Formal Verification of Firmware-Based System-on-Chip Modules

Formale Verifikation von firmwarebasierten System-on-Chip-Modulen

Vom Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

von
M.Sc. Carlos Villarraga
geboren in Ibague, Kolumbien

D 386

Dekan:	Prof. Dr.-Ing. Hans D. Schotten
Vorsitzender der Prüfungskommission:	Prof. Dr. Gerhard Fohler
Gutachter:	Prof. Dr.-Ing. Dr. rer. nat. Wolfgang Kunz
	Prof. Dr. Rolf Drechsler
Tag der Einreichung:	26. Oktober 2016
Tag der Disputation:	7. Dezember 2016

Abstract

In current practices of system-on-chip (SoC) design a trend can be observed to integrate more and more low-level software components into the system hardware at different levels of granularity. The implementation of important control functions and communication structures is frequently shifted from the SoC's hardware into its firmware. As a result, the tight coupling of hardware and software at a low level of granularity raises substantial verification challenges since the conventional practice of verifying hardware and software independently is no longer sufficient. This calls for new methods for verification based on a joint analysis of hardware and software.

This thesis proposes hardware-dependent models of low-level software for performing formal verification. The proposed models are conceived to represent the software integrated with its hardware environment according to the current SoC design practices. Two hardware/software integration scenarios are addressed in this thesis, namely, speed-independent communication of the processor with its hardware periphery and cycle-accurate integration of firmware into an SoC module. For speed-independent hardware/software integration an approach for equivalence checking of hardware-dependent software is proposed and an evaluated. For the case of cycle-accurate hardware/software integration, a model for hardware/software co-verification has been developed and experimentally evaluated by applying it to property checking.

Acknowledgments

This thesis is the result of several years of intense and interesting research at the Electronic Design Automation group at the University of Kaiserslautern. Different persons have contributed to this work and I would like to thank them.

Firstly, I would like to express my sincere gratitude to Prof. Wolfgang Kunz for giving me the great opportunity to conduct this research under his supervision. I am truly indebted to him for his unflagging support and guidance during my research. I would also like to specially thank Prof. Dominik Stoffel for the valuable discussions and his guidance during the entire period of this research.

Many thanks to Prof. Rolf Drechsler for his interest in reviewing my thesis as well as for the important and helpful feedback. I also thank Prof. Gerhard Fohler for chairing my thesis committee.

I am very thankful to the German Academic Exchange Service (DAAD) for supporting and funding the first 3 and a half years of this research.

Special thanks go to Dr. Jörg Bormann for his collaboration and essential input at the beginning of my research.

Next, I would like to thank my colleagues Sacha Loitz, Binghao Bao, Oliver Marx and Christian Bartsch for the interesting discussions and fruitful collaborations. I am especially grateful with Bernard Schmidt, with who I closely worked while doing this research. The ideas of this thesis have been enriched during the nice and valuable discussions we had. I would like to thank Max Thalmaier for the time he spent proof-reading this thesis as well as Oliver Marx for helping me with the German summary.

Finally, I would like to thank Carmen Vicente-Fess, Matthias Legrom and Andreas Christmann for the good and kind support.

To my family

Contents

Chapter 1

Introduction

Embedded systems play today an integral role in nearly every aspect of modern life. They are employed in safety-critical products such as medical devices, vehicles, and airplanes, in manufacturing and security systems, as well as in consumer products such as mobile devices and home appliances. Thanks to the advances in semiconductor industry, embedded systems are commonly implemented by means of *system-on-a-chips* (SoCs). An SoC integrates into a single chip different *intellectual property* (IP) cores like processors, memories, communication buses, and communication interfaces.

Continuous development of embedded systems technology is driven by the intense product competition in system features and capabilities caused by a more and more demanding society. To be competitive, new designs must exhibit increases in functionality, performance, and reliability and declines in features like power consumption and size. As a result of this and supported by the advances in silicon technology, an increasing number of IP cores can be integrated into a single SoC. Likewise, the subsystems inside an SoC are becoming more and more complex. In many cases, a single IP block can by itself be considered as an embedded system. Current microprocessors, for instance, integrate apart form processor cores: caches, memory controllers, and communication interfaces [BC11].

Besides the use of high-integrated silicon systems, shrinking time-to-market windows of the highly competitive global electronics market places additional pressure on design teams. Increasingly complex systems need to be developed in decreasing amounts of time. In response to this, different design strategies continuously emerge that allow designers to tackle the challenges in SoC markets. In particular, in recent years the programmability of SoCs has continuously grown. Because of its inherent flexibility and reconfigurability, software offers an attractive solution to SoC developers.

Software-based solutions allow also to obtain products that exhibit increased functionality to the users. Because of this fact, in automotive and aerospace industries, for example, mechanical systems get continuously replaced or combined with electric and software-based components. As a result of this, the number of software-implemented control units in cars

and airplanes keeps growing across the new generations of models.

These trends do not only allow for creating application embedded software with growing complexity, but also increase the complexity of software at lower design levels [EJ09].

On the one side, the firmware in embedded systems becomes more complex in order to manage the increasing number of integrated units in an embedded system. Furthermore, the growing functionality of these units contributes to the complexity increase. For instance, complex device drivers are required to control and communicate with highly configurable hardware peripherals available in current SoC architectures.

On the other side, changes in design practices for SoCs at lower design levels contribute also to the increased programmability of embedded systems. Different chip-wide control functions of an SoC are no longer implemented in register-transfer level (RTL) hardware, but as firmware running on service processors that are instantiated particularly for this purpose [ZES13, WPL+12, CJ09]. Among others, control functions include chip (or system) initialization, power management, and the control of infrastructures for test and system diagnosis. Similarly, implementation of communication structures is shifted more and more from hardware to the low-level software of the system. As an advantage, a firmware-based design of these functions permits quick product updates and late engineering changes to the designs because firmware is much easier to change than RTL hardware.

These trends have created new interest in techniques for formal firmware verification, not only among software developers but also in the hardware design community [KG14, Gru13, WCGP12, Kro07]. Because of the tight coupling between firmware and hardware, verification techniques as they have been developed for application-level software are not always appropriate. Techniques are required that help to analyze the mutual effects of hardware and firmware on each other.

Motivated by these observations, this thesis tackles the problem of performing formal firmware verification by following a combined hardware/software analysis. The next section of this introduction presents integration scenarios of firmware and hardware in current architectures for embedded systems. Then, Section 1.2 gives an overview of currently reported techniques for formal SoC verification. Finally, Section 1.3 presents the motivation of this thesis and summarizes the main contributions of it.

1.1 Firmware Integration in SoC Designs

In the context of this thesis the term *firmware* is employed to refer to the part of the software in an embedded system that directly interacts with the system's hardware. Because of this fact, firmware will also be referred to as *hardware-dependent* software in the sequel without extra notice.

Firmware interacts closely with its hardware environment and the timing behavior of hardware/software interactions depend on how firmware and hardware are integrated into the

system. In the following, two main firmware integration scenarios present in modern SoCs are described.

1.1.1 Speed-Independent Integration Scenario

The first scenario is found in traditional SoC design flows. Processor cores are integrated into the hardware system as components of a CPU bus. They usually communicate with the rest of the system in a *speed-independent* way using some bus protocol with handshake mechanisms in order to accommodate for different access latencies. There exists a number of standard buses (cf. [ARM99, IBM99]) that can be instantiated in the design so that processors can be easily integrated with different IP cores in an SoC.

Speed-independent communication is key since the execution time in pipelined processors, especially when advanced architectures based on out-of-order execution are employed, is difficult to predict. Similarly, caches have a difficult-to-predict timing behavior and provide another reason for speed-independent bus communication.

In this scenario, a fine-grained timing analysis of the software is not performed. Instead of that, techniques for worst-case-execution-time (WCET) analysis [WEE+08] are employed to ensure that the software, executed on its hardware platform, responds in the correct time limits.

1.1.2 Cycle-Accurate Integration Scenario

Besides conventional design styles where firmware and hardware are integrated by employing CPU buses, there are also new design approaches for which a clock *cycle-accurate* analysis is required. This introduces a second scenario which is described in the following.

When designing SoCs it has become increasingly popular to replace dedicated RTL hardware components by a firmware-based design [ZES13, WPL+12, CJ09]. For this purpose, service processors in addition to the main processor are instantiated, implementing the sub-functions that were formerly performed by the hardware component. Commonly employed processors have a timing behavior that is fully predictable. For example, processors in the style of the Intel 8051 or the Xilinx PicoBlaze are popular in ASIC-based and FPGA-based design flows, respectively.

The firmware executed on its hardware platform implements a finite-state machine (FSM) that conceptionally behaves just like a pure RTL hardware design. However, the implemented control structure is not fixed in hardwired state transitions, but is embodied in firmware executed by the processor. Control-flow decisions occur in branch instructions and references to the hardware are explicit in load and store operations. The control of the surrounding hardware is not done in a speed-independent way but performed cycle-by-cycle by transactions generated by the firmware. This firmware-based design approach offers

advantages:

1. The design time is reduced and product updates can be made more easily by making changes in the firmware.

2. Especially for FPGAs, due to a well-optimized design of the processor, the resulting implementation may need less chip area when compared to a conventional implementation with standard RTL hardware.

The software running on the instantiated cores is usually not meant to be visible to the users. It is provided as firmware with the design and may be loaded into a read-only memory. In many cases, the cores are not directly connected to the rest of the system using (standardized) communication interfaces like SoC buses but instead are embedded into the surrounding system using special interface hardware, sometimes called "wrapper RTL" [Xil11a, WCGP12]. Such design styles allow for a tight integration delivering high performance because the exact timing of the processor hardware and its software is known at design time. A firmware-based SoC module designed in this way is shown in Figure 1.1. The module consists of two processor cores tightly integrated with their firmware, wrapper RTL, and some additional hardware.

1.2 State of the Art in SoC Formal Verification

This thesis does not intend to extend the basic proving methodology for formal verification. It mainly leverages the state of the art in this field and focuses on how to integrate low-level software components (firmware) into the computational procedures needed for a combined hardware/software verification. Therefore, computational models and algorithms for low-level hardware-dependent software verification are in the particular interest of this thesis.

Formal methods for hardware verification and for software verification have been active research fields for decades. In the following overview, the focus is kept on (1) those techniques that help to the development of the intended research and on (2) those contributions that are related to the topics of this thesis.

First of all, this thesis is based only on fully automated proof techniques which have been successfully adopted and extended by the Electronic Design Automation (EDA) industry over the last decades. This precludes the use of sophisticated methodologies based on higher-order logic and theorem proving [Har09]. In spite of advances in automation of the proofs, these methodologies still require highly skilled verification engineers that guide the verification process interactively. This added manual effort makes less attractive the practical adoption of such methodologies.

Formal verification methods in general employ mathematical models for describing the *design under verification* (DUV) and the proof goal (the specification). From this description

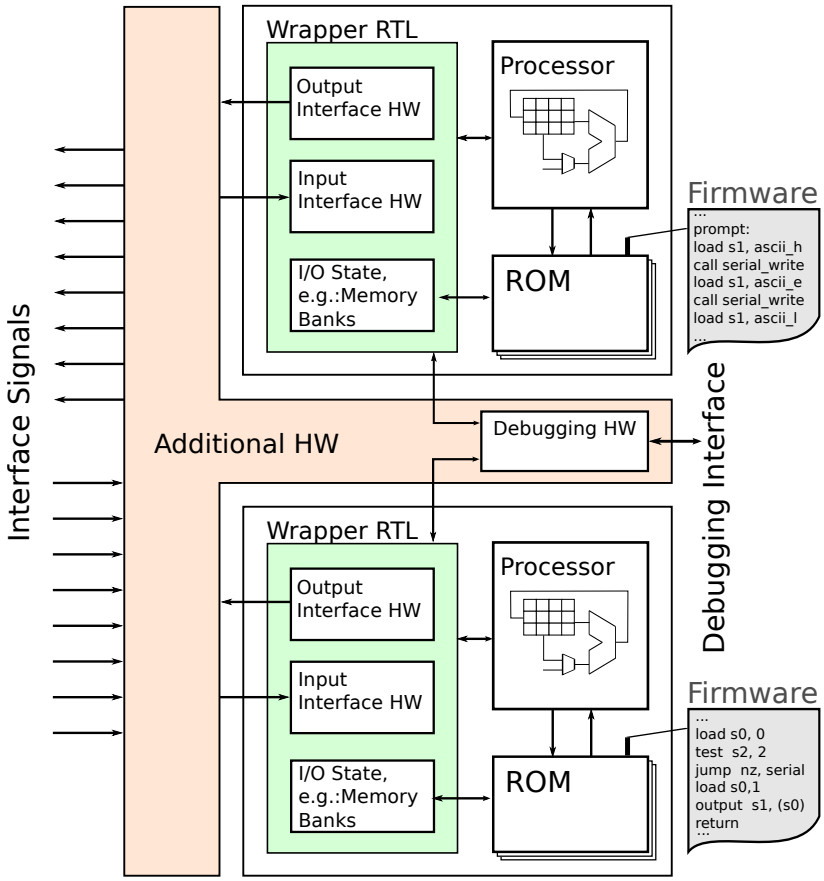

Figure 1.1: Firmware-based IP core

then a formal checker exhaustively inspects whether the proof goal holds or not for all possible input assignments. If the proof does not hold, then a counterexample (e.g., an input assignment for which the proof goal does not hold) is extracted and provided to the user so that the cause of the failure can be derived from it.

There are two main practical applications of formal verification methods, namely *property checking* and *equivalence checking*. In property checking, the DUV is proven to determine whether it complies with the intended behavior or not. So-called *properties* are used in the verification process to formalize the design specification. For its part, equivalence checking is used to prove whether two designs are functionally equivalent or not. The rest of this section gives more details on current reported formal verification techniques by property and equivalence checking for both hardware and software.

1.2.1 Formal Hardware Verification

A possible classification of formal property checking techniques employed for hardware is to distinguish between a *bounded* and an *unbounded* paradigm. In the unbounded paradigm, techniques reason about behaviors of the system over infinite lengths of time. This paradigm is rooted in classical model checking [CE81] based on temporal logics. Numerous extensions to the basic scheme have been proposed to increase its computational power. Based on Binary Decisions Diagrams (BDDs) [Bry86], symbolic methods have been developed to traverse the state space [McM93, CBM89]. The state explosion problem is one of the biggest challenges in this area and is often addressed by automatic abstraction techniques such as [CGJ+03, GS97, JKSC05] that over-approximate the state space. This is related to the problem of finding invariants to identify appropriate state space approximations, e.g., [CHM+96, CNQ04]. Compositional techniques such as [CLM89] have been developed to handle large designs in a divide-and-conquer strategy.

In the bounded paradigm, always a *finite time interval* is considered when formulating a property to describe a piece of design behavior. Bounded Model Checking (BMC) [BCCZ99] and Interval Property Checking (IPC) [NTW+08] are representatives of this paradigm and have in common that the underlying computational model is obtained by unrolling finite state machines for a finite number of steps into an entirely combinational circuit model. This allows for mapping the property checking problem to the Boolean satisfiability problem (SAT) [BHvMW09]. Thereby, verification based on bounded models has benefited from the significant advances in SAT solving technology in the last decades [JLBRS12, KSMS11].

K-step induction [SSS00] can be used to extend verification based on bounded models for proving safety properties also over unbounded time windows. More recent research also explores property checking based on SAT with interpolants [McM03] and property directed techniques [Bra11] to efficiently capture adequate information about the state space. Also, in the bounded paradigm, invariants can play a key role in obtaining general proofs for relevant

parts of the system behavior, e.g., [BC00, TNW+10]. The restrictions resulting from the bounded nature of the computational model have motivated sophisticated methodologies, such as [Cla07, BBM+07], to obtain a global correctness proof for the system.

While the unbounded paradigm is usually adequate to handle systems with no more than a few hundred state variables, the bounded approach is often successfully applied to designs with thousands of state variables. This makes it particularly attractive for industrial use. Current solutions have been able for instance to replace simulation of large hardware modules in industry [KGN+09].

Most of these techniques and their numerous extensions are tailored to be applied to pure hardware descriptions, i.e., they operate on Boolean networks and conventional finite state machine descriptions which are then converted into the required computational models such as those based on Kripke structures [Kri63]. Rooted in classical temporal logics such as CTL [CE81] and LTL [Pnu77], standardized property specification languages for hardware are available such as SystemVerilog Assertions (SVA) [Spe08] and Property Specification Language (PSL) [Acc04].

Equivalence Checking on its part has been widely adopted in industry and has completely replaced logic simulation at the gate level in modern SoC design flows. This success is based on two main facts:

First, design transformations performed by RTL synthesis tools normally preserve the state encoding of the implemented sequential circuit. This allows to reduce the general sequential equivalence problem to a pure combinational one. In practice, the problem is solved using a computational model called *miter* [Bra93] containing: the logic for the reference (golden) and the implemented (revised) circuits, mappings for the primary inputs and outputs as well as mappings between the latches of the designs [vEJ95]. As a result, no space traversals are required and therefore the required proofs become much easier.

Second, synthesis tools perform only local changes to the reference design. Hence, the main structure between the circuits being compared is kept. Equivalence checkers leverage this fact by identifying and pruning out internal equivalences (cut points) from the compared circuits [KK97, JMF95, Kun93].

Besides this, current solvers integrate several engines such as BDDs, SAT solvers, and Automatic Test Pattern Generation (ATPG) to achieve increased proof capabilities.

Due to these facts, current equivalence checkers scale up to verification of the whole SoC RTL hardware. For problems where changes in the state encoding are performed, the works of [vE00, SWWK04] have adapted the combinational approach achieving, to some extent, similar benefits.

1.2.2 Formal Software Verification

There is a large research body on methods for formal software verification. In the following, formal software verification techniques are classified into *hardware-independent* and *hardware-dependent* approaches. Although this thesis is focused on a hardware-dependent software view, methods also from hardware-independent techniques are useful and therefore are covered by this overview.

In hardware-independent software models, the software is often described by "simple programs" (cf. [JM09]) as some kind of (finite) state transition system that is processed by model checking and related techniques. Programs are typically given in high-level languages like C. Also here it is possible to distinguish between methods adopting an unbounded paradigm [BPR01, BHJM07, God05, HP00, Hol97] and a bounded paradigm [BH08, IYG+04, CKY03]. Differences between tools and methods result from the underlying proof methods (enumerative (stateful, stateless), symbolic) and the employed abstraction techniques (iterative abstraction-refinement based on localization reduction and/or predicate abstraction).

In [BPR01, BHJM07] properties of C programs are verified by performing a systematic predicate abstraction [CGJ+03] of the software. While performing a given proof, the abstraction gets refined until a valid answer is returned by the verification algorithm. Unbounded model checking is employed in these approaches for traversing the state space of the resulting finite-state model.

The tools of [IYG+04] and [CKY03] perform bounded model checking of C programs. In [IYG+04], the transition relation of a C program is obtained by performing optimizations such as basic-block extraction, one-hot encoding of program locations, and bit-width reductions by static range analysis. The obtained transition system is then syntactically unrolled following the approach of [BCCZ99]. Additionally, control flow information is provided to the SAT engine in order to improve scalability of the method. In [CKY03] a different approach is taken that unrolls the C program by unwinding its control flow graph. This combines benefits of conventional BMC with path-oriented techniques employed in symbolic execution.

Symbolic execution is a technique widely used for software verification and testing [Kin76]. Good overviews of this technique are given in [PV09] and [CS13]. Symbolic execution traces symbolically the individual execution paths of a program to explore the program behavior. With the goal of avoiding path explosion, techniques have been developed to prune as well as to merge execution paths. In the context of software testing, current approaches combine symbolic and concrete executions in order to obtain test suites with increased coverage [GKS05, CGP+06]. Also, different traversal algorithms have been proposed for improving path coverage [GKS05, SMA05].

In this thesis, some ideas from symbolic execution are borrowed to develop a computational model for software that can be integrated as a component into a hardware description.

When examining the impact of software on the concrete hardware, as is required in firmware designs, the software must be examined at a hardware-dependent level. Literature on hardware-dependent low-level software verification and formal hardware/software co-verification is much sparser than the literature for hardware-independent software.

Previous work on low-level software verification includes [Sch10] which employs explicit unbounded model checking algorithms in order check assembly code against properties specified with computation tree logic.

Different approaches based on symbolic execution have been also proposed. The work of [CFF+06] performs equivalence checking of digital signal processing algorithms by employing symbolic execution together with SMT solving (using the theory of uninterpreted functions with equality). Similarly, in [AEF+05, AEO+08] symbolic execution has been successfully employed to perform microcode verification. In these approaches, due to the explicit enumeration of the individual program paths, as pointed out in [AEF+05], analyzing the reactive behavior of low-level embedded software with its hardware periphery becomes very complex. Unlike in many cases of hardware-independent verification, the analysis can no longer be localized to an individual path but the contribution of all possible execution paths must be considered simultaneously. This typically leads to restrictions on the hardware/software interfaces that can be modeled. The work of [CFF+06], for instance, restricts the formulation of comparing two assembly programs to programs with very similar control flow graphs (CFGs) that can communicate with the environment only at the beginning and at the end of the execution. [AEF+05] restricts to programs that communicate with the environment only at specific exit points (no intermediate interactions with the environment are handled by the approach).

The tool of [CEP00] assumes the manual creation or the availability of an abstract automaton or petri-net model for hardware and software components. These models are disconnected from the actual implementation so that a significant amount of additional efforts would be needed to use such an approach in a standard design flow for firmware designs employing either ASICs or FPGAs.

In [HTV+13] it is proposed to model the combined hardware/software system in terms of C programs. The approach is based on manually extracting models for the hardware from virtual prototypes in C. This level of abstraction is appropriate for performing early verification of hardware/software systems during design exploration. However, for verifying the impact of the software on concrete implementations of hardware/software designs, the modeling task needs to be performed at a different level of abstraction. Otherwise, lifting such high-level models from the design would again require significant additional efforts for the verification.

[GKD06] and [EES04] are based on unrolling the programs by representing each program step as an instance of the processor's (RTL) hardware. These approaches allow for a tightly coupled view on both the hardware and the software, but lead to highly complex models that

only allow for local examinations of the combined system behavior. In order to improve scalability, specific abstraction mechanisms have been proposed in [NWSK11]. However, a main drawback of the latest approach is that the required abstractions need to be provided by the verification engineer for each analyzed program.

In conclusion, there is a large number of formal techniques tailored for verification tasks in the hardware and in the software domain. Many of the basic algorithmic concepts reported in previous work have the potential to contribute to a verification environment for firmware designs in SoCs. However, there is a substantial lack of adequate models and compositional techniques that allow for a joint analysis of the hardware and software components at the appropriate levels of abstraction.

1.3 Motivation and Thesis Overview

Traditional techniques for formal software verification usually adopt a hardware-independent view when verifying software programs written in high-level languages such as C. This is reasonable for a wide range of applications where the main objective is to identify bugs that are specific to the software development process. However, in embedded system design, as a result of the trends described above, it is important to analyze the mutual effects of hardware and software on each other. Therefore, a hardware-dependent software view is needed where the behavior of the firmware is precisely described in terms of its effect on the underlying hardware.

In this thesis, the software is modeled by using program netlists. A program netlist (PN) is a combinational model that compactly represents the behavior of a low-level embedded program in terms of the hardware on which it executes. Following a path-oriented modeling approach, in a program netlist the software behavior is represented along execution paths. More specifically, the program's computation is implicitly represented in a compact and hardware-dependent way for all paths of execution. This is opposed to symbolic execution where the expressions for the state variables and the path conditions are explicitly generated for every simulated path. Like symbolic execution, the approach using program netlists is also based on the enumeration of execution paths. However, this is shifted to a pre-processing phase that neglects all of the program's computation that is not relevant to its control flow. No formulas representing the computation along the paths are generated. Instead, a program netlist includes additional logic which not only simplifies the model but also makes relevant control flow information explicit to the decision procedure (e.g., a SAT solver). This facilitates the reasoning on the model. When solving the actual verification task on the program netlist, the "intelligence" of a SAT solver is used to exploit the additional elements of the computational model to traverse the execution paths with their associated hardware-represented computation in an efficient way.

Chapter 3 details on the characteristics of the program netlist model and on its generation process. This includes modeling aspects such as efficient representation of the data traffic between software and data memory as well as the traffic between software and the surrounding environment. The developments presented in this chapter have been conducted jointly with the dissertation work of Dipl.-Ing. Bernard Schmidt.

Subsequently, in Chapter 4, it is presented how generation of program netlists can be efficiently implemented so that the method scales for practical designs. For that, simulation techniques are mixed with formal SAT-based analyses for finding reachability information of the firmware's control flow as well as for computing the set of memory addresses that are accessed by the software. This information turns out to be crucial for performing simplifications not only in the program netlist, but also in models derived from it.

One important characteristic of the program netlist is that it can be instantiated and combined with other models in order to solve different verification problems. Taking advantage of that, this thesis proposes how formal verification of firmware designs can be performed for the hardware/software integration scenarios described in Section 1.1.

In Chapter 5 an approach for formal verification by equivalence checking for speed-independent integration scenarios (cf. Section 1.1.1) is proposed. For this scenario, assuming the correct implementation of the CPU bus protocol (as can be verified by standard techniques of formal hardware verification [NTW+08]), it is possible to model the software in a time-abstract way. This is exploited in the program netlists of Chapter 3 by creating time-abstract descriptions. In this way, even for complex processor architectures compact models can be obtained.

It is important to note that even though the concrete timing, in terms of HW clock cycles, is abstracted away from the program netlist, the original ordering of the instructions during execution is preserved in the model. This characteristic of the program netlist is particularly important when analyzing the hardware/software interface of reactive programs. Reactive software communicates with the environment continuously at distinct time points and the ordering in which the exchange of information takes place is crucial for the functional correctness of the system behavior. For example, for the case of equivalence checking, as will be shown in Chapter 5, it needs to be proven that two different programs interact in the same way with the environment. Therefore, verification needs to consider not only the data values exchanged with the environment but also the ordering of the data exchange.

Apart from equivalence checking, also property checking for time-abstract scenarios using program netlists has been researched in [SVF+13b].

For firmware-based design approaches for which a clock cycle-accurate analysis is required (cf. Section 1.1.2) extensions to the program netlist are presented in Chapter 6 in order to perform verification by property checking.

For firmware-based design styles, verification is important because the hardware/software interface is usually custom-designed and, thus, error-prone. As pointed out in [WCGP12],

traditional verification approaches based on instruction set hardware/software co-simulation would never fully capture the entire hardware and firmware system behavior in one tool environment. Verifying the firmware in isolation, while possible, would require a hardware bus-functional model interface. This makes the simulation of such firmware-based IPs complicated and creates the need for an additional behavioral test bench [WCGP12]. Similarly, also a formal approach that verifies hardware and software in separation would require the tedious task of modeling the interface between them by a set of constraints.

Therefore, this thesis proposes a formal co-verification approach instead. The program netlists of Chapter 3 are very attractive for this purpose because they can be generated completely automatically. On the other hand, due to their abstract, non-cycle-accurate nature, they cannot be directly integrated into the RTL descriptions of the hardware and they do not allow for a cycle-accurate analysis. In Chapter 6, therefore extensions are presented to make program netlists cycle-accurate and show how to create a joint model for formal co-verification of hardware and firmware by property checking. The developments presented in Section 6.2.3 have been conducted jointly with the dissertation work of M.Sc. Michael Schwarz.

Chapter 2 details the techniques on which this research is based. Notations and basic definitions are given in Appendix A. Chapter 7 concludes this thesis summarizing the proposed approaches and the results obtained as well as describing the future uses of the computational models presented in this work.

