>
</tr>
<tr>
<td>Extract high nibble</td>
<td>y = x &amp; 0xF0;</td>
<td>10011101</td>
<td>11110000</td>
<td>∧</td>
<td>10011101</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11110000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10010000</td>
</tr>
</tbody>
</table>

Table 4.5: How bitmasks are used in embedded software.

```c
1  unsigned char readValueFromIO (void) {
2      /* read port value */
3          value = Port1;
4      /** mask the upper two bits */
5          value &= 0xC0;
6          return value;
7  }
8
9  int main(void) {
10     while(1) {
11         readValueFromIO();
12         /** do something */
13     }
14  }
```

Listing 4.2: Embedded C code example program for the Intel MCS-51 target.

The source code line 5 in Listing 4.2 is now mapped by the compiler (Keil C51 Compiler V8.01) to the opcode #0x53 representing the instruction ANL direct,#immediate. The ANL instruction compares the bits of the internal memory location (0x12) with the immediate value (#0xC0) and sets the corresponding bit in the resulting byte only if the particular bit is set in both of the operands, otherwise the resulting bits are cleared.

```assembly
1  MOV  value(0x12), Port1(0x90)
2  ANL  value(0x12), #0xC0
```

Listing 4.3: Translated assembly code for source code lines 4-5 of Listing 4.2.
In the following, the effect of the assembly code in Listing 4.3 on the abstraction techniques Delayed Nondeterminism and Delayed Nondeterminism with Look Ahead are discussed and compared.

Delayed Nondeterminism helps to avoid generating successor states for the initial MOV instruction by simply copying the actual value as well as the ND-mask from memory location 0x90 to memory location 0x12 (cf. Figure 4.3). Reading from the environment leads always to a full-nondeterministic read, since environment information is not present. However, Delayed Nondeterminism forces us to determinize (creating all possible successors) involved memory locations in preparation for the following ANL instruction. The variable value is unknown, thus, all 8 bits are marked as ND and the model checker invokes the simulator to generate all possible successors arising from this uncertainty. The number of successor states is easily calculated and results in $2^8 = 256$ states. Consequently, Delayed Nondeterminism leads to a wide branch in the computation tree, having a negative impact on the state space and makes the state-explosion problem even worse. This scenario is depicted in Figure 4.4, showing the total number of 256 successor states generated.

![Figure 4.4: The state-explosion problem.](image)

However, the described approach results in a valid over-approximation (cf. Section 4.1.1) by replacing the ND value of memory location value with actual values (one at a time) and performing the ANL afterwards. Nevertheless, this approach lacks a consideration of the second operand included in the operation, i.e., the constant value of the bitmask. As shown in the example code, the bitmask is of value 0xC0. Examining the bitmask on the binary level, it evaluates to $b\ 11000000$. Thus, the only two bits of interest in this calculation are the upper two, i.e., the two most significant bits. The remaining six bits, will evaluate, according to the relations defined in Table 4.4, to false in any case.

Hence, the number of successor states can be reduced from $2^8 = 256$ down to $2^2 = 4$. The resulting values are 0x00, 0x40, 0x80, and 0xC0 as detailed in Table 4.6. Figure 4.5 presents the differences in the number of the resulting system states for the various abstraction techniques when executing the two assembler instructions of Listing 4.3.

The Delayed Nondeterminism with Look Ahead approach helps to avoid over-approximation whenever logical operations are performed over ND memory locations. How-
ever, the promising approach to state space abstraction cannot be applied to all logical
instructions of the microcontroller’s instruction set. An example is the XOR instruction.
To exemplify this on the bit level representation, neither the result of XOR \[1, \text{ND}\] nor
XOR \[0, \text{ND}\] can be decided without knowing the actual value of the ND bit. The same
applies to the negation, i.e., \(\text{NOT} \ [\text{ND}]\).

Nevertheless, considering the frequent I/O accesses and the common method of bit-
twiddling in typical embedded systems code, the presented abstraction technique can be
seen as a promising contributor to state space reduction. Regarding the C51Simulator
implementation Delayed Nondeterminism with Look Ahead is applied to 32 out of 256
instructions in total.

In [13] a saving in overall state space of 99% is achieved by Delayed Nondeterminism with
Look Ahead compared to plain explicit state space building. It should be noted, that this
result is only valid for the chosen example in [13]. Actual savings due to this method are
de pending on the source code structure and the number of accesses to nondeterministic
memory locations, i.e., for source code without any I/O accesses this concept will not
contribute to state space reduction (but won’t increase the state space either).

### Implementation

The actual implementation in the C51Simulator component uses a visitor pattern. The
visitor design pattern is a common way of separating an operation from an object structure
upon which it operates. The major benefit lies in the ability to add new operations to
existing objects without modifying those structures. More on the visitor design pattern is
found in [85].

For the C51Simulator, the whole instruction set implementation is built around a visitor
pattern. Based on the actual abstraction technique, the corresponding instruction visitor
is selected and used to apply the desired abstraction mechanism.

As an example, the Delayed Nondeterminism and Delayed Nondeterminism with Look
Ahead instruction visitors for the \texttt{ANL} [\texttt{direct}, \#immediate] instruction is presented
in the following (the notation uses pseudo Java code). The Delayed Nondeterminism
instruction visitor, shown in Listing 4.4, implements the \texttt{ANL} instruction as stated in the
instruction set manual [46]. First, the involved memory location is read from the internal
memory. Second, the logical \texttt{ANL} operation is performed and the new value is written
back to the destination register. Note that for this particular instruction, the Delayed
Nondeterminism visitor pattern is the same as for plain state space building without any

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Operation</th>
<th>Mask</th>
<th>Result</th>
<th>Combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB</td>
<td>*</td>
<td></td>
<td>1</td>
<td>*</td>
<td>(i) 0x00 ≡ b 00000000</td>
</tr>
<tr>
<td>6</td>
<td>*</td>
<td>∧</td>
<td>1</td>
<td>0</td>
<td>(ii) 0x40 ≡ b 01000000</td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td></td>
<td>0</td>
<td>0</td>
<td>(iii) 0x80 ≡ b 10000000</td>
</tr>
<tr>
<td>4</td>
<td>*</td>
<td></td>
<td>0</td>
<td>0</td>
<td>(iv) 0xC0 ≡ b 11000000</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>LSB</td>
<td>*</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6: Details on the Delayed Nondeterminism with Look Ahead approach.
Abstraction Techniques

Figure 4.5: Successor state generation and resulting system states with options: instantiate immediately, delayed nondeterminism, and delayed nondeterminism with look ahead for the assembly code presented in Listing 4.3.

(a) Instantiate immediately

(b) Delayed Nondeterminism

(c) Delayed Nondeterminism with Look Ahead

Resulting states = 513 (256 + 256 + 1)

Resulting states = 6 (4 + 1 + 1)

Resulting states = 258 (256 + 1 + 1)

Resulting states = 6 (4 + 1 + 1)

Resulting states = 258 (256 + 1 + 1)
4.2 Implementation – Abstraction Techniques for the C51Simulator

abstractions applied. Recall that Delayed Nondeterminism is only applied on data transfer instructions.

```java
public void visit(ANL_Direct_Const instruction) {
    int tmp2 = 0x00;
    //∗∗ Read direct address byte from memory ∗∗
    tmp2 = mcu.readRegisterByAddress(instruction.address);
    //∗∗ Perform AND and write back ∗∗
    mcu.writeRegisterByAddress(instruction.address, instruction.constant & tmp2);
}
```

Listing 4.4: The Delayed Nondeterminism visitor pattern for the ANL [direct, #immediate] instruction.

Consequently, the visitor pattern used by Delayed Nondeterminism with Look Ahead works in a different way, since it takes care about the relations defined in Table 4.4. It works as follows. First, if the involved memory location is deterministic, i.e., none of the bits is masked as ND, the algorithm calls the standard visitor pattern, as introduced in Listing 4.4 and returns. Second, the algorithm iterates over all bits of the involved memory location and extracts the bit values as well as the corresponding ND mask values. Since the ANL [direct, #immediate] instruction involves a constant immediate value, the ND mask of the constant value is always 0x00 (false). Consequently, the gathered information is evaluated according to Table 4.4, and written back to a temporal register, termed resultReg. This procedure continues until all 8 bits of the operand are handled. Finally, the ND mask and the actual value of the resultReg are written to the destination register (cf. lines 24-25 in Listing 4.5).

```java
public void visit(ANL_Direct_Const instruction) {
    if (mcu.isAddressDeterministic(instruction.address)) {
        super.visit(instruction);
        return;
    }
    C51Register resultReg = new C51Register("", 0);
    boolean bitA, tbdA, bitB, tbdB;
    for (int i = 0; i < C51Utilities.STD_REG_LENGTH; i++) {
        bitA = mcu.getRegisterByAddress(instruction.address).bitGet(i);
        tbdA = mcu.getRegisterByAddress(instruction.address).bitGetTBD(i);
        bitB = C51Utilities.extractBitFromByte(instruction.constant, i);
        tbdB = false;
        if ((bitA == false & tbdA == false & tbdB == true) ||
            (bitA == false & tbdA == true & bitB == false & tbdB == false)) {
            resultReg.bitSetTo(i, false);
        } else {
            resultReg.bitSetTBD(i, true);
        }
    }
    mcu.getRegisterByAddress(instruction.address).setValueAndTBD(resultReg.getValue(),
        resultReg.getTBDMask());
}
```

Listing 4.5: The Delayed Nondeterminism with Look Ahead visitor pattern for the ANL [direct, #immediate] instruction.

Summarized, the presented approach of Delayed Nondeterminism with Look Ahead helps to avoid the generation of successor states whenever a microcontroller executes logic operations by taking advantage of the 3-valued memory representation of the [MC]SQUARE model checker.
4.2.3 Nondeterministic Program Status Word

Another abstraction technique implemented by the C51Simulator is termed Nondeterministic Program Status Word. Nondeterministic Program Status Word was first presented in [86]. This approach moves model checking with [MC]SQUARE slightly towards abstract interpretation. In abstract interpretation, a single instruction is only partially executed without performing all the included calculations as defined by the instruction set manual. Thus, the aim of abstract interpretation is to gather information about the semantics of the executed program rather than exploring the program in all its details.

In direct comparison to Delayed Nondeterminism and Delayed Nondeterminism with Look Ahead, the abstraction technique Nondeterministic Program Status Word can also be applied to arithmetic operations. In this approach, the generation of successor states is avoided by introducing massive over-approximations. Again, the concept of the 3-valued memory model is the basis for the abstraction technique. In the following, an example is presented to explain the main idea of Nondeterministic Program Status Word.

Example

The instruction `ADDC [A, R0]` (Add register to Accumulator with carry) with the opcode 0x38 is defined as follows [46]:

`ADDC` simultaneously adds the byte variable indicated, the carry flag and the Accumulator contents, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates that an overflow has occurred. `OV` is set if there is a carry-out of bit 6 but not out of bit 7, or a carry-out of bit 7 but not out of bit 6 – otherwise `OV` is cleared. When adding signed integers, `OV` indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative.

As stated in the instruction set manual, the `ADDC [A, R0]` operation affects the following flags:

- `C` the carry flag
- `OV` the overflow flag
- `AC` the auxiliary carry flag
- `P` the parity flag (is set implicitly whenever the Accumulator is written)

Furthermore, the following memory locations are involved:

- The Accumulator `A`, containing the first operand and serving as destination register after the calculation
- The working register `R0`, holding the second operand
4.2 Implementation – Abstraction Techniques for the C51Simulator

Whenever one of the operands, i.e., \( \text{A} \) or \( \text{R0} \), contains nondeterministic bits, \([mc]\text{SQUARE}\) will force the C51Simulator to create successor states by replacing nondeterminism with actual values and performing the \( \text{ADDC} \ [\text{A}, \ \text{R0}] \) operation one after another. However, the Nondeterministic Program Status Word approach avoids the generation of successor states in this case by setting involved memory locations to nondeterministic. For the \( \text{ADDC} \ [\text{A}, \ \text{R0}] \) operation, Nondeterministic Program Status Word sets the Accumulator \( \text{A} \) and the flags \( C, \ OV, \ AC, \) and \( P \) to nondeterministic. The second operand \( \text{R0} \) is not modified, since it is not actively written by the operation. This is detailed in Table 4.7.

<table>
<thead>
<tr>
<th>Location</th>
<th>Binary value before executing ( \text{ADDC} \ [\text{A}, \ \text{R0}] )</th>
<th>ND-mask</th>
<th>Ternary value after executing ( \text{ADDC} \ [\text{A}, \ \text{R0}] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulator ( \text{A} )</td>
<td>b 11100000</td>
<td>b 00000000</td>
<td>11100000</td>
</tr>
<tr>
<td>Working register ( \text{R0} )</td>
<td>b 00001111</td>
<td>b 11110000</td>
<td>****1111111</td>
</tr>
<tr>
<td>Flags ( C, OV, AC, P )</td>
<td>b 0001</td>
<td>b 0000</td>
<td>0001</td>
</tr>
</tbody>
</table>

Table 4.7: The \( \text{ADDC} \ [\text{A}, \ \text{R0}] \) example.

Thus, Nondeterministic Program Status Word avoids to create successor states even for arithmetic instructions, leading to additional savings in the state space. Nevertheless, whenever a program branching instruction such as \( \text{JC} \) (Jump if Carry flag set) is encountered and the carry flag itself is nondeterministic, two successors are generated to maintain a sound over-approximation. Even though the contribution of this particular abstraction technique to state space reduction is tremendous, additional behavior is added which might not be present when executing the program on the real target hardware (cf. Table 4.2 for a rough overview).

**Implementation**

Nondeterministic Program Status Word is again implemented using a visitor pattern. As an example, the corresponding visitor patterns for the \( \text{ADDC} \ [\text{A}, \ \text{R0}] \) operation are discussed in the following.

As shown in Listing 4.6, the Delayed Nondeterminism visitor pattern executes the instruction as specified in the manual. First, the two operands are read and the addition is performed afterwards. Then, the corresponding flags are set (cf. source code lines 25-55). Finally, the result is written back to the Accumulator. Again, for this particular instruction, the Delayed Nondeterminism visitor pattern behaves exactly like plain state space building without any abstraction at all.

```
public void visit(ADDC_A_Rn_instruction) {
    int tmp0 = 0x00;
    int tmp1 = 0x00;
    int tmp2 = 0x00;
    int tmp3 = 0x00;

    tmp1 = mcu.readAccumulator();
    tmp2 = mcu.readWorkingRegister(instruction.regNumber);
    /** If carry flag set, then add 1 to the result */
    if (mcu.psw.bitGet(C51PSW.FLAG_CY)) {
        tmp0 = 1;
    }

    mcu.writeAccumulator(tmp0);
    mcu.writeWorkingRegister(instruction.regNumber, tmp2);
    mcu.psw.bitSet(C51PSW.FLAG_CY, tmp0);
    mcu.psw.bitSet(C51PSW.FLAG OV, (tmp0 & 0x80) != 0);
    mcu.psw.bitSet(C51PSW.FLAG_AC, (tmp0 & 0x40) != 0);
    mcu.psw.bitSet(C51PSW.FLAG_P, (tmp0 & 0x20) != 0);
}
```
Abstraction Techniques

13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55

Listing 4.6: The Delayed Nondeterminism visitor pattern for the ADDC [A, R0] instruction.

Listing 4.7 shows the Nondeterministic Program Status Word visitor pattern. First, the algorithm checks if all included memory locations are deterministic. If so, the instruction is executed as specified in the instruction set manual and the algorithm returns. In case that nondeterministic memory locations are included, no calculation is performed at all. However, the visitor pattern marks the modified memory location as ND, thus, implementing the concept as described in Table 4.7.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

Listing 4.7: The Nondeterministic Program Status Word visitor pattern for the ADDC [A, R0] instruction.

Summarized, NondeterministicProgramStatusWord shifts the model checking approach of [MC]SQUARE further to the idea of abstract interpretation, i.e., not executing the instructions with all its details. The broad application of over-approximation, by marking involved memory locations as nondeterministic, helps to further shrink the resulting state
space. This is an interesting observation since the massive use of over-approximation due to
the Nondeterministic Program Status Word concept decreases the state space drastically.
This is against what one would expect without knowing the internals of the [MC]SQUARE
approach and the underlying 3-valued memory model. Nevertheless, Nondeterministic
Program Status Word introduces behavior which may not exist in reality, thus, may yield
false-negatives during model checking.
The following chapter focuses on static analysis of embedded systems assembly code. First, a brief introduction to Control Flow Graphs (CFGs) and data-flow analyses is given. Next, the [mc]SQUARE static analysis framework and relevant internals are presented. Then, the adaption and implementation of regular data-flow analysis for the Intel MCS-51 microcontroller and an algorithm for CFG building are described. Later on, a novel data-flow analysis concerning the particular architectural feature of register bank swapping is discussed in length. Finally, remaining challenges in static analysis of Intel MCS-51 assembly code are pointed out and possible approaches to overcome them are stated.

5.1 Background – Static Analysis of Embedded Systems Code

The classical use of static code analysis is to facilitate the construction of compilers generating optimal code. Compiler optimizations aim to minimize the run-time of a program, the amount of memory occupied, and – especially for the embedded systems domain – the power consumption. Similar concepts of analyzing source code without actually executing it are used by static code analysis for verification issues. However, the intention is a rather different one. In traditional static source code analysis one focuses on software inspection, checking the code against syntactical standards, and automated program analysis. Available static analysis tools are taking up the cause of automatically revealing software flaws and supporting the development team to obtain reusable, structured and easy maintainable code. For that purpose, it is often sufficient to focus on low-tech static analysis such as automatic software inspection or checking for syntactical standards. In the presented approach, high-tech static analysis such as data-flow analysis and finite-state verification of CFGs are utilized.

The role of static code analysis in [mc]SQUARE is to compute information about the source code under verification that helps the model checker to reduce the state space [9]. In the following, the underlying concepts needed for static code analysis are described.
5.1.1 Control Flow Graphs

A CFG is essential to most static code analysis. It is a representation of all paths that might be traversed through a program while it is executed. The CFG is a directed graph where the vertices represent basic blocks and edges present possible transfers of control flow from one basic block to another. For instance, transfer of control flow is induced by program branching instructions. Every CFG has two designated nodes through which all control flow enters and leaves the graph, i.e., the entry and the exit node. A basic block is a straight-line sequence of code with a single entry point and only one exit point, i.e., instructions within a basic block are executed consecutively without interruption. In our approach, a single node in the CFG correlates to a single instruction of the program memory.

The process of generating a CFG out of program code is discussed by the example code in Listing 5.1. To simplify matters, the WHILE language is used that serves as a rudimentary, imperative, Pascal-like language as defined in [81].

```
1 [y:=x ];
2 [z:=1];
3 while [y>0] do
4   [z:=z*y]
5   [y:=y-1]
6 od;
7 [y:=0];
```

Listing 5.1: Source code used for CFG building.

The source code uses three variables, a few assignments and a conditional program branch. The resulting CFG is shown in Figure 5.1. It is composed out of six vertices and eight edges.

![Figure 5.1: The resulting CFG for Listing 5.1.](image-url)

There is an edge from the entry to the first executable node of the CFG, that is, to the node coming from the first instruction of the program memory. There is an edge to the exit from any node that contains an instruction that could be the last executed instruction of
the program. If the final instruction of the program is not an unconditional jump, then the node containing the final instruction is one predecessor of the exit node. The same applies to any node that includes a jump to code that is not inside the valid program memory range. CFGs and their importance for various compiler optimizations are discussed in length in [87].

In summary, a CFG is a representation of all paths that might be traversed through a microcontroller program.

5.1.2 Data-flow Analysis

Data-flow analysis refers to a collection of techniques used to gather information about the flow of data along all possible execution paths of a program. Data-flow analysis uses the CFG in order to obtain knowledge about:

- Assignments that produced the value of a variable at a certain program point
- Variables that contain values that are no longer used in the remaining program
- Range of possible values of variables at a certain program point
- Run time values of variables and their dependencies among each other

Basically, depending on the kind of information that is in the focus of the analysis, two rudimentary data-flow concepts exist. The difference is described by the direction the analysis traverses the CFG.

**Forward data-flow analysis** propagates values forward through the CFG following the flow of control. It starts at the entry node and follows all paths until it reaches the exit node. Each node in the CFG has a transfer function, i.e., the semantics of the instruction. We denote the value of a variable prior to a node as *entry value*, and the value of the variable after the node as its *exit value*. Values flow from program points after predecessor nodes to program points before successor nodes. At joint points, values are combined using a join function.

**Backward data-flow analysis** propagates values backward through the CFG against the flow of control. It starts at the exit node and follows all paths in the reverse direction until it reaches the entry node. Each node in the CFG has a transfer function, i.e., the semantics of the instruction. We denote the value of a variable prior to a node as *exit value*, and the value of the variable after the node as its *entry value*. Values flow from program points before successor nodes to program points after predecessor nodes. At solid points, values are combined using a join function.

Data-flow analysis are the basis for classical intraprocedural analysis, such as Reaching Definition Analysis (RDA) and Live Variable Analysis (LVA). In the following, these analyses are described.

5.1.3 Forward Data-flow Analysis - RDA

The aim of RDA is to determine for each program location, which assignments may have been made and not overwritten, when program execution reaches this location along some path [81]. In other words, the *reaching definitions* for a given program location are those
assignments that may have defined the current value of variables. RDA aims at answering
the following questions [88]:

- Which definitions of variable \( x \) reach a given use of \( x \) in an expression?

- Is \( x \) used anywhere before it is defined?

A definition of a variable \( x \) is an operation that assigns, or may assign, an actual value
to \( x \). Furthermore, a definition \( R \) reaches a program location \( l \) if there is a path from
the point immediately following \( R \) to \( l \) such that the definition \( R \) is not redefined along
the path [89]. A variable is redefined between two program locations whenever there is an
assignment that defines a new value of that variable.

Considering the given example code in Listing 5.2, the definition of code line 1 reaches
line 2, but the definition made at code line 3 does not reach code line 5 since \( y \) is redefined
in assignment 4.

\[
\begin{align*}
1 & \quad [y := 3]; \\
2 & \quad [z := y + 1]; \\
3 & \quad [y := 3]; \\
4 & \quad [y := 6]; \\
5 & \quad [z := y + 1];
\end{align*}
\]

Listing 5.2: RDA example code.

The aforementioned informal statements about reaching definitions can be expressed as
data-flow equations [81, 9] (Note that \( l \) denotes the current program location and \( l' \) its
successor):

\[
\begin{align*}
\text{RD}_{\text{entry}}(l) & = \begin{cases} \\
\text{if } l \text{ is the final state,} \\
\bigcup \{ \text{RD}_{\text{exit}}(l') \mid l' \text{ successor of } l \} \text{ otherwise.} \\
\end{cases} \\
\text{RD}_{\text{exit}}(l) & = \left( \text{RD}_{\text{entry}}(l) \setminus \text{kill}_{\text{RD}}(l) \right) \cup \text{gen}_{\text{RD}}(l)
\end{align*}
\]
The presented data-flow equations use two assistant functions, i.e., $kill_{RD}(l)$ and $gen_{RD}(l)$, respectively. Whereas $gen_{RD}(l)$ represents a set of definitions created by an operation at program location $l$, the term $kill_{RD}(l)$ represents a set of definitions destroyed by an operation. $RD_{entry}(l)$ contains the set of definitions that are reaching the entry of program location $l$. The set of definitions that are reaching the exit of program location $l$ are contained in $RD_{exit}(l)$.

For the example code shown in Listing 5.3, the results of $kill_{RD}(l)$ and $gen_{RD}(l)$ as well as the results for $RD_{entry}(l)$ and $RD_{exit}(l)$ are gradually performed and listed in Table 5.1.3. The presented example is based on an example given in [81].

The statement at program location 3 ($l = 3$) simply checks whether the variable $x$ is greater than a constant value, thus, both the $kill_{RD}(3)$ and the $gen_{RD}(3)$ function do not yield any results. Most important, the evaluation of $RD_{entry}(3)$ reveals that immediately before entering location 3, variable $x$ was defined either at program location 1, denoted by $(x, 1)$ or location 5, denoted by $(x, 5)$. Location 5 is included for the case that the body of the while loop was previously executed.

The result of the RDA is an over-approximation of definitions reaching this location. That is a mapping indicating for each variable where it was possibly written the last time.

```
1  [x := 5];
2  [y := 1];
3  while [x > 1] do
4      [y := x * y]
5      [x := x - 1]
6  od;
```

Listing 5.3: RDA example code.

<table>
<thead>
<tr>
<th>$l$</th>
<th>$kill_{RD}(l)$</th>
<th>$gen_{RD}(l)$</th>
<th>$RD_{entry}(l)$</th>
<th>$RD_{exit}(l)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$(x, ?), (x, 1), (x, 5)$</td>
<td>$(x, 1)$</td>
<td>$(x, ?), (y, ?)$</td>
<td>$(y, ?), (x, 1)$</td>
</tr>
<tr>
<td>2</td>
<td>$(y, ?), (y, 2), (y, 4)$</td>
<td>$(y, 2)$</td>
<td>$(y, ?), (x, 1)$</td>
<td>$(x, 1), (y, 2)$</td>
</tr>
<tr>
<td>3</td>
<td>–</td>
<td>–</td>
<td>$(x, 1), (y, 2), (y, 4), (x, 5)$</td>
<td>$(x, 1), (y, 2), (y, 4), (x, 5)$</td>
</tr>
<tr>
<td>4</td>
<td>$(y, ?), (y, 2), (y, 4)$</td>
<td>$(y, 4)$</td>
<td>$(x, 1), (y, 2), (y, 4), (x, 5)$</td>
<td>$(x, 1), (y, 4), (x, 5)$</td>
</tr>
<tr>
<td>5</td>
<td>$(x, ?), (x, 1), (x, 5)$</td>
<td>$(x, 5)$</td>
<td>$(x, 1), (y, 4), (x, 5)$</td>
<td>$(y, 4), (x, 4)$</td>
</tr>
</tbody>
</table>

Table 5.1: Results after solving data-flow equations for source code Listing 5.3.

In the case of [MC]SQUARE, RDA is used to obtain a set of possible values of memory locations within the microcontroller simulator. These variables are used as input data for further analysis, such as the global Interrupt Flag Analysis (IFA) [9] and the Register Bank Analysis (RBA) (see Sections 5.2.6 and 5.2.8).

### 5.1.4 Backward Data-flow Analysis - LVA

The aim of LVA is to determine for each program location, which variables may be live at the exit from that location [81]. A variable is considered as live at a program location if its current value may be read during the remaining execution of the program.

In compiler optimizations, LVA is important for register allocation within basic blocks. After a value is computed in a register, and presumably used within a block, it is not
necessary to permanently store that value if it is known to be dead at the end of the
block [87]. It is important to note, that LVA is used within [MC]SQUARE with a different
intention. The results obtained from the LVA are used by the model checker to combine
single states that do only differ in the value of dead memory locations. Dead memory
locations can be reseted and states that only differ in dead memory locations can be
merged into single states. Thus, this analysis contributes to state space reduction and
helps to contain over-approximation by the model checker.

As aforementioned, information flow for LVA travels backwards through the CFG –
opposite to the flow of control – since the analysis aims to prove that the use of a variable
at program location $l$ is propagated to all points prior to $l$ along an execution path, so that
one may know at the prior point $l'$ that the variable will have its value used.

Similar to RDA, data-flow equations are used to express the LVA problem [81, 9] ($l$
denotes the current program location and $l'$ its predecessor):

\[
\begin{align*}
\mathbf{LV}_{\text{exit}}(l) &= \begin{cases} 
\mathbf{LV}_{\text{entry}}(l') & \text{if } l \text{ is the final state,} \\
\bigcup \{ \mathbf{LV}_{\text{entry}}(l') \mid l' \text{ predecessor of } l \} & \text{otherwise.}
\end{cases} \\
\mathbf{LV}_{\text{entry}}(l) &= (\mathbf{LV}_{\text{exit}}(l) \setminus \text{kill}_{\mathbf{LV}}(l)) \cup \text{gen}_{\mathbf{LV}}(l)
\end{align*}
\]

Within the LVA, $\text{kill}_{\mathbf{LV}}(l)$ represents a set of variables defined by an operation at pro-
gram location $l$ whereas $\text{gen}_{\mathbf{LV}}(l)$ represents a set of variables that are consumed by an
operation. $\mathbf{LV}_{\text{entry}}(l)$ contains the set of variables that are live at the entry of program
location $l$. The set of variables that are live at the exit of program location $l$ are contained
in $\mathbf{LV}_{\text{exit}}(l)$.

1  [x:=2];
2  [y:=4];
3  [x:=1];
4  (if [y>x] then
5  [z:=y]
6  else [z:=y*y]);
7  [x:=z]

Listing 5.4: LVA example code (cf. [81]).

Next, the functions $\text{kill}_{\mathbf{LV}}(l)$ and $\text{gen}_{\mathbf{LV}}(l)$ are evaluated for each location of the program
shown in Listing 5.4. The results are used and applied to the data-flow equation, resulting
in the following statements.
5.1 Background – Static Analysis of Embedded Systems Code

\[ \text{LV}_{\text{entry}}(1) = (\text{LV}_{\text{exit}}(1) \setminus \text{kill}_{\text{LV}}(1)) \cup \text{gen}_{\text{LV}}(1) = \text{LV}_{\text{exit}}(1) \setminus \{x\} = \{\} \]
\[ \text{LV}_{\text{entry}}(2) = (\text{LV}_{\text{exit}}(2) \setminus \text{kill}_{\text{LV}}(2)) \cup \text{gen}_{\text{LV}}(2) = \text{LV}_{\text{exit}}(2) \setminus \{y\} = \{\} \]
\[ \text{LV}_{\text{entry}}(3) = (\text{LV}_{\text{exit}}(3) \setminus \text{kill}_{\text{LV}}(3)) \cup \text{gen}_{\text{LV}}(3) = \text{LV}_{\text{exit}}(3) \setminus \{x\} = \{y\} \]
\[ \text{LV}_{\text{entry}}(4) = (\text{LV}_{\text{exit}}(4) \setminus \text{kill}_{\text{LV}}(4)) \cup \text{gen}_{\text{LV}}(4) = \text{LV}_{\text{exit}}(4) \cup \{x, y\} = \{x, y\} \]
\[ \text{LV}_{\text{entry}}(5) = (\text{LV}_{\text{exit}}(5) \setminus \text{kill}_{\text{LV}}(5)) \cup \text{gen}_{\text{LV}}(5) = (\text{LV}_{\text{exit}}(5) \setminus \{z\}) \cup \{y\} = \{y\} \]
\[ \text{LV}_{\text{entry}}(6) = (\text{LV}_{\text{exit}}(6) \setminus \text{kill}_{\text{LV}}(6)) \cup \text{gen}_{\text{LV}}(6) = (\text{LV}_{\text{exit}}(6) \setminus \{z\}) \cup \{y\} = \{y\} \]
\[ \text{LV}_{\text{entry}}(7) = (\text{LV}_{\text{exit}}(7) \setminus \text{kill}_{\text{LV}}(7)) \cup \text{gen}_{\text{LV}}(7) = \{z\} \]
\[ \text{LV}_{\text{exit}}(1) = \text{LV}_{\text{entry}}(2) = \{\} \]
\[ \text{LV}_{\text{exit}}(2) = \text{LV}_{\text{entry}}(3) = \{y\} \]
\[ \text{LV}_{\text{exit}}(3) = \text{LV}_{\text{entry}}(4) = \{x, y\} \]
\[ \text{LV}_{\text{exit}}(4) = \text{LV}_{\text{entry}}(5) \cup \text{LV}_{\text{entry}}(6) = \{y\} \]
\[ \text{LV}_{\text{exit}}(5) = \text{LV}_{\text{entry}}(7) = \{z\} \]
\[ \text{LV}_{\text{exit}}(6) = \text{LV}_{\text{entry}}(7) = \{z\} \]
\[ \text{LV}_{\text{exit}}(7) = \{\} \]

For example, the term \(\text{LV}_{\text{entry}}(4)\) corresponds to the statement \([y>x]\) found at source code line 4 (cf. Listing 5.4). \(\text{LV}_{\text{entry}}(4)\) evaluates to \(\{x, y\}\), revealing that at the entry point of that particular program location the only two variables live are \(x\) and \(y\). Furthermore, \(\text{LV}_{\text{entry}}(5)\) evaluates to \(\{y\}\) indicating that \(y\) is still alive immediately before statement \([z:=y]\).

For the chosen example, \(\text{LV}_{\text{exit}}(l)\) and \(\text{LV}_{\text{entry}}(l')\) yield the same results, arising from the fact that the presented example code – for the sake of simplicity – lacks any kind of program loops.

<table>
<thead>
<tr>
<th>(l)</th>
<th>(\text{kill}_{\text{LV}}(l))</th>
<th>(\text{gen}_{\text{LV}}(l))</th>
<th>(\text{LV}_{\text{entry}}(l))</th>
<th>(\text{LV}_{\text{exit}}(l))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(x)</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>(y)</td>
<td>–</td>
<td>–</td>
<td>(y)</td>
</tr>
<tr>
<td>3</td>
<td>(x)</td>
<td>–</td>
<td>(y)</td>
<td>(x, y)</td>
</tr>
<tr>
<td>4</td>
<td>–</td>
<td>(x, y)</td>
<td>(x, y)</td>
<td>(y)</td>
</tr>
<tr>
<td>5</td>
<td>(z)</td>
<td>(y)</td>
<td>(y)</td>
<td>(z)</td>
</tr>
<tr>
<td>6</td>
<td>(z)</td>
<td>(y)</td>
<td>(y)</td>
<td>(z)</td>
</tr>
<tr>
<td>7</td>
<td>(x)</td>
<td>(z)</td>
<td>(z)</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 5.2: Results after solving LVA data-flow equations for source code Listing 5.4.

The annotated CFG of the example is shown in Figure 5.3. In here, it becomes obvious that there is no variable marked as live in the first program location, i.e., the statement \([x:=2]\). As a result, the first assignment of value 2 to variable \(x\) is superfluous and can be neglected. The resulting and minimized CFG is shown in Figure 5.3(c). In compiler theory, such a reduced CFG would lead to a smarter code that can be generated by the compiler backend.
5 Static Analysis

![ CFG for Listing 5.4. ]

(5.1) CFG for Listing 5.4.

(a) CFG for Listing 5.4.

(b) Evaluated LVA equations.

IV
entry = \{\{z\}\}
IV
exit = \{\{y\}\}
IV
entry = \{\{z\}\}
IV
exit = \{\{z\}\}
IV
entry = \{\{y\}\}
IV
exit = \{\{z\}\}
IV
entry = \{\{y\}\}
IV
exit = \{\{z\}\}
IV
entry = \{-\}
IV
exit = \{-\}

(c) Minimized CFG.

![ Minimized CFG. ]

Figure 5.3: LVA example.
However, [MC]SQUARE does not use this data-flow information to generate any code but would now mark the memory location where variable \( x \) is saved as dead, thus, resetting the memory location to its initial value. Resetting a memory location to its initial value increases the probability of finding equal states within the generated state space. Equal states do not contribute to the expansion of the state space size, since the model checker simply adds an additional edge to the state space graph.

### 5.1.5 Solving Data-flow Equations

Having discussed the principles of data-flow based static analysis, the issue of solving those equations is still not addressed. In general, algorithms are used that result in the least fixed point of the equations. A least fixed point is the solution of the equations whose assigned values of, e.g., \( \text{RD}_{\text{exit}}(l) \)s and \( \text{RD}_{\text{entry}}(l) \)s are contained in any other solution to the equations. A set \( S' \subseteq S \) is a fixed point of a function \( \tau : \mathcal{P}(S) \rightarrow \mathcal{P}(S) \) if \( \tau(S') = S' \) [28]. In the presented approach, a fixed point iteration algorithm, similar as the one given in [87], is used for solving the RDA data-flow equations.

It works as follows. First, it is started with an estimate such that \( \text{RD}_{\text{exit}}(\text{entry}) = 0 \) for all nodes \( l \). Then, an iteration until the \( \text{RD}_{\text{exit}}(l) \)s converge starts, i.e., when there are no new results for the \( \text{RD}_{\text{exit}} \)s. Thus, a boolean value is used to track changes of \( \text{RD}_{\text{exit}} \)s for every iteration. The algorithm is sketched in Algorithm 1. Line 1 initializes data-flow values and lines 2-4 contain the loop responsible for iterating until convergence and lines 5-10 apply the data-flow equations.

**Algorithm 1**: A fixed point iterating algorithm to solve data-flow equations for the RDA problem [87].

**Input**: A CFG with \( \text{kill}_{\text{RD}}(l) \) and \( \text{gen}_{\text{RD}}(l) \) resolved for each node.

**Result**: \( \text{RD}_{\text{entry}}(l) \) and \( \text{RD}_{\text{exit}}(l) \), the set of definitions reaching the entry and exit of each node \( l \in \mathcal{G} \).

1. \( \text{RD}_{\text{exit}}(\text{entry}) = 0; \)
2. foreach Node \( l \) other than entry do
3. \hphantom{2} \( \text{RD}_{\text{exit}}(l) = 0; \)
4. end
5. while any \( \text{RD}_{\text{exit}}(l) \) changes do
6. \hphantom{5} foreach Node \( l \) other than entry do
7. \hphantom{6} \( \text{RD}_{\text{entry}}(l) = \bigcup \text{RD}_{\text{exit}}(l') \mid l' \) predecessor of \( l \);
8. \hphantom{6} \( \text{RD}_{\text{exit}}(l) = (\text{RD}_{\text{entry}}(l) \setminus \text{kill}_{\text{RD}}(l)) \cup \text{gen}_{\text{RD}}(l); \)
9. end
10. end

For the LVA a similar algorithm is used. As aforementioned, information flow travels backwards through the control flow in the CFG, thus, the LVA algorithm starts by initializing \( \text{LV}_{\text{entry}}(\text{exit}) = 0 \) and the sets \( \text{LV}_{\text{entry}} \) and \( \text{LV}_{\text{exit}} \) have their roles interchanged as shown in Algorithm 2. For more details on the theoretical background of data-flow analysis the interested reader is referred to relevant literature, such as [87, 81].
Algorithm 2: A fixed point iterating algorithm to solve data-flow equations for the LVA problem [87].

**Input**: A CFG with $kill_{LV}(l)$ and $gen_{LV}(l)$ resolved for each node.

**Result**: $LV_{entry}(l)$ and $LV_{exit}(l)$, the set of definitions reaching the entry and exit of each node $l \in CFG$.

1. $LV_{entry}(exit) = 0$;
2. foreach Node $l$ other than exit do
   3. $LV_{entry}(l) = 0$;
3. end
4. while any $LV_{entry}(l)$ changes do
   5. foreach Node $l$ other than exit do
      6. $LV_{exit}(l) = \bigcup LV_{entry}(l') \mid l'$ predecessor of $l$;
      7. $LV_{entry}(l) = (LV_{exit}(l) \setminus kill_{LV}(l)) \cup gen_{LV}(l)$;
6. end
7. end

5.2 Implementation – Static Analysis for the C51Simulator

Having discussed the theoretical foundations of static analysis, the following section focuses on relevant implementation details. The aim of the static analysis in [mc]square is to support model checking by providing information that can be statically extracted from the source code. The generated annotations are used to reduce state spaces by limiting the over-approximation during model checking. One can think of making the model checker intelligent due to the additional knowledge extracted by the static analysis.

Basically, [mc]square implements two kinds of analyses, namely (i) data-flow analysis and (ii) abstraction techniques. A rather high level sketch of the [mc]square static analysis framework is given in Figure 5.4.

![Figure 5.4: The [mc]square static analysis framework for the Intel MCS-51 target.](image)

The [mc]square static analysis framework is composed out of:

**Parser and preparation** handles the interaction with the user. It accepts a compiled and linked *.hex file and a specification given in CTL. Furthermore, it parses common debug formats in order to preserve the connections between the analyzed assembler code and the source code file, which may be written in C, C++, Java, or any other high level language able to be compiled towards assembler machine code for the Intel MCS-51 microcontroller. As aforementioned, a complete and precise CFG is the basis...
for all further data-flow analysis. Consequently, the parser & preparation component is responsible of preparing the analyses and building the CFG.

**Data-flow analyses** performs forward and backward oriented data-flow analyses, such as RDA and LVA. It uses the CFG to execute those analyses. The extracted reaching definitions are further used by the particular abstraction techniques in order to gather a better program comprehension. Further, it includes the novel RBA, a Stack Analysis (SA), and an Interrupt Flag Analysis (IFA).

**Abstraction techniques** use the information gathered by the data-flow analyses to apply state space reductions. A technique called Dead Variable Reduction (DVR) is used to mark dead memory locations\(^1\), prompting the model checker to reset certain memory locations whilst model checking. Another concept is Path Reduction (PR), which aims at combining single successor chains, e.g., of an ISR into a single state.

**Model checking** uses the additional information about the verified program in order to reduce the overall system states.

In the following, a rather conceptional description about the actual implementation of the various analyses into the [mc]square framework is given. For a more detailed insight, the interested reader is referred to the respective source code.

### 5.2.1 Overview

Currently, [mc]square is able of conducting the following static analyses:

- Control Flow Analysis (CFA) [81]
- Stack Analysis (SA) [9]
- Reaching Definition Analysis (RDA) [81]
- Interrupt Flag Analysis (IFA) [9]
- Live Variable Analysis (LVA) [81]
- Dead Variable Reduction (DVR) [81, 90]
- Path Reduction (PR) [91, 90]

The execution order is depicted in Figure 5.4. The SA is used to track dependencies between values pushed onto and popped from the stack. For instance, the PSW is frequently pushed onto the stack at the beginning of a function and then read from the stack at the end. The status of interrupt registers is extracted from the reaching definitions, which then influences the RDA in the next iteration. The RBA described in Section 5.2.6 interacts with the RDA and, in consequence, increases the precision of the RDA and the IFA.