1.4 Publication List

Large parts of this thesis have been already published in the publications listed chronologically below:

1. Bernard Schmidt, Carlos Villarraga, Jörg Bormann, Dominik Stoffel, Markus Wedler, and Wolfgang Kunz. A computational model for SAT-based verification of hardware-dependent low-level embedded system software. In *Proceedings of the 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 711–716, 2013

2. Bernard Schmidt, Carlos Villarraga, Thomas Fehmel, Jörg Bormann, Markus Wedler, Minh Nguyen, Dominik Stoffel, and Wolfgang Kunz. A new formal verification approach for hardware-dependent embedded system software. *IPSJ Transactions on System LSI Design Methodology (Special Issue on ASPDAC-2013)*, 6:135–145, 2013

3. Carlos Villarraga, Bernard Schmidt, Christian Bartsch, Jörg Bormann, Dominik Stoffel, and Wolfgang Kunz. An equivalence checker for hardware-dependent software. In *11. ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 119–128, 2013

4. Christian Bartsch, Carlos Villarraga, Bernard Schmidt, Dominik Stoffel, and Wolfgang Kunz. Efficient SAT/simulation-based model generation for low-level embedded software. In *17. GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 147–157, 2014

5. Carlos Villarraga, Bernard Schmidt, Binghao Bao, Rakesh Raman, Christian Bartsch, Thomas Fehmel, Dominik Stoffel, and Wolfgang Kunz. Software in a hardware view: New models for HW-dependent software in SoC verification and test (invited paper). In *Proc. International Test Conference (ITC'14)*, 2014

6. Binghao Bao, Carlos Villarraga, Bernard Schmidt, Dominik Stoffel, and Wolfgang Kunz. A new property language for the specification of hardware-dependent embedded system software. In *Proc. Forum on Specification And Design Languages (FDL)*, Munich, Germany, Oct 2014. (accepted for publication)

7. Binghao Bao, Carlos Villarraga, Bernard Schmidt, Dominik Stoffel, and Wolfgang Kunz. A new property language for the specification of hardware-dependent embedded system software. In Frank Oppenheimer and Julio Luis Medina Pasaje, editors, *Languages, Design Methods, and Tools for Electronic System Design*, volume 361 of *Lecture Notes in Electrical Engineering*, pages 83–100. Springer International Publishing, 2016

8. Carlos Villarraga, Dominik Stoffel, and Wolfgang Kunz. *Formal System Verification*, chapter Software in a Hardware View: New Models for HW-dependent Software in SoC Verification. Springer (to appear), 2016

9. Christian Bartsch, Niko Rödel, Carlos Villarraga, Dominik Stoffel Stoffel, and Wolfgang Kunz. A HW-dependent software model for cross-layer fault analysis in embedded systems. In *17th Latin-American Test Symposium (LATS)*, pages 153–158, April 2016

10. Oliver Marx, Carlos Villarraga, Dominik Stoffel, and Wolfgang Kunz. A computer-algebraic approach to formal verification of data-centric low-level software. In *14. ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE) (to appear)*, 2016

Chapter 2

Background

This chapter gives a short introduction into different verification techniques and algorithms used in this thesis. It is not the goal of this introduction to be comprehensive, its purpose is to recall the most important concepts and terminologies so that the reading of the subsequent chapters becomes easier for the reader. For a comprehensive discussion, the reader will be provided, in each section of this chapter, with the corresponding literature.

For an introduction to basic mathematical definitions and notations used in this thesis the reader is referred to the Appendix A.

2.1 The Boolean Satisfiability Problem

Many problems in the field of electronic design automation can be mapped to the Boolean satisfiability problem (SAT). In this thesis, verification problems for property and equivalence checking are reduced to instances of the SAT problem. Therefore, this section introduces the SAT problem and summarizes how this problem is currently solved in practice by current techniques. For a deeper treatment, readers are recommended the following reference [BHvMW09].

2.1.1 The Satisfiability Problem

Given a Boolean formula $f(x_1, \ldots, x_n)$, the *Boolean satisfiability problem* poses the following question: is there an assignment to the variables $x_1, \ldots, x_n$ under which f evaluates to *true*?. If the answer is 'yes', f is said to be *satisfiable* and the involved variable assignment is called a *satisfying assignment*. If the answer is 'no', i.e., if no such assignment exists, f is said to be *unsatisfiable*.

Due to efficiency reasons, Boolean formulas are usually represented in conjunctive normal form (CNF) for solving the SAT problem. A CNF formula is satisfied under a given assignment if each of the individual clauses composing it are satisfied. Furthermore, a single

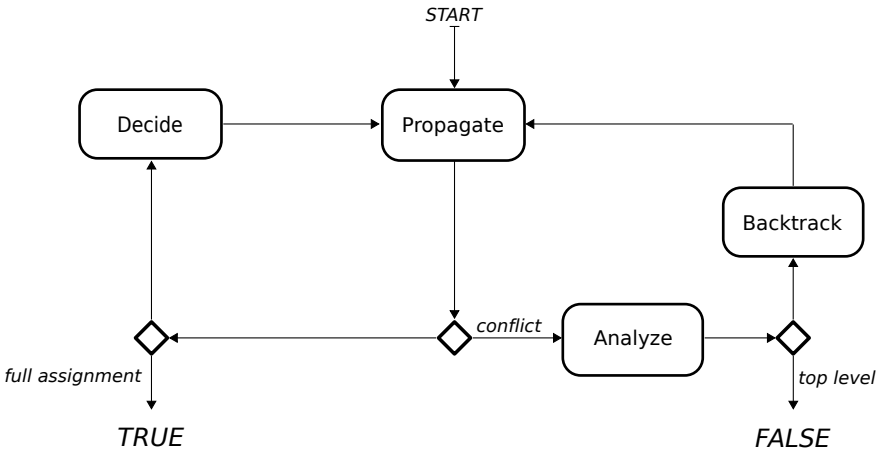

Figure 2.1: Basic structure of a conflict driven SAT solver

clause is satisfied if at least one of its literals evaluates to *true*. On the other hand, a CNF formula is unsatisfied under a given assignment if at least one of its clauses evaluates to *false*. A clause that evaluates to *false* is called a *violated* or a *conflicting* clause. A clause is violated if each of its literals evaluates to *false*.

If every clause of a CNF formula contains at maximum two literals, then the SAT problem can be solved in polynomial time [AB09]. However, for formulas not fulfilling this condition, as it happens with most real-life formulas, the SAT problem is NP-complete and therefore it can be solved in the worst-case in exponential time [Coo71]. Despite this fact, in the last couple of decades, there have been significant advances in developing efficient SAT solvers capable of solving large practical problems in reasonable amounts of time [JLBRS12].

2.1.2 SAT Solvers

Modern SAT solvers work on the basis of the DPLL algorithm [DLL62, DP60]. Broadly speaking, for finding a satisfying assignment the DPLL algorithm assigns one by one the variables of a given CNF formula. Every time a variable gets assigned, the formula is updated by evaluating the logical implications of the assignment. If after an assignment all clauses are satisfied then the algorithm finishes returning *true*. However, if one or more clauses are violated then the algorithm corrects the bad assignment by backtracking. Backtracking moves back to the last assignment and reverts it. Since the algorithm moves back to the last assignment this kind of backtracking is referred to as *chronological backtracking*. After backtracking, the variable that has been wrongly assigned is assigned with the opposite truth value and the algorithm continues. If it happens that clauses are violated for both possible truth assignments then the algorithm backtracks again and repeats. The DPLL returns *false* in case that all variable assignments have been exhausted without success.

In the last decades, the basic DPLL algorithm has been improved to the point that current

SAT solvers are capable of solving problems having millions of variables in practical amounts of time [JLBRS12]. However, there are still problems for which modern SAT solvers run in exponential time. Typical cases are problems containing arithmetic units such as multipliers.

Figure 2.1 presents the core structure of a modern SAT solver. It is composed of one main loop with two branches inside it. The branch on the right is executed when conflicts are produced. On the other side, the left branch is executed when assignments cause no conflicts. Modern SAT solvers implement the engines *Propagate*, *Decide*, *Analyze* and *Backtrack* adopting a number of techniques which have become standard during the last decades. In [KSMS11] an experimental evaluation demonstrates the effectiveness of these techniques. In the following a brief description is given:

- *Analyze* performs a diagnosis of the reason for a conflict based on the variable assignment and the logical implications which activate the conflict [MSS99]. As a result of the diagnosis a so-called *conflict clause* is generated. A conflict clause represents a more concise variable assignment which also activates the conflict. [ES03b] performs further improvements by eliminating redundant literals from the conflict clause. Conflict clauses are added to the CNF formula by *Analyze*. Adding conflict clauses to the CNF is referred to as *learning*. Learning prevents the solver of repeating wrong assignments. In practice, learning has shown to increase importantly the performance of the solver [KSMS11]. Furthermore, from a conflict clause it can be also derived the level to which the solver needs to backtrack. If the backtracking level corresponds to the so-called *top level*, the solver finishes returning *false*.

- *Decide* is responsible for selecting and assigning variables. This process is critical for the performance of a solver [KSMS11]. In [MMZ$^+$01] a selection heuristic based on literal activities is proposed. This strategy favors selection and assignment of literals which try to satisfy first clauses related to the most recent conflicts. Low computational overhead is added to the solver since only literals related to current conflict clauses need to be updated. In [ES03b] overhead is further reduced by associating activities to variables. A possible problem with this approach is that the solver can get stuck in deep search regions without a solution. Techniques based on restarts have been proposed in [Bie08, ES03b] to counter effectively this problem.

- *Propagate* is called to evaluate all logical implications due to a variable assignment. The overall performance of the solver depends largely on the efficiency of the *Propagate* engine since a SAT solver spends most of its run time propagating implications [KSMS11]. In [MMZ$^+$01] a technique based on watching literals by means of pointers is proposed. More specifically for every clause only two literals which are not assigned to *false* are watched. This reduces importantly the bookkeeping of the propagation process since assignments to literals that are not being watched are not

updated. Furthermore, the state of a clause is only updated if the two pointers coincide, i.e. if the clause becomes unit.

- *Backtrack* goes back to the level estimated by *Analyze*. Technically, backtracking is done until the level at which the conflict clause becomes unit. This kind of backtracking is known as *non-chronological backtracking* [MSS99]. Non-chronological backtracking outperforms chronological backtracking as in practice conflicts are caused by assignments performed many levels before and not by the previous assignment. During backtracking, the solver reverts all intermediate assignments and their corresponding implications. For this, the solver benefits from the fact that only non-false literals are tracked and therefore there is no necessity to relocate pointers while backtracking.

SAT solvers integrating these engines are referred to as *conflict-driven clause learning* (CDCL) solvers. State-of-the-art CDCL solvers incorporate different additional techniques such as pre-processing of CNF formulas [JHB12] and rapid restarts [BA15] to increase the overall performance. With these improvements, CDCL solvers can currently handle instances (derived from real-world problems) containing up to tens of millions of variables and clauses [HS15, JLBRS12].

SAT solvers return by default only a single satisfying assignment for CNF formulas that result satisfiable. There are problems however where the set of all satisfying assignments is required. This problem is known as the *All-solutions SAT problem* (All-SAT) [BHvMW09]. In [McM02], an approach based on so-called *blocking clauses* is proposed. It adds incrementally blocking clauses to the clause set of a given problem to avoid getting satisfying assignments that were already found. All-SAT solvers [GSY04, YSTM14] are commonly built on the top of CDCL solvers and employ incremental techniques (cf. next section) to improve performance.

Incremental SAT

Many problems in formal verification require to solve a sequence of related SAT problems [Sht01, ES03a, CLM+10]. Instead of solving each problem independently, incremental SAT solving seeks to propagate useful knowledge collected across the proofs [Hoo93]. In this way, a given proof in the sequence can benefit from information that has been previously learned. A secondary advantage is that the clause set that is shared among the SAT problems does not need to be parsed over and over again. There are a number of incremental SAT solvers such as the ones presented in [Bie08] and [ES03b] that implement efficiently this idea.

After solving a given instance of the SAT problem, an incremental SAT solver allows solving a new SAT problem by adding clauses to the former instance. For solving the new problem, the solver keeps conflict clauses that have been learned in the previous proof(s). These learned conflict clauses then may help to avoid conflicts when solving the new problem.

In practice, mechanisms need to be included that allow a SAT solver to be further called independently of the last performed proof. For this, so-called *assumptions* are used to represent special clauses that do not belong to the general SAT problem. For unsatisfiable instances, implications due to assumptions can be discarded so that the state of the solver, containing among others conflict clauses, can be further used for future calls without the need of a restart.

2.1.3 Converting Circuits into CNF

A combinational circuit can be encoded by a CNF formula using the procedure presented in [Tse68]. For a given circuit, this procedure does not represent directly the output of the circuit. Instead of that, it represents a function that evaluates to *true* only for assignments that are consistent with the circuit (i.e. it represents the characteristic function of the circuit). This ensures that the obtained CNF preserves the satisfiability with respect to the circuit functionality.

For encoding a given circuit, the procedure represents each gate in a (multi-level) combinational circuit as a CNF containing a fixed set of clauses. During the conversion, new extra variables are used to represent the valid assignments of the gate. The resulting CNF is obtained by conjuncting the CNFs of all gates. As a result, the final formula is linear in the size of the circuit.

2.2 Verification Based on Bounded Models

Verification based on bounded models examines the validity of a property $\varphi_l(\pi_l)$ for a set of finite paths $\pi_l = (s_0, s_1, \ldots, s_l)$ in a finite-state transition system, where l is the length of the paths. In general, the paths may also have different lengths within a finite interval of length l. These finite paths all begin in a specific set of starting states and all end in a specific set of ending states, where each state is characterized by appropriate Boolean state predicates. In order to formulate the verification problem we consider:

- $T(s, s')$: the characteristic function of the transition relation of the finite state system.
- $ispath(\pi_l) = \bigwedge_{i=1}^{l} T(s_{i-1}, s_i)$: a Boolean predicate obtained by unrolling the transition relation into l *time frames*. $ispath(\pi_l)$ is *true* if π_l is a valid path, i.e., the characterized state sequences can actually be traversed in the concrete transition system.
- $X_C(s_0)$: a Boolean state predicate of C that imposes the starting state(s) of π_l.

In Bounded Model Checking [BCCZ99], C corresponds typically to the set of initial states (I) of the transition system. In Interval Property Checking [NTW+08], $X_C(s_0)$ does

not constrain the proof to a specific set of reachable starting states. It corresponds to a predicate characterizing an invariant (W) that may only rule out unreachable states at s_0. In the most simple case, the Interval Property Checker may choose $X_W(s_0) = true$, i.e., W is the set of all possible states.

Checking the property $\varphi_l(\pi_l)$ is then reduced to a tautology check of the formula $X_C(s_0) \wedge ispath(\pi_l) \implies \varphi_l(\pi_l)$. In other words, for all paths described by $X_C(s_0) \wedge ispath(\pi_l)$ the property $\phi_l(\pi_l)$ holds. Formulations have been proposed for cases where there is a loop in the examined interval [BCCZ99].

The required tautology check can be computed by finding a satisfying assignment for the formula $X_C(s_0) \wedge ispath(\pi_l) \wedge \neg\varphi_l(\pi_l)$. If $\varphi_l(\pi_l)$ is violated, then there exists a counterexample containing a sequence of states for which the property fails. It should be noted that the sequential verification problem is reduced to checking an entirely Boolean formula. The validity of the property can simply be checked by SAT solving. The computational complexity for the resulting check, in practice, is significantly lower than the computations that would result from applying a generic model checker to the same finite-state transition system.

In verification based on bounded models, properties are expressed in linear temporal logic (LTL) [Pnu77]. The most common application is to check *safety properties* of the form Gp. Where G is the *always* temporal modal operator specifying that the formula p holds along every state on the interval l. In Interval Property Checking properties commonly describe cause-effect operations, written as $Gp = G(a \implies c)$, that should be performed by the transition system. Where a and c correspond respectively to LTL formulas describing the condition that triggers a given operation (assumption part) and the output sequence that has to be produced by the system (commitment part). In practice, standardized property specification languages such as SVA [Spe08], PSL [Acc04], and ITL [One] are available to ease the specification task.

Different optimizations to the basic proving method have been proposed. Techniques to include additional constraints to the problem are used in order to reduce verification runtime [Sht01, Sht02]. Reductions by exploiting design symmetries [BJW04] have been shown to be effective for verifying regular designs such as memories, processors, and arbiters. Motivated by the observation that a property is commonly not influenced by the whole design, optimizations based on cone-of-influence reduction are presented in [BCRZ99]. Broadly speaking, this optimization performs a structural analysis in order to prune out all the logic that does not belong to the transitive fanin of the properties. As a result, the memory footprint required for the proofs can be reduced.

2.3 Symbolic Execution

Symbolic execution is a technique widely used in software verification and testing [Kin76]. A good overview of this technique in the context of software testing is presented in [CS13]. Symbolic execution allows to explore the software behavior for big input spaces which are impractical to cover with techniques based on concrete execution, such as random testing [CH00, CS04, PE05]. On the other hand, compared to other static analysis tools [BPS00, CS05], symbolic execution has the ability to produce a *valid* concrete input together with an execution trace when a problem is discovered (i.e., no false negatives are produced).

Symbolic execution analyzes the behavior of a given program by exploring systematically its execution paths. It employs symbolic variables to represent the program inputs and describes the software behavior as symbolic formulas in terms of these input variables. At branching points, the path exploration splits to examine each possible execution path. During the exploration, conditions (typically assertions) are checked by feeding a decision procedure (e.g. an SMT solver) with formulas for each check. This task is accomplished by a *symbolic execution engine* (often called *symbolic executor*). Figure 2.2 presents an example (taken from [CDE08]) for a simple C program.

During the analysis, the symbolic execution engine keeps internal record of the program state, the program location (i.e., the program counter) and of the so-called *path conditions*. The program state and the path conditions are expressed in terms of the symbolic inputs. For sequential programs, the program state is composed of the program variables. For a simulated path, the path condition is set to *true* when the path is activated. Furthermore, the formula for the path condition corresponds to the conjunction of all branch conditions along the path.

The execution paths can be represented in an execution tree as shown at the bottom of Figure 2.2. Only those paths that can be activated are represented in the graph. A node in the execution tree represents the execution of a program statement which is identified by its program location. Graph edges correspond to transitions between the statements. Each node is associated with its program state and path condition. At the beginning of the simulation, the path condition is set to *true*, since the first program statement is always executed. When encountering a branch, one of the branches is selected by the execution engine, and the corresponding branching condition is conjoined to the path condition.

In order to tackle the path explosion problem techniques for path pruning and merging are available. Path pruning ensures that only feasible paths are explored in the analysis. This is achieved by making solver calls to check whether the path conditions of new explored branches can be satisfied or not. In Figure 2.2 the path along the two consecutive “taken" branches is infeasible since the corresponding branching conditions are contradictory. With path merging the simulation of (different) divergent paths is joined whenever these paths converge again in the execution. In this way, the exploration does not produce a tree but a

```
11:    int bad_abs(int x){
12:      int z;
13:      if(x < 0)
14:        z = −x;
15:      if(x == 1234)
16:        z = −x;
17:      assert(x >= 0);
18:      return z;}
```

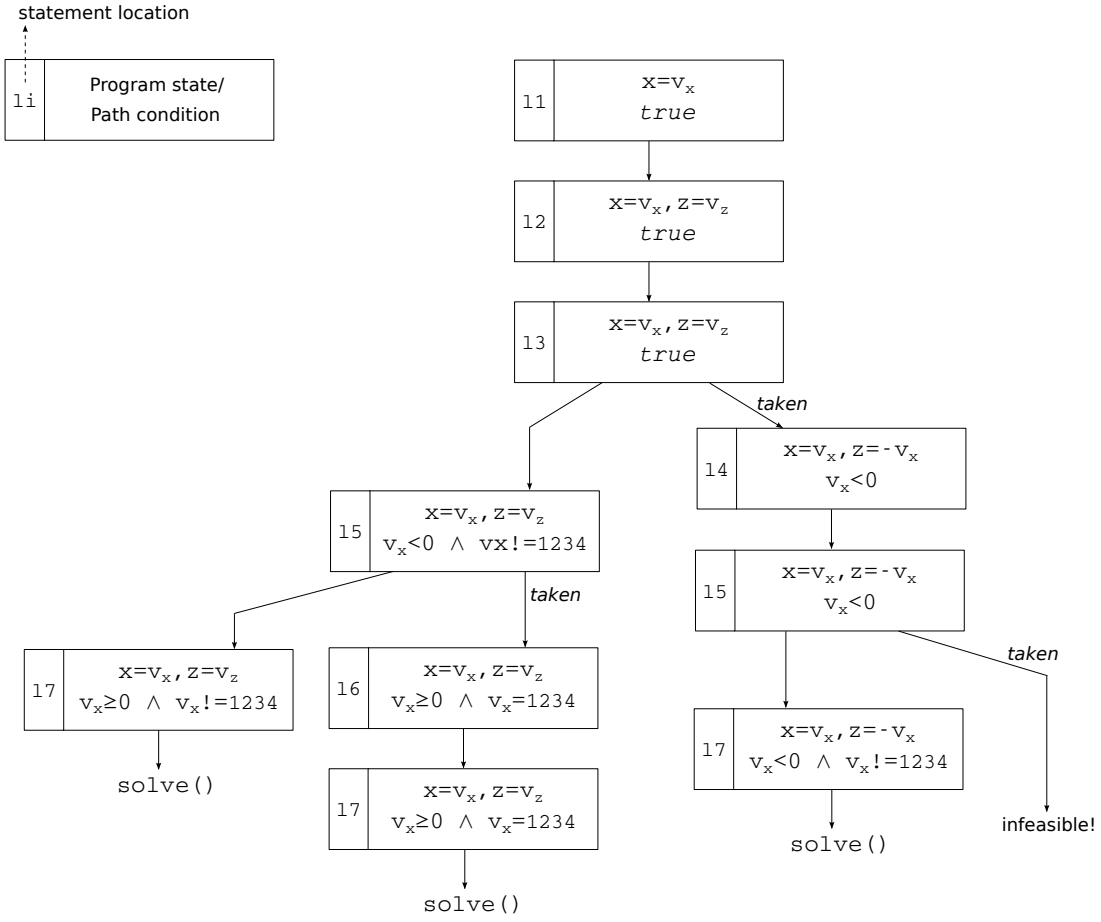

Figure 2.2: Symbolic execution example

```
assert(x >= 0);        →    if (x >= 0) {
                                assert false;
                                abort();
                            }
```

Figure 2.3: Instrumenting assertions in symbolic execution

DAG.

Symbolic execution is commonly used to check assertion violations as well as generic errors such as dangling pointers, buffer overflows, and divisions by zero. For that, the code of the analyzed program is instrumented with assertions. Assertions are typically treated as conditions in order to create explicit error paths. Figure 2.3 presents one example. In this way, the problem of checking assertions is mapped to reachability.

Chapter 3

Program Netlists

This chapter presents a computational model for hardware-dependent low-level software called program netlist. A program netlist represents the behavior of a low-level embedded program in terms of the hardware structures on which it executes. It has been developed for representing programs that are *reactive* to the hardware, i.e., communication may happen not only at the start and at the end of the program execution but also continuously during run-time.

A straightforward approach for verification of hardware-dependent software could be to model the software as binary code stored in a ROM which is connected to the processor hardware. As a result, a hardware model for the entire system is obtained which is represented by its transition relation, T, in the usual way. Verification could be based on Bounded Model Checking [BCC$^+$99] by unrolling this transition relation for a finite number of time steps. For instance, the maximum number of clock cycles along the longest execution path of the program could be chosen for the unrolling. Figure 3.1 presents an example. In order to keep the discussion simple it is assumed that the CPU requires one clock cycle to execute each instruction.

Such a hardware-style BMC approach is attractive for hardware-dependent software verification since the behavior of the software can be represented by hardware structures at the desired level of detail. However, the approach will yield a complex computational model representing the entire processor hardware multiple times, once in each i-th time step. Only very small designs and only short time windows can be examined with such an approach.

Let us examine what would happen if a SAT solver is used to reason on such a model when performing a given proof. Consider the piece of control flow graph (CFG) and the BMC unrolling shown in the top and in the middle of Figure 3.1. The nodes in the CFG represent individual instructions of the machine code. Each T_i in the model describes all software behaviors that could occur in the i-th time step. In time step 1 instruction a is executed. No other instruction can be executed at this point in time. This means that the system can be modeled under the constraint that this particular instruction is performed. This

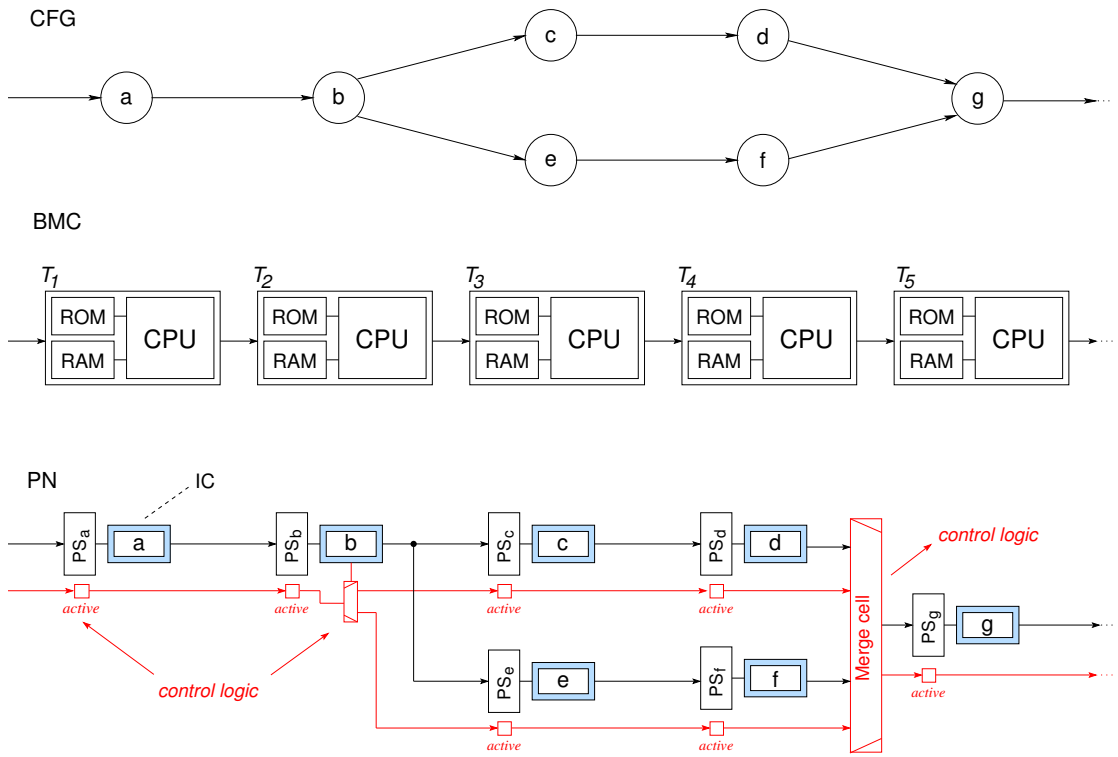

Figure 3.1: HW-style BMC unrolling against program netlist approach

fact can be exploited to drastically simplify the transition relation T_1. The same process can be followed to model the system at time point 2. (Only instruction *b* can be executed at that time point.) Now consider time point 3. At this time point, instruction *c* or instruction *e* can be executed. T_3 can still be simplified but it now needs to model both of these instructions. Hence, fewer simplifications to the transition logic can be performed. It is realized that in more complex control flow graphs with numerous branches and loops the simplification can only benefit from such constraints during a fairly small number of steps in the initial parts of a program. At later time points, there will be many possibilities what instructions can be performed. Therefore, when unrolling the transition relation, the individual T_i will have to model (almost) the entire hardware system, since no (or only few) constraints can be identified. If a SAT solver has to enumerate the search space to prove some property on this model it will obviously suffer from the sheer complexity of this model.

Moreover, there is an additional problem for the SAT solver making the situation even worse. When backtracking through the search space the solver makes assignments to the variables of this model that mix situations occurring in different runs of the program. For example, if instruction *c* is performed at time 3 it is not possible that instruction *f* is performed at time 4. If the SAT solver makes assumptions in its branching decisions relating to instruction *c* at time 3 and instruction *f* at time 4 it will enter the non-solution area of the search space. It may take a large number of backtracks until this is discovered.

In conclusion, a SAT solver needs to deduce from scratch the possible execution paths of the program via clause learning, backtracking, and similar concepts. This is because the model lacks an explicit view on execution paths. Since the program's control flow is represented only implicitly by the unrolled hardware, reasoning on the program will require a high computational effort. It is then apparent that such a model, even if it was small, is computationally inefficient and would require excessive computational resources.

3.1 Basic Idea

The program netlist approach is related to the BMC approach illustrated above, however, key obstacles to scalability are removed. The basic idea is the following. The unrolling of the processor with its instruction and data memory (ROM and RAM respectively) is not done clock cycle by clock cycle, replicating the full transition function for every time frame, but rather *instruction by instruction* (cf. the model on the bottom part of Figure 3.1). At branching points in the software, the unrolled logic is duplicated, modeling each execution branch separately. This instruction-wise unrolling along execution paths allows for a significant reduction in the amount of logic that needs to be replicated: Since the actual instruction in every unrolled logic block is known and fixed, many constants exist that can be propagated in order to simplify the logic block so that all circuitry that is not needed for modeling the instruction behavior is removed.