All analyses are designed as **interprocedural analyses** due to the peculiarities of assembly code. For instance, all memory locations can be accessed globally. Data-flow analyses in [mc]square consist of the following steps:

\(^1\)Recapitulating, a dead memory location is a memory location that is not used anymore in the further progression of the input program.
(i) The static behavior of a function is determined, where the effects of function calls are ignored.

(ii) The static behavior of a called function is propagated from the return statement of a callee into the call site.

(iii) Data-flow information is propagated from a call site into a called function.

All these steps run as fixed point iterations\(^2\) to support recursive function calls. More details of this approach are described in [90, 9].

In what follows, the adaption of these existing static analysis techniques for the Intel MCS-51 target architecture is described.

### 5.2.2 Control Flow Graph Building

Static analysis starts with CFG building. Even at this early stage of analysis one has to take care about architectural peculiarities of the Intel MCS-51 target microcontroller. Since the Intel MCS-51 is a CISC based architecture, single instructions can be of different length. Some are one byte long, others are two bytes and a few are four bytes long. Thus, the program memory content cannot be divided into parts of equal length where each of these parts represent a single instruction. On the other hand, this is possible for (most) Reduced Instruction Set Computer (RISC) architectures. Consequently, the very humble approach of CFG building by linearly stepping through the program memory, may not cover all instructions executed by the CISC based target microcontroller and calls for a tailored treatment.

Interestingly, modern compilers make use of this fact. A common technique – especially with enabled “favor size” options – applied by the compiler tries to reuse equal bytes of program memory for distinct purposes. For example, the lower two bytes of a three byte instruction may equal to another instruction of the microcontroller. In order to save program memory size, the compiler may insert a \texttt{JUMP} instruction to the entry point of the lower two bytes of the three byte instruction. This is especially effective when the shared program memory bytes are used at multiple points of the program.

<table>
<thead>
<tr>
<th></th>
<th>C: 0x0800</th>
<th>78  D3</th>
<th>MOV</th>
<th>R0, #0xD3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>C: 0x0802</td>
<td>E8</td>
<td>MOV</td>
<td>A, R0</td>
</tr>
<tr>
<td>3</td>
<td>C: 0x0803</td>
<td>3520</td>
<td>ADDC</td>
<td>A, 0x20</td>
</tr>
<tr>
<td>4</td>
<td>C: 0x0805</td>
<td>6411</td>
<td>XRL</td>
<td>A, #0x11</td>
</tr>
<tr>
<td>5</td>
<td>C: 0x0807</td>
<td>F4</td>
<td>CPL</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>C: 0x0900</td>
<td>D3</td>
<td>SETB</td>
<td>C</td>
</tr>
<tr>
<td>7</td>
<td>C: 0x0902</td>
<td>E8</td>
<td>MOV</td>
<td>A, R0</td>
</tr>
<tr>
<td>8</td>
<td>C: 0x0903</td>
<td>3520</td>
<td>ADDC</td>
<td>A, 0x20</td>
</tr>
<tr>
<td>9</td>
<td>C: 0x0905</td>
<td>6411</td>
<td>XRL</td>
<td>A, #0x11</td>
</tr>
<tr>
<td>10</td>
<td>C: 0x0907</td>
<td>F4</td>
<td>CPL</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>C: 0x0800</td>
<td>78  D3</td>
<td>E8  35</td>
<td>20  64  11</td>
</tr>
<tr>
<td>12</td>
<td>C: 0x0900</td>
<td>D3  E8</td>
<td>35  20</td>
<td>64  11  F4</td>
</tr>
</tbody>
</table>

Listing 5.5: Code sharing within the program memory.

---

\(^2\)A fixed point iteration is the common approach to solve data-flow equations. Usually, the analyses are repeated until no change can be detected. Most time only the difference between iterations are concerned, in order to avoid redundant steps. The interested reader is referred to [81, 87]
To illustrate that behavior, consider Listing 5.5. In here, it is assumed that the compiler has already translated the high level code to assembly instructions. The program memory ranges 0x0800 - 0x0807 and 0x0900 - 0x0907 contain almost similar code. The only difference is that the Carry flag is set at location 0x0900 before entering the calculations starting at locations 0x0802 and 0x0902, respectively. Thus, in order to save program memory space the compiler might now combine those similar blocks, by replacing the five instructions located from 0x0900 to 0x0907 by an unconditional jump, e.g., an AJMP [0x0801], leading to a dynamic disassembly of the location 0x0801, which turns out to become the same sequence as when executing sequentially from 0x0900 to 0x0907. Thereby, the compiler can save five bytes of program memory, since the AJMP instruction itself is two bytes long.

Considering the discussed compiler optimization of sharing equal program memory bytes, it is not possible to sequentially decode the program memory, i.e., by iterating over the program memory and changing the index pointer by the instruction length. In the present case, a more elaborated approach to CFG building is needed. The implemented algorithm for building the CFG out of a given Intel MCS-51 program memory is given in Algorithm 3.

Algorithm 3: CFG building algorithm for the Intel MCS-51 target.

```
Input : A disassembled program memory content \( P \).
Result: An equivalent CFG representation for \( P \).

initialize CFG;
foreach instruction \( I \) in \( P \) do
   if \( I \) is an indirect branching statement then
      quit CFG building;
   end
   add new node \( l \) to CFG;
   label node \( l \) with detail info of \( I \);
   add new edge \( E \) from \( l \) to all its successors \( l' \);
end
while unknown target addresses exist do
   foreach edge \( E \) in CFG do
      obtain target addresses \( A \) of \( E \);
      if \( A \) does not point to an existing \( l \) in CFG then
         dynamically disassemble \( P \) at address \( A \);
         obtain new instruction \( I = P(A) \);
         if \( I \) is an indirect branching statement then
            quit CFG building;
         end
         add new node \( l \) to CFG;
         label node \( l \) with detail info of \( I \);
         add new edge \( E \) from \( l \) to all its successors \( l' \);
      end
   end
end
```

First, an object capable of storing a CFG is initialized. Then, sequential decoding of the program memory starts. For each instruction a node is added to the CFG.
Moreover, edges are added from the current node to all its successors. For example, for non-program branching instructions one edge is added (from the current program counter location \( l \) to \( l' \) representing the location with the program counter value of \( l + l_{\text{length}} \)). If an indirect branching statement is detected whilst decoding, CFG building is stopped, since one cannot guarantee anymore that the resulting CFG will be complete. Then, the iteration starts over as long as new branching targets are found that do not point to an existing node in the CFG. Later on, the target addresses are extracted and the program memory is re-decoded at this location. Finally, the dynamic decoded instructions are added to the CFG. The loop is left when all target addresses are resolved and map to a node in the CFG, thus, ensuring that the resulting CFG is complete.

### 5.2.3 Action List Building

The presented algorithms for solving data-flow equations expect a CFG as input with the statements \( \text{kill}(l) \) and \( \text{gen}(l) \) resolved for each program location \( l \). Thus, the very first action in the analysis is to evaluate the corresponding \( \text{kill}(l) \) and \( \text{gen}(l) \) statements for each node (instruction) in the CFG. This process is termed Action List Building. Later on, this information is used by all following data-flow analyses. From the implementation point of view, a visitor pattern is used with a visitor for each instruction defined in the instruction set manual [47] of the Intel MCS-51 microcontroller.

For an instruction at program location \( l \) we define:

- \( \text{kill}(l) \) involves any piece of memory written by the instruction. That can involve single bits, bytes, or whole memory areas.
- \( \text{gen}(l) \) involves any piece of memory read by the instruction. Again, that can be any bit inside the microcontroller, byte locations, or whole memory areas.

In the following, a few examples of Action List Building for the Intel MCS-51 target are discussed. The behavior of the instructions below is defined in the datasheet as follows:

**CLR [C]** Clear carry. The indicated bit is cleared (i.e., reset to zero). No other flags are affected [47].

**MOV [dest, src]** Move src byte to dest. The byte variable indicated by the second operand is copied into the location specified by the first operand. The source byte is not affected. No other register or flag is affected [47].

**ADDC [A, direct]** Add direct byte to Accumulator with carry. Simultaneously adds the indicated byte variable, the carry flag, and the Accumulator contents, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred. \( \text{OV} \) is set if there is a carry-out of bit 6 but not out of bit 7, or a carry-out of bit 7 but not out of bit 6, otherwise \( \text{OV} \) is cleared. When adding signed integers, \( \text{OV} \) indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative operands [47].
5.2 Implementation – Static Analysis for the C51Simulator

Table 5.3 shows the evaluated \( \text{gen}(l) \) and \( \text{kill}(l) \) statements for the instructions above. \( \text{ADDC} [\text{A, direct}] \) reads the Accumulator, the specified direct memory location, and the carry flag, thus, \( \text{gen}(l) \) evaluates to \( \{ \text{A, direct, C} \} \). Similar, the instruction writes the Accumulator and the flags carry, auxiliary-carry, overflow, and parity, thus, \( \text{kill}(l) \) evaluates to \( \{ \text{A, C, AC, OV, P} \} \).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>( \text{gen}(l) )</th>
<th>( \text{kill}(l) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{CLR} [\text{C}] )</td>
<td>-</td>
<td>\text{C}</td>
</tr>
<tr>
<td>( \text{MOV} [\text{dest, src}] )</td>
<td>\text{src}</td>
<td>\text{dest}</td>
</tr>
<tr>
<td>( \text{ADDC} [\text{A, direct}] )</td>
<td>{ \text{A, direct, C} }</td>
<td>{ \text{A, C, AC, OV, P} }</td>
</tr>
</tbody>
</table>

Table 5.3: Action List Building – a few examples.

The implementation of the instruction visitors is straightforward. Listing 5.6 shows the corresponding visitor for the mnemonic \( \text{ADDC} [\text{A, direct}] \). The instruction visitors for the remaining instructions work quite similar to the introduced one and the interested reader is referred to the actual source code of [mc]sQuare for details.

```java
1 public void visit(ADDC_A_Direct instruction) {
2     addSingleRead(currentVertex.actionList, true, C51Utilities.REGISTER_ACC);
3     addSingleRead(currentVertex.actionList, true, instruction.address);
4     addPSWBitRead(currentVertex.actionList, true, PSW.CY);
5     addSingleWrite(currentVertex.actionList, true, C51Utilities.REGISTER_ACC);
6     addPSWBitWrite(currentVertex.actionList, true, PSW.CY);
7     addPSWBitWrite(currentVertex.actionList, true, PSW.AC);
8     addPSWBitWrite(currentVertex.actionList, true, PSW.OV);
9     addPSWBitWrite(currentVertex.actionList, true, PSW.P);
10 }
```

Listing 5.6: The Action List Builder visitor pattern for the \( \text{ADDC} [\text{A, direct}] \) instruction.

5.2.4 Live Variable Analysis

For the implementation, a dedicated \( \text{C51LVALatticeElement} \) serves to describe memory locations that are live at a discrete node in the CFG. Within the \( \text{C51LVALatticeElement} \) memory locations are defined that are watched and modeled either on byte level or on bit level. The generic LVALatticeElement is extended by bitwise modeling of the PSW and the Interrupt Enable (IE) register. Bitwise modeling of the PSW is needed by the RBA and the bitwise model of the IE register is needed for the IFA, respectively.

```
<table>
<thead>
<tr>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>C51LVALatticeElement (architectural dependencies)</td>
</tr>
</tbody>
</table>
```

Figure 5.5: The type hierarchy of the C51LVALatticeElement.

Every node in the CFG has a \( \text{C51LVALatticeElement} \) attached. Its type hierarchy is given in Figure 5.5. The main class for LVA is the \( \text{C51LVABuilder} \) responsible for the microcontroller specific extensions of the generic LVA as used in [mc]sQuare. The C51LVABuilder determines for each node in the CFG the set of live variables by iterating backwards over all nodes in the CFG. First, the behavior of a single node (i.e., an
instruction) is determined and in a later step the behavior of functions and interrupts is propagated through their callers in the CFG.

The \textit{C51LVABuilder} applies a LVA on single nodes in the CFG. Usually, a CFG consists of several functions that are called by, e.g., the main routine. Moreover, especially in embedded systems software the use of interrupts is common, thus, a program may respond to various sources of interrupts. In principle there is no difference between interrupts and functions, except that interrupts can occur at every location along the CFG, whereas a function is always explicitly called. Thus, it is not sufficient to consider the set of live variables on the node level, a more broaden approach is needed to obtain usable results by the LVA. This broaden approach is realized by the base classes of C51LVABuilder, namely \textit{LVABuilder} and \textit{BackwardProceduralAnalysis}, respectively. The corresponding type hierarchy is given in Figure 5.6.

![Type Hierarchy of C51LVABuilder](image)

Within those classes, \texttt{mcSQUARE} implements the propagation of LVA relevant behavior from single nodes in the CFG to their predecessors and successors, from functions back to their caller, and from ISR to all those locations within the CFG where interrupts are enabled.

### 5.2.5 Reaching Definitions Analysis

Similar to the LVA, a dedicated \textit{C51RDALatticeElement} serves to store definitions made at discrete nodes of the CFG. Again, bitwise modeling for the PSW and the IE register is performed. The motivation is the same as for LVA. The reaching definitions are mapped into single nodes of the CFG by attaching a C51RDALatticeElement object. The type hierarchy of C51RDALatticeElement is given in Figure 5.7.

![Type Hierarchy of C51RDALatticeElement](image)

In order to respect architectural features of the Intel MCS-51 microcontroller, the generic framework for RDA of \texttt{mcSQUARE} is extended by \textit{C51RDABuilder}, as shown in Figure 5.8.

![Type Hierarchy of C51RDABuilder](image)
5.2 Implementation – Static Analysis for the C51Simulator

More details on the conceptional approach are given in [9], for implementation details the reader is referred to the actual source code of [mc]square.

5.2.6 Register Bank Analysis

Most of the aforementioned analysis techniques can be applied to the Intel MCS-51 target without major modifications. Particular architectural features, however, need to be considered explicitly in order to obtain usable analysis results. In the following, a concept termed Register Bank Analysis (RBA) is introduced. With this concept, the problem of additional over-approximation in the existing data-flow analyses is tackled. The over-approximation originates from the architectural feature of register bank switching as available on the Intel MCS-51 microcontroller. The following is mainly based on our work published in [14].

The internal RAM area of the Intel MCS-51 is separated into three main sections, i.e., (a) the register bank area, (b) the bit-addressable area, and (c) the general user RAM area. The register bank area is located within the bottom 32 bytes of internal data memory, resulting in four 8 byte wide register banks. A certain register bank may be selected by the application software through modifying the register bank selection bits. These bits are termed RS0 and RS1 and both are located in the PSW of the microcontroller. More details of architectural features of the Intel MCS-51 are given in its documentation [46]. The bits RS0 and RS1 act as register bank pointer indicating the active register bank. Register banks and possible register bank pointer configurations are detailed in Table 5.4.

<table>
<thead>
<tr>
<th>IRAM Addresses</th>
<th>Register Bank Memory</th>
<th>Bank Selection Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 ... 0x07</td>
<td>R₀{0, 1, 2, 3, 4, 5, 6, 7}</td>
<td>0 0 0</td>
</tr>
<tr>
<td>0x08 ... 0x0F</td>
<td>R₁{0, 1, 2, 3, 4, 5, 6, 7}</td>
<td>1 0 1</td>
</tr>
<tr>
<td>0x10 ... 0x17</td>
<td>R₂{0, 1, 2, 3, 4, 5, 6, 7}</td>
<td>2 1 0</td>
</tr>
<tr>
<td>0x18 ... 0x1F</td>
<td>R₃{0, 1, 2, 3, 4, 5, 6, 7}</td>
<td>3 1 1</td>
</tr>
</tbody>
</table>

Table 5.4: Register bank configurations of the Intel MCS-51.

Register bank swapping is a frequent approach taken by the compiler for passing data to functions or for saving status information before entering ISRs. Conducting a register bank swap over pushing values of memory locations onto the stack before entering an ISR minimizes interrupt latency, and thus, it is the favored approach for time-critical interrupts.

Programs with embedded assembly code, however, can change the register bank at any program location by bit-wise writing the register bank pointer. Knowing the actual value
5 Static Analysis

of the register bank pointer is a decisive criterion for the precision and usefulness of further
analysis, such as LVA and RDA. For example, in case that a variable resides within memory
area (a) of the microcontroller, the analysis results can be significantly sharpened if precise
values of the register bank pointer are determined.

Consider the Intel MCS-51 instruction MOV [R0, #const], which copies an immediate
value to the working register R0 of the currently active register bank. Apparently, MOV
[R0, #const] reads the immediate #const and writes the working register R0. \( \text{kill}_{LV}(\pi) \)
evaluates to R0 and \( \text{gen}_{LV}(\pi) \) to #const, respectively. In order to assign R0 to a certain
register bank, however, there is a need for special treatment of the register bank pointer
composed out of the control bits RS0 and RS1.

### Motivation

Missing prior information about the actual values of RS0 and RS1 is cumbersome. Any
following data-flow analysis suffers from the generated over-approximation due to the un-
known value of the register bank pointer. This effect is detailed in Table 5.5. For example,
if only a single bit of the register bank pointer is ambiguous (either \( \perp \) or \( \top \), see Section 5.2.6
for definition), none of the working registers can be marked as killed since the active regis-
ter bank is unknown. The actual working register may be located at register banks 0, 1, 2,
or 3. As mentioned in Section 5.1.4, dead variables are reset during state space building,
thus, leading to a greater number of equal states that can be merged. Therefore, a bit-wise
analysis of the bank selection pointer seems worthwhile and actively contributes to smaller
state spaces.

<table>
<thead>
<tr>
<th>Bank Selection Pointer</th>
<th>( \text{kill}_{LV}(\pi) )</th>
<th>Register Bank Memory</th>
<th>Safe?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banks</td>
<td>RS0</td>
<td>RS1</td>
<td>( R_0{0} )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>{0, 2}</td>
<td>| / \perp</td>
<td>0</td>
<td>( R_0{0}, R_2{0} )</td>
</tr>
<tr>
<td>{1, 3}</td>
<td>| / \perp</td>
<td>1</td>
<td>( R_1{0}, R_3{0} )</td>
</tr>
<tr>
<td>{0, 1}</td>
<td>0</td>
<td>| / \perp</td>
<td>( R_0{0}, R_1{0} )</td>
</tr>
<tr>
<td>{2, 3}</td>
<td>1</td>
<td>| / \perp</td>
<td>( R_2{0}, R_3{0} )</td>
</tr>
<tr>
<td>{0, 1, 2, 3}</td>
<td>| / \perp</td>
<td>| / \perp</td>
<td>( R_0{0}, R_1{0}, R_2{0}, R_3{0} )</td>
</tr>
</tbody>
</table>

Table 5.5: Evaluating \( \text{kill}_{LV}(\pi) \) for MOV [R0, #const].

### Bit-wise Modeling

The register bank pointer is modeled at bit-level granularity to capture the effects of
bit-wise operations on the PSW. For most of the other analyses, registers are modeled
at byte-level granularity, which turned out to be accurate enough. Based on ideas from
abstract interpretation [92], a single bit is represented using a complete lattice as shown
in Figure 5.9. In the following, the lattice for a single bit depicted on the right-hand side
is denoted by \( L^1 \).
5.2 Implementation – Static Analysis for the C51Simulator

The lattice $L^1$ is composed of the values 0 (false), 1 (true), a top element $\top$ (all), and a bottom element $\bot$ (unknown). The top element $\top$ represents a bit that may have the value 0 or 1, and the bottom element $\bot$ states that no information is available at all. A 4-valued approach of bit-wise modeling is required since merging different paths in the CFG forces the analysis to generate a safe over-approximation. Branches in the CFG originate from conditional branching instructions, which change the flow of program execution. Examples are JZ (jump if accumulator zero), CJNE (compare jump if not equal), and DJNZ (decrement jump if not zero). Merging multiple predecessors in the CFG and combining their individual contributions at confluence points is performed by a join-operation as illustrated in Figure 5.10.

![Figure 5.9: Bit-wise modeling of the register bank selection pointer.](image)

### Formal Description

In the presented approach, the existing RDA is extended to analyze the register bank pointer at bit-level granularity. In a first run of the analysis the reaching definitions for the bits $\text{RS0}$ and $\text{RS1}$ are gathered by a RDA at bit-level. In the first pass, all register banks are assumed to be active in each program location. Then, in further iterations, the application of the join-operator introduced in Table 5.5 leads to more precise results. The join-operator is implicitly encoded in the equations explained in the remainder of this section.