In fact, this analysis can be moved to a pre-processing step before unrolling (Section 3.3 details on this analysis). As a result of it, information about reachable execution paths of the software is directly encoded into the control logic of the program netlist (cf. logic blocks on the bottom of Figure 3.1). These specific control structures make execution paths and the program's control flow visible to the verification engine.

For a given instruction set architecture and machine program, the behavior of the processor can be precisely modeled for each individual instruction of the program. A logic block that models atomically the effects of an individual instruction on a set of state variables is called an *instruction cell* (IC). The set of state variables that the cell modifies depends on the type of instruction and includes registers from the general-purpose register file, status bits, and flags as well as memory locations associated with data variables of the program and input/output registers. (The memory model is described in Section 3.4.) These state variables constitute the *program state* (PS) of the programmable hardware/software system. The subset of these state variables which are internal to the processor are referred to as the *architectural state* (AS) variables.

Connecting instruction cells together and duplicating paths at branches is, by itself, not sufficient for efficiency because the resulting model can become of exponential size in the number of branches. Instead of building a *tree* of instructions, a netlist that has the structure of a directed acyclic graph (DAG) is created. As will be shown next, so-called *merge cells*,

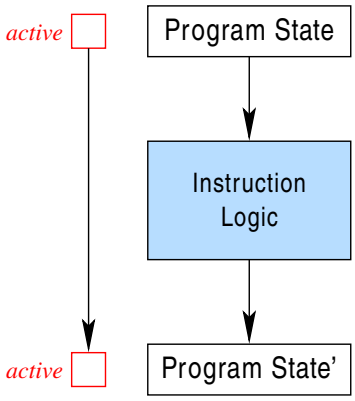

Figure 3.2: Instruction Cell for instruction without branching

as the one placed at the fanin of instruction cell *g* on Figure 3.1, are used for recombining paths in the program netlist in order to avoid exponential growth of the model.

3.2 Instruction Cells

Instruction cells are the building blocks of the *program netlist*, i.e., the unrolling of the processor's behavior under the control of the program. Instruction cells are parameterized with registers, operation modes, and similar objects so that they can be optimized during model generation, in particular by constant propagation. This leads to compact combinational circuit models.

Instruction cells can be described at different levels of abstractions depending on the level of detail that is required by the given verification tasks. In this work, manually-written abstract instruction cells are used to capture behavior according to the programming model at the instruction set architecture (ISA) level. Instruction cells can be also created for modeling the concrete behavior of the RTL implementation of a specific processor architecture.

Figure 3.2 and Figure 3.3 show examples of instruction cells with and without branching. Combinational logic circuitry changes the program state including architecture registers and variables in the data memory. Instruction cells are connected together at the Program State interfaces (cf. program netlist on Figure 3.1). A connection between two instruction cells indicates a possible transition from a CFG instruction to the next one.

Additionally, an instruction cell models the *control flow* of the program using a special state variable called *active*. In the program netlist, all instructions lying on an actual program execution path have their *active* flag set. A BRANCH instruction as shown in Figure 3.3 produces two possible program states, one for the branch taken and one for the branch not taken. The *active* flag is distributed into the branch selected by the program, as controlled by the instruction logic (signal *J* in Figure 3.3).

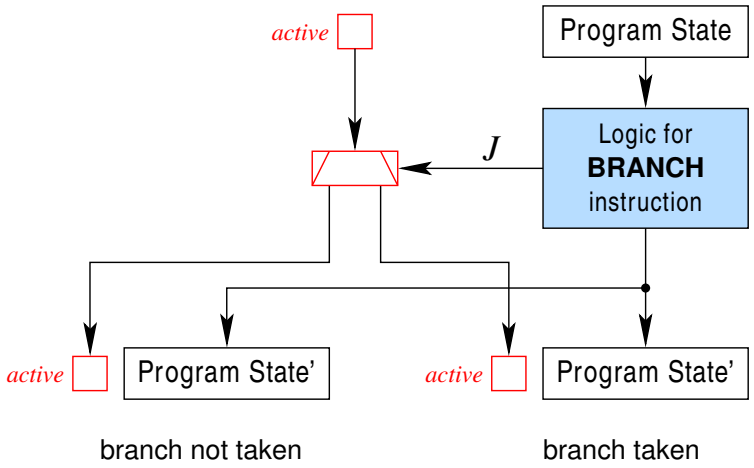

Figure 3.3: Instruction Cell for BRANCH instruction

Modeling the control flow in this way is crucial to the performance of SAT-based reasoning on the program netlist as it makes execution paths explicit to the SAT solver. By asserting or de-asserting the *active* signal, whole paths or path segments spanning many time frames can be taken into or out of consideration simultaneously. This gives significant performance improvements over an implicit, unguided enumeration of execution paths as in the straightforward approach discussed above.

3.3 Model Generation

Unrolling of the program involves two steps, as illustrated in Figure 3.4. The unrolling process begins with a representation of the control flow graph of the program. It is obtained, e.g., from the machine code of the program or from an assembler program. (As will be discussed in Chapter 4 the control flow graph may not be complete.) The control flow graph is unrolled into an *execution graph* (EXG). This execution graph is then used to build the program netlist by instantiating and interconnecting instruction cells corresponding to the nodes in the EXG.

These two steps are not taken one after the other but instead are carried out in an interleaved fashion. The incomplete program netlist, while it is being built, is used to control the unrolling of the execution graph. The key idea is to determine whether a branch can actually be taken at a particular node in the unrolling. For example, a loop is unrolled until the loop end condition is reached. A SAT solver is used to check whether there exist executions where the active flag of the loop-back branch can (still) become active. Consider for instance the unrolling of the branch instruction at CFG node c. In the two first instances of this instruction (EXG nodes c_0 and c_1) both possible branches are alive. In the last instance (EXG node c_2), however, the branch to the instruction at CFG node e results dead, i.e., there is no

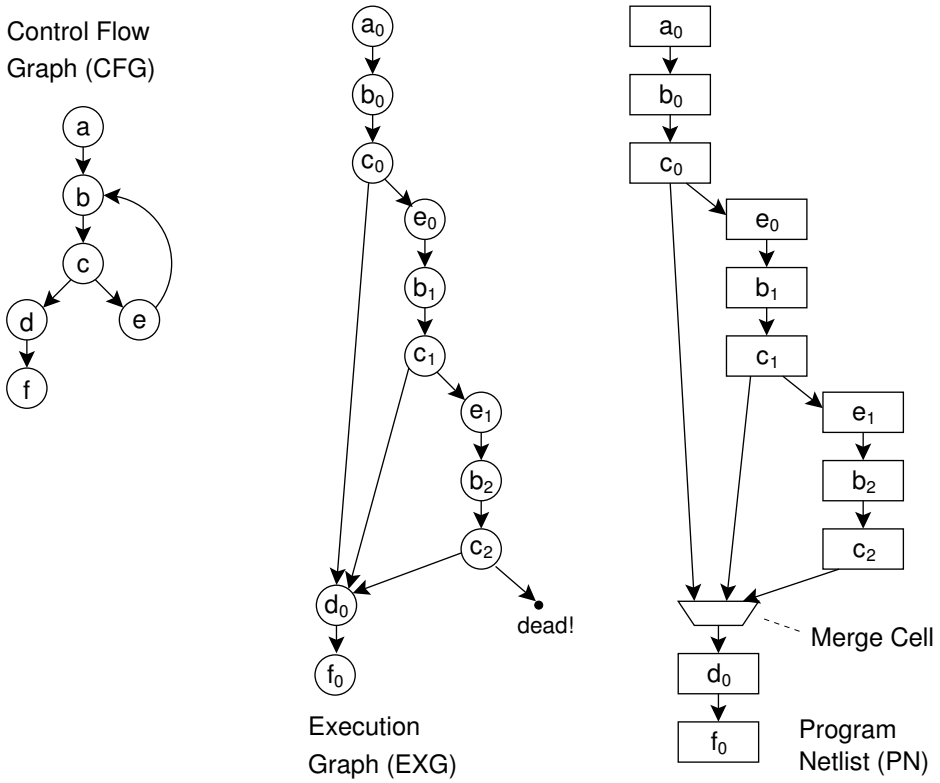

Figure 3.4: Generating a program netlist

input assignment to the program that makes this branch active. Similarly, branch destination addresses can be computed on the incomplete program netlist to trace the actual flow of control in the program.

In this way, the depth of the unrolling is determined dynamically at every branch node. Loop conditions may depend on external inputs. Similarly as in common approaches to hardware verification or worst-case-execution-time analysis [WEE+08], user-defined environment constraints can be employed to bound the number of unrollings of a loop in such a case.

An important component of the model building process is *merging of nodes* in the execution graph. Whenever the control flow modeled in an unrolled path segment reaches a program location that has been visited before, no new node is created but instead, the program state variables are connected to the existing instruction cell for that location, provided this does not introduce a cycle in the graph. This is done using multiplexers in the program netlist called *merge cells* (see example in Figure 3.4). Merging keeps the model compact by sharing of sub-graphs and produces a DAG with reconvergent paths (rather than a tree). Note that the sub-paths being merged in a merge cell can never be active simultaneously. This is guaranteed by construction of the program netlist with *active* flags.

The program netlist model obtained in this way allows for efficient SAT-based reasoning in applications like equivalence checking (cf. Chapter 5) or property checking (cf. Chapter 6) for low-level hardware-dependent software. Key to efficiency is the fact that most of the control flow related information is computed beforehand and is built into the model. The program netlist is an explicit representation of all possible execution paths in the software, while the data path information is still contained implicitly in the combinational circuitry inside the instruction cells. This makes the model particularly amenable to SAT-based proof algorithms.

3.4 Modeling Data Memory and Input/Output

Modeling accesses to the data memory and to the environment (e.g., to hardware peripherals) is a key element of the program netlist. On one side, efficient modeling of the data traffic of between data memory and processor is crucial for the scalability of the method. On the other side, precise input/output modeling is required to inspect the details of the software behavior at the hardware/software interface.

Figure 3.5 shows an instruction cell modeling a read or write access to data memory. The shown selection logic creates an access path from the LOAD or STORE instruction cell for the given address, *addr*. In case of a LOAD, the architectural state is modified with the selected *data*. In case of STORE, the data memory is updated with *data* from the architectural state. Modification is enabled if the *active* flag is set.

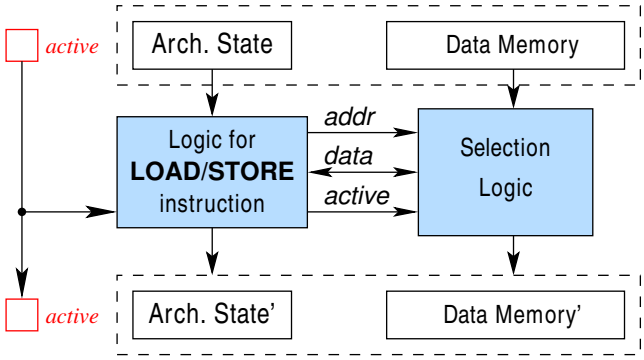

Figure 3.5: Instruction cells for memory accesses

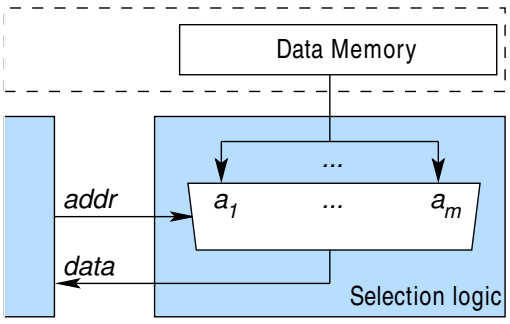

Figure 3.6: Selection logic for a LOAD instruction cell

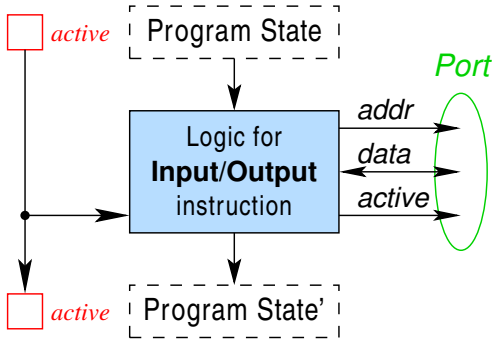

Figure 3.7: Input/output instruction cells for accessing the environment

In order to keep the selection logic block compact, a combined simulation/SAT-based algorithm, discussed in Chapter 4, is used to compute the set of addresses the LOAD/STORE instruction at the given program location can actually access. The computed reachable set of addresses can be used to reduce the size of the selection logic greatly because only the corresponding data memory locations need to be multiplexed for a given instruction cell. For the LOAD instruction cell shown in Figure 3.6 for instance, the size of the multiplexer is reduced because it selects only among memory data addresses, $\{a_1, a_2, \ldots, a_m\}$ that can be actually accessed by the instruction cell. Logic for memory locations not reached by the instruction cell is not included in the model. Although, in principle, the range of addresses accessed by a program can be huge, in the intended application domain of the proposed technique (low-level, hardware-dependent software) the number of addresses used by the software is usually limited and restricted by design.

This representation of the program state can be viewed as an associative memory model like the one proposed in [EES04] for formal verification of assembler code by Bounded Model Checking. Note, however, that in the approach of this work the associated memory entries are created statically during model generation and not dynamically at proof time through value assignments in the reasoning engines. The resulting logic in the program netlist can be much simpler and can lead to more efficient SAT reasoning during verification. This is a direct benefit of the instruction-wise and path-oriented unrolling of the software in the program netlist as opposed to a BMC-style unrolling of the processing hardware as in [EES04].

Figure 3.7 illustrates how input/output is modeled using instruction cells. The input/output instruction cell provides an interface for the program netlist called *port*.

Definition 1. *The* port *of an input/output instruction cell i is a set of three logic signals called i.addr, i.data and i.active.* □

The *addr* signal provides the address of an environment location, e.g., a peripheral device register. The *active* flag indicates when an access occurs and data is transferred through

the *data* signal. Using such ports, the sequences of input and output accesses of the software along different execution paths can be modeled. In the same way as for memory accesses, a simulation/SAT-based algorithm is used to compute the possible addresses an input/output instruction can access (cf. Section 3.4). This is used for instance in Chapter 5 for equivalence checking to reduce the amount of logic necessary to model all possible access sequences exhibited at the hardware/software interface. It is also used in Chapter 6 for keeping the logic needed for a cycle-accurate input/output model as small as possible.

3.5 Modeling Interrupt-Driven Systems

Program netlists can be seen as combinational blocks that can be instantiated and combined with other blocks to create new kinds of computational models. The work of [SVF+13a] takes advantage of this fact to propose a compositional approach for modeling interrupt-driven systems.

The approach proceeds in two steps as follows. In the first step, the program netlist of each software component is generated. (Software components include interrupt service routines (ISRs) and the main program.) For this purpose, the control flow graphs of the software components are extracted from the machine code. Starting from the control flow graphs, program netlist generation is done as explained in Section 3.3.

Subsequently, in the second step, the overall program netlist representing the behavior of the whole composed system is created by instantiating and interconnecting the individual program netlists created in the first step. In the composition, complexity problems are tackled by instantiating program netlists only at instruction cells where communication takes place. This simplification can be performed if the ISRs communicate with the rest of the system uniquely via shared memory and if the communications points between software components can be precisely detected. The former condition can be checked by proving the transparency of each ISR with respect to the elements of the architectural state. The last condition is satisfied by taking into account that after program netlist generation the set of data memory addresses accessed by the instruction cells is known (cf. Section 3.4). Finally, the number of program netlist instances can be further restricted by using environment constraints that help to determine the maximum number of times a given ISR can occur between subsequent accesses to shared memory [SVF+13a].

Chapter 4

Efficient Generation of Program Netlists

This chapter presents a method for efficient generation of program netlists. The method is used to compute information that is required for performing the control flow graph unrolling presented in Chapter 3. On the one hand, the method computes control flow reachability information such as destination addresses (targets) of jump instructions as well as liveness checks at branch instructions. On the other hand, it computes addresses accessed by memory instructions and by input/output instructions. The method uses an instruction set simulator to find the information for most of the cases. It resorts to SAT-based techniques for computing the remaining, hard-to-determine cases.

Compared to the analysis of high-level code, for low-level software the assumption that a complete control flow graph is obtained after parsing the source code is not always valid [RLT+10]. A typical case is a jump instruction whose destination is stored in a CPU register (e.g. *jump [Rm]*). Without making a semantic analysis of the software it cannot be clearly determined during the unrolling which path the execution will take. This situation occurs in code constructs commonly used in low-level software where jump targets are computed. Typical examples are low-level implementations of branch tables and call-back functions.

Moreover, for generating compact models for software it is required to perform different optimizations during the unrolling. In particular, path pruning [CS13] allows to exclude unreachable paths of the software from the model. In order to perform path pruning, it is necessary to perform specific checks for every conditional branch found during the unrolling that determine whether the corresponding branch can be reached or not.

Finally, low-level software makes heavy use of indirect memory addressing (e.g. *load Rn, [Rm]*) which involves arithmetic on memory addresses. A common case is an instruction accessing an element of an array or buffer stored in data memory. This complicates the construction of efficient data memory models [VSB+13]. Again, without a semantic analysis it is not clear what memory address(es) can be accessed by the software.

All the previous situations occur thousands of times while unrolling complex low-level

software. In consequence, an effective processing strategy is required so that model generation becomes feasible for industrial applications.

In practice, several characteristics of hardware-dependent software resulting from typical embedded system applications turn out to be beneficial for an efficient model generation. For instance, the software neither makes calls to unbounded recursions nor does it dynamically allocate memory. Moreover, the possible address ranges of the system are limited and predefined by design. Also, it is a common practice in low-level software to use registers also for storing constant addresses, for example when jumping to a subroutine or when accessing a register of the hardware periphery. Finally, analyzing unrolled versions of the control flow graph reduces the complexity as more constants can be detected. For instance, accesses to arrays, as mentioned before, normally result in a single constant memory address value.

In this chapter, these characteristics are exploited to only include the feasible behavior of the software into the computational model. This significantly reduces the state space when performing verification. Instruction set simulation techniques are employed to propagate and discover constant values. Only for difficult cases where simulation does not succeed because the considered values are not constant, e.g., when there is a dependency on primary inputs, SAT is employed to determine the range of possible values. SAT is much more costly than simulation and should be avoided whenever possible. As shown in the results, the proposed combination of techniques provides a very effective solution for the intended applications of this work.

4.1 Model Generation Using Incomplete CFGs

Program netlist generation requires two main elements (Chapter 3): a set of instruction cells describing formally the processor behavior for each program instruction and an execution graph (a fully unrolled control flow graph) containing all possible execution paths of the software.

Instruction cells are described at the ISA level by means of a tailored instruction cell language (ICL). This ICL includes different elements that facilitate the description and that allow automatic translation into other formats. In particular, generation of C++ code is done so that instruction cells can be executed by the instruction set simulator (cf. Section 4.3). Instruction cells can be also translated into an internal data structure from which clauses can be automatically generated in order to perform SAT solving (cf. Section 4.2).

In order to unroll an incomplete control flow graph into an execution graph, as explained in the previous section, efficient processing of indirect jumps and of indirect accesses to data memory as well as liveness checks for path pruning are required. When it comes to a control flow graph that is extracted from a machine program, the resulting control flow graph will most likely be incomplete. The reason is that not all necessary information is directly available in the machine code because it depends on the data flow of the program. In those

cases, a semantic analysis of the software has to be performed in order to deduce the missing information.

Two cases need to be distinguished here: evaluation of the control flow, including indirect jumps and conditional branches, and resolution of the memory behavior, including indirect accesses to memory.

If at a given point of the unrolling an analysis of the semantics of the program is required, the proposed method proceeds in two steps. First, it calls the instruction set simulator. In turn, the simulator will proceed by forward propagating all constant values contained in the program netlist. If the call succeeds then the information is added to the control flow graph and the unrolling process can continue. Otherwise, if the simulation fails, the SAT solver is called to compute the required information. This second call will always succeed.

This approach speeds up the overall generation process as fast simulation is used to save SAT calls that can be costly in terms of generation run-time.

For control flow analysis, extracting successors inside an instruction sequence without conditional branches and jumps, i.e., a basic block, is straightforward. However, if a jump instruction is encountered the possible destinations (successors) may not be given directly by the instruction code because that information can be stored in a register. Two possible situations may arise for the jump target value. In the simpler one, a constant value loaded from memory and stored in a register is used to calculate the target address. In the second and more complex case, the jump instruction can have different values for the target address. They can depend on the particular execution paths taken by the program or on operations performed by preceding instruction cells to those values. In both cases all possible destinations need to be precisely determined, otherwise performing formal verification on the resulting model can lead to wrong results.

Consider the example of Figure 4.1. The destination of the jump instruction at node h is unknown in the control flow graph. However, in the execution graph the destination of h_0 is the instruction at node c_1 (which corresponds to c in the control flow graph). This is known in the approach only after examining the semantics of the program by using the combined simulation/SAT solution.

If a conditional branch is found in the unrolling the possible destinations are known in the control flow graph (cf. instruction d in Figure 4.1). However, it is still necessary to check whether each branch can be dead or alive. As explained in Section 3.3, this is required to remove unreachable execution traces from the model. For conditional branches, two calls to the simulation/SAT engine are made, one for each branch. In Figure 4.1, both branches (g_0 and e_0) are determined live after the checks performed at branch node d_0 in the execution graph.

Finally, for resolving data memory accesses the complexity of the model can be reduced by making data memory values path dependent. This means that memory logic is created for a given program variable residing in memory only on paths with instructions accessing that

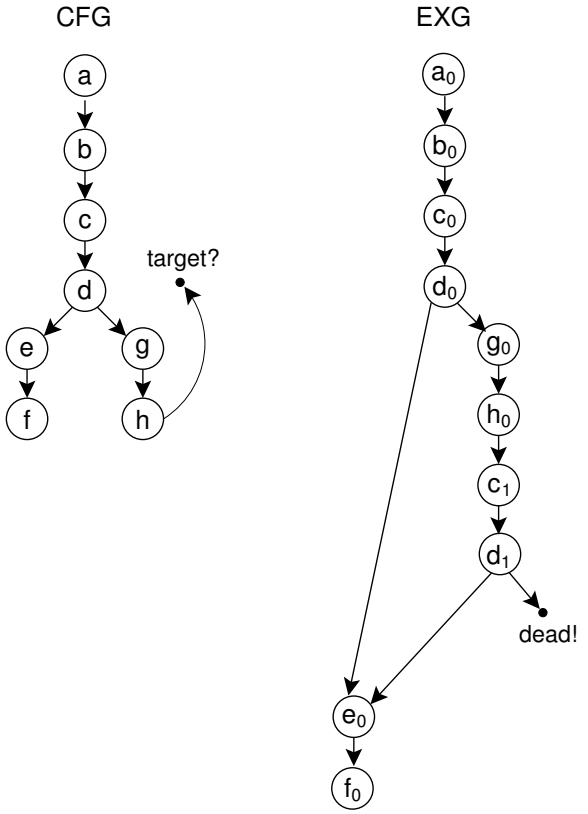

Figure 4.1: Model generation example: unrolling an incomplete CFG into an EXG

variable. Back to the example of Figure 4.1, assume that the instruction cell at node *c* reads a memory value stored at address *addr_x*. If it is also assumed that the instruction at node *g* changes the value stored at the same memory address, then, due to the jump from node *h* to node *c* the control flow graph involves two different accesses to memory location *addr_x*. However, after unrolling has been performed execution graph nodes c_0 and c_1 require only one memory access respectively. In the resulting program netlist, g_0 and c_1 will be directly connected by using memory logic and the memory value read at c_0 corresponds to the initial value of the memory variable. Before taking these decisions, however, it has to be determined that execution graph nodes g_0, c_0, and c_1 access the same memory location *addr_x*. In each case, this location value can depend on the data flow of the program and may be stored in a CPU register. Consequently, this information needs to be calculated to define the memory ranges that have to be modeled. In the particular example, three calls to the simulation/SAT engine need to be made.

4.2 Computing Addresses Using SAT Techniques

There is a large research body on the problem of finding all satisfying values of a propositional formula [JS05]. In this section, the basic enumeration algorithm is presented and how it can be employed to solve the particular problem of completing control flow graph information.

Let us assume *addr* is a bit vector in the program netlist representing the signal that needs to be inspected. Particularly, *addr* can be the memory address signal (cf. Figure 3.5 and Figure 3.7) or the program counter of a given instruction cell modeling a jump instruction. The enumeration algorithm obtains on every iteration a new satisfying value for *addr*. Let us also assume that v_k is the k-th satisfying value of *addr* obtained after the k-th iteration of the algorithm. Then, a new element satisfying value v_{k+1} can be extracted from a counterexample refuting the following property.

$$p_{k+1} = \bigvee_{v_i = v_0, v_1, \ldots, v_k} (addr = v_i) \tag{4.1}$$

The iterative algorithm will iterate until the property of Equation 4.1 holds, i.e., no new counterexamples are returned by the SAT solver. The constraints in the disjunction are converted to blocking clauses in the SAT instance preventing the solver to visit previous solutions. In the implemented algorithm incremental SAT solving is employed to speed up the process by reusing information learned by the solver in previous iterations.

This proposed solution produces the exact set of values *addr* can take. Obviously, in a general setting, this scheme can run into complexity problems since instructions cells, in principle, could address huge ranges of data memory or jump to many different destinations. However, for the intended application domains, as explained at the beginning of this chapter, the possible solution ranges are limited and restricted leading to a tractable number of SAT calls when computing Equation 4.1.

4.3 Constant Propagation Using Instruction Set Simulation Techniques

The SAT-based algorithm of the previous section is powerful in the sense that it finds solutions for all cases, independently whether values are constant or not. However, using the SAT solver is very time-consuming. The situation is even worse since the number of calls required for unrolling complex software can be large, e.g., in the order of thousands of calls. In order to reduce run times a simulator is implemented and integrated on an appropriate abstraction level. In particular, performance problems that arise when simulating code on low abstraction levels (e.g., at bit level) are avoided. The proposed simulator employs simulation techniques on a higher abstraction level in order to compute memory addresses, jump targets, and

inactive program paths.

Since for program netlist generation instruction-based models like assembler code, control flow graphs, and execution graphs are used, the ISA level is an adequate level of abstraction. However, the performance boost obtained on this abstraction level is bought by loss of information. Particularly, information about single bit values and their relationship is lost.

The simulator is capable of simulating arbitrary programs that have been unrolled completely or partially into an execution graph. For achieving as much performance boost as possible, descriptions of instruction cells are generated in a high-level programming language such as C++. In this way, it is possible to take advantage of compiler optimizations for significantly reducing the actual number of processor instructions (of the host machine) needed to simulate a single instruction cell.

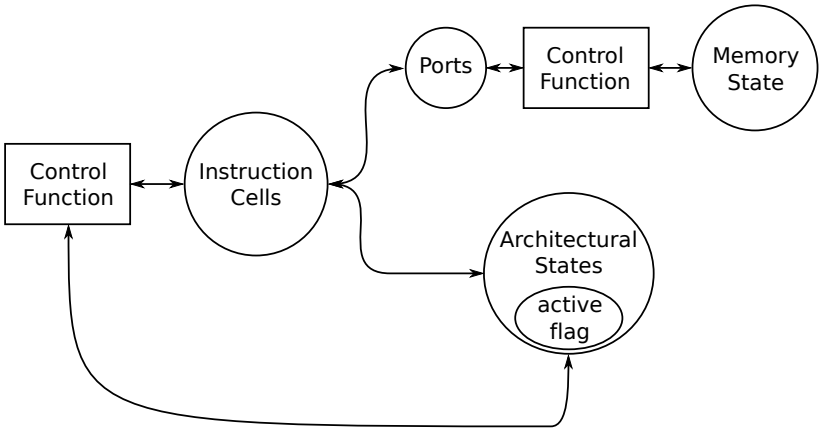

Figure 4.2: Components of the simulation engine

Figure 4.2 depicts the basic components of the proposed simulator. The core of the simulation engine comprises a set of simulation functions (more specifically C++ methods), each of them simulating a corresponding instruction cell. Besides this, the simulator contains an internal data structure modeling the architectural state and the memory state as well as functions for controlling the flow of the simulation. One instruction can have several successors and therefore the simulator needs to represent them individually. Besides the registers belonging to the architectural state, the simulator keeps track of the active flag (cf. Section 3.1) whose value indicates the activity of a corresponding simulated execution trace. By using this flag, the simulator can find out what program paths are active at a given point of the simulation. Once it is called, the simulator runs for all active program paths. Importantly, every execution path which is not active is considered dead and not reachable by the program. As shown in Figure 4.2, information of simulated values for registers of the architectural states can be directly accessed from the instruction cells. Data residing in memory has to be accessed via specific ports representing address and data buses (cf. Figure 3.5). This

facilitates the collection of information of addresses related to memory accesses.

Two functions are required to control the simulation flow. One control function decides which instruction cells has to be simulated next. After this, a call to the simulation method of the corresponding instruction cell is made. Simultaneously, information about active and inactive architectural states is collected. The second control function tracks the memory accesses initiated by the memory ports of the instruction cells. It also collects information for every memory access (data and addresses) which is necessary for the overall model generation.

4.3.1 Simulating Undetermined Data and Control Flow

A program netlist represents the behavior of a given program along all possible execution paths. Whether a particular execution path is executed or not depends on the concrete assignments to the program inputs. When using formal methods, all valid input assignments need to be considered. In the program netlist values of the CPU registers or of the memory that depend on the primary inputs are undetermined. This needs to be addressed when simulation methods are applied.

For solving this problem a new flag is introduced in the simulation model. This flag indicates whether the value of a register is valid or not. A valid value corresponds to a constant word stored in the register. Undetermined (invalid) values occur when there exists a dependency with the primary inputs. With this flag, it is possible to identify whether all bits or just a subset of them are undetermined.

Detecting individual undetermined bits is used extensively in the case of branch instructions. Imagine that all bits of the status register of the CPU are invalid except the ones required for branch decisions. Without an analysis on register values at bit granularity simulation could lead to false conclusions, for example that both branches of a branch instruction can be reached by the program even though in reality this is true only for one of them. Nevertheless, situations, where both branches of a conditional branch instruction can be reached by the program, can occur. This situation is recorded in the simulation engine by setting the flags for the corresponding bits in the status register of the CPU. Consequently, both branches are marked active. In these cases, the simulator then simulates both paths until no one can be simulated any further. This happens when the program has ended or when an unresolved control flow instruction (a new branch or jump instruction) is found for the first time by the simulator. This is the topic of the next section.

4.3.2 Computing Successors of Control Flow Instructions

As explained in Section 4.1 there are two situations where the control flow of a program needs to be evaluated while unrolling incomplete control flow graphs, namely indirect jumps

and conditional branches. In order to handle them the simulation engine implements an additional analysis in order to improve performance.

If a new control flow instruction is found, while unrolling, then a new call to the simulator is made in order to define the possible successors. For the case of a conditional branch instruction, this means to check the liveness of each branch. For this purpose, the simulator traverses every preceding path ending at the control instruction that is being inspected. Taking into account that the simulation is based on concrete values, if there is a merge point preceding the control instruction, then the simulator does not merge the concrete simulated values of the architectural state. Instead of that, each path is simulated individually until the control flow instruction is reached. Note that merging concrete simulation values could introduce nondeterminism into the simulation. By avoiding this, unnecessary calls to the SAT solver are avoided. For jump instructions, every path simulation can find at maximum one destination value. In the case that at least one of the simulated paths result in an invalid value then the simulation fails to resolve the control flow and consequently the SAT engine is called.

4.4 Tool

The techniques for efficient generation of program netlists presented in this chapter have been implemented and integrated into the formal verification platform FCK (Formal Checker Kaiserslautern). FCK is implemented in C++ and can be used for automatic verification of low-level hardware-dependent software. Figure 4.3 shows the main components of the tool.

At the front end, the block *CFG Parser* processes the machine code that can be generated from C, assembler, or a mixture of both. As a result of this, a control flow graph data structure is produced. From the control flow graph, the execution graph is generated by the block *CFG Unroll*. In the unrolling process, whenever the semantics of the program needs to be evaluated in order to resolve control flow or accesses to data memory, *CFG Unroll* makes the corresponding calls to the *Simulation Engine*.

For constructing the simulator a set of instruction cells described as a set of C++ functions is used. These are generated automatically from the input description in the ICL language. Instruction cells in ICL are provided as a library which is described manually by the user (for a detailed description of the syntax of ICL see https://www.eit.uni-kl.de/fileadmin/eis/pdf/icl.pdf). An example of an instruction cell described in ICL is shown in Figure 4.4. The code describes the behavior of the ADD instruction of the SuperH-2 ISA [Ren05].

The *SAT Solver* is called uniquely if the simulation engine fails. CNF instances for the solver are generated by the block *SAT Interface* by taking an intermediate version of the program netlist and properties which are automatically generated by *CFG Unroll*. These properties correspond to the active checks for path pruning and the properties related to Equation 4.1.

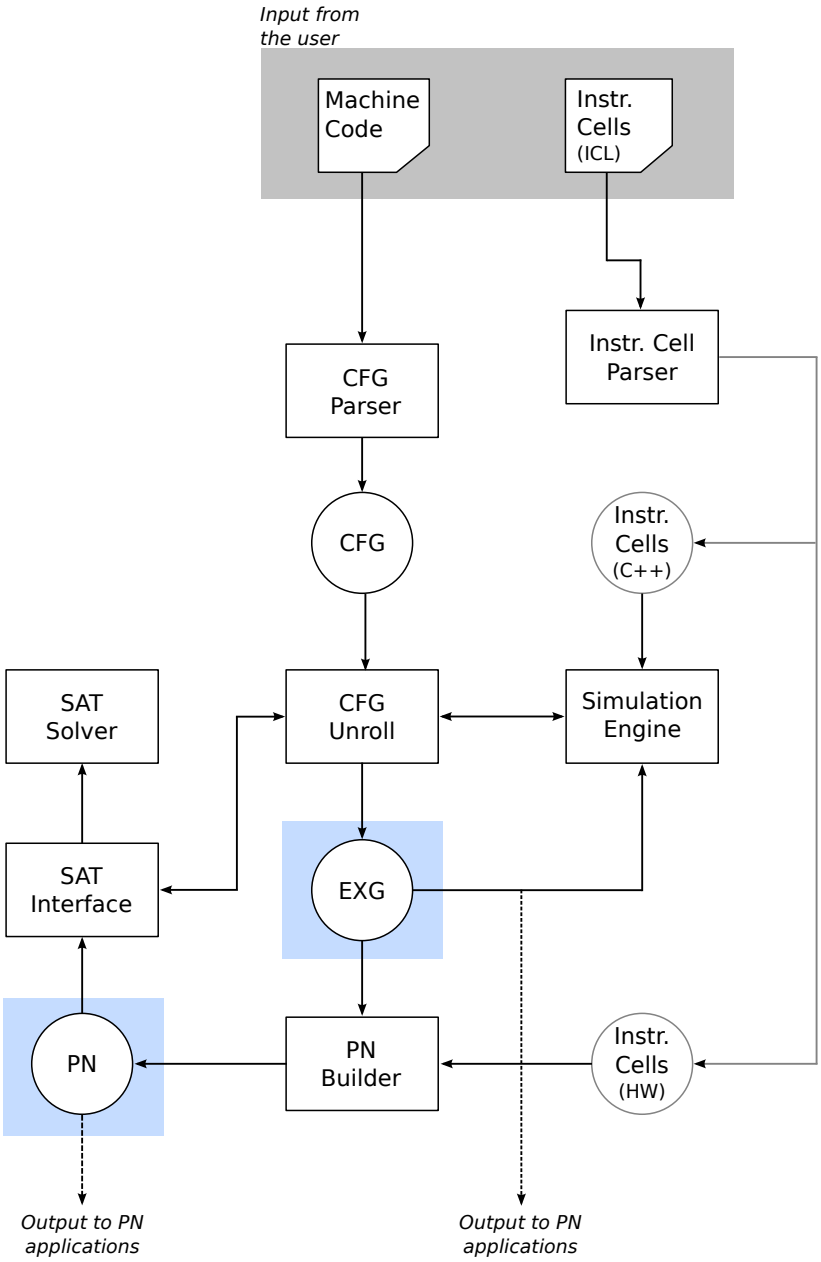

Figure 4.3: Tool for efficient program netlist generation

```
ADD_SH2(const u<4> Rm, const u<4> Rn, in PS S, out PS Z)
{
    Z' = S;
    Z.RegisterFile[Rn] =
        S.RegisterFile[Rn] + S.RegisterFile[Rm];
}
```

Figure 4.4: ADD Instruction Cell described in ICL

The final program netlist together with its corresponding execution graph is output by the tool. Both outputs are used to perform property verification or equivalence checking as will be shown in the coming chapter.

4.5 Experimental Results

The following experiments evaluate the efficiency of the model generation with the improvements of the proposed simulation/SAT approach. The low-level software used in the experiments are a LIN driver (LIN), a serial-to-parallel converter (S2PCONV) and a multiplier (MULT).

The serial-to-parallel converter takes a 32-bit serial input sequence and writes it in parallel as one 32-bit word to the output. The detection of the bit sequence is done via repeated polling on a device register. It is guaranteed by the environment that after at most 5 polling accesses a valid input bit is read in.

The multiplier implements a simple 32x32-bit multiplication subroutine using addition and shifting instructions.

The LIN driver (developed by Infineon Technologies AG) implements a master node as low-level software [LIN02]. For these experiments code was adapted to run on the open-source version of the SuperH-2 processor [Ren05] Aquarius [Ait03]. The processor has a datapath of 32-bit and a 5-stage pipelined. The driver comprises about 1350 lines of hardware-dependent, low-level C code and inline assembly. It can be configured such that both, transmission and reception modes, are allowed. The driver interacts on one side with a UART containing different registers and on the other side with an application using shared memory. The GCC compiler was used for applying three different optimization levels to the source code, starting from the level zero LIN (l0) with no optimizations being activated, and increasing the aggressiveness of the compiler optimizations up to the maximum level two LIN (l2). LIN (multi) corresponds to a version of the LIN driver for which the ID of the message can be configured by the application.

All experiments were performed on an Intel Xeon E5420 CPU at 2.5 GHz with 16 GB RAM.

Table 4.1 shows the time required to generate each program netlist. Additionally, the

Program	CPU time (s.)		Calls (Using Simulation)		
	SAT only	SAT and simulation	address	successors	active bit
LIN (multi)	1791.44	41.27	574 (572)	181 (175)	424 (424)
LIN (l0)	5805.59	98.10	1136 (1136)	161 (161)	401 (401)
LIN (l1)	1062.28	27.02	569 (569)	158 (158)	398 (398)
LIN (l2)	630.74	17.89	496 (496)	75 (75)	342 (342)
S2PCONV	2920.59	2878.97	1448 (263)	0 (0)	2018 (360)
MULT	2.83	0.4	0 (0)	0 (0)	99 (99)

Table 4.1: CPU times and solver calls for program netlist generation

table presents the times needed for finding memory addresses, jump successors and to decide whether branches are alive or not.

As can be observed, using the combined approach for finding constant values drastically reduces the CPU time needed for model generation for the multiplier as well as for all variants of the industrial LIN driver. For model generation of the serial-to-parallel converter, however, the run times with and without simulation are roughly equal, as can be observed from Table 4.1. The reason for this lies in the repeated polling on the input register which results in a number of program paths that is exponential in the number of access attempts. The simulator needs to enumerate all these paths. For preventing a runtime that is longer than a SAT-only model generation the simulation is stopped if its runtime exceeds a certain limit. In this case, the rest of the model generation is done using only SAT. This also explains the low number of found addresses and active bits using simulation for the S2PCONV.

Table 4.1 also shows the total number of calls made to the combined engine, with the number of calls for which simulation succeeded in parentheses. In most experiments, the simulator was able to find all needed information so that the SAT engine was not needed at all. In LIN(multi) not all addresses and successors could be calculated using simulation. The reason is that some address computations in this benchmark were done using bit manipulation instructions. The simulator, however, cannot analyze values at the bit level and was, therefore, unable to compute the unknown targets. For those cases, the SAT solver successfully found the address values.

These experimental results confirm that the memory addressing mechanisms, as they are employed here and in similar applications, can result in a large number of constant address values so that the generation of program netlists remains tractable even in the presence of a large address space.

Chapter 5

Equivalence Checking of HW-Dependent Software

During the design of an embedded system, the software usually undergoes several transformations. For instance, embedded software is frequently optimized, automatically or manually, in order to meet design requirements on program execution speed, memory footprint (code size) and power consumption. Furthermore, often during the design or field time of a product, features are added to a given software, extending the existing functionality. Finally, embedded software is also often ported to different hardware platforms. For all these cases, equivalence checking is a valuable tool since it can be used to certify that the original functionality of the software is not damaged or altered by the applied transformations.

Figure 5.1 depicts the general equivalence checking problem for comparing hardware-dependent programs at the machine level. A revised program R is obtained after performing a certain number of transformations to the original (golden) program G. Similar as in hardware design flows, transformations can be either implemented by a compiler or performed manually by the designer. Equivalence checking is used for determining whether R is functionally equivalent to G or not.

This chapter presents a fully automated method to formally prove the functional equivalence of hardware-dependent programs. The chapter focuses on describing the main ideas, originally presented in [VSB+13, VSK16], that enable building an efficient computational model called *software miter*.

The equivalence criterion for the proposed method establishes that two programs are equivalent if for any input sequence read by both programs, the output sequences produced by the programs are equal. Input (output) sequences of a program contain the values read from (written to) the environment and the corresponding orderings. According to this equivalence criterion, it is not only certified that the data values exchanged by the programs with the environment are equal but also that data are exchanged in the exact same order. This second element of the proof is especially relevant if the reactiveness of the software is taken into

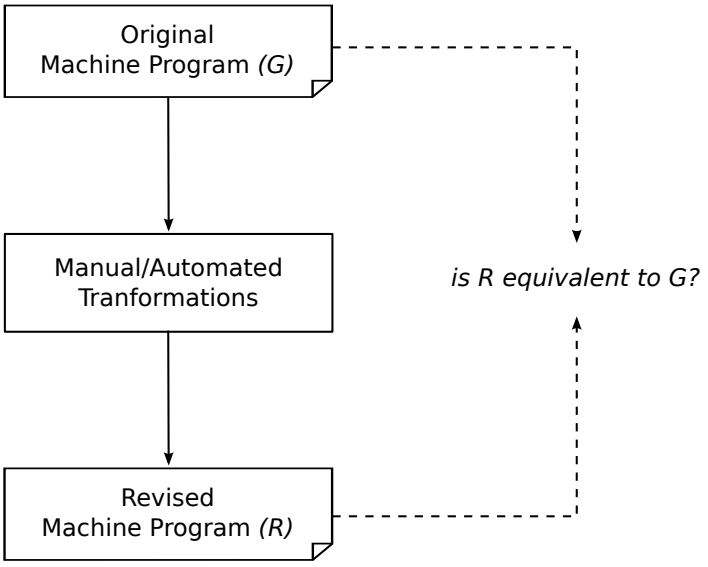

Figure 5.1: General view of the problem of comparing two machine programs

account.

For taking into account reactive behavior, as will be shown in this chapter, the program netlist is extended with a global model of the input/output behavior based on access sequences. This global model describes in a time-abstract way the transactions performed by the software at its interfaces, e.g., at the hardware/software interface. More specifically, time modeling is not performed in terms of clock cycles but in terms of abstract time points.

Note that formulating the equivalence problem in terms of input/output sequences becomes trivial for the case of transformational programs (e.g., arithmetic algorithms, sorting algorithms, etc.). For transformational software, input and output sequences contain each only one element because communication with the environment takes place at only at two points in time, namely at the beginning of execution when the inputs of the software are read and at the end of execution when outputs are returned. Therefore matching the sequences for two different transformational programs is straightforward. This situation clearly differs from the case of hardware-dependent software where input/output sequences describing interactions with the hardware or with other software components can become complex.

The following general steps are carried out to check equivalence of two machine programs G (for "golden") and R (for "revised") which may run on different hardware platforms:

- The program netlists for G and R are generated independently from the corresponding machine programs and the corresponding instruction cells of the processor on which the programs run. For this, the generation process presented in Chapter 3 is followed.

- Each generated program netlist is extended with a sequence-based model that represents the behavior of a given machine program at its interface. The logic necessary for

constructing the sequence model is simplified by exploiting information about the possible execution traces of the software contained in the EXG as well as information about the address space reached by the software. Section 5.1 details on how the sequence model is constructed.

- A software miter is built by using the program netlists, the sequence-based input/output model and a bijective mapping of input/output data environment locations from G to R which is provided by the user. For constructing the software miter, the proposed method takes advantage of the fact that program netlists are compositional. Section 5.2 gives the details on this step.

- Finally, a decision procedure, in particular a SAT solver, is called iteratively to prove the equivalence of G and R. Section 5.3 shows how incremental SAT solving techniques can be employed to reduce verification run-time.

At the end of this chapter, Section 5.5 presents different experiments evaluating the effectiveness of the proposed method for industrial low-level software in relevant equivalence checking scenarios such as automated/manual code transformations and code porting.

5.1 Sequence-Based Input/Output Model of Low-Level Software

This section describes a model for representing in a time-abstract way the transactions initiated by the software at its interfaces. The model employs the concept of input/output sequences to represent the exchange of data of the software with its environment.

Since the software is embedded in a reactive environment, it is necessary to include in the interface model a representation of the orderings describing the timing of the data exchange. A good example of a reactive program is a software-implemented bus agent. According to the bus specification, the bus agent must transfer frames of information ensuring that the transmission order of the individual fields composing a frame is preserved. If the transmission ordering produced by the agent does not comply with the bus specification, then the correctness of the system's behavior might be seriously compromised.

Broadly speaking, a hardware-dependent program exchanges information with its environment by reading or writing environment locations that belong to the system's address space. The set of data environment locations accessed by a given program is denoted by $A = \{a_1, a_2, \ldots, a_m\}$. Each a_j is an address of such a location. These locations are read (input data location) or written (output data location) by the software to communicate with the rest of the system. As shown in Figure 5.2, data environment locations for a hardware-dependent program correspond to registers in hardware devices or to shared

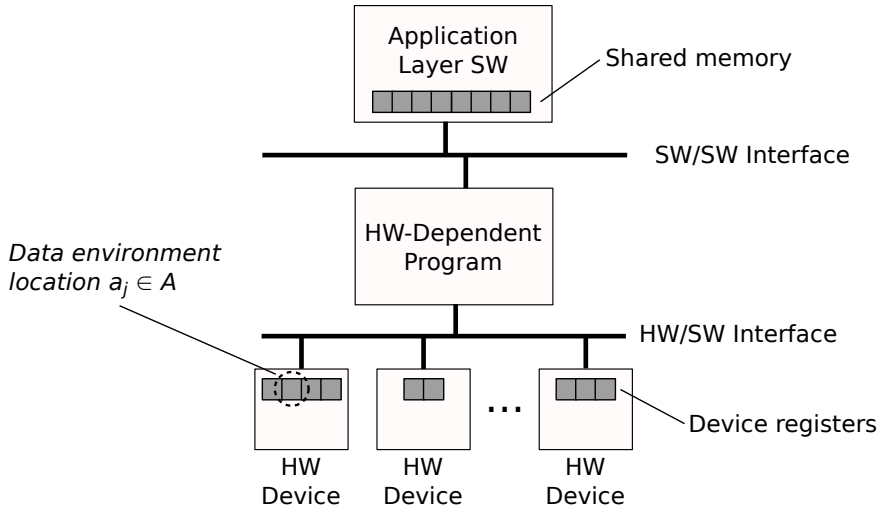

Figure 5.2: Data environment locations of a HW-dependent program

memory locations used for exchanging data with other software components or layers (e.g., the application layer).

According to its definition, A is a subset of the set of all data locations accessed by the software. It contains only data locations which are relevant to the external input/output behavior. Data locations corresponding to software variables in main memory that are not externally visible are not contained in A. Consequently, the interface model describes only the sequences of accesses that the software makes to the data locations in A. Since the proposed method focuses on representing transactions generated by software that typically has finite execution time (e.g., interrupt-based drivers, tasks with deadlines in an RTOS-based system, etc.), finite sequences are considered in the following formulation.

For each $a_j \in A$ a data sequence of accesses $(data_{a_j}(0), data_{a_j}(1), data_{a_j}(2), \ldots, data_{a_j}(n-1))$ is generated by the software such that for each consecutive pair of sequence points $(data_{a_j}(k), data_{a_j}(k+1))$ it holds that $data_{a_j}(k+1)$ occurs later in time than $data_{a_j}(k)$. As an example, Figure 5.3 shows the data sequence generated by a software implemented LIN (local interconnect network) [LIN02] master agent when performing a write transaction. For this example, the software sends data to the LIN bus by writing the transmission buffer (with address *TxBuff*) of a UART peripheral. Values written to the transmission buffer, denoted as $data_{TxBuff}$, by the software depict an output data sequence as shown in Figure 5.3.

The values of the input/output sequences can be only modified by input/output instructions of the software, i.e., instructions that access the data locations in A. In memory-mapped input/output systems, input/output instructions typically correspond to load/store instructions transferring information from/to device registers in hardware peripherals or IP cores. Other CPU architectures dispose of dedicated instructions for controlling input/output CPU ports.

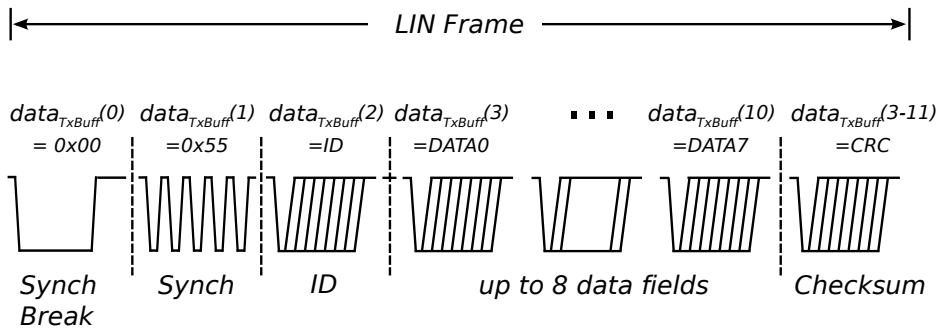

Figure 5.3: Data sequence for a software-implemented LIN master agent

The interface model presented here is independent of the kind of mechanisms used by the CPU to communicate with the environment.

For creating the sequence model, in a first step, input/output instructions of a given machine program need to be identified so that the sequence model is expressed only in terms of these instructions. This helps to keep the size of logic building the sequence model compact since only instructions defining the software's external behavior are considered. In a less efficient approach, the sequence model could also consider instructions different to input/output instructions, for instance, instructions addressing data memory. In this case, the sequence logic would need to resolve additionally whether the considered instructions address the locations of A or not. As a result, the obtained sequence model would be valid but the logic for representing it would be more complex.

In the program netlist, input/output instructions can be easily identified because after program netlist generation, the complete address space accessed by the software is known. As shown in Section 3.4, for each input/output instruction the values taken by the *port* signal *addr* (cf. Definition 1) are computed during generation of the program netlist by using simulation together with enumerative SAT, as described in Chapter 4. In practice, input/output instruction cells are identified as follows. From the program netlist only instruction cells having a port are inspected (instruction cells without port are discarded). If the values taken by the port signal *addr* (cf. Figure 3.7) belong to the set of data locations A then the corresponding instruction cell is marked as input/output instruction cell. Furthermore, for each address the location a_j the set $I_{a_j} = \{i_1, i_2, \ldots, i_{n_j}\}$ of all instruction cells accessing a_j is recorded. This complementary information will be further used in the sequel for building the sequence model.

5.1.1 Abstract Time Modeling of Input/Output Software Operations

In order to represent the ordering of accesses for each data location $a_j \in A$ a new time variable t_{a_j} is added to the program netlist. This variable represents the position (index)

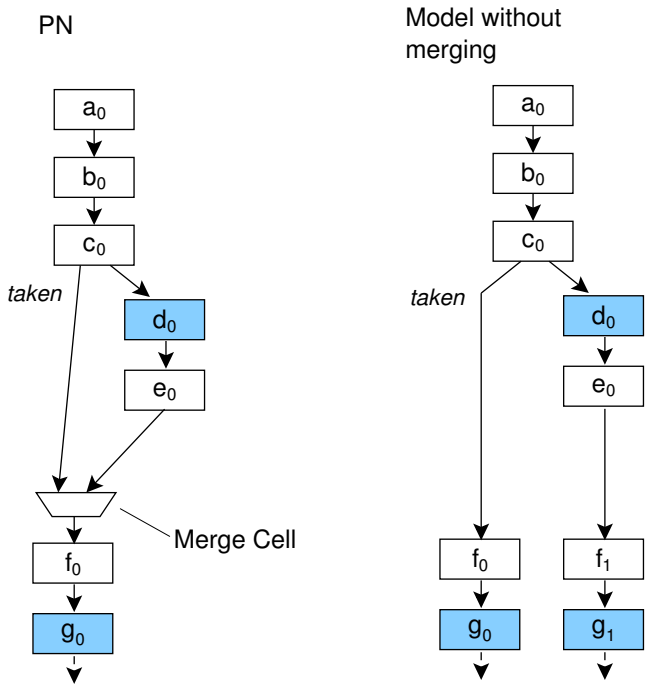

Figure 5.4: Non-unique access time points in merged PNs

of a particular element in the sequence of accesses to a given data location a_j. It can be understood as an abstract access time point.

In the program netlist, time variables (t_{a_j}) are necessary for describing precisely the values written or read by a given instruction cell as a function of the time. Since execution paths reconverge at merge cells, single instruction cells can belong to different execution paths. Hence, a single input/output instruction cell can access a given environment location at different time points and the data values exchanged by the instruction cell may vary as a function of the time point. Figure 5.4 illustrates this for a small segment of a program netlist (shown on the left side of the figure). In this example the same environment location is written at instruction cells d_0 and g_0. Therefore, instruction cell g_0 can write to the given environment location at time points one (when the branch is taken) and two (when the branch is not taken). Furthermore, the value issued by g_0 can vary depending on the particular time point at which the access is performed. This is because the value written at g_0 at time point two can be modified, with respect to the value written at g_0 at time point one, by one of the instruction cells which are on the not-taken branch, for instance, instruction cell e_0.

This situation is caused by the merge performed at the fanin of instruction cell f_0. If no merging was done, then instruction cells would access the environment at single unique time points. This is the case of the model shown on the right side of Figure 5.4. In this equivalent model, two different instruction cells g_0 and g_1 are included for representing each of the possible accesses. In particular, g_0 represents the access at time point one and g_1 the access

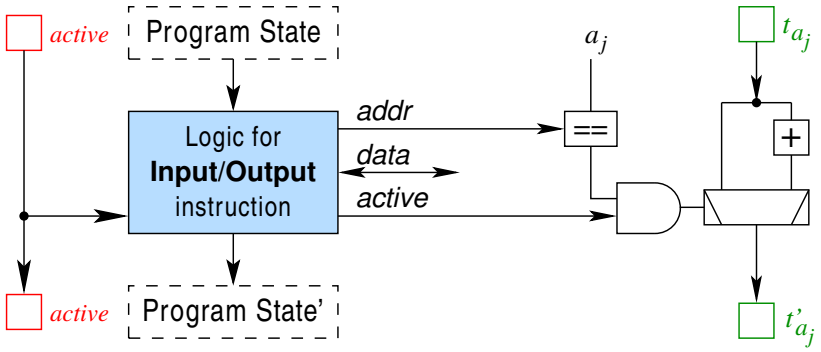

Figure 5.5: Logic for updating time variables used for data sequences

at time point two. Instead of removing merge cells of the program netlist (what notably would increase the size model), the proposed approach employs the new time variables above introduced. This allows representing sequences precisely while maintaining the compactness of the program netlist.

In the program netlist, time variables are updated uniquely at input/output instruction cells that access a particular data location a_j. For updating the abstract time variable t_{a_j}, the logic for input/output instruction cells is extended with incrementers as shown in Figure 5.5. The logic increments the value of the time variable by one if the given instruction cell accesses the location a_j and if the corresponding active signal is *true*, i.e., the instruction cell belongs to the path executed by the program. In the sequel, the value of the time variable at the instruction cell i is denoted by $i.t_{a_j}$.