In the following, the notation of Nielson et al. [81] is used for the definition of the functions $\text{genRBA}$ and $\text{killRBA}$, which form the basis of the extension of the RDA. The function $\rho : L^1 \rightarrow 2^{\{0,1\}}$ is used to project lattice elements representing register bank configurations to the domain of values they represent.
\[ \rho(r) = \begin{cases} 
\{0, 1\} & \text{if } r = \top, \\
\emptyset & \text{if } r = \bot, \\
r & \text{otherwise.} 
\end{cases} \]

The function \( \rho \) is used in function \( \tau : L^1 \times L^1 \to 2^{\{0,1,2,3\}} \), which computes integer representations of possible register bank configurations. Here, \( \tau(r_0, r_1) \) returns the set of all register banks that may be active due to the values of \( r_0 \) and \( r_1 \).

\[ \tau(r_0, r_1) = \{2 \cdot x_0 + x_1 | x_0 \in \rho(r_0), x_1 \in \rho(r_1)\} \]

A reaching definition is a pair \((v, \pi)\), where \( v \) represents a memory location or a register and \( \pi \) represents an instruction. Reaching definitions with register bank analysis are computed in several iterations. In the following, the values of RS0 and RS1 in program location \( \pi \) after the \( i \)th iteration are denoted by \( R^i_{RS0}(\pi) \) and \( R^i_{RS1}(\pi) \). They can be extracted from the results of the \( i \)-iteration of the analysis. It is initially \( R^0_{RS0} = \emptyset, R^0_{RS1} = \top \), which means that all register banks are assumed to be active. This leads to a conservative over-approximation of reaching definitions in the first iteration.

For an assignment to \( R_?\{k\} \), for instance through an instruction MOV [R0, #0x80], a reaching definition for register \( R_0 \) on register bank \( b \), denoted by \( R_b\{k\} \), is generated in program location \( \pi \) using \( gen^{i+1}_{RBA} \), if there exists a register bank configuration \( b \). The notation \( R_?\{k\} \) denotes that from the instruction itself, no knowledge about the active register bank is present.

\[ gen^{i+1}_{RBA}(\pi) = \{(R_b\{k\}, \pi)|R_?\{k\} \text{ is assigned a value in } \pi \land r_0 \in R^i_{RS0}(\pi) \land r_1 \in R^i_{RS1}(\pi) \land b \in \tau(r_0, r_1)\} \]

In case the register bank configuration is ambiguous, an over-approximation of the real behavior is generated because a reaching definition is generated for each possible register bank configuration. In like manner, a reaching definition is deleted by \( kill^{i+1}_{RBA} \) only if the register bank configuration is unambiguous. This means, a definition can only be overwritten if only a single register bank configuration is possible. Otherwise, no reaching definitions can be killed in order to guarantee an over-approximation.

\[ kill^{i+1}_{RBA}(\pi) = \{(R_b\{k\}, \pi)|R_?\{k\} \text{ is assigned a value in } \pi \land \exists r_0 \in R^i_{RS0}(\pi), \exists r_1 \in R^i_{RS1}(\pi) : b \in \tau(r_0, r_1) \land |\tau(r_0, r_1)| = 1\} \]

In case no assignment to a memory location addressable using register banks is found, the common equations for RDA are used.
5.2 Implementation – Static Analysis for the C51Simulator

\[
\text{kill}^i_{\text{RDA}}(\pi) = \{(R_k, \psi) | R_k \text{ is assigned a value in } \pi \land k \geq 8 \land (R_k, \psi) \in \text{RDA}^{i-1}(\pi) \}
\]

\[
\text{gen}^i_{\text{RDA}}(\pi) = \{(R_k, \pi) | R_k \text{ is assigned a value in } \pi \land k \geq 8 \}
\]

The entry- and exit-functions for RDA using RBA are then expressed in such a way that the specific equations \(\text{gen}^i_{\text{RBA}}\) and \(\text{kill}^i_{\text{RBA}}\) are only used for those memory locations addressable through register banks. That is, these functions are only used for \(R_b\{k\}\) with \(0 \leq b \leq 3\) and \(0 \leq k \leq 7\). Hence, RBA is used for absolute memory addresses from 0x00 to 0x1F. For all other memory locations, \(\text{gen}^i_{\text{RDA}}\) and \(\text{kill}^i_{\text{RDA}}\) are used.

\[
\text{RDA}^i_\Omega(\pi) = (\text{RDA}^i_A(\pi) \setminus (\text{kill}^i_{\text{RDA}}(\pi) \cup \text{kill}^i_{\text{RBA}}(\pi))) \cup \text{gen}^i_{\text{RDA}}(\pi) \cup \text{gen}^i_{\text{RBA}}(\pi)
\]

\[
\text{RDA}^i_A(\pi) = \bigcup \{\text{RDA}^i_\Omega(\pi') | (\pi', \pi) \in \text{CFG} \}
\]

The results are refined in further iterations. Due to monotony, the results become smaller after each iteration and eventually stabilize after a finite number of iterations. In practice, a fixed point was reached for all programs checked already after the second iteration, but it is possible to construct programs where more iterations are required. After each iteration, concrete values for \(\text{RS}_0\) and \(\text{RS}_1\) are extracted from the RDA – if possible – and used in the next iteration.

Consequently, RBA is conducted at least twice: The first time to collect reaching definitions for \(\text{RS}_0\) and \(\text{RS}_1\) and further times to refine the analysis results by actively using the previously extracted values of the register bank pointer for read and write accesses on working registers. For example, \(\mathcal{R}^i_{\text{RS}_0}(\pi)\) is a reaching definition for the bit \(\text{RS}_0\) at a certain program location \(\pi\), containing a set of all definitions \(\text{gen}^i_{\text{RDA}}(\pi)\) detected for \(\text{RS}_0\) through the program. The definitions originate from the predecessors of \(\pi\) in the CFG.

In like manner, RBA also contributes to the precision of LVA by allowing a more precise reasoning about which values are read in which locations. This enhanced precision is demonstrated using an example in the following section.

**Example**

To highlight the effectiveness of the introduced RBA, the analysis of an assembler program that alters the register bank pointer is described. In particular, the contribution of RBA to the precision of LVA is evaluated. The program is given in Listing 5.7.

<table>
<thead>
<tr>
<th>C: 0x0000</th>
<th>020100</th>
<th>LJMP</th>
<th>STARTUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>C: 0x0100</td>
<td>STARTUP:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C: 0x10100</td>
<td>75D000</td>
<td>MOV</td>
<td>PSW,#0x00</td>
</tr>
<tr>
<td>C: 0x10103</td>
<td>7880</td>
<td>MOV</td>
<td>R0,#0x80</td>
</tr>
</tbody>
</table>

\(^3\)A function \(f : L_1 \to L_2\) between partial ordered sets \(L_1 = (L_1, \sqsubseteq_1)\) and \(L_2 = (L_2, \sqsubseteq_2)\) is monotone if \(\forall l, l' \in L_1 : l \sqsubseteq_1 l' \Rightarrow f(l) \sqsubseteq_2 f(l')\). \(\sqsubseteq\) denotes partial ordering [81].
During the start-up code, the PSW is initialized with 0x00 (see source code lines 3-9), thus, the initial register bank is register bank 0 (RS0=0, RS1=0). In source code line 16, however, the register bank pointer is altered and for lines 16-19 the active register bank is register bank 1 (RS0=0, RS1=1). For the remaining program, register bank 0 remains active. The results of the LVA are listed in Table 5.6.

For the sake of illustration, ISRs are not considered within the example to keep things simple and focus on comprehension of the main idea of the analysis. Functions and ISRs require the propagation of local analysis results to call-sites. In consequence, this makes the intermediate steps and results difficult to follow.

The results are evaluated in two ways. First, the results with the support of RBA are described. These results are compared to the original analysis without the additional information gathered by RBA. The new analysis successfully generates the desired over-approximation and narrows the analysis results. For example, instead of adding registers R0{3}, R1{3}, R2{3}, R3{3}, and P3 to the set of live variables at line 17 (program location 0x10f), the RBA reveals that the exact set of live variables at this location is only composed of R0{3} (working register 3 on bank 0) and P3. Hence, the number of live variables reduces from 5 down to 2, which is a significant improvement. Consequently, reducing the number of live variables increases the number of variables that can be marked as dead.

The same applies to program location 0x118 where the RBA successfully determines register bank 1 as active. Thus, instead of setting registers R0{3}, R1{3}, R2{3}, R3{3}, and P2 to the set of live registers down to R1{3} and P2.

Moreover, for the program locations 0x115, 0x118, 0x11a, and 0x11b, the analysis manages to recognize the change of the register bank pointer from 0 to 1, which is conducted by the instruction MOV [PSW, #0x08] in program location 0x115.

Evaluating the resulting set of live variables in this example reveals that without the new RBA one is unable to make precise propositions about the register bank pointer.
Figure 5.11: The corresponding CFG as generated with |MC|SQUARE for the assembly code in Listing 5.7.
configuration. In this case, the highest degree of over-approximation for working registers has to be applied, i.e., all four register bank combinations are added to the set of live variables (see Table 5.6). The RBA, however, significantly improves the LVA results.

The actual contribution to state space reduction is difficult to state due to the strong interdependence of the analyses. It is simple to construct example codes where an enabled RBA leads to significant state space reductions. On the other hand, examples exist where RBA fails to further shrink the state space. To give an estimation for the example code at hand, the overall state space – without static analysis – consists of 263,683 states. However, in case static analysis supported by RBA is activated the state space shrinks down to 196,740 states leading to a reduction of appr. 25% for this specific example. Note that the large number of states results from the fact that the application reads three I/O ports.

<table>
<thead>
<tr>
<th>PC</th>
<th>with RBA</th>
<th>without RBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>$R_0 {3}, R_1 {3}, P_1, P_2, P_3$</td>
<td>$R_0 {0,3}, R_1 {0,3}, R_2 {0,3}, R_3 {0,3}, P_1, P_2, P_3$</td>
</tr>
<tr>
<td>0x100</td>
<td>$R_0 {3}, R_1 {3}, P_1, P_2, P_3$</td>
<td>$R_0 {0,3}, R_1 {0,3}, R_2 {0,3}, R_3 {0,3}, P_1, P_2, P_3$</td>
</tr>
<tr>
<td>0x103</td>
<td>$R_0 {3}, R_1 {3}, P_1, P_2, P_3$</td>
<td>$R_0 {0,3}, R_1 {0,3}, R_2 {0,3}, R_3 {0,3}, P_1, P_2, P_3$</td>
</tr>
<tr>
<td>0x105</td>
<td>$R_0 {0,3}, R_1 {3}, P_1, P_2, P_3$</td>
<td>$R_0 {0,3}, R_1 {0,3}, R_2 {0,3}, R_3 {0,3}, P_1, P_2, P_3$</td>
</tr>
<tr>
<td>0x106</td>
<td>$R_0 {3}, R_1 {3}, P_1, P_2, P_3$</td>
<td>$R_0 {0,3}, R_1 {0,3}, R_2 {0,3}, R_3 {0,3}, P_1, P_2, P_3$</td>
</tr>
<tr>
<td>0x108</td>
<td>$R_0 {0,3}, R_1 {3}, P_1, P_2, P_3$</td>
<td>$R_0 {0,3}, R_1 {0,3}, R_2 {0,3}, R_3 {0,3}, P_1, P_2, P_3$</td>
</tr>
<tr>
<td>0x109</td>
<td>$R_0 {3}, R_1 {3}, P_1, P_2, P_3, A$</td>
<td>$R_0 {0,3}, R_1 {0,3}, R_2 {0,3}, R_3 {0,3}, P_1, P_2, P_3, A$</td>
</tr>
<tr>
<td>0x10b</td>
<td>$R_0 {3}, R_1 {3}, P_1, P_2, P_3$</td>
<td>$R_0 {3}, R_1 {3}, R_2 {3}, R_3 {3}, P_1, P_2, P_3$</td>
</tr>
<tr>
<td>0x10d</td>
<td>$R_0 {3}, R_1 {3}, P_2, P_3, A$</td>
<td>$R_0 {3}, R_1 {3}, R_2 {3}, R_3 {3}, P_2, P_3, A$</td>
</tr>
<tr>
<td>0x10f</td>
<td>$R_1 {3}, P_2$</td>
<td>$R_0 {3}, R_1 {3}, R_2 {3}, R_3 {3}, P_2$</td>
</tr>
<tr>
<td>0x111</td>
<td>$R_0 {3}, R_1 {3}$</td>
<td>$R_0 {3}, R_1 {3}, R_2 {3}, R_3 {3}$</td>
</tr>
<tr>
<td>0x112</td>
<td>$R_0 {3}, R_1 {3}$</td>
<td>$R_0 {3}, R_1 {3}, R_2 {3}, R_3 {3}$</td>
</tr>
<tr>
<td>0x115</td>
<td>$R_0 {3}, P_3$</td>
<td>$R_0 {3}, R_1 {3}, R_2 {3}, R_3 {3}, P_3$</td>
</tr>
<tr>
<td>0x118</td>
<td>$R_0 {3}, P_3$</td>
<td>$R_0 {3}, R_1 {3}, R_2 {3}, R_3 {3}, P_3$</td>
</tr>
<tr>
<td>0x11a</td>
<td>$R_0 {3}, R_1 {3}$</td>
<td>$R_0 {3}, R_1 {3}, R_2 {3}, R_3 {3}$</td>
</tr>
<tr>
<td>0x11b</td>
<td>$R_0 {3}, R_1 {3}$</td>
<td>$R_0 {3}, R_1 {3}, R_2 {3}, R_3 {3}$</td>
</tr>
<tr>
<td>0x11e</td>
<td>$R_0 {3}, R_1 {3}, A$</td>
<td>$R_0 {3}, R_1 {3}, R_2 {3}, R_3 {3}$</td>
</tr>
<tr>
<td>0x120</td>
<td>$R_0 {3}, A$</td>
<td>$R_0 {3}, R_1 {3}, R_2 {3}, R_3 {3}, A$</td>
</tr>
<tr>
<td>0x121</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 5.6: Comparison of resulting live variables.

Summarizing, the described approach of RBA is a powerful contribution to narrow data-flow analysis results for the Intel MCS-51 architecture. It can be applied to a variety of programs, showing its precision whenever the compiler makes use of instructions involving register banks. For the Intel MCS-51 microcontroller, the RBA handles all of the 160 (out of 256) instructions that use register banks.

### 5.2.7 Stack Analysis

Stack overflows are a common source of software failures within embedded systems code. In order to detect possible stack corruptions, [MC]SQUARE implements a simple check to verify that all those locations pushed onto the stack are later on popped from the stack in
5.2 Implementation – Static Analysis for the C51Simulator

the correct order. For the SA [9], the needed adaption for the Intel MCS-51 target were little, thus, not elaborated in this thesis.

5.2.8 Interrupt Flag Analysis

As aforementioned, control-flow behavior is propagated from ISRs to all nodes of the CFG with interrupts enabled. In order to alleviate over-approximation it is particular important to determine the actual value of the IE bit for each location in the CFG. For locations with interrupts disabled there is no need to consider the additional behavior originated from ISRs. Similar to SA, the existing concepts for the IFA [9] fit well for the Intel MCS-51 target, thus, only minor extensions were made.

5.2.9 Path Reduction

PR [91, 90] is an abstraction technique that is used to compress single successor chains, i.e., paths of states that have only single successors, into single states. Hence, only the first and the last state of these chains are stored by this abstraction technique. As usual, this abstraction technique is performed in order to reduce the overall state space. Figure 5.12 illustrates the principle.

In microcontroller code such single successor chains are, for example, found in ISRs. Although this abstraction contributes greatly to state space reductions, a minor drawback still exists. The validity of the CTL next operator is not preserved due to the compression of successor chains into single states. Thus, using path reduction leads to a restriction of applicable CTL statements and may lead to incomplete counterexamples that are difficult to understand. The subset of statements which can be used in combination of PR are called CTL-X, as described by Yorav and Grumberg in [91].

For the process of collapsing multiple successors to a single state, a set of rules for path reduction for the Intel MCS-51 target was established:
1. PR cannot be applied if the given CTL specification makes use of the neXt operator.
2. PR cannot be applied if one of the successors is an ISR. Similar to rule #4.
3. Any state in which a register is written that is part of the CTL specification cannot be collapsed, since the model checking algorithm has to evaluate the value of this register at this state in order to prove or falsify the specification.
4. Any branch in the CFG determines an end point such as $S_4^*$ in Figure 5.12. Thus, path reduction must preserve the full control flow of the program.

The idea of PR is implemented in two components of [mc]square, i.e., the C51Simulator and the Path Compressor. The C51Simulator part is called the dynamic and the Path Compressor part is called the static component of PR.

The Path Compressor iterates over the nodes in the CFG and tracks whether the node writes a memory location involved in the CTL formula, thus, covering rule #3. The obtained results are back-annotated into the CFG. Indirect control flow and successors branching to ISRs are detected on-the-fly whilst state space building by the C51Simulator, covering rule #2 and rule #4. Finally, rule #1 is checked by the input parser.

5.2.10 Implementation Summary

In this section, a novel data-flow analysis termed Register Bank Analysis was presented to master the architectural feature of register bank swapping for the Intel MCS-51 microcontroller. This new analysis supports static assembly code analysis within [mc]square. In particular, the approach leads to more precise RDA and LVA results, which allows the detection of additional dead variables. Hence, the number of overall system states is reduced during model checking. The effectiveness of this approach was shown by an example. Typical data-flow analyses for high-level languages cannot be applied to assembly code one to one, thus, it is necessary to take architectural peculiarities into account during the analysis to achieve precise results. Analyses such as RDA or LVA have to be adapted to be applicable to assembly code [90]. Especially the concept of Action List Building proved to be a major contributor for the portability of existing static analyses within [mc]square for future microcontroller families.

5.3 Remaining Challenges in Static Analysis of (Intel MCS-51) Assembly Code

Although the introduced RBA is an advantageous concept to limit the over-approximation on architectures featuring register bank swapping, there are still some obstacles to overcome. In what follows, four major challenges are outlined and possible approaches to overcome them are highlighted. These challenges are mostly concerned with indirect control, indirect memory access, and loops.

5.3.1 Indirect Addressing

Data structures such as tables, lists, or arrays are often accessed by changing the address of an operand on-the-fly, i.e., during the execution of a program. In high-level programming languages, these addresses are known as pointers. Indirect addressing is a powerful
addressing mode, which provides flexibility for the compiler. For the Intel MCS-51 the working registers R0 or R1 may serve as base registers for indirect addressing.

Again, when executing instructions such as MOV [A, @R0], it is the register bank pointer that selects the corresponding base register from one of the four register banks. Thus, for resolving the actual base register the aforementioned RBA is used. Without actually executing the code prior to the instruction, which uses indirect addressing, it is in most cases not trivial to predict the content of the base register. This uncertainty forces our analysis to generate a conservative over-approximation. Consider the assembly snippet depicted in Listing 5.8.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CLR A</td>
</tr>
<tr>
<td>2</td>
<td>MOV A, 0x25</td>
</tr>
<tr>
<td>3</td>
<td>ADD A, 0x26</td>
</tr>
<tr>
<td>4</td>
<td>MOV R0, A</td>
</tr>
<tr>
<td>5</td>
<td>MOV 0x44, @R0</td>
</tr>
</tbody>
</table>

Listing 5.8: Intel MCS-51 assembly snippet.

In fact, resolving the destination register is a rather challenging problem for the static analysis. The destination register is the value held by R0 in line 5 in Listing 5.8. To resolve the actual value one has to consider:

(i) The actual value of memory locations 0x25 and 0x26 needs to be detected. In order to do so, one has to trace back the operations performed on these memory locations until one can reason about the circumstances under which the actual values are generated.

(ii) The exact semantics of the involved instructions is required. Although guessing the effect of certain instructions on the memory content of the microcontroller seems to be obvious for instructions such as CLR and MOV, it is a challenging task for complex instructions such as ADD – at least without explicitly executing the instruction, for instance, by a target platform simulator.

(iii) Embedded systems communicate actively with their environment, thus, various interrupt sources are likely to interfere with execution of the main process. Special care has to be taken in this case. Interrupt handlers may alter the values of memory locations 0x25, 0x26, or even the value of the working register R0. This situation becomes more challenging on target architectures supporting nested interrupts.

(iv) Even though the presented assembly code does not contain any program branching instructions, the remaining program memory may contain direct and indirect jumps targeting any of the program locations stated in Listing 5.8. For instance, a branching instruction may target the program location holding MOV [R0, A] with an entire different register configuration compared to the sequential execution of the program fragment.

### 5.3.2 Indirect Control Flow

A precise CFG is the foundation of any kind of data-flow analysis. Building a complete CFG is challenging in presence of indirect control. A precise CFG requires all control branches in the input program to be mapped into the resulting CFG. Due to information that is not
5 Static Analysis

statically computable such as target addresses of indirect branches the analysis framework is forced to generate a conservative over-approximation. For example, the unconditional indirect jump statement \texttt{JMP [A+DPTR]} would add edges to all possible program locations reachable by the \texttt{JMP [A+DPTR]} instruction. Indirect branches to dynamically calculated targets are fragments commonly used by the compiler in order to generate optimized code. As a matter of fact, in the embedded systems domain highly optimizing compilers are used due to prevailing resource constraints.

An interesting aspect when dealing with indirect control flow is the fact that a target addresses of branch instructions can origin from either (i) the environment or (ii) from lookup tables stored in the program memory. The latter is the more common one, since reading branch target addresses from the environment is rarely found in real life applications.

```
void main() {
    ...
    switch(var){
        case 0xAA: foo1(); break;
        case 0xBB: foo2(); break;
        case 0xCC: foo3(); break;
        case 0xDD: foo4(); break;
        case 0xEE: foo5(); break;
        case 0xFF: foo6(); break;
        case 0xC0: foo7(); break;
        default: break;
    }
    while(1);
}
```

Listing 5.9: C source code containing switch statement.