5.1.2 Logic for Modeling Input/Output Sequences

Based on the time variables it is possible to construct the logic for each sequence element. The k-th written data value, denoted by $data_{a_j}(k)$, to a given location a_j, is described by the following if-then-else construct, built for the set I_{a_j}.

$$
\begin{aligned}
&data_{a_j}(k) := \\
&\quad \textbf{\textit{if}}\ (i_1.active\ \textbf{\textit{and}}\ i_1.addr = a_j\ \textbf{\textit{and}}\ i_1.t_{a_j} = k)\ \textbf{\textit{then}}\ i_1.data \\
&\textbf{\textit{else if}}\ (i_2.active\ \textbf{\textit{and}}\ i_2.addr = a_j\ \textbf{\textit{and}}\ i_2.t_{a_j} = k)\ \textbf{\textit{then}}\ i_2.data \\
&\quad \ldots \\
&\textbf{\textit{else}}\ i_{n_j}.data
\end{aligned}
$$

The logic for $data_{a_j}(k)$ builds a cascade of multiplexers where each multiplexer is connected to a single input/output instruction cell belonging to I_{a_j}. The selection signal of each multiplexer is set to *true* if three conditions are met, namely, (1) the instruction cell is active, (2) location a_j is addressed and (3) the time variable has the value k. The last two conditions are necessary because an instruction cell can access more than one data address

and, also, can perform the accesses at different abstract time points. For a concrete execution trace of the program only one port signal *data* is selected by the logic. When the select signal of a given instruction cell's multiplexer is *true* then the data value of the sequence at time point k is assigned the value of the port signal *data*.

The logic for $data_{a_j}(k)$ can be simplified because normally for a given sequence element k just a single instruction cell or a small subset of I_{a_j} can actually access the interface at the abstract time point k [VSB+13]. Therefore, not all instruction cells belonging to the set I_{a_j} need to be considered in the logic for $data_{a_j}(k)$ and consequently the amount of multiplexers can be reduced. Figure 5.6 presents one example. A program, whose CFG is shown on the right side of the figure, writes to the environment location *TxBuff* at instruction b. From the corresponding program netlist, shown on the right hand side of Figure 5.6, it can be deduced that $I_{TxBuff} = \{b_0, b_1, b_2\}$. Furthermore, the longest sequence for *TxBuff* has three elements. This happens when the program takes the right-most path through the program netlist, visiting the b-instruction three times. Figure 5.6 shows the resulting data sequence for this example. As can be seen, for each element of the access sequence there is only a single b-instruction cell that can write to the location and therefore, no multiplexer chain needs to be constructed for the building the sequence model.

This simplification not only reduces the size of the sequence model but also speeds up verification run-time since the decision procedure does not waste time anymore analyzing and eventually discarding instruction cells which are now known to be irrelevant for a particular sequence element.

For identifying the set of instruction cells, $\tilde{I}^k_{a_j} \subseteq I_{a_j}$, that access the location a_j at time point k the algorithm of Figure 5.7 is employed. The algorithm takes as input the EXG which is topologically sorted (V and E refer to the sets of nodes and edges of the EXG, respectively), the set of EXG root nodes S, and the set I_{a_j}. For a given location a_j, the algorithm propagates sets of access count values, i.e., sets of possible values of the index variable t_{a_j} as introduced above, through the EXG in topological order beginning at the root node(s). At a EXG node i_v the set of count values is denoted as $Kprop_{i_v}$. Whenever an instruction cell is visited that accesses address a_j, every access count in the set is incremented. At such an instruction cell, the set represents the possible indexes of the elements in the output sequence for a_j that are affected by the instruction. For instructions that do not access address a_j, the set of access counts is propagated without modification. After the traversal, for each sequence element k, the set of instruction cells $\tilde{I}^k_{a_j} \subseteq I_{a_j}$ that affect k can be computed. For the example of Figure 5.6 the algorithm returns the sets $\tilde{I}^0_{TxBuff} = \{b_0\}$, $\tilde{I}^1_{TxBuff} = \{b_1\}$, and $\tilde{I}^2_{TxBuff} = \{b_2\}$.

Another auxiliary signal also included in the sequence model is $active_{a_j}(k)$. It is asserted in any execution path where the k-th read or write access to a data location a_j occurs. As will be explained later, this signal helps to speed up the verification by ensuring that only the execution paths related to the k-th access sequence point are considered by the decision

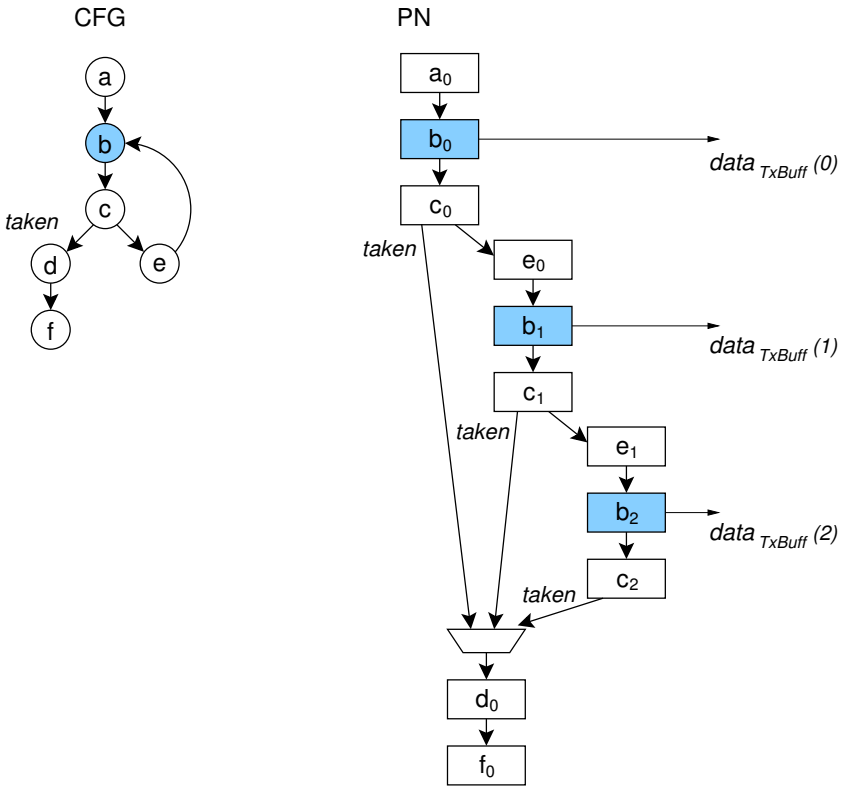

Figure 5.6: Example of a simplified sequence model

```
1: COMP_SEQ_POINTS(V, E, S, I_{a_j}) {
2:     initialize start nodes of EXG
3:     for each i_s ∈ S {
4:         Kprop_{i_s} ← {0};
5:     }
6:     traverse the nodes of the EXG
7:     for each i_v ∈ V {
8:         if i_v ∉ S {
9:             Kprop_{i_v} ← ∅;
10:        }
11:        for each i_p ∈ Pred[i_v] {
12:            Kprop_{i_v} ← Kprop_{i_v} ∪ Kprop_{i_p};
13:        }
14:        check if i_v accesses address a_j
15:        if i_v ∈ I_{a_j} {
16:            K̃^{i_v}_{a_j} ← Kprop_{i_v};
17:            Ktemp ← ∅;
18:            for each k ∈ Kprop_{i_v} {
19:                Ĩ^k_{a_j} ← Ĩ^k_{a_j} ∪ {i_v};
20:                increment sequence point values
21:                Ktemp ← Ktemp ∪ {k + 1};
22:            }
23:            Kprop_{i_v} ← Ktemp;
24:        }
25:    }
26: }
```

Figure 5.7: Algorithm for precomputing the sequence points of a given environment location

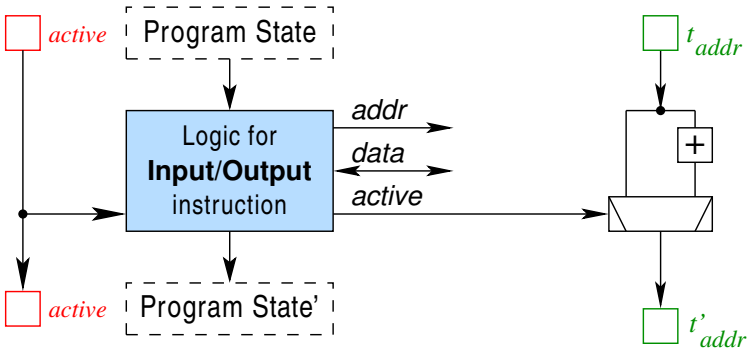

Figure 5.8: Logic for updating time variables used for address sequences

procedure in a particular proof. All other paths (if any) related to other time points will be disregarded by the solver.

$$
\begin{aligned}
&active_{a_j}(k) := \\
&\quad \textbf{\textit{if}}\ (i_1.active\ \textbf{\textit{and}}\ i_1.addr = a_j\ \textbf{\textit{and}}\ i_1.t_{a_j} = k)\ \textbf{\textit{then true}} \\
&\textbf{\textit{else if}}\ (i_2.active\ \textbf{\textit{and}}\ i_2.addr = a_j\ \textbf{\textit{and}}\ i_2.t_{a_j} = k)\ \textbf{\textit{then true}} \\
&\quad \ldots \\
&\textbf{\textit{else if}}\ (i_{n_j}.active\ \textbf{\textit{and}}\ i_{n_j}.addr = a_j\ \textbf{\textit{and}}\ i_{n_j}.t_{a_j} = k)\ \textbf{\textit{then true}} \\
&\textbf{\textit{else false}}
\end{aligned}
$$

While the previous analysis describes the sequences of data exchanges for single data locations, in some cases it is also important to consider the interleaving of accesses to different data locations. For example, for the bus agent introduced above, it can be required that the initialization of the surrounding hardware takes place before the data payload can be transferred, otherwise, the hardware periphery would not be ready to communicate with the agent. Therefore, verification needs to prove that initialization is executed before the data payload transmission takes place. For this purpose, a model for address sequences can be employed. For the bus agent example, the first element of the address sequence should equal the value of the address used for initialization and the rest of the elements of the sequence should correspond to the address used for transferring the payload and so on. The model generation for the address sequence follows the same strategy based on incrementers as was introduced for the data sequences (see Figure 5.5). The logic added for updating time points of address sequences is shown in Figure 5.8. This logic is again added to all input/output instruction cells of the program netlist. However, note that for this case, comparators are not required as all environment locations in the set A need to be considered simultaneously. Likewise, the corresponding logic for each element of the address sequence is described as follows.

$addr(k)$:=
  if $(i_1.active$ ***and*** $i_1.t = k)$ ***then*** $i_1.addr$
else if $(i_2.active$ ***and*** $i_2.t = k)$ ***then*** $i_2.addr$
  ...
else $i_{n_j}.addr$

The functions for the signals $data_{a_j}(k)$, $active_{a_j}(k)$ and $addr(k)$ encapsulate the interface of the software with the environment and are used next for solving the equivalence checking problem.

5.2 Software Miter

The previous model of the software interface makes it possible to formulate the *equivalence* of hardware-dependent programs in a straightforward way as follows.

Consider two low-level hardware-dependent programs G and R. For these programs, the inputs and outputs are defined by the user as sets of input data locations X_G, X_R and output data locations Y_G, Y_R.

The user provides additionally a bijective mapping that assigns to every input data location $x_G \in X_G$ of program G an input data location $x_R \in X_R$ of R. Also, a bijective mapping assigning elements of Y_G to elements of Y_R must be provided. Then, the program netlists for G and R and the corresponding sequence models are created with respect to the user-defined environment locations. Finally, the two program netlists together with their corresponding interface models are combined as follows.

Mapped inputs are set equal by connecting every input sequence element $data_{x_G}(k)$ of program G with the corresponding sequence element $data_{x_R}(k)$ of program R. This ensures that the input assignments of the programs are equal as established in the equivalence criteria defined at the beginning of this section. At this point, it is expected that the sequence lengths are the same for both programs. If this is not the case, then no sequence mapping is possible and the programs are not considered equivalent.

Similarly, for each sequence element of the mapped outputs $data_{y_G}(k)$, $data_{y_R}(k)$ and their respective active signals $active_{y_G}(k)$, $active_{y_R}(k)$ the following set comparisons are implemented.

$$\begin{aligned} &\text{equiv}(y_G, y_R, k) = \\ &(active_{y_G}(k) = active_{y_R}(k))\ \textbf{\textit{and}} \\ &(active_{y_G}(k)\ \textbf{\textit{implies}}\ data_{y_G}(k) = data_{y_R}(k)) \end{aligned} \tag{5.1}$$

The function equiv(y_G, y_R, k) can be seen as a property, in particular a Boolean predicate. The first condition in Equation 5.1 expresses that a sequence element must be produced by both programs under exactly the same input conditions. Remember that both, $\mathit{active}_{y_G}(k)$ and $\mathit{active}_{y_R}(k)$, are asserted for exactly the input conditions that make the respective program, G or R, produce the output sequence element k. They are deasserted if k is not produced. The second condition states that whenever the output sequence element k is produced then its data value must be the same in both programs.

For checking equivalence of the address interleavings the same kind of comparison can be implemented for the sequence elements $\mathit{addr}_G(k)$ and $\mathit{addr}_R(k)$ of the programs G and R, respectively.

The final model results in a software miter with a set of mapped inputs and a vector of comparison for the outputs. It must be checked whether or not all of these comparisons always yield *true*. If this is the case then both programs G and R are equivalent.

5.3 Equivalence Checking Using SAT

In order to compute the proofs expressed by Equation 5.1, the procedure takes each pair of mapped outputs (y_G, y_R) and calls the SAT solver iteratively for all related sequence points as shown in the algortihm of Figure 5.9. As for the inputs, sequence lengths for the outputs are also expected to be the same for each environment location, i.e., for the mapped locations y_G, y_R, the corresponding sequence lengths m_G, m_R must have the same value.

```
1:   for k ← 1 to m_G do {
2:     solve: SAT(¬equiv(y_G, y_R, k));
3:   }
```

Figure 5.9: Iterative SAT proofs for a given data environment location

For each iteration k the SAT solver enumerates all involved execution paths by using the *active* signal mechanisms described in Section 3.1. The Equation 5.1 places Boolean constraints on the *active* signals of the compared output sequence elements. By taking decisions and propagating values into the *active* signals of the elements of the program netlist, the SAT solver explores the consequences of these constraints on the control flow of each of the programs, G and R. The SAT search is, thereby, guided to consider only those execution paths that are related to the equivalence check of the currently considered output sequence element k. By construction, the *active* flags in both program netlists are assigned by the SAT solver such that only related execution paths in the two program versions are considered simultaneously. Multiple executions paths can be implicitly considered at once. Clauses

can be learned that express relationships between corresponding execution paths in both programs. All of this helps to significantly enhance the efficiency of the SAT proof.

It is clear that the cone of influence of the proofs in Figure 5.9 grows incrementally with k because, if an access happens at sequence index $k+1$ then an access to the same port has happened at sequence index k in the same execution path. This means that the cone of influence of the constraint equiv$(y_G, y_R, k+1)$ contains the cone of influence of the constraint equiv(y_G, y_R, k).

The procedure takes advantage of this fact and employs *incremental SAT* techniques to re-use the knowledge acquired by the SAT solver when proving equiv(y_G, k) for the subsequent proof of equiv$(y_G, k+1)$. The individual points in the sequence of equivalence proofs can be seen as "internal equivalences" for all later proofs and have a similar speed-up effect as internal equivalences in combinational hardware equivalence checking.

5.4 Tool

The functionality of the verification platform FCK has been extended in order to implement the methods proposed in this chapter. Two new main components are added to FCK. The first component, shown in Figure 5.10, creates the input/output sequence model as described in Section 5.1. This component is employed twice (one for each of the programs to be compared) by the tool. The block called *Sequence Points Computation* implements the algorithm of Figure 5.7. The EXG and the PN, which are inputs to this component, are generated after running the tool presented in Section 4.4. Information about the reachable address space, also produced during program netlist generation, is also taken as input by the tool. This information is part of the data structure holding the information of the EXG.

The second component of the tool, seen in Figure 5.11, builds the software miter from the program netlists and their sequence models. This job is done by the block called *SW Miter Builder* based on the mapping information provided by the verification engineer. If required, mappings and comparison functions can be adjusted by the user.

The block *SAT Interface* calls iteratively the SAT solver for proving incrementally the checks of Equation 5.1. MiniSAT [ES03b] is used in the current implementation of the tool. In the case of a bug, the tool presents a counterexample as a pair of active program traces showing a scenario in which both programs behave differently. For every trace, the tool presents the values of the program states along active paths on the program netlist corresponding to a given input assignment of the mapped inputs. Note that there can be no false counterexamples because the program netlists in the software miter exactly represent all execution paths beginning at the initial states of the programs and there are no approximations of state sets in the model.

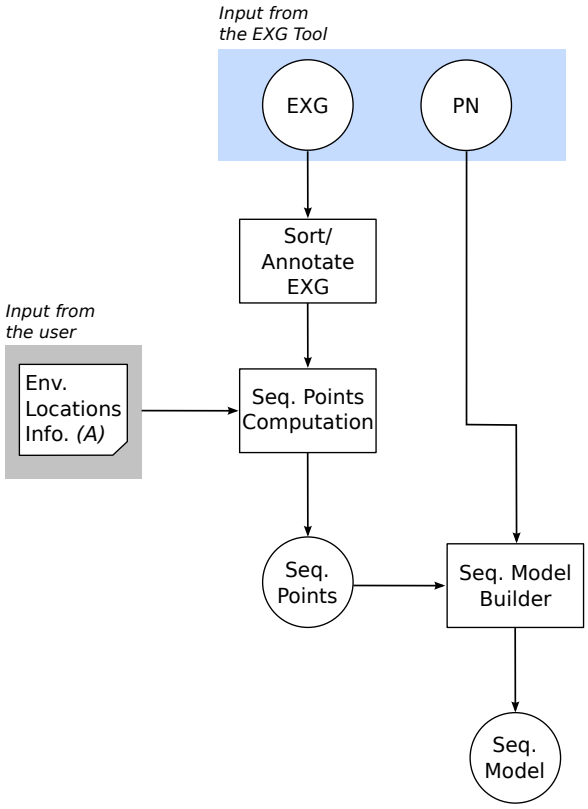

Figure 5.10: Tool for generating the input/output sequence model

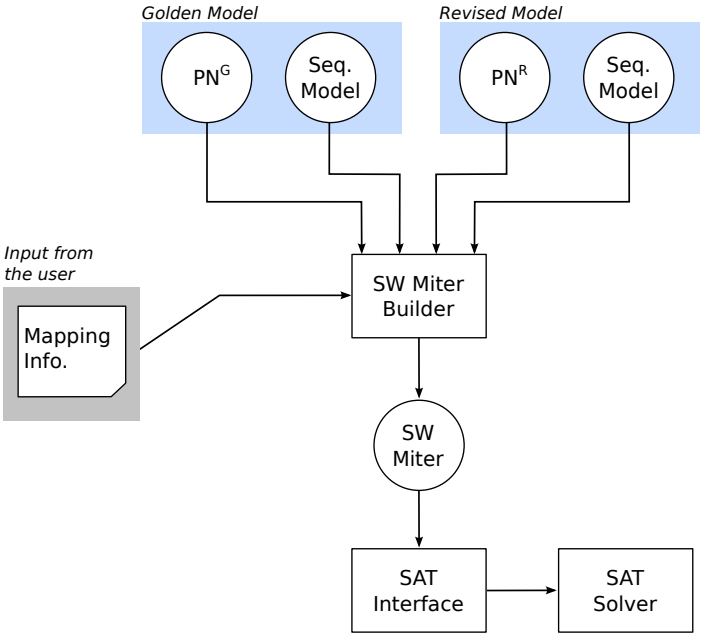

Figure 5.11: Tool for generating the software miter

5.5 Experimental Results

The following experimental evaluation demonstrates that it is feasible to perform equivalence checking of industrial hardware-dependent programs using the approach presented above. The experiments are based on two relevant examples of hardware-dependent software, namely, an industrial software implementation of the automotive protocol LIN and a serial synchronous interface. Both examples are mainly control driven.

All experiments are performed on an Intel Xeon E5420 CPU at 5 GHz with 16 GB RAM.

5.5.1 LIN Driver

The described equivalence checking approach was applied to the LIN driver considered in Section 4.5. The following description summarizes the main characteristics of the LIN node.

The LIN node can be configured such that transmission and reception modes are allowed, data-length is variable up to 8 bytes, and the used IDs can be modified. The driver interacts with the LIN bus by means of an UART containing status, configuration and data registers. The UART is accessible as a memory-mapped input/output device. The driver also interacts with the user application via shared memory consisting of the received data, the data to be transmitted and additional status information (e.g., the status of the transmission).

Different verification scenarios were considered in which the code was subjected to automated and/or manual transformations. The GCC compiler was used applying three different optimization levels to the source code, starting from level zero (LIN l0, cf. Table 5.3) with no optimizations being activated and increasing the aggressiveness of the compiler optimizations up to the maximum level two (LIN l2). Engineering changes were introduced into the code in different parts of the program (LIN modif.). For all these cases the program versions were verified to keep the same input/output behavior. Experiments were also conducted with a version of the driver containing an error in the computation of the checksum (LIN error). This code was obtained based on the code version modified by engineering changes (LIN modif.) and by making further manual changes that introduced an error.

Table 5.1 shows the size of the program netlists and the times required to generate them. The tool for program netlist generation presented in Chapter 4 has been used in this step. Generation times include the times necessary to explore the address spaces accessed by each instruction cell that interacts with the environment or with data memory.

Before calling the SAT solver it was checked that all versions of the LIN driver have the same number of access sequence points. This was a first indication that the programs are equivalent in all cases. Table 5.2 presents information on the interface model. Times to build the access sequence, as explained in Section 5.1, were negligible since only a simple graph traversal of the EXGs are needed in order to identify input/output instruction cells.

For each output sequence point, a SAT check was then performed. Data for the software

Program	Instr. Cells	CPU time (s.)
LIN (l0)	7856	26.8
LIN (l1)	5908	12.4
LIN (l2)	5342	11.6
LIN (modif.)	6032	12.9
LIN (error)	6007	12.1
SER (orig.)	6655	698.4
SER (ported)	5105	645.0

Table 5.1: CPU times for PN model generation

miters and the proof times are shown in Table 5.3. In all cases, except for the comparison LIN (l1) vs. LIN (error), the programs were proven equivalent according to the proposed formulation. For the equivalence proof of LIN (l1) vs. LIN (error) a counterexample was returned by the SAT solver. This counterexample was composed of two $active$ execution traces: one for LIN (l1) and the other for LIN (error). The counterexample presents an input assignment to the mapped inputs of both programs which, in the considered case, produced a mismatch of the values written to the UART's transmission buffer. The value written to the UART buffer corresponded to the checksum field of the LIN-protocol and, specifically, it could be observed that the erroneous behavior occurred at the 12th time point of the output access sequence belonging to this buffer.

For the proofs the technique described in Section 5.3 was employed. By using internal equivalences detected by incremental SAT, CPU times could be reduced to about 36% on average.

5.5.2 Serial Synchronous Interface

The interface implements a serial synchronous receiver using a round-robin scheme that iteratively samples a clock synchronization signal and a data-input serial line. In every transfer, the data is passed byte-wise to the user application until a 32-bit word has been received. In order to guarantee a finite unrolling a model generation constraint was added that limits the number of sampling actions to ten (five for each clock-phase).

The code was initially developed for the Aquarius (SER (orig.)) and was later ported to run on the ARM7-TDMI architecture (SER (ported)). Then, the equivalence of both versions of the code was formally proven .

The serial synchronous interface provides an interesting case study since it contains a complex nested-loop structure with a high number of branches. On the other hand, when compared to the LIN driver, this program has low traffic with data memory. Therefore, the times for the model generation in this case were dominated by the checks on the *active*

Program	No. locations input	No. locations output	No. seq. points input	No. seq. points output
LIN	6	5	25	42
SER	2	4	992	4

Table 5.2: Program netlists: interface model

signals performed for path pruning at every branch during the unrolling. The times for model generation were: 698.4 s for SER (orig.) and 645.0 s for SER (ported).

Table 5.2 shows information on the interface model. Both versions of the serial interface presented the same number of access points on the interfaces. Input sequence points correspond to the individual samples of the serial data line and of the clock signal. Output points of the interface relate the corresponding storing accesses of the received data to the user application.

Table 5.3 presents the information on software miter construction and proof times. The programs were proven to be equivalent. Due to incremental SAT run times were reduced to 27%.

Golden	Revised	Miter size (inst. cells)	Proof time (s.)	Memory usage (MB)
LIN (l0)	LIN (l1)	13764	692.3	777.5
LIN (l0)	LIN (l2)	13198	766.2	698.1
LIN (l1)	LIN (l2)	11520	419.5	343.0
LIN (l1)	LIN (modif.)	11904	500.5	470.5
LIN (l1)	LIN (error)	11915	295.1	336.2
SER (orig.)	SER (ported)	11760	188.2	404.6

Table 5.3: Equivalence checking: proof results

Chapter 6

Cycle-Accurate HW/SW Co-Verification of Firmware-Based Designs

This chapter deals with hardware/software co-verification by property checking of firmware-based designs where processor and firmware are directly integrated with the rest of the system without using standard bus interfaces. As described in Section 1.1, commonly for these design approaches, processors with high timing predictability are employed allowing the firmware behavior to be integrated into the surrounding hardware in a cycle accurate way by means of a wrapper RTL (cf. Figure 1.1).

Hardware/software co-verification of firmware-based designs becomes an important but also difficult task when the software is reactive, i.e., when the tight interaction between the processor executing the software and the surrounding hardware needs to be examined in sufficient detail. A straightforward approach capable of delivering cycle-accurate precision is to model the processor with its program and data memory at the hardware RT level and to use standard hardware verification techniques such as Bounded Model Checking [BCCZ99] or Interval Property Checking [NTW+08] to verify properties for this model. Figure 6.1 illustrates the unrolling performed in standard BMC for a firmware-based design. At every clock cycle, there is a copy of the system's transition logic including gate-level representations of the processor, the instruction memory with the firmware (stored as machine code), the data memory, the wrapper RTL, and the additional system hardware. A formal property can be expressed as a propositional logic formula over arbitrary signals of the unrolled circuit and can be added as combinational circuitry to the model, (this is not shown in Figure 6.1). The resulting problem instance is converted to a single formula in conjunctive normal form and checked using a SAT solver.

Verification based on standard BMC has been shown to be appropriate for verifying pure hardware designs [PBG05, KGN+09]. However, as pointed out in Chapter 3, employing standard BMC technology in a straightforward way also to software, and in particular to a hardware/software system as the one shown in Figure 6.1, bears important complexity

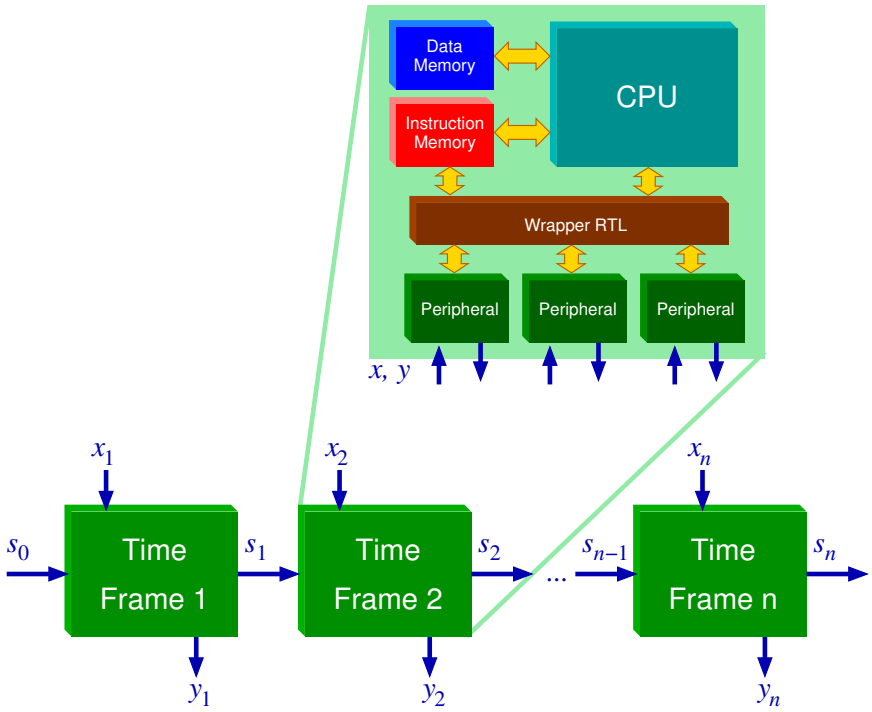

Figure 6.1: Straightforward BMC-style verification approach

challenges. These complexity challenges arise from the fact that the unrolling of BMC implicitly models all possible executions of the software for all possible inputs without any guidance regarding the possible execution traces the software can take. A time frame i represents all possible system states at clock cycle i including the possible states of the program, which, obviously, depend on the possible program states at earlier clock cycles. Therefore, the SAT solver implicitly enumerates all these possible states for proving the property, and, since it has no guidance of any form, does this very inefficiently. Furthermore, the model can usually not be simplified by constant propagation because a time frame represents not a single location in the program but many possible locations. Approaches relying on a straightforward BMC-style unrolling therefore can work only for small problem instances, even for systems with simple processors as the ones used in firmware-based design styles. Experiments presented at the end of this chapter confirm this for different case studies.