Lookup tables, among others, are used by the compiler to realize switch-case statements as shown in Listing 5.9.

The program in Listing 5.10 shows the resulting assembler code generated by the Keil C51 Compiler v8.01, without any optimizations enabled. The assembler routine \texttt{C?CCASE} is called from the main method to achieve the required behavior needed for the switch statement (cf. Listing 5.9, line 14). The call of the subroutine (indirectly) pushes the current program counter value (\texttt{0x0808}) on the stack. Thereafter, the new data pointer value of \texttt{0x0808} is loaded from the stack by two consecutive \texttt{POP} statements. Next, the Accumulator holding the value of variable \texttt{var1} is loaded into working register \texttt{R0} and the Accumulator is cleared afterwards. Line 7 fetches a byte from program memory at address \texttt{C:0x0808}. Listing 5.11 presents a memory dump of the particular program memory section, revealing that the instruction in line 7 actually reads the byte \texttt{0x8}.

A conditional jump to address \texttt{C:0x0860} is executed. Instructions at source lines 22 and 23 loading the comparison value \texttt{0xA} for the first case branch (cf. Listing 5.9, line 14). The comparison value residing in \texttt{R0} and the Accumulator are XORed, thus, carrying out a compare of the two values. In case the two values are equal, the Accumulator is set to zero after the comparison. For the first run the comparison value (\texttt{0xA}) does not match the value of variable \texttt{var1 (0xC0)}, therefore, program flow reaches lines 26 to 29, incrementing the data pointer by three bytes.

The aforementioned sequence is repeated for the comparison values \texttt{0xBB} and \texttt{0xC0}, respectively. Both do not match the value of variable \texttt{var1}. Next, the program code is
executed with the comparison value of 0x0c0 that matches the actual value of variable \textit{var1}. The comparison of the two values in line 24 evaluates to a cleared accumulator in line 25, thus, forcing a jump to program location \texttt{C:0x0855} (cf. Listing 5.10, line 14). Lines 14 to 17 are loading the address of the corresponding function \texttt{C:0x082B} and the following two instructions reset the data pointer (DPTR). Finally, the indirect jump in line 21 branches to the selected function \texttt{void foo3(void)}, which resides at program address \texttt{C:0x082B} (cf. Listing 5.12, line 3).

```
1 C: 0x0805 120845 LCALL C?CCASE(C:0845)
2 C: 0x0845 D083 POP DPH(0x83)
3 C: 0x0847 D082 POP DPL(0x82)
4 C: 0x0849 F8 MOV R0,A
5 C: 0x084A E4 CLR A
6 C: 0x084B 93 MOV A,@A-DPTR
7 C: 0x084C 7012 JNZ C:0860
8 C: 0x084E 7401 MOV A,#0x01
9 C: 0x0850 93 MOV A,@A-DPTR
10 C: 0x0851 700D JNZ C:0860
11 C: 0x0853 A3 INC DPTR
12 C: 0x0854 A3 INC DPTR
13 C: 0x0855 93 MOV A,@A-DPTR
14 C: 0x0856 F8 MOV R0,A
15 C: 0x0857 7401 MOV A,#0x01
16 C: 0x0859 93 MOV A,#0x01
17 C: 0x085A F582 MOV DPL(0x82),A
18 C: 0x085C 8883 MOV DPH(0x83),R0
19 C: 0x085E E4 CLR A
20 C: 0x085F 73 JMP @A-DPTR
21 C: 0x0860 7402 MOV A,#0x02
22 C: 0x0862 93 MOV A,@A-DPTR
23 C: 0x0863 68 XRL A,R0
24 C: 0x0864 60EF JZ C:0855
25 C: 0x0866 A3 INC DPTR
26 C: 0x0867 A3 INC DPTR
27 C: 0x0868 A3 INC DPTR
28 C: 0x0869 80DF SJMP C:084A
```

Listing 5.10: Switch Statement Assembler code snippet.

The corresponding lookup table assembled by the compiler contains the comparison values \texttt{AA}, \texttt{BB}, \texttt{CC}, \texttt{DD}, \texttt{EE}, and \texttt{FF} (see Listing 5.11) for the case statements as well as the entry addresses of the called functions (see Listing 5.11 and Listing 5.12).

```
1 C: 0x0808 08 21 AA 08 26 BB 08 3F
2 C: 0x0810 C0 08 2B CC 08 30 DD 08
3 C: 0x0818 35 EE 08 3A FF 00 00 08
4 C: 0x0820 42 xx xx xx xx xx xx
```

Listing 5.11: Program memory content.

Again, at least for this particular case, it seems feasible to reason about actual target locations by searching for the pattern of comparison values in the program memory. Such naive approaches, however, may work for a very limited number of configurations, but they
heavily depend on the compiler version, optimization levels, etc. Most important, they are only applicable for a certain target architecture. Consequently, a rather holistic approach needs to be found to overcome the problem of generating a precise CFG for programs containing indirect control flow.

```
1 C: 0x0821  void foo1 (void);
2 C: 0x0826  void foo2 (void);
3 C: 0x082B  void foo3 (void);
4 C: 0x0830  void foo4 (void);
5 C: 0x0835  void foo5 (void);
6 C: 0x083A  void foo6 (void);
7 C: 0x083F  void foo7 (void);
8 C: 0x0842  while(1);
```

Listing 5.12: Entry addresses for called functions.

### 5.3.3 Self-Modifying Code

Self-modifying code is error-prone as well as difficult to read, understand, test, and maintain and hard to port to different target microcontrollers. In general, generating self-modifying code is not supported by compilers, thus, it is a design pattern introduced by sloppy application engineering. Fortunately, it is rarely seen in real life applications and widely considered as bad programming style.

As [mc]square aims toward an universal static analysis framework for assembly code, we have to consider the possibility of self-modifying code. Determining the exact dependencies and behavior of self-modifying code is a challenging task. The Intel MCS-51 target realizes a traditional Harvard architecture with program and data memory strictly separated. Hence, the problem of self-modifying code can be abandoned by architectural considerations.

Nevertheless, modified Harvard architectures and von Neumann based targets are open for self-modifying code. Fortunately, those architectures targeted by [mc]square use dedicated instructions to actively alter the program memory content at runtime. For the ATMELE ATmega16 target, modifying program code is only possible by a single instruction, namely SPM (store to program memory). Consequently, a rather straightforward approach is applied to deal with self-modifying code. Whenever these instructions are detected during CFG building, the data-flow analyses are aborted since self-modifying code will show a behavior that cannot be analyzed statically.

Thus, in the presented approach, the analyses are limited to constant, non self-modifying code. Other approaches exist, for instance, Anckaert et al. [93] introduced state-enhanced CFGs as a new program representation in presence of self-modifying code.

### 5.3.4 Loop Bounds

A precise data-flow analysis requires upper bounds of loop iterations. Respective approaches are referred as loop bound analysis in literature. Existing approaches aim at determining loop bounds widely automatic, but approximating loop bounds for all conceivable loop constructs such as non constant increment and decrement of counter variables or multiple nested loops depending on each other remains a challenge. Some of the existing tools [94, 95] avoid these pitfalls by requiring the user to annotate certain program
remaining challenges in static analysis of (Intel MCS-51) assembly code

locations prior to the analysis, or by directly specifying upper loop bounds for constructs
that cannot be analyzed automatically.

Considering the enormous amount of research already performed on detecting loop
bonds, we are eager to reuse the existing knowledge. For our particular purpose, we
are interested in the following cases:

(i) For a precise pointer analysis, a narrow approximation of loop bounds is required.
   This necessitates to take the effects of complete sequences of instructions into account
   as the conditions of branching instructions are frequently computed in a sequence of
   instructions.

(ii) Other techniques such as program slicing [96, 77] require the detection of loop ter-
    mination. During program slicing for model checking, instructions that have no
    influence on the validity of a specification are removed from the program. Divergent
    behavior, i.e., non-termination, of the original program must remain visible in the
    sliced program. Hence, loops for which termination cannot be proven cannot be
    sliced. Moreover, statements that influence loop conditions cannot be sliced, which
    strongly affects sizes of program slices.

In summary, a combination of detecting loop bounds and loop termination is required
in order to compute precise results for slicing.

5.3.5 Summary

Even though application tailored analysis methods, such as RBA, allow a significant nar-
rowing of data-flow analyses, there are still difficulties in static analysis for assembly code
to overcome. A scalable and precise pointer analysis would be a major boost for analysis
precision within [MC]SQUARE. In consequence, future research incentives are required to
develop a widely generic approach for resolving indirect read and write accesses on assem-
fty code level. One requirement for such an approach is that only minor modifications
to the analysis are required to take peculiarities of different target architectures into ac-
count. Such a framework could as well be used for predicting actual target addresses on
the assembly code level in order to accomplish CFG building of programs featuring indirect
control.
6 Real Life Case Study

Program testing can be used to show the presence of bugs, but never to show their absence. We ... take the position that it is not only the programmer’s task to produce a correct program but also to demonstrate its correctness in a convincing manner.

(Edsger W. Dijkstra)

In what follows, a real life industry case study is conducted with the $\text{[MC]SQUARE}$ model checker. The case study focuses on (i) assessing the feasibility of $\text{[MC]SQUARE}$ when applied to real life embedded applications, (ii) evaluating the effects of the implemented abstraction techniques on the resulting state space size, and (iii) identifying future research directions. First, an introduction and a motivation for the case study is given. Next, hardware and software components of the application are presented. Later on, the used communication protocol is sketched. Then, temporal logic properties are postulated that correspond to the given textual specification. Finally, results and findings are presented.

6.1 Introduction and Motivation

In the following, a real life embedded systems application is introduced and its software is model checked with $\text{[MC]SQUARE}$. The model checking process is supported by the previously described abstraction and static analysis techniques for the Intel MCS-51 target. We use this case study to evaluate strengths and weaknesses of our approach. Most of the previously conducted case studies (cf. [49, 12, 13]) were of smaller code size or less complexity.

On the other hand, the results of the case study are used to assess the individual contribution to state space reduction of the aforementioned abstraction techniques. Furthermore, we expect the case study as a significant indicator to define further research incentives in order to make $\text{[MC]SQUARE}$ a mature embedded software verification tool that can be used by embedded software designers within their day-to-day software engineering routine.

Besides all the technical concepts implemented in $\text{[MC]SQUARE}$, in our believe, it is of upmost importance that a certain degree of usability is preserved throughout the development process of a verification tool. A few existing tools – especially those with strong academic background – are, from the technical point of view, highly professional but from the user point of view quite hard and challenging to operate and manage. Not surprisingly, the long term vision of $\text{[MC]SQUARE}$ is a fully automatic, push-button verification tool.

The analyzed source code is provided by Texion Software Solutions, a company located
in Aachen, Germany. Texion Software Solutions' core business is the development of individual, application tailored embedded hardware and software solutions. Their product portfolio includes the software ProFab, a software system for production data acquisition. Their customers use the software system for networking of textile knitting machines. A textile knitting machine produces various types of knitted fabrics of varying degrees of complexity. Modern knitting machines usually contain highly complex electronics controlling the needles and the yarn. Figure 6.1(a) shows such a machine.

As in every industrial application software reliability is a major concern. Thus, formal verification of the knitting machine's software is worthwhile, since failures caused by software faults are costly in terms of production losses and the associated additional maintenance effort.

The target application of the case study is the software for a knitting machine monitoring device, as shown in Fig. 6.1(b). The source code for the knitting machine monitoring device was selected for the case study based on the following considerations:

- Good conformance of the application with the applications we are aiming at. The application targets the Intel MCS-51 microcontroller, uses several on-chip peripheral modules, interacts with its environment, and makes use of various interrupt sources.
- Commonality of interests with the developers and their willingness to cooperate. Our industry partner supported us by providing the full source code and a sample device. Furthermore, we can access all accompanying documents such as the software specification and the hardware schematics.
- Criticality and the need of high reliability. As aforementioned, flawless software is crucial, since every malfunction is costly in industrial practice.
- Complexity. The number of source code lines is within our reach, i.e., from the conceptional point of view, [MC]SQUARE is able to handle applications of this size. It should be noted, that the source code line count is a rather unsuited indicator, whether an application can be successfully model checked or the model checker will run out of resources (state-explosion problem) whilst examining the code. It is almost solely the source code complexity that is crucial.

### 6.2 The Knitting Machine Monitoring Device – Hardware Overview

The knitting machine monitoring device is composed out of the following hardware modules (cf. Figure 6.2):

**Knitting machine** is the machine that is observed and monitored by the knitting machine monitoring device.

**Input module** connects the knitting machine monitoring device with the knitting machine. It features eight input lines. Each input is decoupled and an inverting Schmitt trigger\(^1\) acts as a pulse shaper. The input module connects the input lines with the

---

\(^1\)A Schmitt trigger is a comparator circuit that incorporates positive feedback. When the input is higher than a certain threshold, the output is high. When the input is below another (lower) threshold, the output is low. When the input is between the two, the output retains its value [97].
6.2 The Knitting Machine Monitoring Device – Hardware Overview

(a) A modern knitting machine (image property of Texion Software Solutions).

(b) Knitting machine monitoring device.

Figure 6.1: The target application.
corresponding I/O pins of the microcontroller (four pins of Port 3 and four pins of Port 1).

**Microcontroller** executes the software subject to verification.

**Serial Interface** provides the physical link to the host application through a RS232 interface.

**Host application** uses the data gathered by the monitoring device for further processing.

**Miscellaneous** (not depicted in Figure 6.2)

**Watchdog module** is a hardware timing module that triggers the reset input of the microcontroller due to a faulty condition. The fault condition is reached if the watchdog hardware timer overflows. The timer overflow can be avoided if the microcontroller application resets the watchdog module periodically.

**Power and clock generation** provides an inverse-polarity protection and a 5 V filtered and stabilized power supply. Furthermore, this module contains a quartz-controlled clock generation.

**Light Emitting Diode (LED) module** operates three LEDs, signaling serial communication traffic and the “liveness” of the application.

**Potential separation** uses a photo-coupler for electrical isolation and performs the needed voltage level adjustment.

---

**6.3 The Knitting Machine Monitoring Device – Software Overview**

The application is of small/medium size and is designed following the foreground/background design pattern (cf. [98, 99]). The system consists of a super-loop, i.e., an infinite loop that calls individual modules (functions) to perform the desired operations (the background part). Asynchronous events (the foreground part) are handled through ISRs. Thus, time critical operations are performed by the ISRs to ensure that they are dealt within the given timing constraints. Timing correctness is met by interrupting the background part of the software at predefined points in time, e.g., when a timer expires or a character is received over the serial interface. The used foreground/background design pattern is sketched in Figure 6.3.
6.3 The Knitting Machine Monitoring Device – Software Overview

Background

Foreground

```
void main (void){
   InitBoard();
   SendTxt(STX,'R',ETX);
   SetTime();
   while(1){
      Watchdog();
      Liveness();
      UpdateInputs();
      EvaluateRPM();
      RSM(readCmd());
   }
}
```

Figure 6.3: The foreground/background design pattern.

6.3.1 The Main Building Blocks

**RPM module**
- **external ISR**
- pulse counting

**Timer module**
- **timer ISR**
- **time management**

**State machine**
- communication controller
- receiver state machine

**Serial interface**
- serial ISR
- receive and transmit characters

**In-Output module**
- read environment
- reset watchdog
- liveness LED

Figure 6.4: The software components.

From the conceptional point of view, the source code can be divided into five building blocks (cf. Figure 6.4):

**Revolutions Per Minute (RPM) module** manages two external interrupts to count pulses from external rotary encoders. It initializes the two interrupt sources and defines their interrupt priority. The pulse count is internally mapped to a 16 bit wide unsigned data type.

**Timer module** uses the Timer 0 peripheral module of the microcontroller to provide a system tick and four software timers. Furthermore, it provides trivial functions for time management like reset(), set(), and get_time().

**State machine** implements the serial communication protocol and performs the needed housekeeping.

**Serial interface module** initializes the serial communication device of the microcontroller to 9600 Baud and uses dedicated circular buffers for managing receive and transmit queues. It provides methods for sending and receiving characters.

**In-Output module** reads the eight input ports for the monitoring function and handles the watchdog reset. Furthermore, it toggles the “liveness” LED.
The full source code of the case study consists of about 600 lines of C-code (i.e., 1400 lines of assembly code).

6.3.2 Serial Receive and Transmit Ringbuffer

The application uses software circular buffers to compensate the lack of a hardware First In First Out (FIFO) memory. The circular buffer manages buffering of characters received from and sent to the serial port. A circular buffer is a common data structure that uses a single, fixed-size buffer as if it is connected end-to-end (cf. Figure 6.5).

![Figure 6.5: A software circular buffer model.](image)

The read pointer indicates the element that is read next and the write pointer determines the location, which will be filled with the next character. Altogether, the case study uses two dedicated circular buffer structures, i.e., one for receiving and one for sending characters. The C code macros and the initialization calls are given in Listing 6.1.

```c
/* Header macro */
#define RingBuffer (Name, DataType, IndexType, Exp, Attribute) 

struct {
  IndexType ReadIndex;
  IndexType WriteIndex;
  IndexType Mask;
  DataType Buffer[(1 << Exp) − 1];
} Attribute Name = {0, 0, (1 << (Exp)) − 1};

/** Ring buffer initialization */
RingBuffer (RxBuffer, char, word, 2, ); /* char−RingBuffer for receiver */
RingBuffer (TxBuffer, char, word, 2, ); /* char−RingBuffer for transmitter */
```

Listing 6.1: Ringbuffer C code macro.

Considering the initialization code in Listing 6.1, it is easily seen that four byte-wide buffers are used. According to Table 6.1, a single RingBuffer element consists of 10 bytes altogether. As [MC][SQUARE] reads and parses relevant debug info, it allows C-code variable names to be included into the temporal specification, i.e., CTL formulas. Thus, the column Formula name in Table 6.1 refers to the actual expression that is used within the CTL formulas.
### 6.3.3 The Communication Protocol

The communication between the knitting machine monitoring device and the host application follows a well defined protocol. It is a straightforward master-slave approach where the host application operates as master. Thus, every communication is initiated by the host application, with one exception: the knitting machine monitoring device sends a status message to the master after powerup. The communication protocol does not include any data integrity checks such as checksums. Figure 6.6 shows the corresponding communication sequence chart and Table 6.3.3 states the specified commands and the expected reply. RPM\(_1\) denotes the first byte of the *Revolutions* variable, RPM\(_2\) refers to the second byte, respectively. The same applies to CNT\(_1\), CNT\(_2\), VER\(_1\), VER\(_2\), and VER\(_3\).

### 6.4 Extracting CTL Properties Out of the Textual Specification

One of the most crucial steps in model checking, as in any formal verification method, is the process of creating a formal specification (as interpreted by the model checker, e.g., CTL) that relate to a given textual specification. Again, it is important to realize that any formal verification is only as good as the stated claims. The remainder of this section reveals that finding a formal CTL counterpart for a textual representation of the systems behavior is non-trivial and sometimes challenging. The precise meaning of the used variables and symbols within the properties is given in Table 6.3.

#### 6.4.1 The Given Textual Specification

In our case, an initial specification is part of the project. The initial specification is given in German. For the sake of clarity, it was translated into English first. This was done by the best of the author’s knowledge and special care was taken to preserve the original meaning of the specification. Strictly speaking this might already introduce some kind of inconsistency and misinterpretation. As the given specification is rather informal and in a textual form, we had to identify relevant properties first and translate them to CTL.

#### 6.4.2 CTL Properties

In the following, CTL properties are presented that originate from the given textual specification. These properties will be model checked by [mc]SQUARE later on. Each property is given in the form of:

<table>
<thead>
<tr>
<th>Element</th>
<th>C code type</th>
<th>Length [byte]</th>
<th>Formula name</th>
</tr>
</thead>
<tbody>
<tr>
<td>IndexType ReadIndex</td>
<td>word</td>
<td>2</td>
<td>(Tx/RxBuffer_{0,1})</td>
</tr>
<tr>
<td>IndexType WriteIndex</td>
<td>word</td>
<td>2</td>
<td>(Tx/RxBuffer_{2,3})</td>
</tr>
<tr>
<td>IndexType Mask</td>
<td>word</td>
<td>2</td>
<td>(Tx/RxBuffer_{4,5})</td>
</tr>
<tr>
<td>DataType Buffer[1«Exp]</td>
<td>char</td>
<td>4</td>
<td>(Tx/RxBuffer_{6...9})</td>
</tr>
<tr>
<td>Sum</td>
<td>RingBuffer</td>
<td>10</td>
<td>(Tx/RxBuffer_{0...9})</td>
</tr>
</tbody>
</table>

Table 6.1: Ringbuffer elements and their size.
Figure 6.6: Communication sequence chart.
6.4 Extracting CTL Properties Out of the Textual Specification

<table>
<thead>
<tr>
<th>#</th>
<th>Master Command</th>
<th>Bytes</th>
<th>Slave Response</th>
<th>Bytes</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$ D #</td>
<td>3</td>
<td>$ D [RPM$_1$ RPM$_2$] revolutions per minute [2 bytes]</td>
<td># 5</td>
<td>returns the current revolutions per minute</td>
</tr>
<tr>
<td>2</td>
<td>$ R #</td>
<td>3</td>
<td>$ R system reset [0 bytes]</td>
<td># 3</td>
<td>resets the device</td>
</tr>
<tr>
<td>3</td>
<td>$ Z #</td>
<td>3</td>
<td>$ Z [CNT$_1$ CNT$_2$] counter value [2 bytes]</td>
<td># 5</td>
<td>returns the current pulse counter value</td>
</tr>
<tr>
<td>4</td>
<td>$ E #</td>
<td>3</td>
<td>$ E [INP] input representation [1 byte]</td>
<td># 4</td>
<td>returns the current input representation</td>
</tr>
<tr>
<td>5</td>
<td>$ V #</td>
<td>3</td>
<td>$ V [VER$_1$ VER$_2$ VER$_3$] version string [3 bytes]</td>
<td># 6</td>
<td>returns the software version number as string</td>
</tr>
</tbody>
</table>

Table 6.2: The master-slave communication protocol.
### Table 6.3: Case study variables and their meaning.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Scope</th>
<th>Initial</th>
<th>Length</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Revolutions</td>
<td>global</td>
<td>0xffff</td>
<td>2 byte</td>
<td>Holds the current RPM.</td>
</tr>
<tr>
<td>RxBuffer_i</td>
<td>global</td>
<td>0x00</td>
<td>1 byte</td>
<td>i(^{th}) byte of receive buffer memory area</td>
</tr>
<tr>
<td>RxBuffer_1</td>
<td>global</td>
<td>0x00</td>
<td>1 byte</td>
<td>The receive circular buffer read pointer</td>
</tr>
<tr>
<td>RxBuffer_4</td>
<td>global</td>
<td>0x00</td>
<td>1 byte</td>
<td>The receive circular buffer write pointer</td>
</tr>
<tr>
<td>TxBuffer_j</td>
<td>global</td>
<td>0x00</td>
<td>1 byte</td>
<td>j(^{th}) byte of transmit buffer memory area</td>
</tr>
<tr>
<td>TxBuffer_1</td>
<td>global</td>
<td>0x00</td>
<td>1 byte</td>
<td>The transmit circular buffer read pointer</td>
</tr>
<tr>
<td>TxBuffer_4</td>
<td>global</td>
<td>0x00</td>
<td>1 byte</td>
<td>The transmit circular buffer write pointer</td>
</tr>
<tr>
<td>Command_state</td>
<td>local</td>
<td>0x00</td>
<td>1 byte</td>
<td>Holds the actual state of the state machine.</td>
</tr>
<tr>
<td>startUpCodeFinished</td>
<td>global</td>
<td>0x00</td>
<td>1 byte</td>
<td>Set to 1 when <code>main()</code> is entered</td>
</tr>
<tr>
<td>mark</td>
<td>global</td>
<td>0x00</td>
<td>1 byte</td>
<td>Supplementary variables inserted for model checking.</td>
</tr>
<tr>
<td>mark</td>
<td>local</td>
<td>0x00</td>
<td>1 byte</td>
<td>Indicates when entering certain PC locations</td>
</tr>
<tr>
<td>mark</td>
<td>global</td>
<td>0x00</td>
<td>1 byte</td>
<td>Indicates when entering certain PC locations</td>
</tr>
<tr>
<td>mark</td>
<td>local</td>
<td>0x00</td>
<td>1 byte</td>
<td>Indicates when entering certain PC locations</td>
</tr>
<tr>
<td>mark</td>
<td>global</td>
<td>0x00</td>
<td>1 byte</td>
<td>Indicates when entering certain PC locations</td>
</tr>
<tr>
<td>mark</td>
<td>local</td>
<td>0x00</td>
<td>1 byte</td>
<td>Indicates when entering certain PC locations</td>
</tr>
</tbody>
</table>

Variables used by the target application (excerpt)
<table>
<thead>
<tr>
<th>Property #</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Textual representation</td>
</tr>
<tr>
<td>#1</td>
<td>CTL representation ([MC]SQUARE notation)</td>
</tr>
<tr>
<td>#1</td>
<td>Comment</td>
</tr>
</tbody>
</table>

Property #1

**Textual representation**
The variable Revolutions is initialized to \texttt{0xffff}.

**CTL representation** (\[MC\]SQUARE notation)
\[(\text{AG}(\text{startUpCodeFinished}=0 \& \text{Revolutions}=\texttt{0x0000} \Rightarrow \text{Revolutions}=\texttt{0x0000} \U \text{Revolutions}=\texttt{0xff00}) \& \text{AG}(\text{startUpCodeFinished}=0 \& \text{Revolutions}=\texttt{0xff00} \Rightarrow \text{Revolutions}=\texttt{0xff00} \U \text{Revolutions}=\texttt{0xffff}) \& \text{EF(Revolutions}=\texttt{0x0000} \& \text{EF(Revolutions}=\texttt{0xff00})\]

**Comment**
If variable Revolutions is \texttt{0x0000}, then it remains \texttt{0x0000} until it becomes \texttt{0xff00}. If variable Revolutions is \texttt{0xff00}, then it remains \texttt{0xff00} until it becomes \texttt{0xffff}. There is a path where variable Revolutions is \texttt{0x0000} and there is a path where Revolutions is \texttt{0xff00}. The initialization must be completed before the startup code is left.

Property #1a

**Textual representation**
The variable Revolutions is initialized to \texttt{0xffff}.

**CTL representation** (\[MC\]SQUARE notation)
\[\text{AF(Revolutions}=\texttt{0xffff} \& \text{startUpCodeFinished}=1\]

**Comment**
On all paths the variable Revolutions is of value \texttt{0xffff} when the startup code is left.

Property #1b

**Textual representation**
The variable Revolutions is initialized to \texttt{0xffff}.

**CTL representation** (\[MC\]SQUARE notation)
\[\text{AF(Revolutions}=\texttt{0xffff}\]

**Comment**
On all paths there is a state where variable Revolutions is set to \texttt{0xffff}.
Property #2

**Textual representation**
Initialization of the receive circular buffer.

**CTL representation ([MC]square notation)**
\[ AF(RxBuffer_0=0x00 \& RxBuffer_1=0x00 \& RxBuffer_2=0x00 \& RxBuffer_3=0x00 \& RxBuffer_4=0x00 \& RxBuffer_5=0x03 \& RxBuffer_6=0x00 \& RxBuffer_7=0x00 \& RxBuffer_8=0x00 \& RxBuffer_9=0x00 \& startUpCodeFinished=1) \]

**Comment**
On all paths within the startup code there is finally a state where all bytes of \( RxBuffer_i \) where \( i = \{0 \ldots 9\} \) are initialized to 0x00, except \( RxBuffer_5 \) which is initialized to 0x03, since it acts as the circular buffer mask. See Listing 6.1 for details.

---

Property #3

**Textual representation**
Initialization of the transmit circular buffer.

**CTL representation ([MC]square notation)**
\[ AF(TxBuffer_0=0x00 \& TxBuffer_1=0x00 \& TxBuffer_2=0x00 \& TxBuffer_3=0x00 \& TxBuffer_4=0x00 \& TxBuffer_5=0x03 \& TxBuffer_6=0x00 \& TxBuffer_7=0x00 \& TxBuffer_8=0x00 \& TxBuffer_9=0x00 \& startUpCodeFinished=1) \]

**Comment**
On all paths within the startup code there is finally a state where all bytes of \( TxBuffer_j \) where \( j = \{0 \ldots 9\} \) are initialized to 0x00, except \( TxBuffer_5 \) which is initialized to 0x03, since it acts as the circular buffer mask. See Listing 6.1 for details.

---

Property #4

**Textual representation**
It is possible to reach the “sleep” state of the application where the application idles in an endless loop.

**CTL representation ([MC]square notation)**
\[ EF\ mark=MARK_SLEEP \]

**Comment**
There is a state where the application reaches the “sleep” state. Note that, this formula can also be expressed by involving the PC into the property, such as AG (EF PC=0xc0ffee). For the sake of clarity, however, the variable \( mark \) is introduced to allow self-explanatory CTL expressions.
Property #5

**Textual representation**

It is possible to reach the “send version” state of the application where the application sends its version string to the host application.

**CTL representation ([MC]square notation)**

\[
\text{EF } \text{mark}=\text{MARK\_SENDVERSION}
\]

**Comment**

There is a state where the application reaches the “send version” state.

---

Property #6

**Textual representation**

It is possible to reach the “send inputs” state of the application where the application sends the actual value of the digital input lines to the host application.

**CTL representation ([MC]square notation)**

\[
\text{EF } \text{mark}=\text{MARK\_SENDINPUTS}
\]

**Comment**

There is a state where the application reaches the “send inputs” state.

---

Property #7

**Textual representation**

It is possible to reach the “send pulse count” state of the application where the application sends the actual value of the pulse counter to the host application.

**CTL representation ([MC]square notation)**

\[
\text{EF } \text{mark}=\text{MARK\_SENDPULSCNT}
\]

**Comment**

There is a state where the application reaches the “send pulse count” state.

---

Property #8

**Textual representation**

It is possible to reach the “send RPM” state of the application where the application sends the actual RPM value to the host application.

**CTL representation ([MC]square notation)**

\[
\text{EF } \text{mark}=\text{MARK\_SENDRPM}
\]

**Comment**

There is a state where the application reaches the “send RPM” state.
Property #9

Textual representation
The default path of the receiver state machine in function \texttt{readCommand()} (cf. Listing 6.2) is executed at least once.

CTL representation ([MC]square notation)
\[ \text{EF} \; \text{mark} = \text{MARK\_DEFAULT} \]

Comment
There is a state where the default path of the switch statement in function \texttt{readCommand()} is executed.

Property #10

Textual representation
The receiver state machine may only reside in states 0, 1, or 2. All other states are invalid.

CTL representation ([MC]square notation)
\[ \text{Inv:} (\text{Command\_state}=0 \mid \text{Command\_state}=1 \mid \text{Command\_state}=2) \]

Comment
On all paths the actual state of the state machine is either 0, 1, or 2. The term \text{Inv} stands for invariant model checking. An equivalent expression of this formula is \( \text{AG} (\text{Command\_state}=0 \mid \text{Command\_state}=1 \mid \text{Command\_state}=2) \)

Property #11

Textual representation
Changes in the receiver state machine can only follow the following patterns: 
0 \(\rightarrow\) 1 \(\rightarrow\) 2 \(\rightarrow\) 0 or 0 \(\rightarrow\) 1 \(\rightarrow\) 0. All other transitions are invalid.

CTL representation ([MC]square notation)
\[
\begin{align*}
\text{AG} & (\text{Command\_state}=0 \Rightarrow A \text{ Command\_state}=0 U \text{ Command\_state}=1 \mid \text{Command\_state}=0) \& \\
\text{AG} & (\text{Command\_state}=1 \Rightarrow A \text{ Command\_state}=1 U \text{ Command\_state}=0 \mid \text{Command\_state}=2 \mid \text{Command\_state}=1) \& \\
\text{AG} & (\text{Command\_state}=2 \Rightarrow A \text{ Command\_state}=2 U \text{ Command\_state}=0 \mid \text{Command\_state}=2) \& \text{AF Command\_state}=0
\end{align*}
\]

Comment
If \text{Cmd\_state} = 0, \text{Cmd\_state} remains 0 until it changes to 1, if \text{Cmd\_state} = 1 then \text{Cmd\_state} remains 1 until it changes to 0 or 1, if \text{Cmd\_state} = 2, \text{Cmd\_state} remains 2 until it changes to 0. There is a path where \text{Cmd\_state} initially becomes 0 and the receiver state machine may always remain in its current state.
6.4 Extracting CTL Properties Out of the Textual Specification

Property #12

--- Textual representation ---
The serial receive circular buffer read and write pointer may never exceed the circular buffer bounds.

--- CTL representation (\[mc\]square notation) ---
AG (RxBuffer_1<4 & RxBuffer_0=0 & RxBuffer_3<4 & RxBuffer_4=0)

--- Comment ---
RxBuffer_1 (the read pointer low byte) and RxBuffer_3 (the write pointer low byte) are on all paths lower than the circular buffer bound, i.e., 4 bytes. Moreover, the high byte of the read and write pointer (RxBuffer_0 and RxBuffer_4) remain 0.

Property #13

--- Textual representation ---
The serial transmit circular buffer read and write pointer may never exceed the circular buffer bounds.

--- CTL representation (\[mc\]square notation) ---
AG (TxBuffer_1<4 & TxBuffer_0=0 & TxBuffer_3<4 & TxBuffer_4=0)

--- Comment ---
TxBuffer_1 (the read pointer low byte) and TxBuffer_3 (the write pointer low byte) are on all paths lower than the circular buffer bound, i.e., 4 bytes. Moreover, the high byte of the read and write pointer (TxBuffer_0 and TxBuffer_4) remain 0.

Property #14

--- Textual representation ---
The microcontroller application sends '$ R #' to the host application after power-up.

--- CTL representation (\[mc\]square notation) ---
EF (TxBuffer_6='$' & TxBuffer_7='R' & TxBuffer_8='#' & TxBuffer_9=0)

--- Comment ---
There is a path where the transmit circular buffer is filled with the sequence $ R #.

6.4.3 Comments

It is notable, that property #1 reveals one of the major strengths of our assembly code model checking approach. As the verification process is based on machine instructions, it is even possible to verify the exact initialization sequence of the 16 bit wide variable Revolutions. Property #1 requires that the high byte (located on the higher address) is initialized first and then the low byte is initialized, as it is the usual way on little-endian processor architectures. In contrast, a model checker targeting C code is most times not able to make assumptions on byte/memory location granularity due to the missing details about the target platform. Property # 1a is a property suited for C code model checkers. However, as the property EF Revolutions=0xffff & startUpCodeFinished=1 only verifies that variable Revolutions will eventually reach the value 0xffff within the startup sequence, it excludes the details on how the initialization of Revolutions is accomplished.
For example, it is possible that the variable Revolutions is – for various reasons – first set to 0xfc0f and later on set to 0xffff. As a result, property #2 will evaluate to true, since the initialization sequence is not sufficiently specified. However, as [MC|SQUARE allows CTL properties to include single memory locations, property #1 will evaluate to false showing the erroneous behavior within the startup code as counterexample. The same applies to properties #12 and #13.

### 6.4.4 Reviewing Properties #4a to #8a

Properties #4 to #8 claim for a single path where the application will finally reach the specified state, e.g., EF mark=MARK_SLEEP. We might tighten this property in a way that, globally, from every state in the program, it must be possible to finally reach the “sleep” state. Properties of this form, are termed resetability in literature [28, 29]. As a result, we extend the properties #4 to #8:

<table>
<thead>
<tr>
<th>Property #4a</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Textual representation</strong></td>
</tr>
<tr>
<td>It is always possible to reach the “sleep” state of the application where the application idles in an endless loop.</td>
</tr>
<tr>
<td><strong>CTL representation</strong> ([MC</td>
</tr>
<tr>
<td>AG(EF mark=MARK_SLEEP)</td>
</tr>
<tr>
<td><strong>Comment</strong></td>
</tr>
<tr>
<td>On every path the application finally reaches the “sleep” state.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property #5a</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Textual representation</strong></td>
</tr>
<tr>
<td>It is always possible to reach the “send version” state of the application where the application sends its version string to the host application.</td>
</tr>
<tr>
<td><strong>CTL representation</strong> ([MC</td>
</tr>
<tr>
<td>AG(EF mark=MARK_SENDVERSION)</td>
</tr>
<tr>
<td><strong>Comment</strong></td>
</tr>
<tr>
<td>On every path the application finally reaches the “send version” state.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property #6a</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Textual representation</strong></td>
</tr>
<tr>
<td>It is always possible to reach the “send inputs” state of the application where the application sends the actual value of the digital input lines to the host application.</td>
</tr>
<tr>
<td><strong>CTL representation</strong> ([MC</td>
</tr>
<tr>
<td>AG(EF mark=MARK_SENDINPUTS)</td>
</tr>
<tr>
<td><strong>Comment</strong></td>
</tr>
<tr>
<td>On every path the application finally reaches the “send inputs” state.</td>
</tr>
</tbody>
</table>
6.4 Extracting CTL Properties Out of the Textual Specification

Property #7a

**Textual representation**

It is always possible to reach the “send pulse count” state of the application where the application sends the actual value of the pulse counter to the host application.

**CTL representation** (\([mc]\)square notation)

\[ \text{AG(} \text{EF}\, \text{mark} = \text{MARK\_SENDPULSCNT}) \]

**Comment**

On every path the application finally reaches the “send pulse count” state.

Property #8a

**Textual representation**

It is always possible to reach the “send RPM” state of the application where the application sends the actual RPM value to the host application.

**CTL representation** (\([mc]\)square notation)

\[ \text{AG(} \text{EF}\, \text{mark} = \text{MARK\_SENDRPM}) \]

**Comment**

On every path the application finally reaches the “send RPM” state.

6.4.5 Communication Protocol Verification

Setting up valid CTL formulas for properties such as “In case $Z#$ is received the microcontroller must answer with $CNT_1 \, CNT_2$” is a rather challenging task, due to (i) fairness issues (see Section 6.4.5 for details) and (ii) the prevalent circular buffer implementation of the application.

However, consider the following (humble) specification:

Property #invalid

**Textual representation**

After receiving $Z#$, answer with $CNT_1 \, CNT_2$.

**CTL representation** (\([mc]\)square notation)

\[ \text{AG}\, (\text{RxBuffer}_6 = 0x24 \, \&\, \text{RxBuffer}_7 = 0x5A \, \&\, \text{RxBuffer}_8 = 0x23) \Rightarrow \text{AF}\, (\text{TxBuffer}_5 = 0x24 \, \&\, \text{TxBuffer}_6 = CNT_1 \, \&\, \text{TxBuffer}_7 = CNT_2 \, \&\, \text{TxBuffer}_8 = 0x23) \]

**Comment**

Whenever the receive circular buffer is filled with $Z#$ there is always a path where the transmit circular buffer finally carries the values $CNT_1 \, CNT_2$.

The above stated property is invalid and incomplete due to several reasons:

- The property does not consider the value of the read and write pointers of the circular buffer.
  - What if the read pointers $Tx/RxBuffer_{\{0, 1\}}$ do not point to the first element in the circular buffer?
  - What if the transmit write pointer $TxBuffer_{\{2, 3\}}$ does not point to the first element in the circular buffer?
• What if the model checker encounters a path where the serial interrupt is never fired?

• The property does not consider the value of the circular buffer mask, i.e., $Tx|RxBuffer_{\{4,5\}}$.

• The property does not consider the value of the fourth byte of the receive buffer, i.e., $RxBuffer_{9}$.

• How to make sure that, after once decoding $Z\#$, the receive circular buffer is not altered anymore?

• Once the receive circular buffer is filled with $Z\#$, how to make sure that the application does read the circular buffer content in a sequential way? What if, i.e., the read pointer is incremented twice, or it is not incremented at all?

It is obvious that creating valid CTL expressions for the communication protocol verification – at least without additional knowledge of software internals – is quite challenging. It might be possible for some corner cases, however, the resulting formulas are of unhandy length and complex to understand in their full details. By all means, there is no way to express fairness in plain CTL model checking.

The Unfair Path

Recapitulating, [MC]SQUARE abstracts from time, i.e., can be categorized as timeless, pure CTL model checker (cf. Section 3.5). Consider the property AG (AF $mark=\text{MARK\_SENRPM}$), which claims that on every path it must always be possible to finally reach the desired mark, e.g., $mark=\text{MARK\_SENRPM}$. As the target application uses interrupts and the Intel MCS-51 allows interrupt nesting, it might be possible that the model checker may get stuck within an interrupt loop, where immediately after an interrupt is executed, the ISR is re-entered again. Such an interrupt loop is depicted in Figure 6.7 by the path $\{I_1, I_2, I_3, I_4, I_1, I_2, ...\}$, which we call an unfair path.

The model checker now may find a counterexample where the property AG (AF $mark=\text{MARK\_SENRPM}$) is disproved due to this unfair path by getting stuck inside the interrupt loop. Clearly, such unfair paths are very unlikely when executing the code on the actual target hardware, albeit theoretical possible. As a matter of fact, the serial interrupt may only occur at multiple time instances of the selected serial baud rate, and thus, is very unlikely to produce unfair paths. As [MC]SQUARE follows a timeless model checking approach, we cannot use timing constraints to eliminate this behavior. In fact, we have to overcome the lack of fairness in CTL in order to obtain meaningful results for the present case study.