This chapter proposes a new modeling approach which preserves the precision of standard BMC while also modeling explicitly the possible instruction sequences that can actually occur in real program runs. For this purpose, the software and the hardware on which it runs (CPU, instruction memory, and data memory) are modeled by means of the program netlist. The program netlist is then extended with a cycle accurate model of the hardware/software interface to create a computational model of the system that is cycle accurate and therefore compositional in RTL hardware descriptions. The techniques used in this chapter for creating

this compositional model are similar to the ones presented in Chapter 5 with the difference that here the resulting representation is cycle accurate.

This chapter is organized as follows. Section 6.1 gives a global overview of the proposed model for hardware/software co-verification. Section 6.2 presents the abstractions performed to model the hardware/software interface in terms of an *timed interface model*. A short description of the extensions done to the FCK verification platform is presented in Section 6.3. Finally, Section 6.4 shows the experimental evaluation of the proposed model.

6.1 Joint Hardware/Firmware Model

As shown in Figure 1.1 and Figure 6.1, systems are composed of processor cores, their surrounding hardware, also called "uncore" hardware, and the software to be executed. In the following analysis, the uncore hardware (wrapper RTL, peripherals, and relevant IP components) should be distinguished from the core hardware (CPU, data memory, instruction memory, and communication infrastructure) with its software. For the sake of a simple terminology, in the sequel, the term *hardware* is only used for the uncore parts of the system. In the hardware-dependent software view of this chapter, since the software behavior is described completely in terms of the core hardware, the term *software* or *firmware* subsumes not only the considered program but also the core hardware on which it runs.

The model to be presented precisely describes the functional behaviors of the system cycle by cycle over finite-time windows. For taking both the hardware and the firmware into account, an approach is taken that combines two different kinds of unrollings as depicted in Figure 6.2. On the one hand, the uncore hardware is unrolled in a classical BMC fashion by instantiating a copy of the associated transition relation at every time step (lower part of Figure 6.2). On the other hand, the unrolled software is modeled by a program netlist, instruction by instruction, representing all possible executions of the programmable system (upper part of Figure 6.2). Time granularity in this part of the model is given by processor instructions and not by clock cycles.

Every core architecture contains input/output instructions that allow interaction of the software with the uncore hardware. In the proposed model, input/output instructions are modeled by means of input/output instruction cells which are equipped with *ports* as introduced in Definition 1. In Figure 6.2, for example, input/output instructions are represented by instruction cells i_1, i_4, i_6 and i_8 and their corresponding ports $port_{i_1}, port_{i_4}, port_{i_6}$ and $port_{i_8}$.

For constructing the hardware/software model, the proposed method takes advantage of the fact that a program netlist can be instantiated as a hardware component and can be extended with a new model of the processor's interface, called *timed interface model* (see Figure 6.2), that allows to combine the program netlist together with the hardware into a single model.

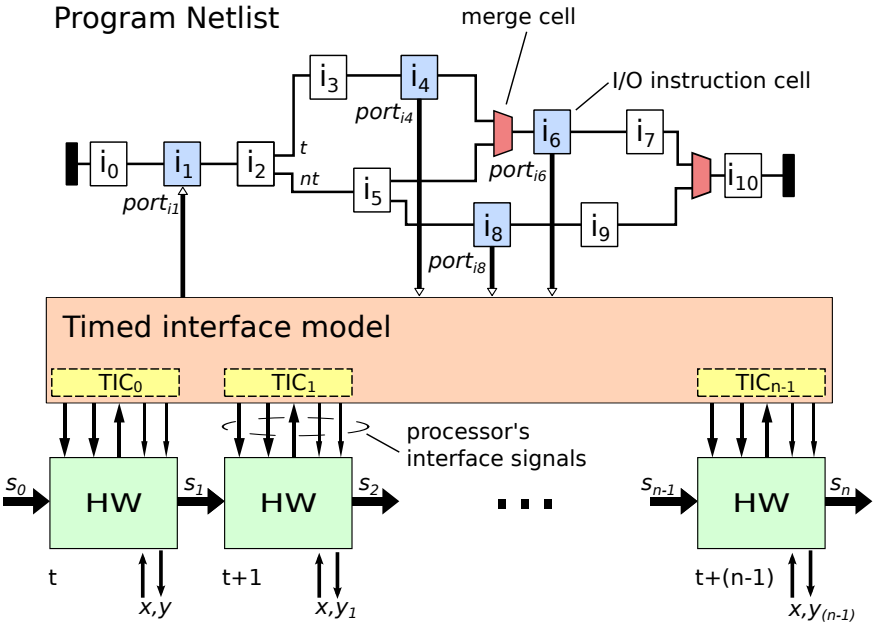

Figure 6.2: Mixed unrolling approach for efficient HW/SW modeling

The timed interface model represents the input/output operations (such as read, write, etc.) that can take place between the hardware and the software at every time frame of the unrolling. Different simplifications (to be presented in the rest of this chapter) are performed in order to reduce the amount of logic required to create the timed interface model and to ease the reasoning on the resulting composed hardware/software model. These simplifications are done under the premise that the set of input/output instructions of the firmware that can interact with the relevant system's uncore hardware at a specific time point of the unrolling is known.

With this timing information the logic for modeling the interface can be reduced because at a given time frame the required logic depends only on the relevant input/output instructions that actually interact with the hardware at the time frame and not on other instructions accessing the interface of the processor at other time frames or accessing other domains that do not correspond to the relevant uncore hardware. Since this information is explicitly added to the interface model, the decision procedure used to reason on the model can directly exploit this information instead of deducing it from another more complex representation.

In general, developing such a model of the interface may appear complicated. However, for the approach proposed in this chapter, it turns out doable, since the required algorithms can employ information readily available as a result of the generation procedures for program netlists. In particular, when determining the timing of input/output instructions, the proposed method benefits from working directly on the execution graph which contains explicit information about the control flow of the firmware (in terms of valid instruction traces taken by the firmware) and its corresponding accessed memory address space.

6.2 Timed Interface Model

Most of the challenges of creating a combined hardware/software model stem from the combination of two unrolling styles with different temporal resolution (cf. Figure 6.2). Since the hardware is unrolled in a cycle-accurate manner, state variables and, in particular, signals connecting to the processor are contained in the model for every analyzed time frame. For the firmware, however, the situation is different because the program netlist as presented originally in [SVF+13b] is a time-abstract model. As explained in Section 3.2, instruction cells atomically represent how a given ISA instruction modifies the program state, abstracting from any intermediate steps carried out by the CPU during instruction execution. This is also true for input/output instructions (cf. Figure 3.7), which do not model how the interaction between the processor and the hardware specifically takes place in time. In the same way, even though a program netlist represents sequences of instructions executed by the processor, the state of a program at a particular absolute time point is not known because it depends on the inputs of the program and thus on the execution path taken by the firmware.

These issues can be resolved by adding a cycle-accurate model of the processor's interface to the overall model that accurately represents the interface signals of the processor for each time frame of the unrolling. In the following, it is shown in detail how such a processor's interface model can be constructed by (1) adding new abstractions describing the behavior of the processor's interface and by (2) creating additional resolution logic for deciding the time points and the values communicated between hardware and software.

6.2.1 Timed Interface Cells

A *timed interface cell* (TIC) is defined as an abstract model representing the state of the signals belonging to the processor's interface at a particular clock cycle. TICs (the small boxes inside the timed interface model in Figure 6.2) are hardware-dependent models specific to each processor architecture. TICs can be classified into DATA-TICs and IDLE-TICs. DATA-TICs transport input/output information such as data, addresses and control values between the software and the uncore hardware. IDLE-TICs represent the processor's interface when there is no exchange of information between hardware and software. The signals of a TIC instance connect to the corresponding copy of the uncore hardware at a given time frame in the unrolling (cf. Figure 6.2).

Similar as with the instruction cells (cf. Section 3.2), TICs are combinational models that can be described using a hardware description language (HDL). A DATA-TIC typically propagates values between the software and the uncore hardware without performing any modification.

At every time frame of the unrolling exactly one TIC is instantiated that models the interactions that can occur at the time frame. A DATA-TIC is instantiated if there is data

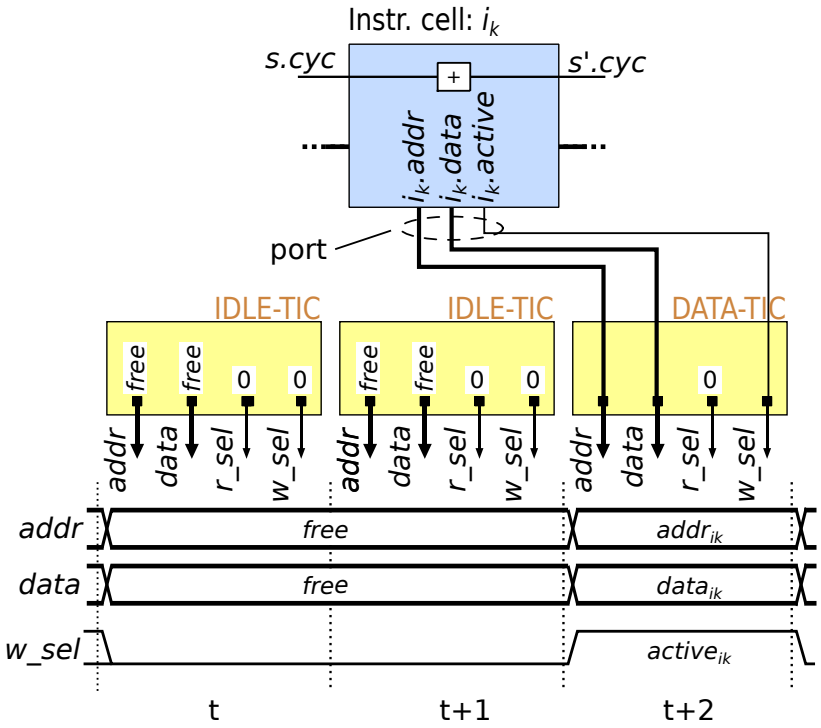

Figure 6.3: Example of TICs for a non-pipelined multi-cycle architecture

exchange at the given time frame, otherwise, an IDLE-TIC is used. Information about what kind of TIC should be instantiated at a specific time frame is obtained from the timing analysis described in Section 6.2.3. If at certain time frame there is at least one input/output instruction cell in the program netlist that accesses the processor's interface, then a DATA-TIC is placed at the time frame. On the other side, if no input/output instruction cell can access the interface at certain time frame then an IDLE-TIC is placed at the time frame.

Since an instruction can be executed at different time points, depending on the particular path executed by the software, a single input/output instruction cell can connect to different DATA-TICs. In the same way, a single DATA-TIC can connect to different input/output instruction cells since for different execution paths different input/output instruction cells can be active at the time point defined by the related DATA-TIC. Section 6.2.2 presents how input/output instruction cells connect to DATA-TICs by means of *resolution logic blocks*. Contrary to DATA-TICs, IDLE-TICs do not require connection with any instruction cell of the program netlist.

In Figure 6.2 the instruction cells i_1, i_4, i_6 and i_8 correspond to input/output instructions and therefore connect to DATA-TICs. In the same example, the white instructions do not exchange data with the uncore hardware and therefore IDLE-TICs are instantiated in the interface model for representing them. There is no connection between IDLE-TICs and the white instruction cells.

Figure 6.3 details on how a STORE instruction writing to the hardware is modeled using different TICs. In this example, the processor has a non-pipelined three-phase multi-cycle architecture. A write operation is performed at the third clock cycle. For each ISA instruction, three TIC instances are needed. As shown in the figure, the first two clock cycles are represented by IDLE-TICs since during these clock cycles no data is exchanged. Also, there is no connection of these IDLE-TICs with the program netlist. Data transfer happens at the third clock cycle which is modeled by a DATA-TIC. This last TIC connects the uncore hardware at the third time frame with the port of the STORE instruction cell i_k.

Figure 6.3 also shows how non-determinism is used in modeling the processor's input/output interface through TICs: The control signal *w_sel* specifies the validity of the output data *data*. At time points where *w_sel* is 0 the *data* signal is left undetermined (modeled by an unconstrained "free" input). In this way, details of the processor's implementation which are not relevant to the model are abstracted away.

6.2.2 Resolution Logic

In order to interconnect DATA-TICs and program netlist resolution logic blocks are employed. For resolving program netlist outputs, i.e. for output signals belonging to the ports of input/output instruction cells such as address, output data and active bit, *output resolution blocks* are employed. Similarly, for program netlist inputs *input resolution blocks* are used.

Output Resolution Logic

If two or more input/output instructions cells can access the interface of the processor at the same time frame then extra control logic is needed in order to resolve the outputs issued by the software to the uncore hardware. This situation is in general possible, since for different execution paths, different instructions can be executed at the same clock cycle. Figure 6.4 shows an example where instruction cells i_4, i_6 and i_8 (from Figure 6.2) can write to the hardware at the same clock cycle if the CPU timing of Figure 6.3 is assumed. For cases like this, an output resolution logic block (RL) is instantiated that decides which of these input/output instructions cells drive the involved processor's interface signals.

An output resolution logic block takes as input all output signals belonging to the ports (c.f. Definition 1) of the involved input/output instruction cells and decides which of them connects to the corresponding DATA-TIC. In general, this decision depends on the active signal values of the ports (which in turn depend on the inputs of the program) and on whether accesses are performed at the given time point. This last condition needs to be regarded, since a single instruction may access the interface at two or more different time points and thus the values written by the instruction to the environment may depend on the particular time at which the input/output access is performed.

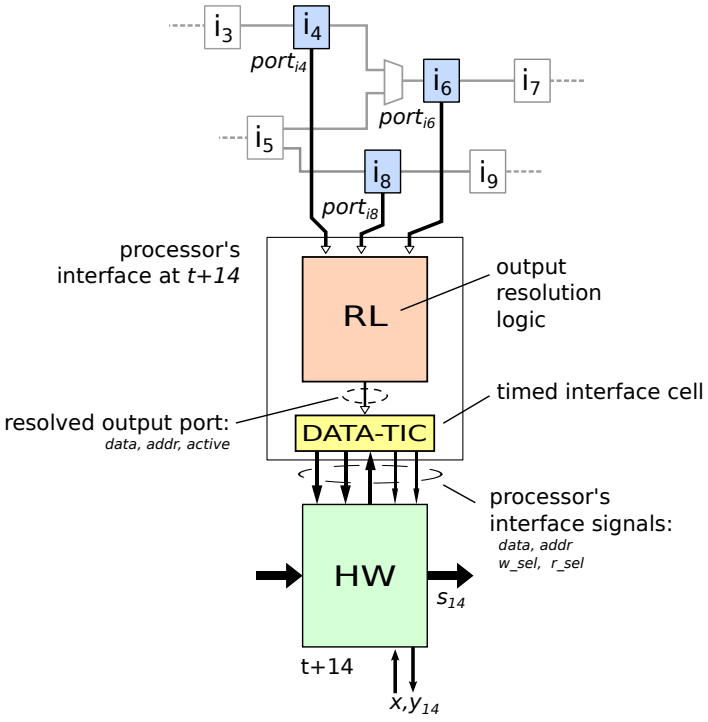

Figure 6.4: Example of unrolled model containing an output resolution block

For resolving the output values issued by input/output instructions at a particular time point a time variable, called *cyc*, is added to the program state (see the instruction cell in Figure 6.3). This variable encodes the clock cycles at which input/output instructions cells access the processor's interface. Section 6.2.3 gives more details on how this time variable is incorporated to the program netlist.

If at clock cycle k the set of input/output instruction cells $W_k = \{i_1, i_2, ..., i_m\}$ can write to the processor's interface then the resolution logic for the output data value, $data(k)$, is defined as follows.

$$
\begin{aligned}
&data(k) := \\
&\quad \textbf{\textit{if}}\ (i_1.active\ \textbf{\textit{and}}\ (i_1.cyc = k))\ \textbf{\textit{then}}\ i_1.data \\
&\textbf{\textit{else if}}\ (i_2.active\ \textbf{\textit{and}}\ (i_2.cyc = k))\ \textbf{\textit{then}}\ i_2.data \\
&\quad \ldots \\
&\textbf{\textit{else if}}\ (i_m.active\ \textbf{\textit{and}}\ (i_m.cyc = k))\ \textbf{\textit{then}}\ i_m.data \\
&\textbf{\textit{else}}\ free
\end{aligned}
$$

This logic describes a chain of multiplexers in which $i_l.data$ and $i_l.active$ belong to the port signals of the instruction cell i_l and $i_l.cyc$ is the time variable previously introduced.

For resolving the values of the port signals *address* and *active*, a similar multiplexer chain is constructed. For this purpose, the logic of the multiplexer's select signals is reused, from

the selection logic implemented for *data*, and the signals to select are changed from *data* to *address* and *active* in each of the multiplexer chains.

By using W_k the amount of logic is reduced since output resolution logic blocks are instantiated only at time points when they are in fact needed. Furthermore, each resolution block takes into account input/output instruction cells contained in W_k and not other instructions which can not reach the interface of the processor at the particular time frame. Note that if W_k was unknown then output resolution blocks would be required at every time point and additionally for each block all possible input/output instruction cells belonging to the program netlist would need to be regarded. Section 6.2.3 details on how the elements of W_k can be determined.

Input Resolution Logic

As shown previously, resolution logic has been employed to resolve signals issued by the software to the uncore hardware. For signals read by the software, a similar approach based on multiplexers can be followed.

For the case of input/output instruction cells reading data from the uncore hardware, it needs to be considered that for different execution paths, a given input/output instruction can be active at different time points and therefore the read operation performed by the instruction can take place at different time points. Figure 6.5 presents one example where it is assumed that instruction cell i_6, from Figure 6.2, reads data from the uncore hardware. As shown in the picture, the instruction cell can read data at two different time points, namely $t + 14$ and $t + 17$. These time points are obtained if the CPU in the example of Figure 6.3 is employed. The fact that i_6 belongs to two different execution paths makes it possible that i_6 can read values from the uncore hardware at two different time points. For this example, an input resolution logic block is instantiated to define the value of the input signal of $port_{i_6}$.

The input resolution block in the example connects only to DATA-TICs at time points that can be actually reached by i_6. Entries for other time points such as $t + 15$ and $t + 16$ are disregarded in the model avoiding additional unnecessary decision logic.

In general if an instruction cell i_l can read from the environment at clock cycles belonging to the set $C_l = \{c_1, c_2, \ldots, c_q\}$, then the logic for the data read by the instruction cell l, denoted as i_l.*data*, can be represented by a chain of multiplexers as follows.

i_l.*data* :=
  if $(i_l.active$ ***and*** $(i_l.cyc = c_1))$ ***then*** $data(c_1)$
  else if $(i_l.active$ ***and*** $(i_l.cyc = c_2))$ ***then*** $data(c_2)$
  $\ldots$
  else if $(i_l.active$ ***and*** $(i_l.cyc = c_q))$ ***then*** $data(c_q)$
  else *free*

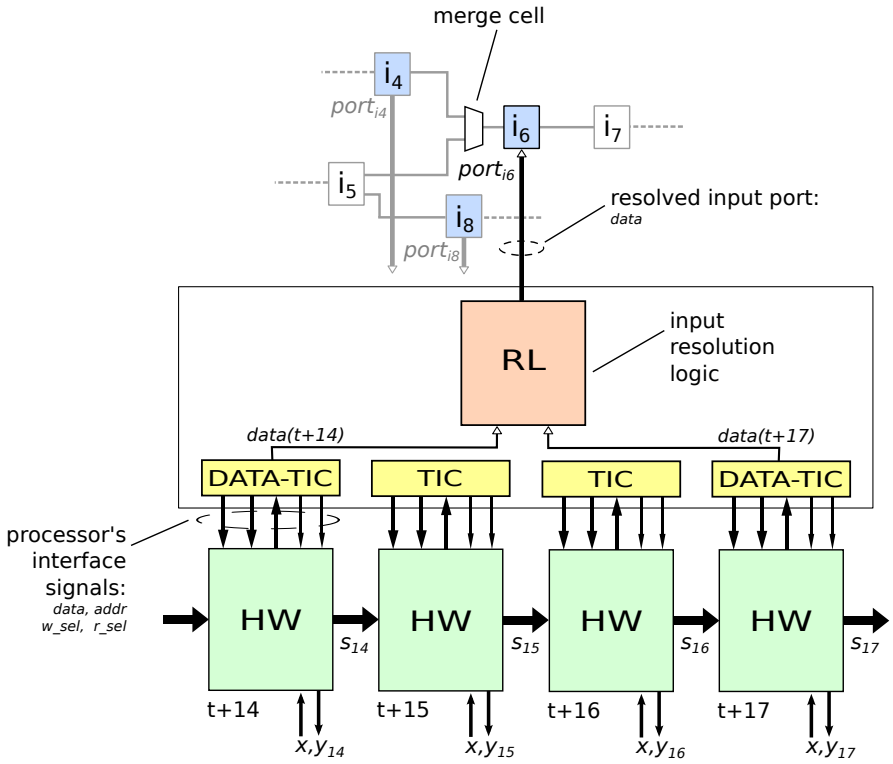

Figure 6.5: Example of unrolled model containing an input resolution block

Where $data(k)$ represents the data read by the instruction cell from the DATA-TIC placed at the time frame k.

By using the information provided by the set C_l the resolution logic is simplified because time frames which can not be reached by a given input/output instruction cell are disregarded. Section 6.2.3 gives the details about how C_l can be obtained.

6.2.3 Timing of Software Input/Output Operations

This section presents a time model that is included into the program netlist in order to represent, in a cycle-accurate manner, the input/output operations that occur at the hardware/software interface. In particular, processor architectures with high time predictability present in firmware based designs are contemplated in the following analysis. Furthermore, this section describes how information about the timing of input/output operations can be statically computed by using the elements of the created time model. As previously discussed in this chapter, important simplifications can be done to the resolution logic if the possible time points at which software input/output operations take place are known.

Time Model of Software Input/Output Operations

Precise cycle-accurate input/output modeling of the software requires to represent correctly the data values read or written by input/output instructions as a function of time. In general, the values read or written by an input/output instruction depend on the time points at which the instruction is executed. These time points depend, in turn, on two factors. First, they depend on the control flow of the program, i.e., the paths executed by the program that can reach the input/output instruction. Second, they depend on the time needed by the processor to execute each of the instructions which precede the input/output instruction. In order to see this more clearly, let us examine instruction cell i_6 from Figure 6.2 and assume again that we have a three stage non-pipelined processor where the input/output access phase takes place at the third clock cycle (cf. Figure 6.3). Depending on the branch taken by the program at instruction cell i_2, it is observed that the input/output access at i_6 can be performed at time point $t + 14$ when the branch is not taken, or, at $t + 17$ when the branch is taken. Besides this, the value of the outputs issued by i_6 vary as a function of the time if the instructions executed on one of the branches, for example i_3 and i_4, change the output values issued by i_6 with respect to the values issued by the same instruction on the other branch.

For tracking the time points (clock cycles) at which input/output instructions of the software access the processor's interface the time variable *cyc* (defined in Section 6.2.2) is incorporated into the program netlist. The time variable *cyc* is appended to the program state (cf. instruction cell on Figure 6.3) and propagated through the program netlist in terms of the control structures employed for describing the active bit. This ensures that the values of *cyc* depend on the path taken by the program, as it happens with the signals belonging to the architectural state of the processor. The value of *cyc* at a particular instruction cell i_l is denoted as $i_l.cyc$ or as $s_l.cyc$ (where s_l denotes the set of signals of the program state at instruction cell i_l). At an input/output instruction cell, $i_l.cyc$ describes the time at which the instruction cell initiates the input/output operations. At instruction cells other than input/output instruction cells, the value of *cyc* has no special meaning, since no input/output operations are performed at those instructions.

The dependency of *cyc* on the execution time of the software instructions can be modeled in different ways. A possible approach is to include an incrementer at every instruction cell in the program netlist. At a given instruction cell the incrementer updates the value of *cyc* by a predetermined amount of delay. Figures 6.6 and 6.7 show the logic for instructions cells with and without branching. The delay added at instruction cells without branching is represented by a constant value *d* as shown on Figure 6.6. For branch instructions, the delay value can depend on the branch that is taken as normally happens in pipelined processors. Therefore, for conditional branches two delay constant values d_t and d_{nt} for the taken and not taken branches respectively are used, as shown in Figure 6.7. Delays for unconditional branches can be modeled in a similar way.

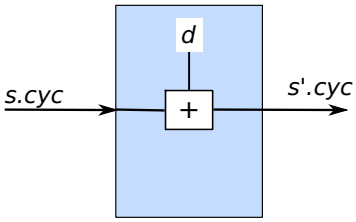

Figure 6.6: Logic for updating the time variable *cyc* in an instruction cell

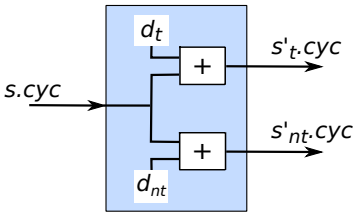

Figure 6.7: Logic for updating the time variable *cyc* in an branching instruction cell

The values of the delays for the instruction cells depend on the particular processor architecture under consideration. Processors used in firmware-based designs have a high time predictability and, therefore, obtaining delay values is, in principle, not a difficult task. In the sequel, the analysis concentrates on non-pipelined and statically-pipelined processor architectures. For the case of processor architectures implementing dynamic pipelining mechanisms techniques based on integer linear programing (cf. [LRM06]) can be employed for computing such delay values.

For non-pipelined multi-cycle architectures, the delay values can be easily derived from the number of clock cycles per instruction as well as from the timing specific to phases where input/output operations take place (e.g., the memory access phase). Since no stalls occur in non-pipelined architectures, the delay values of the instruction cells are independent of the execution scenarios of the instructions. Hence, the delay value set for each instruction cell in the program netlist is commonly the same.

Figure 6.8 shows the resulting time model for the example of a small piece of firmware which runs on the Picoblaze processor of Xilinx [Xil11b]. Execution graph nodes for input/output instructions are marked in blue. The Picoblaze processor implements a non-pipelined 2-phase architecture. All instructions, independently of their kind, require two clock cycles to execute. Hence, the delay values for all instructions cells (including the delays for branch instruction cells) are set to two in the program netlist. Input/output operations take place at the second clock cycle. In the example, the time at which i_6 reads data from the environment ($s_6.cyc$) can be expressed in relation to the value of *cyc* at i_3 as $s_3.cyc + 6$. Note also that, the merge cell at the fanin of i_7 recombines the values of *cyc* for each of the two possible execution paths. Based on this, the input/output access at i_8 ($s_8.cyc$) can happen

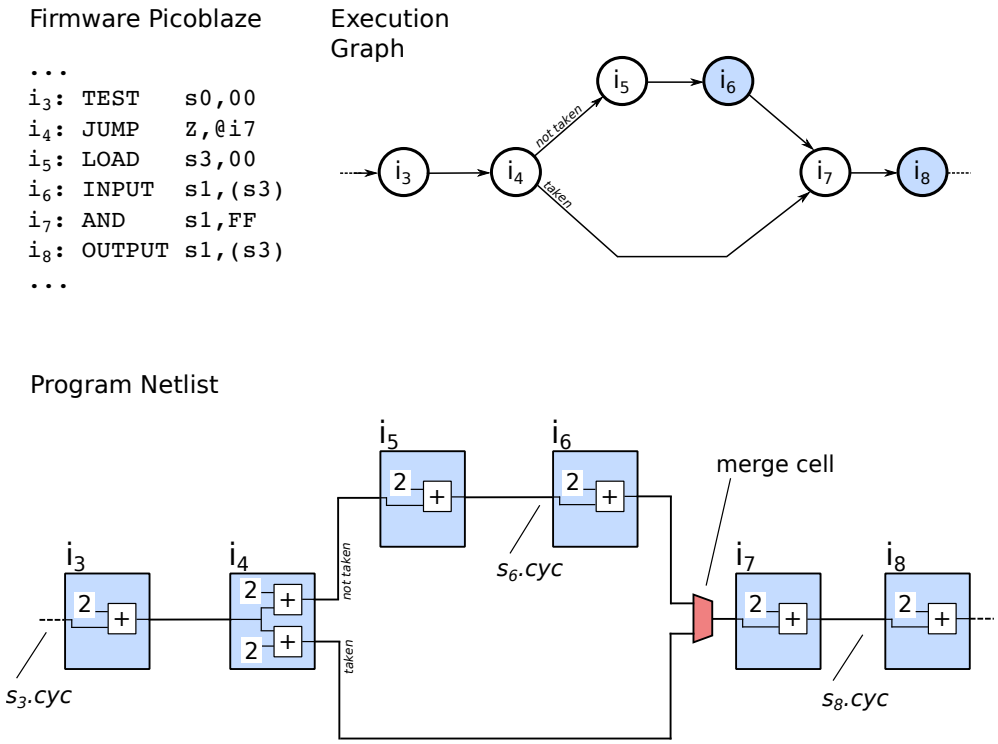

Figure 6.8: Obtaining delays for PicoBlaze instructions

either at $s_3.cyc + 10$ for the not taken branch or at $s_3.cyc + 6$ for the taken branch.