The Lack of Fairness in CTL

When using formal verification tools, one is often only interested in proving a property over fair paths. Thus, certain paths that are considered to be unrealistic for the actual target hardware, in our case the Intel MCS-51 microcontroller, need to be ruled out. In literature [29, 40] an unfair computation is described as an unreasonable computation that ignores certain transition alternatives forever and all the others are described as fair. In order to express fairness, fairness constraints [29] are used that operate on a path level and
6.4 Extracting CTL Properties Out of the Textual Specification

Figure 6.7: The unfair path.

replace the standard meaning for all paths with for all fair paths and there exists a path with there exists a fair path.

Unfortunately, such fairness properties cannot be expressed directly in CTL [100, 36, 101] but can be expressed in CTL*. In contrast, fairness assumptions can be easily added as a premise to an LTL formula. In LTL a fairness assumption can be stated in the form of (fairness)\(\Rightarrow\) (property), e.g., (GF enabled)\(\Rightarrow\) (GF occurs).

However, Clarke et al. [28] show how a Kripke structure can be enriched by fairness constraints in order to enable fairness in CTL. In their approach, a fair path must contain an element of each fairness constraint infinitely often. A path is fair if each constraint is true infinitely often along the path. Consequently, they restrict path quantifiers in the logic to those fair paths. We use, from the conceptual point of view, a similar approach of introducing fairness into CTL model checking with [mc]square. In the following, we present the introduction of fairness through a model of the microcontroller environment.

Introducing Fairness through Environment Modeling

As [mc]square implements CTL model checking algorithms, we are limited to plain CTL without fairness constraints, thus, fairness must be introduced via an additional concept. To that end, we make use of environment modeling.

Within [mc]square this particular feature is termed User Defined Environment (UDE) modeling [102, 103]. The UDE constrains the inputs read from the environment to a manually specified set of values and allows to control the occurrence of interrupt sources. An automata is used to define input values as well as interrupt and value transitions. The use of UDE leads to the adapted model checking workflow as shown in Figure 6.8.

---

In their approach, a fair Kripke structure is a 4-tuple \(\mathcal{M} = (\mathcal{S}, \mathcal{R}, \mathcal{L}, \mathcal{F})\), where \(\mathcal{S}\), \(\mathcal{R}\), and \(\mathcal{L}\) are defined as in Section 3.4.2 and \(\mathcal{F} \subseteq 2^\mathcal{S}\) is a set of fairness constraints. \(\pi\) is a path in \(\mathcal{M}\) and \(\text{inf}(\pi)\) is defined as \(\text{inf}(\pi) = \{s \mid s = s_i, \text{ for infinitely many } i\}\). A path \(\pi\) is fair iff for every \(\mathcal{P} \in \mathcal{F}\), \(\text{inf}(\pi) \cap \mathcal{P} \neq \emptyset\).
The model checking workflow is now enriched by a third input, namely the environment automata $U$.

An UDE is realized by a communicating finite state machine [104], which interacts with a representation of the microcontroller, i.e., the C51Simulator component (see Section 3.6). During model checking, this automata represents the environment, thus, influences the behavior of the C51Simulator. From the user point of view, UDE automata are created with a graphical editor inside the GUI of [mc]SQUARE. It works like any other automata drawing tool, e.g., the user adds states and transitions through toolbars onto a canvas. Alternatively, UDEs can be defined by using an environment description language [102]. Both approaches have the same expressiveness [103].

For the present case study, an UDE is used to define an exact sequence of values read from the serial port. Moreover, we block or fire the serial ISR at certain points to ensure fairness.

![Figure 6.8: The model checking workflow of [mc]SQUARE with UDE (cf. Figure 3.3).](image)

**An Environment Automata for Fairness in the Communication Protocol Verification**

Having discussed the principles of UDEs in [mc]SQUARE, the remainder of this section is dedicated to the procedure of finding a suitable UDE automata for fairness in verifying the communication protocol of the case study. Two requirements for the desired UDE automata are derived:

(i) Values are read from the serial interface according to the communication protocol specification (cf. Figure 6.6).

(ii) The unfair path of interrupt loops (see Section 6.4.5) is avoided through blocking of the corresponding ISRs for at least a single instruction after executing a RETI
instruction. In other words, progress in the background part of the application (cf. Figure 6.3) is assured.

With respect to this requirements, we can generate the environment automata $U_1$, as shown in Figure 6.9. State changes among $\{S_0, S_1, S_2, S_3\}$ are triggered whenever the application reads a value from the serial communication interface. For example, the transition label $\text{SBUF 35!}$ indicates that the UDE forces the simulator to determinize the serial receive register SBUF to the value of 35, i.e., ASCII ‘#’ (cf. Table 6.4). The states $\{B_0, B_1, B_2, B_3\}$ are responsible for blocking the serial interrupt, after the serial ISR is left (transition “ISR leave”). The execution of the next instruction triggers the transition (“Instr leave”) back to one of the states of $\{S_0, S_1, S_2, S_3\}$.

Figure 6.9: A first UDE automata proposal ($U_1$).

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBUF</td>
<td>The serial transmit/receive register</td>
</tr>
<tr>
<td>d 35</td>
<td>Decimal equivalent of ASCII ‘#’</td>
</tr>
<tr>
<td>d 36</td>
<td>Decimal equivalent of ASCII ‘$’</td>
</tr>
<tr>
<td>d 68</td>
<td>Decimal equivalent of ASCII ‘D’</td>
</tr>
<tr>
<td>d 69</td>
<td>Decimal equivalent of ASCII ‘E’</td>
</tr>
<tr>
<td>d 86</td>
<td>Decimal equivalent of ASCII ‘V’</td>
</tr>
<tr>
<td>d 82</td>
<td>Decimal equivalent of ASCII ‘R’</td>
</tr>
<tr>
<td>d 90</td>
<td>Decimal equivalent of ASCII ‘Z’</td>
</tr>
</tbody>
</table>

Table 6.4: Definitions for UDE modeling.

Nevertheless, the automata $U_1$ is still not sufficient for our protocol verification venture, due to the following consideration – What happens if the serial (receive) interrupt does not occur at all?

In fact, this configuration is possible on the real target hardware. If the host application does not initiate any serial communication at all, the knitting machine monitoring device will not send any answer to the host. This can be seen as the idle state of the application.
However, as we are interested in the communication sequences rather than the idle state, the path where no serial (receive) interrupt occurs is unfair too. As a result, when model checking the communication protocol with the UDE automata \( U_1 \), \([mc]\)SQUARE disproves the properties by presenting counterexamples where the serial interrupt is never activated.

In order to eliminate those paths, we extend the automata \( U_1 \) to \( U_2 \), as shown in Figure 6.10. Considering automata \( U_2 \), the states \( \{ F_0, F_1, F_2, F_3 \} \) actively trigger the execution of the serial interrupt. Again, \( \{ B_0, B_1, B_2, B_3 \} \) are states where the serial interrupt is blocked. Basically, there are four sequences that are equivalent, i.e., \( \{ S_0, B_0, F_0 \} \), \( \{ S_1, B_1, F_1 \} \), \( \{ S_2, B_2, F_2 \} \), and \( \{ S_3, B_3, F_3 \} \). Transitions among those sequences \( \{ F_0, S_1 \}, \{ F_1, S_2 \}, \{ F_2, S_3 \}, \) and \( \{ F_3, S_0 \} \) are used to determinize the serial receive register SBUF to the values of the protocol as defined in Table 6.3.3.

The transitions labeled with "ISR leave" from \( \{ F_0, F_1, F_2, F_3 \} \) back to \( \{ B_0, B_1, B_2, B_3 \} \) are needed due to an implementation detail of the application. In case the circular buffer is full, the application skips incoming serial data bytes, thus, it may happen that a serial receive interrupt occurs, but the application does not read the value of the SBUF register. Hence, the transition is needed to prevent the automata \( U_2 \) from being stuck in the states where the receive circular buffer is full and the serial interrupt is fired again, i.e., one of \( \{ F_0, F_1, F_2, F_3 \} \).

Note that the transition SBUF \{35,36,82\}! is used to send any of these three bytes between a communication sequence. Thus, in our UDE model possible sequences are \{35,36,82,35\} (’#,$’,R,‘#’), \{35,36,82,36\} (’#,$’,R,’$’), \{35,36,82,82\} (’#,’R,’R’), \{35,36,82\} (’#,’R,’$’), …

We can easily extend this claim to a full nondeterministic read between a communication sequence, e.g., by changing the transition to SBUF \{0 .. 255\}!, however, as it will turn out in the remainder of this section this is not needed for our verification process.

It should be noted that without detailed knowledge of the application’s software structure it is almost impossible to obtain a proper UDE automata – at least for the example code at hand. As we are aiming towards a formal verification tool that can be used as early as during the development phase, the software insight is brought into by the software development team.

After demonstrating that an UDE automata is capable of introducing fairness to the model checking process, in the following, properties for the communication protocol verification are stated that are model checked with support of the automata \( U_2 \). The extended \([mc]\)SQUARE workflow is used as shown in Figure 6.8.

Note that, automata \( U_2 \) exactly corresponds to property \#Comm2. For the remaining properties, we adapt the transitions with the actual values of the command, i.e., we replace the transitions SBUF 82! and SBUF \{35,36,82\}! of property \#Comm2 with SBUF 68! and SBUF \{35,36,68\}! to obtain the UDE automata for property \#Comm1.
Figure 6.10: The final UDE automata (U2).
Property #Comm1

Textual representation

After receiving $D\#$ the knitting monitoring device answers with $D\ RPM_1\ RPM_2\#$, i.e., sends the variable Revolutions.

CTL representation ([mc]square notation)

$AG(AF\ mark=MARK\_SEND\_RPM)$

Comment

If the application reads $D\#$ from the serial port the application always reaches the state MARK\_SEND\_RPM.

Property #Comm2

Textual representation

After receiving $R\#$ the knitting monitoring device answers with $R\#$, i.e., enters the reset state.

CTL representation ([mc]square notation)

$AG(AF\ mark=MARK\_RESET)$

Comment

If the application reads $R\#$ from the serial port the application always reaches the state MARK\_RESET.

Property #Comm3

Textual representation

After receiving $Z\#$ the knitting monitoring device answers with $Z\ CNT_1\ CNT_2\#$, i.e., sends the current pulse counter value.

CTL representation ([mc]square notation)

$AG(AF\ mark=MARK\_SENDCOUNTER)$

Comment

If the application reads $Z\ CNT\#$ from the serial port the application always reaches the state MARK\_SENDCOUNTER.

Property #Comm4

Textual representation

After receiving $E\#$ the knitting monitoring device answers with $E\ INP\#$, i.e., sends the current input lines value.

CTL representation ([mc]square notation)

$AG(AF\ mark=MARK\_SEND\_INPUTS)$

Comment

If the application reads $E\#$ from the serial port the application always reaches the state MARK\_SEND\_INPUTS.
6.5 Results

In what follows, the results of the case study are presented. Note that – for clarity’s sake – only a summary of the most significant results is given in this section.

6.5.1 Numbers

Table 6.5 shows the results of the first [MC]SQUARE model checking run. The item States created refers to the overall states that are created by the model checker. The item States stored comprises the states that are stored in main memory. It is compiled out of the total states created (i.e., the item States created) shortened by the number of state revisits, i.e., single states that are already present in the main memory. The column Time refers to the time needed by [MC]SQUARE to finish model checking.

The numbers were generated on a Dual-Core AMD Opteron™ Processor 8220 with 2.80 Ghz (8 cores), equipped with 256 GB of RAM running 64 bit Windows Server Enterprise Edition and a Java™ server virtual machine version 1.6.0 (with settings -server -Xmx200G -Xss120M). Source code revision 4338 of [MC]SQUARE was used.

6.5.2 Stack Analysis

As mentioned in Section 5.2.7, [MC]SQUARE performs a static stack analysis in order to detect stack corruptions. For the case study, no stack corruptions were detected, thus, the stack is safe. That means, all those bytes pushed onto the stack, are again popped from the stack in the right order. The upper stack bound of the application is 0x81 and the lower stack bound evaluated to 0x73, hence, the maximal stack size is 8. The results obtained from static analysis were cross-proved in the model checking run, as the maximum stack size during model checking evaluated to 8, too.

6.5.3 The Circular Buffer Implementation

The circular buffer implementation is covered by the properties #2, #3, #12, and #13. Whereas #2 and #3 refer to the correct initialization of the circular buffer, properties #12 and #13 make propositions about the range of the read and write pointers. Referring to the resulting figures in Table 6.5, these properties could be verified on the analyzed source code. Thus, with respect to the postulated CTL properties, the circular buffer implementation is error-free.
### Table 6.5: Case study results.

<table>
<thead>
<tr>
<th>Property</th>
<th>Stored</th>
<th>Created</th>
<th>Revisits</th>
<th>Time</th>
<th>Abstraction techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>✓</td>
<td>2,107,785</td>
<td>22,627,672</td>
<td>721,167</td>
<td>00:05:11</td>
</tr>
<tr>
<td>1a</td>
<td>✓</td>
<td>230</td>
<td>634</td>
<td>0</td>
<td>00:00:01</td>
</tr>
<tr>
<td>1b</td>
<td>✓</td>
<td>146</td>
<td>145</td>
<td>0</td>
<td>00:00:01</td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
<td>230</td>
<td>634</td>
<td>0</td>
<td>00:00:01</td>
</tr>
<tr>
<td>3</td>
<td>✓</td>
<td>230</td>
<td>634</td>
<td>0</td>
<td>00:00:01</td>
</tr>
<tr>
<td>4</td>
<td>✓</td>
<td>123,242</td>
<td>126,486</td>
<td>3,244</td>
<td>00:00:02</td>
</tr>
<tr>
<td>5</td>
<td>✓</td>
<td>123,304</td>
<td>126,551</td>
<td>3,247</td>
<td>00:00:02</td>
</tr>
<tr>
<td>6</td>
<td>✓</td>
<td>104,234</td>
<td>107,051</td>
<td>2,817</td>
<td>00:00:02</td>
</tr>
<tr>
<td>7</td>
<td>✓</td>
<td>142,248</td>
<td>156,924</td>
<td>3,676</td>
<td>00:00:02</td>
</tr>
<tr>
<td>8</td>
<td>✓</td>
<td>161,195</td>
<td>165,300</td>
<td>4,106</td>
<td>00:00:03</td>
</tr>
<tr>
<td>9</td>
<td>x</td>
<td>2,325,333</td>
<td>20,099,919</td>
<td>525,454</td>
<td>00:04:25</td>
</tr>
<tr>
<td>10</td>
<td>✓</td>
<td>2,333,585</td>
<td>20,065,707</td>
<td>524,632</td>
<td>00:04:28</td>
</tr>
<tr>
<td>11</td>
<td>✓</td>
<td>2,333,585</td>
<td>20,065,707</td>
<td>524,632</td>
<td>00:04:20</td>
</tr>
<tr>
<td>12</td>
<td>✓</td>
<td>2,497,397</td>
<td>19,999,441</td>
<td>524,632</td>
<td>00:04:33</td>
</tr>
<tr>
<td>13</td>
<td>✓</td>
<td>2,359,645</td>
<td>20,018,732</td>
<td>524,632</td>
<td>00:04:35</td>
</tr>
<tr>
<td>14</td>
<td>✓</td>
<td>1,519</td>
<td>344</td>
<td>14</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

Revisited Properties #4 to #8

<table>
<thead>
<tr>
<th>Property</th>
<th>Stored</th>
<th>Created</th>
<th>Revisits</th>
<th>Time</th>
<th>Abstraction techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>4a</td>
<td>✓</td>
<td>2,325,333</td>
<td>20,099,919</td>
<td>525,454</td>
<td>00:04:30</td>
</tr>
<tr>
<td>5a</td>
<td>x</td>
<td>20,459</td>
<td>150,896</td>
<td>3,247</td>
<td>00:00:03</td>
</tr>
<tr>
<td>6a</td>
<td>x</td>
<td>20,456</td>
<td>150,890</td>
<td>3,247</td>
<td>00:00:02</td>
</tr>
<tr>
<td>7a</td>
<td>x</td>
<td>23,893</td>
<td>173,084</td>
<td>3,676</td>
<td>00:00:02</td>
</tr>
<tr>
<td>8a</td>
<td>x</td>
<td>27,328</td>
<td>195,275</td>
<td>4,106</td>
<td>00:00:03</td>
</tr>
</tbody>
</table>

Comm. Protocol Verification with UDE (faulty receiver implementation)

| Comm1    | x      | 4,810   | 4,811   | 1 | 00:00:01 | ✓ | x |
| Comm2    | x      | 4,810   | 4,811   | 1 | 00:00:01 | ✓ | x |
| Comm3    | x      | 4,810   | 4,811   | 1 | 00:00:01 | ✓ | x |
| Comm4    | x      | 4,810   | 4,811   | 1 | 00:00:01 | ✓ | x |
| Comm5    | x      | 4,810   | 4,811   | 1 | 00:00:01 | ✓ | x |

Comm. Protocol Verification with UDE (fixed receiver implementation)

| Comm1    | ✓      | 101,277 | 101,616 | 339 | 00:00:02 | ✓ | x |
| Comm2    | ✓      | 48,513  | 48,672  | 159 | 00:00:01 | ✓ | x |
| Comm3    | ✓      | 101,205 | 101,544 | 339 | 00:00:01 | ✓ | x |
| Comm4    | ✓      | 100,989 | 101,328 | 339 | 00:00:02 | ✓ | x |
| Comm5    | ✓      | 101,133 | 101,472 | 339 | 00:00:02 | ✓ | x |
6.5.4 The Receiver State Machine

The receiver state machine, responsible for decoding input commands from the serial interface, is considered by properties #10 and #11. These properties were successfully verified by [mc]SQUARE on the analyzed source code (see Table 6.5), thus, the variable Command_state is proven to remain within its bounds. Furthermore, the receiver state machine implementation follows the claimed transition sequence.

6.5.5 Properties #4a to #8a

As shown in Table 6.5, property #4a is valid. The property claims that it is always possible to reach the “sleep” state of the application.

Surprisingly, properties #5a to #8a were disproved by [mc]SQUARE. The counterexample shows that it is not possible to reach the states MARK_SENDVERSION (property #5a), MARK_SENDINPUTS (property #6a), MARK_SENDPULSCNT (property #7a), and MARK_SENDRPM (property #8a) in case the host application previously sent the command $ R # (go to sleep state) to the microcontroller application. The sleep state of the application is implemented as a while(true) endless loop, that can only be left through an external reset of the microcontroller by the watchdog module. As we do not consider the watchdog module in the verification process, the endless loop cannot be left, thus, properties #5a to #8a are – correctly – disproved by [mc]SQUARE.

6.5.6 The Communication Protocol

The proper implementation of the communication protocol is covered by properties #Comm1 to #Comm5. As aforementioned, we used the UDE automata $U2$ in order to rule out unfair paths and to overcome the lack of fairness in plain CTL model checking. As stated in Table 6.5, all those properties were falsified by [mc]SQUARE. The tool is capable of presenting counterexamples either in a step-by-step way by using the Intel MCS-51 simulator or by drawing a graphical counterexample path.

Studying the counterexample reveals that the communication protocol implementation is erroneous. In the rare case the host application sends an even number of start bytes ($), the microcontroller application skips the following command, thus, fails to send a reply message to the host.

Consequently, erroneous sequences are:

- \{$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$,\$
host application sends an odd number of start bytes, i.e., \{[2n+1]_{n>0}|\times$, ?, #\}. Listing 6.2 shows the erroneous implementation of the receiver state machine.

The counterexample presented by [mc]SQUARE reveals that a sequence of \{[2n]_{n>0}|\times$, ?, #\} resets the variable \textit{Command\_state} to 0 after the initial state is left for the first time. In fact, the \textit{readCommand()} implementation is now only sensitive to start bytes ($\times$), but the host application already starts transmitting the command part (one of \{D, R, Z, E, V\}) of the communication sequence. As a result, the following protocol bytes are skipped as long as a new valid command sequence (\{[2n + 1]_{n>0}|\times$, ?, #\}) is received. The root cause of the failure lies in source code line 26 of Listing 6.2. The application waits for either a start byte (STX) or a stop byte (ETX), thus, an even number of start bytes resets the variable \textit{Command\_state} to 0 – over and over again, as shown in source code line 28.

    char readCommand(void)
    {
        static byte Command\_state = 0;
        static char Command;
        char c;

        if (char available())
            c = rcvchar();
        else
            return 0;

        switch (Command\_state)
        {
            case 0:
                /* Initial State */
                if (c == STX)
                    Command\_state = 1;
                break;
            case 1:
                if ((c == STX) || (c == ETX))
                    Command\_state = 0; /* STX repeat or ETX */
                else
                    Command = c; /* Command found */
                    Command\_state = 2;
                break;
            case 2:
                Command\_state = 0;
                if (c == ETX)
                    return Command; /* Command returned if ETX received */
                break;
            default:
                Command\_state = 0;
                mark = MARK\_DEFAULT;
                break;
        }
        return 0;
    }

Listing 6.2: The erroneous receiver state machine implementation.