For pipelined architectures, the delay added by an instruction cell depends on the possible execution scenarios since stalls may occur. Therefore, a description of the possible stall scenarios is needed when dealing with pipelined processors. It should be noted that statically pipelined processors commonly get rid of most of the hazards by using forwarding units and other related techniques that keep the total number of stall scenarios low. This reduces the complexity of obtaining the delays introduced by instruction cells since fewer stall scenarios need to be considered in the analysis.

With the information about stall scenarios, the computation of the delay for each instruction cell in the program netlist can be implemented by the algorithm of Figure 6.9. The algorithm takes the execution graph (V and E refer to the sets of nodes and edges of the EXG, respectively) and traverses it node by node. When a new execution graph node is processed then the execution scenarios for the related instruction are examined. The processed instruction is analyzed together with a determined number of successors. This is done by the function *ANALYZE_DELAYS*. The number of successor nodes analyzed depends on the depth of the pipeline. If stalls are detected, then the delay values are incremented correspondingly. Note that if no stalls occur and assuming that the minimum cycles per instruction is one, then the delay value is one. This algorithm can be understood as the process of moving a window through the execution graph instruction by instruction (the size

of the window corresponds to the depth of the pipeline). If there are branches, the window moves through each of the instructions on the branch so that all execution scenarios are covered (cf. line 13 in Figure 6.9).

```
1: COMP_DELAYS(V, E) {
2:     list ← {}
3:     for each i_v ∈ V {
4:         FIND_PATHS(V, E, i_v, list);
5:     }
6: }

8: FIND_PATHS(V, E, i_v, list) {
9:     list_1 ← list;
10:    INSERT(list_1, i_v);
11:    if SIZEOF(list_1) = depth {
12:        ANALYZE_DELAYS(list_1);
13:    }
14:    else {
15:        for each i_s ∈ Succ[i_v] {
16:            FIND_PATHS(V, E, i_s, list_1);
17:        }
18:    }
19: }
```

Figure 6.9: Algorithm for computing execution delays of I/O instruction cells

Figure 6.10 shows the resulting time model including the delays for a small piece of firmware. In this case, the code runs on the SuperH-2 processor of Renesas [Ren05]. As can be seen, a stall occurs as a result of the data dependency between i_6 and i_7. The delay value at i_6 is incremented by one time unit to reflect the extra delay caused by this stall. Note that the branch instruction at i_4 has two different delay values for each of the branches. The taken branch has a delay value of three, as two instructions in the pipeline are discarded by the processor for this execution scenario.

Static Computation of Timing Information for Software Input/Output Operations

Computing information about the possible set of clock cycles at which input/output instructions communicate with the environment is important for reducing the amount of resolution logic built in to the timed interface model (cf. Section 6.2.2). By using the time model presented above, timing information can be easily derived. Note that the set of clock cycles

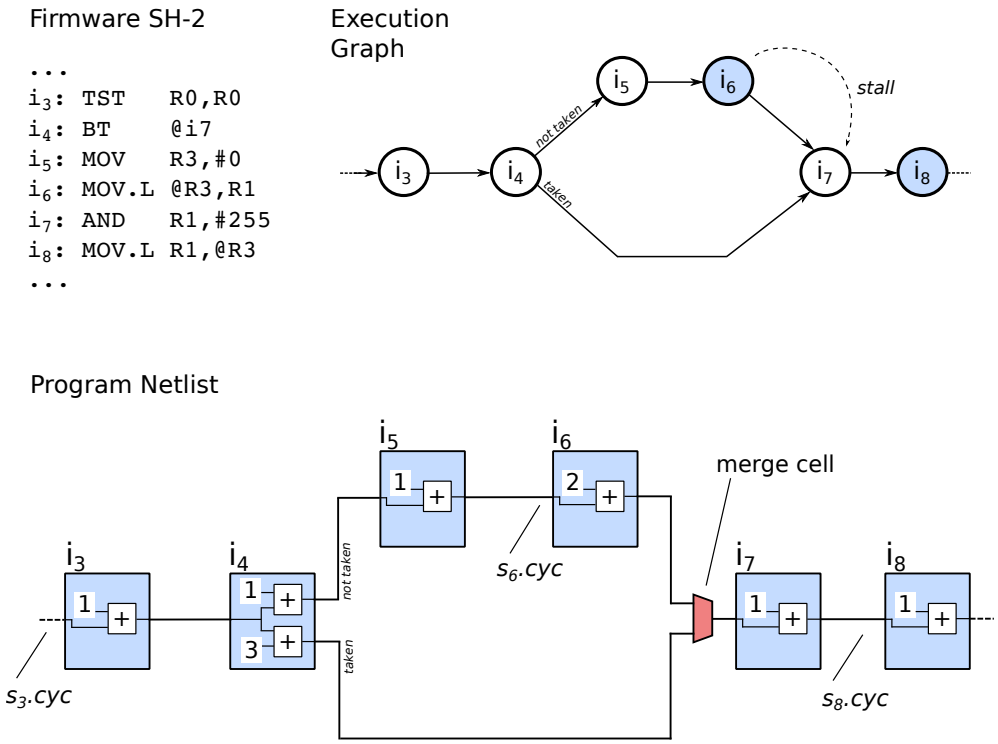

Figure 6.10: Obtaining delays for SuperH-2 instructions

at which an input/output instruction cell i_l communicates with the environment corresponds to the set of values the variable $i_l.cyc$ can take. This is the set $C_l = \{c_1, c_2, \ldots, c_q\}$ which has been defined in Section 6.2.2.

In order to compute C_l for each input/output instruction cell i_l an algorithm that propagates the execution times for each individual instruction cell through the execution graph is employed. Figure 6.11 presents the pseudo-code of this algorithm. The algorithm takes as inputs the execution graph (V and E refer to the sets of nodes and edges of the EXG, respectively) and the time delays associated to every instruction cell. In particular, for each direct successor u of a node q, $delay_{(q,u)}$ represents the delay added by node q when the program takes the path to u.

The algorithm traverses the nodes of the execution graph which is topologically sorted. Associated with every execution graph node q there is a set of possible execution times C_q containing the values the variable $i_q.cyc$ can take. When visiting a new execution graph node q, the set C_q is computed by joining the execution times $Cprop_{(p,q)}$ of each direct predecessor p of q (see line 11 in Figure 6.11), where $Cprop_{(q,u)}$ contains the set of possible execution times that result when the program takes the path from q to u. Since the execution graph is topologically sorted, the set of execution times for all predecessors of a given node q are known when that node is visited.

In theory the sizes of the sets computed by the algorithm can grow exponentially with

```
1: COMP_CYCLES(V, E, S) {
2:     initialize start nodes of EXG
3:     for each i_start ∈ S {
4:         C_{i_start} ← {c_0};
5:     }
6:     traverse the nodes of the EXG
7:     for each i_v ∈ V {
8:         if i_v ∉ S {
9:             C_{i_v} ← ∅;
10:            for each i_p ∈ Pred[i_v] {
11:                C_{i_v} ← C_{i_v} ∪ Cprop_{(i_p,i_v)};
12:            }
13:        }
14:        compute clock cycles for each successor
15:        for each i_s ∈ Succ[i_v] {
16:            Cprop_{(i_v,i_s)} ← ∅;
17:            for each c ∈ C_{i_v} {
18:                Cprop_{(i_v,i_s)} ← Cprop_{(i_v,i_s)} ∪ {c + delay_{(i_v,i_s)}};
19:            }
20:        }
21:    }
22: }
```

Figure 6.11: Algorithm for computing timing information of I/O Operations

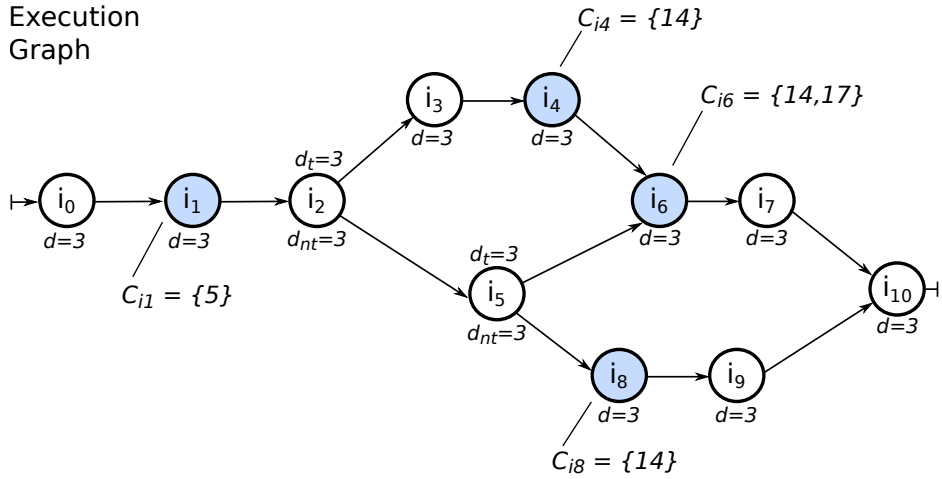

Figure 6.12: Computing timing information of input/output instruction cells

the number of execution paths since a given input/output access can be performed at a different time for each path leading to the corresponding input/output instruction. However, for practical firmware the number of possible execution times of an input/output accesses are often limited by design constraints (e.g. WCET constraints) ensuring that input/output accesses are completed in a specified time window.

It is pointed out that this algorithm operates independently of the type of processor used. Specific information about the timing of the processor is taken into account by the delays resulting from the time model above presented.

Consider again the example of Figure 6.2 and assume a three-phase multi-cycle architecture (cf. Figure 6.3). Figure 6.12 shows the execution graph with the delays for each instruction cell. The algorithm begins with computing $C_{i_0} = \{t\}$ (where t is the initial time reference), then $C_{i_1} = \{t + 3\}$ and so on. Once i_6 is processed, it is already known that $C_{i_4} = \{t + 12\}$, $C_{i_5} = \{t + 9\}$ and therefore $C_{i_6} = \{t + 12, t + 15\}$ is computed. The algorithm can then further continue to compute $C_{i_7} = \{t + 15, t + 18\}$. The traversal ends as soon as the possible execution times are calculated for all execution graph nodes.

Since the memory access phase takes place at the third clock cycle (i.e., it has a delay of two time units) the initial time reference is set to two. Consequently the final access times for the input/output instruction cells are $C_{i_1} = \{5\}$, $C_{i_4} = \{14\}$, $C_{i_6} = \{14, 17\}$ and $C_{i_8} = \{14\}$.

From the computed information, the data required to reduce the resolution logic can be directly derived. For example, it can be seen that during $t + 12$ and $t + 14$ the input/output instruction cells i_4, i_6, i_8 can be executed. More specifically, since in the last phase the data is written to the hardware it can be determined that the set of instruction cells that can write at the clock cycle number 14 is $W_{14} = \{i_4, i_6, i_8\}$, as shown in Figure 6.4. All other time points of the unrolling can be processed in the same way.

6.3 Tool

The verification platform *Formal Checker Kaiserslautern* (FCK) was extended with the algorithms and modeling elements presented in this section. Figure 6.13 presents the block diagram of the extended platform.

For program netlist generation, FCK takes as input the machine code of the firmware and the instruction cells modeling the core hardware. Section 4.4 presents the details of the part of the platform employed for program netlist generation. For creating the unrolled hardware/software co-verification model, the resulting program netlist together with the corresponding execution graph are taken as inputs. This is shown on the top of Figure 6.13.

In order to generate the timed interface model (see *Timed Interface Builder* block in the figure), FCK requires an RTL description of the TICs of the processor together with the specific timing information of the CPU necessary for predicting instruction execution times. In the current implementation this information is provided directly as a data structure. TICs are provided in ICL for each supported CPU architecture. Both pieces of information, TICs and CPU timing specification data, should be provided by the user (see the gray box on the block diagram of FCK).

With all these inputs FCK fully automatically generates the hardware/software model for performing formal verification. The block called *Timed PN Builder* constructs the cycle-accurate time model as explained in Section 6.2.3. Static computation of execution times for input/output instructions is performed by the block *Timing Information Computation*.

The elements in the green box show the flow for unrolling the uncore hardware. An RTL description of the uncore hardware is read in this part of the tool. From this description, the transition logic is extracted and unrolled as done in most BMC-based hardware verification tools. In the current implementation, the property checker of Onespin [One] is employed for this purpose.

The tool can export the combined hardware/software model, including the program netlist with its timed model incorporated and the timed interface model, as VHDL RTL code so that it can be used as input in a standard property checker.

6.4 Experimental Results

The following experimental evaluation considers an application domain where high predictability of the processor's timing behavior allows for a firmware-based design style, in which the processor is directly integrated into the hardware without the use of a standard bus interface. In the experiments the commercial property checker OneSpin 360 DV [One] was used. All experiments were conducted on an Intel Xenon running at 2.83 GHz with 32 GB of main memory.

Two different case studies were conducted employing the PicoBlaze processor from

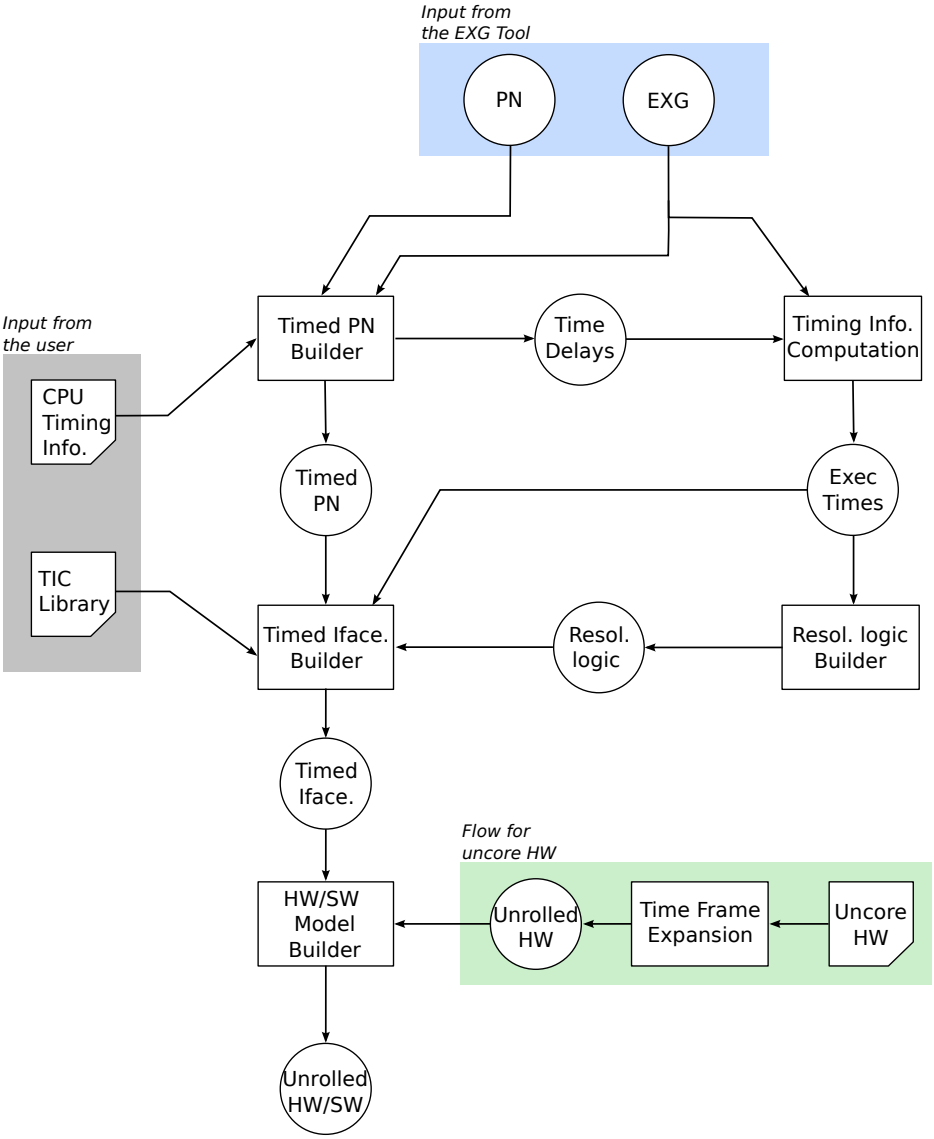

Figure 6.13: Tool for generating the HW/SW co-verification model

Design	LoC SW	LoC HW	PN size	DATA-TICs	HW inputs	state vars.
FPI slave	172	2908	380	188	113	1149
Control unit	253	2734	474	171	18	1148

LoC HW: lines of HDL code, LoC SW: lines of assembly code, PN size: number of instruction cells

Table 6.1: Characteristics of the designs and models

Xilinx. Properties written in the experiments specify global behavior of the designs, e.g., complete transactions. OneSpin's property language ITL was used to describe the properties.

In all experiments, the approach presented in this section was compared with the classical hardware Bounded Model Checking technique of Figure 6.1. The author is not aware (at the time of submitting this thesis) of any other tools or methods reported in the literature that could be applied in this context.

The first case study is a firmware-based implementation of a *slave interface* for the *Flexible Peripheral Interconnect* (FPI) bus protocol [Inf]. The FPI bus is a pipelined SoC bus developed by Infineon. The slave serves to connect a peripheral device with the FPI bus. In this design, the main control of the slave agent is implemented as firmware while the data path functionality is implemented in hardware. The slave agent interacts synchronously with the bus and asynchronously with the peripheral. In the system, the surrounding hardware (wrapper RTL in Figure 1.1) captures the signals from the FPI bus when a request occurs. Subsequently, the firmware checks from the wrapper RTL for new incoming requests and informs the peripheral about it. Once the answer from the peripheral device arrives, the firmware finishes the transaction by setting the correct values on the wrapper RTL. The outputs of the wrapper RTL drive the signals of the FPI bus when data and control values are read from the slave agent.

Table 6.1 summarizes the characteristics of the design and the models produced by the FCK tool. Proofs were conducted for two different safety properties *slave_read* and *slave_write*, specifying a read and a write transaction, respectively, after a system reset sequence. Both properties describe control and data values as specified in the FPI bus documentation. Table 6.2 presents the results obtained for proving both properties.

The second case study is a control unit resembling typical non-mainline functionality implemented with the use of firmware. The control unit interacts on one side with a master SoC module (e.g., a processor or some other hardware module) which sends commands to specific hardware devices through the control unit. Addresses of the destinations (hardware units) are also sent to the control unit by the same master SoC module. The firmware of the control unit receives command and destination information, analyzes its validity (checking

Property	length (cycles)	PN-based time (min.)	PN-based mem. (GB)	BMC-style time (min.)	BMC-style mem. (GB)
slave_write	461	01m21s	1.93	45m02s	19.11
slave_read	460	01m50s	1.82	43m34s	20.78
trans_ok	729	04m02s	1.62	TO[1]	23.40[2]
trans_valid	700	03m04s	1.46	TO[1]	21.05[2]

[1]*time-out = 24h,* [2]*memory usage at time-out*

Table 6.2: Property checking results

parity) and then synchronously sends the command to the corresponding hardware devices. Each of the hardware devices runs an independent finite state machine recognizing whether commands are actually intended for it or not. In the case of a match, the command is latched by the hardware unit and a control signal is activated to indicate the arrival of the valid command. At the end, the firmware of the control unit informs the master SoC module whether the transaction has been completed successfully or not.

For improving performance, two pipeline stages are inserted to the read/write transactions initiated by the CPU. This pipelining is implemented in the RTL wrapper and splits the address decoding in two phases. In the first stage, the address issued by the CPU is latched and decoded. In the second stage, the address is validated against the *enable* signal of the CPU.

Table 6.1 summarizes the characteristics of the design and the resulting models. Two safety properties *trans_ok* and *trans_valid* were proved. The first property specifies that (after a reset sequence was applied) if the data obtained from the master SoC module is valid then at the end of the transaction the corresponding commands and control signals are activated in the correct hardware unit. The second property specifies that the correct finish condition is sent to the master SoC module depending on the validity of the command and destination values. Table 6.2 contains the results obtained for proving both properties.

All properties were finally proven correct after a number of system bugs had been identified by the method and were corrected. For instance, in the FPI slave interface there was a situation in which after the reset, the slave could be selected before the system was properly initialized by the firmware and therefore wrong values of the FPI control signals were issued by the interface. Also, the firmware of the control unit contained a bug due to a wrong sequence of reading operations. For a certain execution scenario of the firmware, it was possible that the acknowledge was read after the command value. This could trigger invalid read commands by the control unit.

As can been observed from the experiments, the proposed technique outperforms a straightforward BMC approach in all cases. The properties for the control unit turn out to be more difficult for the property checker (i.e., for the SAT engine) than the properties for the FPI bus. An explanation for this is that the control signals depend on the parity computation performed by the firmware which increases the computational challenges for the solver.

In both case studies, the firmware was required to react as fast as possible so as to cause only a minimum number of wait states. Therefore, the run times for complete transactions, in both cases, are short, resulting in program netlists of small size. Their generation required less than one minute in all cases. Taking into account that verification based on much larger program netlists can be performed, as shown in Section 5.5 of this thesis, there is promise that also significantly larger designs integrating firmware as described in Section 6.1 can be handled.

Chapter 7

Summary and Future Work

7.1 Summary

This thesis described hardware-dependent models of firmware for formal verification. The proposed models are conceived to represent the firmware integrated with its hardware environment according to the current SoC design practices. Two hardware/software integration scenarios have been addressed, namely, speed-independent communication of the processor with its hardware periphery and cycle-accurate integration of firmware into an SoC module.

The proposed models are based on the program netlist (Chapter 3). A program netlist is a combinational model that represents the software behavior in terms of its underlying hardware platform. For a given software, the program netlist compactly represents the behavior along all possible execution paths and models the program computation by instantiating instruction cells of the corresponding processor architecture. An instruction cell describes the behavior of the hardware for executing a given ISA instruction. Information about the execution paths reachable by the software is collected by performing an automated analysis of the control flow. The resulting information is then encoded into the control logic of the program netlist so that it can be directly exploited by verification engines. Modeling of the data traffic between the processor and memory is a another component in the program netlist. For improving scalability of the memory model, only sets of reachable data memory locations accessed by load/store instructions are regarded in the logic representing the memory behavior.

For program netlist generation, information on addresses, such as jump/branch destinations and data memory accesses, needs to be computed many times. In Chapter 4, a method for quickly finding this information is proposed. The method exploits several characteristics of firmware that significantly reduce the state space when performing model generation. For instance, firmware commonly neither makes calls to unbounded recursions nor does it dynamically allocate memory. Moreover, the possible address ranges of the system are limited and predefined by design. Also, it is a common practice in firmware to use registers also for storing constant addresses, for example when jumping to a subroutine or when

accessing a register of the hardware periphery. As a result, most of the addresses that need to be computed correspond to constant values. Instruction set simulation techniques are employed in order to discover quickly these constants. For hard-to-determine addresses where simulation fails, the method resorts to enumerative SAT-based techniques.

The proposed methods in Chapters 5 and 6 benefit from information provided by program netlists to model the hardware/software interface for different time granularities. In particular, information about the data memory address spaces accessed by the software is used to identify possible hardware/software interactions. Additionally, explicit control flow representation is employed for determining temporal information about those hardware/software interactions.

In Chapter 5, for speed-independent communication schemes, a time-abstract model of the hardware/software interface has been proposed for checking equivalence of reactive hardware-dependent programs. For this purpose, a model of the input/output sequences of accesses to data environment locations is incorporated into the program netlist. The logic for representing the sequences is constructed in terms of the control logic of the program netlist. This enables efficient program path exploration while performing proofs. Based on the input/output model, a software miter is created in order to perform the equivalence check. In particular, two programs are proven to be equivalent, if for the same sequence of inputs both programs produce the same sequence of outputs. Incremental SAT solving is employed in order to reduce verification run time.

For cycle-accurate integration (cf. Chapter 6), a time interface model representing the input/output behavior of the processor is constructed and integrated into the program netlist. This allows to integrate the firmware together with its hardware environment in a single model. The obtained model preserves the precision of standard techniques based on finite unrollings of concrete hardware descriptions while also modeling explicitly the possible instruction sequences that can actually occur in real firmware runs. The resulting computational model has been used for performing hardware/software co-verification by property checking.

The experimental evaluation demonstrates that with the approaches presented in this thesis, it is feasible to perform relevant formal verification tasks for hardware/software systems as they are designed in industry. As shown by the experiments, low-level software of realistic complexity can be handled by the proposed approaches. In most cases verification of the designs was completed in few minutes. This is achieved mainly by the three following factors. Firstly, significant simplifications are performed to the computational models before the actual verification tasks take place. These simplifications avoid that the employed decision procedures waste time in exploring state spaces which are not relevant to the performed proofs. Secondly, computational models allow verification engines to explore efficiently the execution paths of the firmware. This is achieved by an explicit encoding of the control flow of the software in the proposed computational models. More specifically, models of the hardware/software interface are built in terms of these control structures so that program paths can be efficiently explored for verification problems where not only the firmware but

also the hardware environment is taken into account. Thirdly, abstraction performed for modeling the processor hardware reduces the verification effort. Models of the processor (including CPU, memory subsystem, and input/output interfaces) consider uniquely logic that is relevant to represent precisely the hardware/software behavior. Details of the processor internals, such as intermediate pipeline logic, are abstracted away in the employed models reducing the complexity when making proofs.

Furthermore, for the experiments of Chapters 4 and 5, an off-the-shelf SAT solver has been employed. No special targeted heuristics or pre-processing techniques have been built into the solver. Similarly, in Chapter 6, a general Interval Property Checker has been used, instead of a specialized back end. These facts leave room for additional optimizations to the proposed approaches.

These results encourage further investigations in the field of formal verification and also in other application domains such as testing and formal safety analysis of hardware/software systems.

7.2 Future Work

7.2.1 Perspectives in Formal Verification

The approach of Chapter 6 targets verification of designs with high timing predictability. This approach can be adapted to perform also hardware/software co-verification of designs where the exact timing of the software is not relevant or too difficult to predict. In such scenario, no cycle-accurate model of the hardware/software interface needs to be created. Instead, a model mixing different time resolutions could be constructed.

The envisioned approach would proceed in a compositional way as follows. For time intervals where the hardware environment idles waiting for commands to be issued by the firmware, no hardware unrolling takes place. In those intervals, the behavior of the complete system would be modeled by a program netlist describing the firmware operations performed in oder to start a new hardware transaction. Those software operations are atomic, i.e., they are abstracted in time. Correspondingly, for intervals where the firmware waits for the hardware environment to react, the system's behavior can be represented uniquely in terms of the hardware environment. In this situation, the hardware is unrolled in a cycle-accurate way following a Bounded Model Checking strategy. As a result, a composed hardware/software model could be obtained that represents complete system transactions. Complementary proofs establishing the correctness of the "idle" operations would ensure the soundness of the final model. This approach works under assumption that the interface between processor and hardware environment is correct. This allows model generation to abstract away such an interface by interconnecting directly the processor with its hardware environment.