\textbf{Revising the Receiver State Machine Implementation}

In order to correct the receiver state machine implementation, source code lines 25-35 of Listing 6.2 are adapted to lines 35-39 of Listing 6.3. Whenever the host now sends sequences of the form \{[2n]_{n>0}|\times$, ?, #\}, the variable \textit{Command\_state} is not erroneously reset to 0, but set to 1 which forces the state machine to wait for the following command byte.
6.5 Results

```c
char readCommand(void) {
    static byte Command_state = 0;
    static char Command;
    char c;
    if (charavail()) {
        c = rcvchar();
    } else {
        return 0;
    }
    switch (Command_state) {
    case 0:
        /* Initial State */
        if (c == STX) {
            Command_state = 1;
        }
        break;
    case 1:
        if (c == ETX) {
            Command_state = 0; /* ETX received */
        } else if (c == STX) {
            Command_state = 1; /* STX repeat */
        } else {
            Command = c; /* Command found */
            Command_state = 2;
        }
        break;
    case 2:
        Command_state = 0;
        if (c == ETX) {
            return Command; /* Command returned if ETX received */
        }
        break;
    default:
        Command_state = 0;
        mark = MARK_DEFAULT;
        break;
    }
    return 0;
}
```

Listing 6.3: The revised receiver state machine implementation.

6.5.7 Compiler Criticism

Not surprisingly, property #9 was disproved by [mc]square. It follows, that the application never enters the default path of the receiver state machine as shown in lines 47-50 in Listing 6.3. Even though it seems a little far-fetched, an intelligent compiler would recognize the default path as dead code and remove it from the *.hex file.

6.5.8 Comparison of Abstraction Techniques

In the following, a comparison of the implemented abstraction techniques is presented. It is used to assess the individual contribution to state space reduction of the different concepts. In order to achieve comparability among the abstraction techniques, one abstraction technique is enabled at a time. Furthermore, the formula AG (true) is used that is equivalent to plain state space building of the application.

Table 6.6 shows the results of the second [mc]square model checking run by using the same hardware configuration as described in Section 6.5.1. Note that the first model checking round was canceled after 24 hours of runtime.
Based on the results in Table 6.6, the following consequences are drawn:

(i) For the present case study, it is not feasible to build the state space without any abstraction techniques applied. This is not surprising since the application heavily interacts with the environment.

(ii) The model checking run with enabled Delayed Nondeterminism option was canceled after running two days on the server. Thus, Delayed Nondeterminism provides not enough abstraction to build the state space within reasonable time and resource constraints. The same applies to the option Delayed Nondeterminism with Look Ahead.

(iii) The state space could only be built when enabling the options Delayed Nondeterminism with Look Ahead and Nondeterministic Program Status Word. These options drastically reduced the state space and consequently the run time.

(iv) Enabling static analysis additionally helps to mitigate the state-explosion problem. Especially the option Path Reduction is a great contributor to state space reduction.

(v) For the conducted case study the option Dead Variable Reduction leads only to a minor reduction of system states. This can be explained by the size of the source code. Static code analysis gets coarser with increasing source code complexity. Nevertheless, this result can be seen as an indicator that there is still vast room for improvements in the existing data-flow analyses.

(vi) Due to the implemented abstraction techniques for the Intel MCS-51 target, the state space could be reduced to a number that can easily be handled by conventional desktop computers. There is no need for a dedicated server in order to build the state space.

Note that the presented results are only valid for the investigated case study, actual savings of the individual abstraction techniques heavily depend on the source code structure, complexity, and the number of accesses to nondeterministic memory locations.
7 Remaining Challenges and Future Work

This section summarizes remaining challenges of the [MC]SQUARE approach and highlights future research possibilities. First, the problem of finding understandable counterexamples is highlighted. Next, the issue of verifying the simulator implementation is discussed and the idea of automatically generating simulators out of high level descriptions is presented. Then, the need of counterexample validation is clarified. Finally, coping with the state-explosion problem is discussed.

7.1 Local Model Checking and Resulting Counterexamples

Due to the local model checking algorithm implemented in [MC]SQUARE (cf. Section 3.4.6), the tool presents only a single counterexample. Depending on the actual implementation of the search algorithm that builds and traverses the state space, the presented counterexample is very likely not an optimal counterexample. Technically, an optimal counterexample is of minimum length, thus, the one with the smallest number of states. However, from the user point of view a practical counterexample is one which can easily be understood by the tool user. Finding such an understandable counterexample is a rather challenging task.

Enabling a variation of counterexamples requires either a global model checking algorithm or the possibility to continue the search for further counterexamples within the local model checking algorithm. At best, the tool generates a number of counterexamples and the user gets the possibility to study any of those counterexamples. Thus, the following conclusions are derived:

(i) Further research is needed in order to algorithmically find an understandable counterexample, or to define characteristics to assess the understandability of a counterexample.

(ii) In order to do so, the internal model checking algorithms of [MC]SQUARE have to be revised.

7.2 Getting the Intel MCS-51 Simulator Implementation Right

A major characteristic of the [MC]SQUARE approach is the handmade CPU simulator that serves for state space building. Generating a simulator of a modern, high performance
microcontroller is a challenging, lengthy, and error-prone task. Consequently, verification of the simulator is tricky, too. Whereas the instruction set part of the implementation can be easily verified against other commercial available target simulators (cf. Section 3.6.3), reasonable verification of the customized parts of the target simulator still remains an open issue.

It is fair to state that implementation errors residing in the simulator itself are very likely to be uncovered during model checking since the model checker urges the simulator to execute single instructions with a huge number of input configurations. As a result, bugs that stem from the simulator implementation are revealed by wrong counterexamples presented by [mc]square. This is an especially effective method to get the simulator bug-free in the early stages of an implementation. However, the following conclusions are noted:

(i) Strategies for handling the verification of the customized CPU simulator have to be proposed.

(ii) Derive methods for an automatic verification of the CPU simulator.

7.3 The Automatic Generated Target Simulator

Lowering the implementation effort for new microcontroller families is a major target of future research. The following approaches are taken into consideration:

(i) Derive new simulator models out of a high level description of the microcontroller, for example out of a behavioral hardware model.

(ii) Interfacing to an existing hardware model (e.g., RTL code and IP cores).

(iii) A generic simulator generator.

7.4 Counterexample Validation

As [mc]square uses a model of the actual target microcontroller, one has to make sure that the presented counterexamples are indeed real ones that occur in the field. The various abstraction techniques help to alleviate the state-explosion problem, however, they also introduce over-approximation, i.e., generate behavior that would not occur on the real target microcontroller within its working environment.

In practice, counterexamples are manually validated by the test engineer by executing the code at the target platform and trying to reconstruct the counterexample at the real life application. Thus, an automatic approach of proving a counterexample to be a real one might be the last missing part of a fully automatic embedded systems software formal verification process.

At the best of the author’s knowledge, no feasible approach has been adopted so far to automatically cross-check a given error trace on the real target hardware. Mercer and Jones [8] are using the GNU debugger for state space generation in their model checking approach. Nevertheless, their approach suffers from state-explosion and rather slow execution times.
Thus, an innovative approach for counterexample validation is needed. In the long run, it might be feasible to tie model checking to the root where all software errors are emerging from – to the hardware unit whose software is subject to verification itself. It might be promising to extend existing microcontroller IP cores in a way to support state space building and the automatic validation of counterexamples. However, the following conclusions are noted:

(i) Further research is needed to automatically validate a given counterexample on the real target hardware.

(ii) Evaluate possibilities of generating a customized IP core for state space generation and counterexample validation, based on available microcontroller implementations.

(iii) Strategies to contain massive over-approximations due to abstraction techniques are needed.

7.5 Coping the State-Explosion Problem

Although the available abstraction techniques (cf. Section 4.2 and 5.2) lead to tremendous state space reductions, the state-explosion problem is still one of the heavy-weighted challenges to overcome in order to scale up the [MC]SQUARE approach to huge code bases.

(i) Promote assembly code static analysis in order to obtain greater savings in the state space.

(ii) Focus on invariant model checking and try to disprove certain properties as early as possible by the preceding static analysis.

(iii) Consider further architectural peculiarities of the target microcontrollers and develop tailored abstraction techniques.
8 Conclusion

The main contribution of the present master thesis is the enhancement of the existing C51Simulator component of the [mc]SQUARE model checker. The Intel MCS-51 simulator is extended by (i) state-space abstraction techniques and (ii) integrated into the static analysis framework of [mc]SQUARE. Finally, (iii) a real life embedded systems application is formally verified with [mc]SQUARE by taking advantage of the implemented abstraction techniques.

Regarding (i), a novel abstraction technique, termed Delayed Nondeterminism with Look Ahead is proposed that when applied to formal verification of I/O intensive embedded systems assembly code is able to achieve a quite notable state space reduction. In particular, this approach helps to avoid the generation of successor states whenever a microcontroller executes logic operations. The presented approach centers around the coherence among the boolean operators $\land$, $\lor$, and $\neg$ with particular regard to 3-valued logic.

Regarding (ii), existing static analyses are adapted to the Intel MCS-51 architecture. Furthermore, a novel data-flow analysis, termed Register Bank Analysis is introduced. This new analysis is used to support static assembly code analysis within [mc]SQUARE. In particular, the approach leads to more precise Reaching Definition Analysis and Live Variable Analysis results, which allows the detection of additional dead variables. Thus, the number of overall system states is reduced during model checking. Typical data-flow analysis for high-level languages cannot be applied to assembly code one to one. Analyses such as Reaching Definition Analysis have to be adapted to be applicable to assembly code. The approach shows that it is necessary to take architectural peculiarities into account during the analysis to achieve precise results.

Regarding (iii), a real life industrial application is model checked by [mc]SQUARE. A specification in Computational Tree Logic (CTL) is derived out of a textual specification. It is possible to reveal an implementation error concerning the receiver state machine, responsible for decoding incoming data bytes from the serial interface. The found error is very likely to go unnoticed during traditional testing methods, since the erroneous behavior only shows up in the rare case the host application sends sequences in the form of $\{[2n]_{n>0}\times\$, ?, #\} to the target microcontroller, i.e., sequences with an even number of start bytes. It turns out that some of the system properties cannot be sufficiently specified in CTL, due to the lack of fairness in CTL. Unfair paths in the microcontroller program are determined and ruled out by taking advantage of a concept termed User Defined Environment (UDE). With UDE it is possible to introduce the required fairness constraints into the [mc]SQUARE model checking process. A solution to fix the existing implementation error is given and is proved to be correct in a further model checking run.

[mc]SQUARE proved to be a promising approach for model checking and static analysis of Intel MCS-51 assembly source code. It aims at a push-button formal verification approach of embedded systems code by relying on custom, highly optimized simulator components. When compared to traditional model checking approaches, the effort for the verification of a system or software is shifted from the user and a system model towards the model.
checking tool and the implementation itself.

Nevertheless, as recognized by Gerth in [3], the real challenge – besides all the technical issues that have to be solved – in formal verification lies in convincing the design teams that devoting some of their verification resources to formal methods leads to a higher design quality. Thus, the major future challenge is to move formal verification upstream in the embedded systems design flow. Contributors, such as ever shortening design cycles and stringent time to market requirements, strongly support the claim for formal verification even at very early design stages. It is about time to transform projects successful in research and academia into practical tools ready to be used within the day-to-day (embedded) software engineering practice.

To conclude, a vague and rather incomplete personal outlook on future trends in formal verification is given:

(i) The holy grail of full program verification has been abandoned - It will probably remain abandoned for the next years.

(ii) Less ambitious tools like [MC]SQUARE might emerge and become more widely used to formally verify sensitive parts of the application software.

(iii) Future tools will exploit ideas from various analysis disciplines, such as abstract interpretation, static analysis, and model checking.

(iv) Future tools will aim at alleviating the chicken-and-egg problem of writing specifications.
Bibliography


## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Formal verification methods classification.</td>
<td>7</td>
</tr>
<tr>
<td>3.2</td>
<td>CTL examples and intuitions.</td>
<td>10</td>
</tr>
<tr>
<td>3.3</td>
<td>The model checking workflow.</td>
<td>11</td>
</tr>
<tr>
<td>3.4</td>
<td>The coffee vending machine example.</td>
<td>13</td>
</tr>
<tr>
<td>3.5</td>
<td>The model checking workflow of the [MC]SQUARc approach (cf. Figure 3.3).</td>
<td>15</td>
</tr>
<tr>
<td>3.6</td>
<td>The [MC]SQUARE framework.</td>
<td>18</td>
</tr>
<tr>
<td>3.7</td>
<td>C51Simulator verification process.</td>
<td>19</td>
</tr>
<tr>
<td>3.8</td>
<td>Software architecture of the C51Simulator.</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Over- and under-approximation in abstraction [81].</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>Nondeterministic state space representation.</td>
<td>29</td>
</tr>
<tr>
<td>4.3</td>
<td>The Delayed Nondeterminism approach of handling the MOV [0xA, 0xB] instruction.</td>
<td>31</td>
</tr>
<tr>
<td>4.4</td>
<td>The state-explosion problem.</td>
<td>34</td>
</tr>
<tr>
<td>4.5</td>
<td>Successor state generation and resulting system states with options: instanti-ate immediately, Delayed Nondeterminism, and Delayed Nondeterminism with Look Ahead for the assembly code presented in Listing 4.3.</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>The resulting CFG for Listing 5.1.</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>Data-flow analysis.</td>
<td>46</td>
</tr>
<tr>
<td>5.3</td>
<td>LVA example.</td>
<td>50</td>
</tr>
<tr>
<td>5.4</td>
<td>The [MC]SQUARE static analysis framework for the Intel MCS-51 target.</td>
<td>52</td>
</tr>
<tr>
<td>5.5</td>
<td>The type hierarchy of the C51LVALatticeElement.</td>
<td>57</td>
</tr>
<tr>
<td>5.6</td>
<td>The type hierarchy of the C51LVBUILDER.</td>
<td>58</td>
</tr>
<tr>
<td>5.7</td>
<td>The type hierarchy of the C51RDA LatticeElement.</td>
<td>58</td>
</tr>
<tr>
<td>5.8</td>
<td>The type hierarchy of the C51RDABuilder.</td>
<td>59</td>
</tr>
<tr>
<td>5.9</td>
<td>Bit-wise modeling of the register bank selection pointer.</td>
<td>61</td>
</tr>
<tr>
<td>5.10</td>
<td>The join-operator and a simple CFG.</td>
<td>61</td>
</tr>
<tr>
<td>5.11</td>
<td>The corresponding CFG as generated with [MC]SQUARE for the assembly code in Listing 5.7.</td>
<td>65</td>
</tr>
<tr>
<td>5.12</td>
<td>The principle of PR.</td>
<td>67</td>
</tr>
<tr>
<td>6.1</td>
<td>The target application.</td>
<td>77</td>
</tr>
<tr>
<td>6.2</td>
<td>The knitting machine monitoring device.</td>
<td>78</td>
</tr>
<tr>
<td>6.3</td>
<td>The foreground/background design pattern.</td>
<td>79</td>
</tr>
<tr>
<td>6.4</td>
<td>The software components.</td>
<td>79</td>
</tr>
<tr>
<td>6.5</td>
<td>A software circular buffer model.</td>
<td>80</td>
</tr>
<tr>
<td>6.6</td>
<td>Communication sequence chart.</td>
<td>82</td>
</tr>
<tr>
<td>6.7</td>
<td>The unfair path.</td>
<td>93</td>
</tr>
<tr>
<td>6.8</td>
<td>The model checking workflow of [MC]SQUARE with UDE (cf. Figure 3.3).</td>
<td>94</td>
</tr>
</tbody>
</table>
6.9 A first UDE automata proposal ($\mathcal{U}1$).
6.10 The final UDE automata ($\mathcal{U}2$).
# List of Tables

3.1 Memory representation in [MC]SQUARE. .................................................. 20
3.2 ND memory representations and resulting value combinations. ................. 21
4.1 Data memory size and resulting system states. ........................................... 27
4.2 Comparison of abstraction techniques for the C51Simulator. ...................... 30
4.3 Memory contents before and after the MOV instruction. .......................... 30
4.4 Truth table for 3-valued logic. ................................................................. 32
4.5 How bitmasks are used in embedded software. ......................................... 33
4.6 Details on the Delayed Nondeterminism with Look Ahead approach. .......... 35
4.7 The ADDC [A, RO] example. ................................................................. 39
5.1 Results after solving data-flow equations for source code Listing 5.3. ......... 47
5.2 Results after solving LVA data-flow equations for source code Listing 5.4. .. 49
5.3 Action List Building – a few examples. .................................................... 57
5.4 Register bank configurations of the Intel MCS-51. .................................... 59
5.5 Evaluating $\text{kill}_{LV}(\pi)$ for MOV [RO, #const]. ............................... 60
5.6 Comparison of resulting live variables. .................................................. 66
6.1 Ringbuffer elements and their size. ....................................................... 81
6.2 The master-slave communication protocol. ............................................. 83
6.3 Case study variables and their meaning. ............................................... 84
6.4 Definitions for UDE modeling. ............................................................... 95
6.5 Case study results. ............................................................................... 100
6.6 Case study results for plain state space building. .................................... 104
## List of Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Algorithm Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A fixed point iterating algorithm to solve data-flow equations for the RDA problem [87].</td>
<td>51</td>
</tr>
<tr>
<td>2</td>
<td>A fixed point iterating algorithm to solve data-flow equations for the LVA problem [87].</td>
<td>52</td>
</tr>
<tr>
<td>3</td>
<td>CFG building algorithm for the Intel MCS-51 target.</td>
<td>55</td>
</tr>
</tbody>
</table>
Listings

4.1 Assembly code excerpt. .............................................. 28
4.2 Embedded C code example program for the Intel MCS-51 target. ..... 33
4.3 Translated assembly code for source code lines 4-5 of Listing 4.2. ..... 33
4.4 The Delayed Nondeterminism visitor pattern for the ANL [direct, #immediate] instruction. ........................................ 37
4.5 The Delayed Nondeterminism with Look Ahead visitor pattern for the ANL [direct, #immediate] instruction. ............................... 37
4.6 The Delayed Nondeterminism visitor pattern for the ADDC [A, R0] instruction. .................................................. 39
4.7 The Nondeterministic Program Status Word visitor pattern for the ADDC [A, R0] instruction. ........................................ 40

5.1 Source code used for CFG building. .................................. 44
5.2 RDA example code. ................................................... 46
5.3 RDA example code. ................................................... 47
5.4 LVA example code (cf. [81]). ........................................ 48
5.5 Code sharing within the program memory. .......................... 54
5.6 The Action List Builder visitor pattern for the ADDC [A, direct] instruction. .................................................. 57
5.7 Example assembly code. .............................................. 63
5.8 Intel MCS-51 assembly snippet. ..................................... 69
5.9 C source code containing switch statement. ......................... 70
5.10 Switch Statement Assembler code snippet. ....................... 71
5.11 Program memory content. .......................................... 71
5.12 Entry addresses for called functions. ................................ 72

6.1 Ringbuffer C code macro. ........................................... 80
6.2 The erroneous receiver state machine implementation. ............... 102
6.3 The revised receiver state machine implementation. ................ 103
List of Abbreviations

ACM Association for Computing Machinery
ASCII American Standard Code for Information Interchange
ASIC Application Specific Integrated Circuit
BDD Binary Decision Diagrams
CFA Control Flow Analysis
CFG Control Flow Graph
CISC Complex Instruction Set Computer
COTS Commercial Off The Shelf
CPU Central Processing Unit
CTL Computational Tree Logic
CTL* Computational Tree Logic*
DND Delayed Nondeterminism
DNDIA Delayed Nondeterminism with Look Ahead
DVR Dead Variable Reduction
ECM Electronic Control Module
FIFO First In First Out
FPGA Field Programmable Gate Array
FSM Finite State Machine
GNU GNU is not Unix
GUI Graphical User Interface
IC Integrated Circuit
IE Interrupt Enable
IFA Interrupt Flag Analysis
IP Intellectual Property
IRAM Internal Random Access Memory
ISR Interrupt Service Routine
LED Light Emitting Diode
LTL Linear Temporal Logic
LVA Live Variable Analysis
ND Nondeterministic
NDPSW Nondeterministic Program Status Word
PC Program Counter
PDAG Propositional Directed Acyclic Graph
PLC Programmable Logic Controller
POR Partial Order Reduction
PROMELA Process or Protocol Meta Language
PR Path Reduction
PSW Program Status Word
RAM Random Access Memory
RBA Register Bank Analysis
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDA</td>
<td>Reaching Definition Analysis</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>RPM</td>
<td>Revolutions Per Minute</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>SA</td>
<td>Stack Analysis</td>
</tr>
<tr>
<td>SDCC</td>
<td>Small Device C Compiler</td>
</tr>
<tr>
<td>SFR</td>
<td>Special Function Register</td>
</tr>
<tr>
<td>SIES</td>
<td>Symposium on Industrial Embedded Systems</td>
</tr>
<tr>
<td>STE</td>
<td>Symbolic Trajectory Evaluation</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver Transmitter</td>
</tr>
<tr>
<td>UDE</td>
<td>User Defined Environment</td>
</tr>
<tr>
<td>VHDL</td>
<td>Very High Speed Integrated Circuit Hardware Description Language</td>
</tr>
</tbody>
</table>