The program netlist can be seen as a formal specification of the concrete hardware when it

executes a certain piece of software. In this context, methods could be developed that formally check whether or not a given program runs correctly on the given processor and its hardware platform. This would allow a *software-driven* approach to hardware design verification. A possible technique to implement this idea would be to use equivalence checking for examining if the software execution on the platform complies with the generated specification (i.e., the program netlist). Particularly, comparisons of the program netlist against unrolled versions of the hardware platform could be performed for finding discrepancies between the models. These comparisons would be enabled by extensions based on the methods of Chapter 6 for representing the memory subsystem with the required timing accuracy. If a discrepancy between the models is found, it means that a design error in the hardware is detected. If no bug is found, it means that the hardware is correct with respect to the specific program. The advantage of the approach is that it is fully automatic. No properties need to be written. Instead, test programs can be written reflecting the software that will actually run on the system.

7.2.2 Perspectives in Test and Safety

The models presented in this thesis are created by computational processes that need to be conducted only once for a given design. After the computation has been completed, a formal analysis by SAT-based techniques can be conducted efficiently. Such analysis may go beyond a merely functional verification. In the following, it is outlined how the proposed models can also be used in test and when designing fault-resilient systems.

Program netlists can be extended to model the behavior of a program in the event of hardware faults. Based on this, a formal analysis employing the equivalence checker of Chapter 5 can be accomplished in order to evaluate the effects of the faults on the global software behavior. Comparisons of the model fault-free system against the model containing different injected faults would be possible. The results of the performed analysis can be used to develop *application-dependent* strategies for test and error resilience of hardware/software systems which target only those errors that can actually modify the correct software behavior. As a result, the overhead associated to error detection and resilience mechanisms can be (in principle) reduced, since logic for test is uniquely built for relevant scenarios. First experimental results presented in [BRV+16] demonstrate the feasibility of this approach.

A program netlist includes explicit information for a given program on: all possible execution paths, all possible input/output access sequences to hardware peripherals and to shared memory, the address spaces reached by every instruction, and all possible effects of the program on the program-visible hardware registers. This could form the basis of a hardware/software cross-layer approach for assessing the effect of hardware faults at the system level. New techniques can be envisioned for providing formal guarantees that certain hardware faults will always be detected by specific system-level mechanisms when executing

the system's software. Design of resilience mechanisms at the software level may benefit from the information whether the targeted hardware faults are completely unaffected by some system-level resilience mechanisms, possibly masked in some testing scenarios and/or always safely covered during functional operation.

Chapter 8

Deutsche Zusammenfassung

Im aktuellen Ablauf des Entwurfs von System-on-Chips (SoC) ist ein Trend zur immer stärker werdenden Integration von low-level Software Komponenten in der Systemhardware feststellbar. Die Implementierung der wichtigen Kontrollroutinen und Kommunikationsstrukturen wird häufig in der Firmware anstatt der SoC Hardware realisiert. Dies hat zur Folge, dass diese enge Bindung von Hard- und Software auf dieser niederen Ebene den Verifikationsaufwand erheblich vergrößert, da der bisherige getrennte Verifikationsansatz von Hard- und Software nicht mehr ausreichend ist. Daher werden neue Methodiken für eine gemeinsame Verifikation von Hard- und Software benötigt.

Diese Arbeit stellt hardwareabhängige Modelle für low-level Software zur formalen Verifikation vor. Die vorgeschlagenen Modelle wurden entwickelt um Software, die in eine Hardwareumgebung integriert ist, bezüglich der gängigen SoC Entwurfsverfahren zu repräsentieren. In dieser Arbeit werden zwei Hardware/Software Integrationsszenarien adressiert, nämlich die geschwindigkeitsunabhängige Kommunikation des Prozessors mit seiner Peripherie und die zyklengenaue Integration der Firmware in ein SoC Modul. Für geschwindigkeitsunabängige Hardware/Software Integration wird ein Verfahren zum Äquivalenzvergleich hardwareabhängiger Software vorgestellt und evaluiert. Für den Fall der zyklengenauen Hardware/Software Integration wurde ein Modell zur Hardware/Software Co-Simulation entwickelt und experimentell evaluiert, indem es für Eigenschaftstests genutzt wurde.

8.1 Programmnetzliste

Während die meisten Techniken zur Softwareverifikation auf einer hardwareunabhängigen Ebene arbeiten, beschreibt diese Arbeit ein hardwareabhängiges Softwaremodell, genannt Programmnetzliste. Eine Programmnetzliste modelliert formal das Verhalten eines Prozessors bezüglich eines bestimmten Softwareprogramms. Diese repräsentiert als kombinatorische Logik alle möglichen Programmausführungen unter allen möglichen Eingaben mit den entsprechenden Ausgaben. Da dieses Modell kombinatorisch ist, kann Boole'sche Beweis-

führung mittels Erfüllbarkeitstest genutzt werden, um beliebiges Programmverhalten zu verifizieren.

Der Vorgang zur Erzeugung einer Programmnetzliste ist vollautomatisch und besteht aus zwei Schritten. Der erste Schritt ist das Abrollen eines Kontrollflussgraphen, der das Programm repräsentiert. Der Kontrollflussgraph kann durch Extraktion aus Maschinen- oder Assemblercode erhalten werden. Jeder Knoten im resultierenden Graphen repräsentiert eine einzelne Prozessorinstruktion. Der abgerollte Kontrollflussgraph wird Ausführungsgraph genannt und ist der Ausgangspunkt für den zweiten Schritt. In diesem zweiten Schritt wird jeder Knoten des Ausführungsgraphen durch einen entsprechenden Logikblock ersetzt, der das Verhalten des Prozessors für die Instruktion modelliert. Ein solcher Logikblock wird Instruktionszelle genannt.

Eine Instruktionszelle besitzt einen Eingang und einen Ausgang, der mit der vorangehenden bzw. nachfolgenden Zelle verbunden ist. Der Eingang einer Instruktionszelle repräsentiert den aktuellen Programmzustand. Dieser umfasst die Werte der Programmvariablen im Speicher und den Inhalt der CPU Register, die vom Programm angesprochen werden können, bevor die entsprechende Instruktion ausgeführt wird. Der Ausgang entspricht dem nächsten Programmzustand, d.h. dem Zustand nachdem die Instruktion ausgeführt wurde. Die Kommunikation mit dem Speicher sowie alle anderen Eingangs- und Ausgangsoperationen werden durch spezielle Schnittstellen in den Instruktionszellen modelliert.

Die Größe des erhaltenden Ausführungsgraphen wird durch zwei wichtige Optimierungen reduziert. Erstens werden Ausführungspfade, welche für das Programm nicht erreichbar sind, vom Graphen ausgeschlossen. Dies wird erreicht indem der Abrollprozess mit einer Kombination aus Simulation und Erreichbarkeitsanalyse, welche tote Zweige (Zweige, die nicht unter der gegebenen Eingangsbelegung erreicht werden können) identifiziert, für Kontrollinstruktionen verschachtelt wird. Zweitens wird eine Reduzierung der Größe des Graphen erreicht, indem Pfade wenn möglich durch Rekombination verschmolzen werden. Auf diese Weise wird anstatt eines Aussführungsbaumes ein gerichteter, azyklischer Graph (DAG) erzeugt.

Ein entscheidendes Element der Programmnetzliste is die Art der Modellierung des Kontrollflusses. Ein einbittiges Signal, genannt Aktivsignal, wird dem Programmzustand hinzugefügt. Für jede Instruktionszelle ist das Signal wahr, wenn die entsprechende Instruktionszelle zum ausgeführten Pfad gehört. Ansonsten ist das Aktivsignal falsch. Das Aktivsignal wird nur an Entscheidungsstellen, wie zum Beispiel Verzweigungsinstruktionen, aktualisiert. Für diese Instruktionen ist eine Kontrolllogik in der entsprechenden Instruktionszelle eingefügt, die sicherstellt dass abhängig von der Verzweigungsentscheidung nur ein ausgeheneder Zweig aktiviert werden kann. In anderen Instruktionszellen wird das Aktivsignal lediglich ohne Veränderung durchgereicht. Es wird dazu keine Kontrolllogik benötigt. Indem einem Aktivsignal ein Wert in der Programmnetzliste zugeordnet wird, können ganze Pfadsegmente aktiviert und deaktiviert werden. Im Gegensatz zu Methoden basierend auf symbolischer

Ausführung, welche Beweise durch explizite Traversierung der Ausführungspfade eines gegebenen Programms führen, wird für Programmnetzlisten ein Erfüllbarkeitsentscheider für die Pfadtraversierung genutzt. Ein Erfüllbarkeitsentscheider profitiert vom Kontrollfluss in der Programmnetzliste, da dieser sich nur auf die Ausführungspfade konzentriert, die für die gegebene Probleminstanz entscheidend sind und die irrelevanten Pfade direkt ignoriert. Dies wird erreicht nur durch Wertezuordnung zu den Aktivsignalen. Die Effektivität dieses Ansatzes wurde an Hand von durchgeführten Experimenten in dieser Arbeit belegt.

Die Modellierung von Datentransfer zwischen Prozessor und Speicher ist ein weiteres wichtiges Element in der Programmnetzliste. Um die Skalierbarkeit des Speichermodells zu verbessern wird das Speicherverhalten nur für erreichbare Speicherstellen von Daten, auf die Lade- und Speicherinstruktionen zugreifen, durch Logik modelliert. Dies wird ermöglicht, indem der Satz von Speicheradressen, welche von den Lade- und Speicherinstruktionen in der Programmnetzliste adressiert werden, mittels einer Kombination aus Simulation und Erreichbarkeitsanalyse berechnet.

Neben der Berechnung von Informationen von erreichbaren Adressen von Datenspeicherzugriffen, müssen auch Ziele der Verzweigungen von der kombinierten Simulations- und Erfüllbarkeitsanalyse berechnet werden. Der Kontrollflussgraph, der als Ausgangspunkt für die Modellerzeugung genutzt wird, kann unvollständig sein. (Beispielsweise können Verzweigungsziele unbekannt sein auf Grund von indirekter Adressierung.) Dies ist oft der Fall für Kontrollflussgraphen, die aus einer realen Maschine oder Assemblercode extrahiert wurden. Diese gemischte Analyse nutzt verschiedene Eigenschaften von low-level Software, welche den Zustandsraum während der Modellgenerierung stark reduziert. Zum Beispiel ruft eine Software normalerweise weder unbegrenzte Rekursionen auf noch reserviert diese dynamisch Speicher. Zudem sind die Addressräume des Systems begrenzt und vordefiniert beim Entwurf. Es ist ebenfalls üblich, dass Register zur Speicherung von konstanten Addressen genutzt werden, z.B. beim Sprung in eine Unterroutine oder beim Zugriff auf ein Register einer peripheren Hardware. Infolgedessen entsprechen die meisten zu berechnenden Addressen konstanten Werten. Simulationstechniken für Instruktionssätze werden zur schnellen Ermittlung dieser Konstanten genutzt. Für schwer zu berechnende Adressen, bei denen die Simulation fehlschlägt, wird auf enumerative erfüllbarkeitsbasierte Techniken zurückgegriffen.

Die resultierende Programmnetzliste enthält alle benötigten Informationen um die low-level Softwareverifikation in eine Hardwareverifikationsumgebung zu integrieren. Zusätzlich kann eine Programmnetzliste als Block instanziiert und mit weiteren Blöcken zusammengesetzt werden, um verschiedene Arten von Verifikationsproblemen zu lösen. Der Rest der Zusammenfassung erklärt, wie diese Eigenschaften in dieser Arbeit ausgenutzt werden.

8.2 Äquivalenzvergleich von hardwareabhängiger Software

Diese Arbeit schlägt eine vollautomatische Methode für formale Beweise der funktionalen Äquivalenz von hardwareabhängigen Programmen, welche in reaktiven Umgebungen eingebettet sind, vor. Äquivalenzvergleich ist ein wertvolles Werkzeug, da es zur Überprüfung genutzt werden kann, dass die ursprüngliche Funktionalität weder beschädigt noch verändert wurde nach: Transformationen zur Codeoptimierung, Hinzufügen neuer Funktionalität zum Code oder Portierung des Codes zu neuen Hardwareplattformen. Das Äquivalenzkriterium für die vorgeschlagene Methode ist, dass zwei Programme äquivalent sind, wenn für jede Eingangssequenz, die von beiden Programmen gelesen wird, die produzierte Ausgabesequenz der Programme gleich ist. Eingabesequenzen (Ausgabesequenzen) eines Programms enthalten die Werte, welche von (zu) der Umgebung gelesen (geschrieben) werden und die entsprechende Reihenfolge. Entsprechend dieses Äquivalenzbegriffes wird nicht nur sichergestellt, dass die von dem Programms mit der Umgebung ausgetauschten Datenwerte gleich sind, sondern dass die Reihenfolge des Datenaustauschs ebenfalls die gleiche ist.

Um reaktives Verhalten berücksichtigen zu können, wird die Programmnetzliste erweitert um ein globales Modell des seqeunzbasierten Ein- und Ausgabeverhaltens. Dieses globale Modell beschreibt zeitabstrakt die Transaktionen, welche von der Software an den Schnittstellen, dass heißt den Hardware/Software Schnittstellen, ausgeführt werden. Das Zeitmodell wird nicht an Hand von Taktzyklen sondern von abstrakten Zeitpunkten beschrieben. Die folgenden, allgemeinen Schritte werden für den Äquivalenzvergleich zweier Maschinenprogramme G (für “golden") und R (für “revised"), welche auch auf verschiedenen Hardwareplattformen ausgeführt werden dürfen, ausgeführt:

- Die Programmnetzlisten für G und R werden unabhängig voneinander aus den entsprechenden Maschinenprogrammen und den dazugehörigen Instruktionszellendes Prozessors, auf dem das Programm ausgeführt wird, erzeugt.
- Jede generierte Programmnetzliste wird durch ein sequenzbasiertes Modell erweitert, welches das Verhalten des Maschinenprogramms and seinen Schnittstellen bschreibt.
- Ein Software-Miter wird durch Instanziierung der Programmnetzliste, des sequenzbasierten Ein- und Ausgabemodells und einer bijektiven Zuordnung der Ein- und Ausgabeumgebung von G und R, welche vom Benutzer bereitgestellt wird, erstellt. Zur Erzeugung des Software-Miters wird in dieser Arbeit ausgenutzt, dass Programmnetzlisten kompositional sind.
- Zum Abschluss wird eine Entscheidungsprozedur, vornehmlich ein Erfüllbarkeitsentscheider, iterativ aufgerufen, um die Äquivalenz von G und R zu

beweisen. Insbesondere kann inkrementelles Erfüllbarkeitstesten genutzt werden, um die Laufzeit der Verifikation zu reduzieren.

Zur Erzeugung des sequenziellen Modells werden zeitabstrakte Variablen zur Programmnetzliste hinzugefügt. Diese Variablen repräsentieren die abstrakten Zeitpunkte (d.h. die Sequenzindizes) zu denen die Software auf die Umgebung zugreift. Für jede Umgebungsadresse, auf die vom Programm zugegriffen wird, existiert eine entsprechende zeitabstrakte Variable, welche die Anzahl der Zugriffe auf die Adresse verfolgt. Jedes Mal, wenn eine Lade- oder Speicherinstruktion auf eine gegebene Umgebungsadresse zugreift, wird die entsprechende zeitabstrakte Variable inkrementiert. Dies hat zur Folge, dass die Logik, welche zur Repräsentation der Ein-und Ausgabesequenzen genutzt wird, verkettete Multiplexer erzeugt, welche als Eingang die Schnittstellensignale der Lade- oder Speicherinstruktionszellen zusammen mit der zeitabtrakten Variable erhält, um die Werte zu ermitteln, die zwischen der Software und der Umgebung an jeder Sequenzstelle ausgetauscht werden. Die notwendige Logik zur Konstruktion des sequenziellen Modells wird vereinfacht durch die Ausnutzung der Informationen aus dem Ausführungsgraphen über mögliche Ausführungspfade der Software. Durch die Traversierung des Ausführungsgraphen ist es möglich, eine Untermenge von Ein- oder Ausgabeinstruktionszellen, welche auf die Umgebung zu gegebenen Zeitpunkten zugreifen können, zu identifizieren. Das endgültige Sequenzmodell enthält ausschließlich Logik zur Modellierung der relevanten Instruktionszellen.

8.3 Zyklengenaue Hardware/Software Co-Verifikation

Diese Arbeit stellt ein Rechenmodell für Hardware/Software Coverifikation vor zum Eigenschaftstest für firmwarebasierte Designs, falls Prozessor und Firmware gemeinsam mit dem Rest des Systems integriert werden ohne Nutzung von Standardbusprotokollen. Für diese Entwurfsansätze werden Prozessoren mit einer hohen Vorhersagbarkeit des Zeitverhaltens genutzt, um das Verhalten der Firmware in die umgebende Hardware in einer zyklengenauen Weise intregrieren zu können mittels eines sogenannten Wrapper RTL.

Um sowohl Hardware als auch Firmware gemeinsam berücksichtigen zu können, wird ein Ansatz gewählt, der zwei verschiedene Arten von Abrollvorgängen kombiniert. Einerseits werden die Firmware und die Haupthardware (einschließlich der CPU, des Instruktionsspeichers und des Datenspeichers) von einer Programmnetzliste modelliert. Andererseits wird die Nebenhardware (einschließlich der Wrapper RTL und der Peripherie) mittels einer Art klassischen Bounded Model Checkings abgerollt, indem für jeden Zeitpunkt eine Kopie der zugehörigen Übergangsfunktion instanziiert wird. Um die Programmnetzliste mit der abgerollten Nebenhardware in ein berechenbares Modell kombinieren zu können, wird die

Programmnetzliste um ein zyklengenaues Modell der Hard- und Softwareschnittstelle erweitert. Dieses Modell, bezeichnet als zeitlich festgelegtes Schnittstellenmodell, repräsentiert die Ein- und Ausgabeoperationen (wie z.B. read, write, idle, usw.), welche zwischen der Hardware und der Software zu jedem Zeitpunkt des Abrollens stattfinden.

Die folgenden Elemente werden in das Modell integriert, um das zeitlich festgelegte Schnittstellenmodell erzeugen zu können. Zuerst wird eine zyklengenaue Zeitvariable in die Programmnetzliste eingebaut. Diese Variable bestimmt zu welchen Taktzyklen die Ein- und Ausgabeoperationen, die von der Software initiiert werden, stattfinden. Als zweites werden abhängig von der Zeitvariablen Resolutionslogikblöcke erzeugt um die Ein- und Ausgabewerte zu bestimmen, die zwischen der Hardware und der Software ausgetauscht werden zu jedem Zeitpunkt des Abrollvorgangs. Drittens werden sogenannte zeitliche Schnittstellenzellen, welche den Status aller Ein- und Ausgabesignal der CPU repräsentieren, instanziiert und mit den Resolutionsblöcken verbunden. Das resultierende Modell, einschließlich der Programmnetzliste und des zeitlichen Schnittstellenmodells, beschreibt präzise das funktionale Verhalten der Firmware während der Laufzeit auf der Hardwareplattform Takt für Takt über einen begrenztes Zeitfenster.

Verschiedene Vereinfachungen werden unternommen um die Menge an Logik zu reduzieren, die benötigt wird um das zeitliche Schnittstellenmodell zu erzeugen und das Schlussfolgern über das resultierende, zusammengesetzte Hardware-/Softwaremodell zu vereinfachen. Diese Vereinfachungen werden unter der Prämisse ausgeführt, dass die Menge der Ein- und Ausgabeinstruktionen der Firmware, welche mit der relevanten Systemhardware zu festen Zeitpunkten des Abrollens interagieren kann, bekannt ist. Mit diesen Informationen über das zeitliche Verhalten wird die Logik zur Modellierung der Schnittstelle reduziert, weil zu einem gegebenen Zeitfenster die Logik nur von den relevanten Ein- und Ausgabeinstruktionen abhängt, welche tatsächlich mit der Hardware in dem Zeitfenster interagieren, und nicht von weiteren Instruktionen, welche zu Zeitpunkten außerhalb des Zeitfenstern auf die Schnittstelle zugreifen oder andere Bereiche, welche nicht der relevanten Nebenhardware entsprechen. Da diese Information explizit zum Schnittstellenmodell hinzugefügt wird, kann das Entscheidungsverfahren, welches zur Beweisführung für das Modell genutzt wird, diese Information unmittelbar abgreifen, anstatt diese aus anderen, komplizierteren Repräsentationen herzuleiten.

Dieses endgültige Berechnungsmodell bewahrt die Genauigkeit eines normalen BMC Ansatzes, in dem das gesamte konkrete System einschließlich der im Instruktionsspeicher geladenen Firmware abgerollt wird. Allerdings modelliert das vorgestellte Modell auch explizit die möglichen Instruktionssequenzen, die tatsächlich während realer Programmausführungen auftreten können. Dies ermöglicht den Erfüllbarkeitsentscheider direkt diese Ausführungspfade während des Prüfungsprozesses zu untersuchen.

Appendix A

Preliminaries

A.1 Mathematical Background

This section presents different mathematical concepts which are relevant to this work. The description aims to give basic definitions and the corresponding notations. A more comprehensive description of these concepts can be found in [GT96, Ros11].

A.1.1 Sets

A set is a collection of distinct elements. Sets are denoted using curly braces and the elements of a set are enclosed inside the braces. For example the set $X = \{a, b, c\}$ contains the elements a, b, and c. Sets can also be defined by using quantification expressions such as $W = \{p : p\ is\ prime\ and\ p \leq 5\}$. If an object belongs to a set then it is said that the object is an *element* of the set. This membership is indicated with $\in$. On the other side, non-membership is indicated with $\notin$. For instance, in the given example it holds that $a \in X$ and also that $d \notin X$. The *cardinality* of a set defines the number of elements of it and is denoted for a set X as $|X|$, in the above example for instance $|X| = 3$.

If all elements of a set X are completely contained in set Y, then X is a *subset* of Y. This containment is denoted as $X \subseteq Y$ (similarly not being a subset is denoted with the symbol $\not\subseteq$). The *empty set* $\emptyset$ is defined as a set with no elements in it.

For two sets X and Y the following operations are defined. $X \cup Y$ defines the *union* as the set $\{z : z \in X\ or\ z \in Y\}$. Likewise, the *intersection* is defined as $X \cap Y = \{z : z \in X\ and\ z \in Y\}$. The *Cartesian product* is a set defined as $X \times Y = \{(x, y) : x \in X\ and\ y \in Y\}$. Elements (x, y) of $X \times Y$ are called *tuples* (note that the order matters in a tuple). The presented set operations can be extended to a finite number of operands W_i with $i = 1, 2, \ldots, n$ and $n \geq 2$ as $\bigcup_{i=1}^{n} W_i$, $\bigcap_{i=1}^{n} W_i$ and $\prod_{i=1}^{n} W_i$, representing the union, the intersection and the Cartesian product respectively. For the case of the Cartesian product, each element of the resulting set is a tuple of n elements $(w_1, \ldots, w_i, \ldots, w_n)$ with $w_i \in W_i$.

If all sets of the Cartesian product are identical, i.e. $Wi = W$ for every i, then the result is written W^n.

A.1.2 Relations, Functions, and Sequences

A *relation* R from two sets X and Y is a subset of $X \times Y$. Relations can also be defined on a particular set X. In this case, the relation R on X is a subset of $X \times X$.

A relation R on a set X is *reflexive* if it holds that $(x, x) \in R$ for all $x \in X$. The relation R is *antisymmetric* if for $x, y \in X$ it holds that if $(x, y) \in R$ and $(y, x) \in R$ implies $x = y$. Furthermore, R is *transitive*, if for $x, y, z \in X$ it holds that if $(x, y) \in R$ and $(y, z) \in R$ implies $(x, z) \in R$. A *partial order* is a relation that is simultaneously reflexive, antisymmetric, and reflexive. It is called "partial order", since not every possible couple of elements of X can be compared with the relation. On the contrary, for a *total order* any pair of elements $x, y \in X$ can always be compared, i.e., for all $x, y \in X$ there is a tuple (x, y) or (y, x) belonging to the relation.

A relation $f \subseteq X \times Y$ is called a *function*, commonly written as $f : X \mapsto Y$, if for every $x \in X$ there is an unique $y \in Y$ such that the tuple $(x, y) \in f$. This means that for every $x \in X$, there is exactly one tuple $(x, y) \in f$ and no other tuple (x, y_1) with $y \neq y_1$ can be contained in f. Instead of writing $(x, y) \in f$, the notation $f(x) = y$ is preferably used. For functions, X is called the *domain* and Y the *codomain*. The *range* of a function is the set of elements of the codomain for which there exits a tuple (x, y). A function is *surjective* if its range is equal to the codomain. A function is *injective* iff for all $x_1, x_2 \in X$ it holds that if $f(x_1) = y$, $f(x_2) = y$ implies that $x_1 = x_2$. A *bijective* function is both surjective and injective.

Sequences are defined over sets and can have finite or infinite *length*. Sequences are denoted by using normal brackets. For example $(1, 1, 0)$ and $(0, 0, 1, 0)$ are sequences with three and four elements respectively. In general, a sequence of length n is a tuple with n elements. Therefore, it is possible to say that a sequence of length n over the set X must be a member of the Cartesian product X^n. The order in a sequence matters, therefore $(1, 1, 0)$ and $(1, 0, 1)$ are considered different. Likewise, repetition of elements is important. For instance $(1, 1, 0)$ and $(1, 1, 0, 0)$ correspond to different sequences.

A.1.3 Graphs

A *graph*, $G = (V, E)$, consists of the nonempty set of *nodes* V and the set of *edges* E. An edge $e \in E$ connects two nodes $u, v \in V$. While in a *directed graph* edges are ordered pairs (u, v), i.e. $E \subseteq V \times V$, in a *undirected graph* edges are unordered pairs $\{u, v\}$.

Graphs which are directed and have finite sets of nodes and edges are especially relevant to this work and are discussed more in detail in the following. For a given edge $e = (u, v)$

in a directed graph, the node u is the *immediate predecessor* of v and the node v is called the *immediate successor* of u. A *path* $p = (v_1, \ldots, v_k)$ of length k of G is a sequence (of length k) over V with the property that for each pair of consecutive sequence elements v_i, v_{i+1} it holds that $(v_i, v_{i+1}) \in E$. It is said that v_k is *reachable* from v_1 (via p), if there is a path p from v_1 to v_k. In this case, v_1 is called *predecessor* of v_k and v_k is called *successor* of v_1. In a directed graph, a path $(v_1, \ldots, v_k)$ forms a *cycle* if $v_1 = v_k$ and the path contains at least one edge.

A directed graph with no cycles is called *directed acyclic graph* (DAG). Under the property of reachability above introduced a DAG forms a partial order. A node in a DAG is called *root* node if the node has no predecessors. Similarly, a node is denoted as *end* node in a DAG if it has no successors.

A.2 Graph Algorithms

There are two basic methods to traverse a graph: *bread-first search* (BFS) and *depth-first search* (DFS). The following sections introduce the main ideas of these methods for the case of DAGs showing the corresponding implementations. While in practice the algorithms explore the nodes of a graph according to a particular application (e.g. distance computation, specific node finding, etc.), here the focus is on the simple task of visiting all nodes of the graph. The text summarizes the ideas presented in [CLR94]. The algorithms assume that the graphs to be traversed are represented by using *adjacency lists*. If a graph $G = (V, E)$ is considered, then the adjacency list $Adj[u]$ stores all direct successors of a node $u \in V$. Furthermore, each node $u \in V$ has associated a Boolean variable $marked[u]$ which guides the exploration of the graph. This variable is set to *true* after visiting the node u.

Section A.2.3 presents the *topological sorting* algorithm. A topological sort can be used, for instance, to ensure that nodes of a DAG are processed according their precedence which results beneficial in many applications.

A.2.1 Breadth-First Search

The BFS works based on the notion of distance. The distance is the number of nodes (e.g. successors) that need to be traversed in order to reach another node in the traversal. In particular, the BFS traverses nodes in ascending distance with respect to a starting node s. This means that if the current node v is examined, then the algorithm first visits all direct successors of v (nodes at distance one) before visiting any other node. Subsequently, the search is expanded by visiting the nodes which are at distance two of v and so on. The BFS ends after all nodes have been traversed, i.e., the variable $marked[u]$ is set to *true* for each node u.

Figure A.1 shows the basic BFS algorithm. Note that $marked[u]$ avoids that the node u is visited more than once. The algorithm employs a queue structure Q to record the nodes whose direct successors are to be visited next.

```
1: BFS(V, E, s) {
2:     for each u ∈ V ∧ u ≠ s {
3:         marked[u] ← false;
4:     }
5:     Q ← {s};
6:     while Q ≠ ∅ {
7:         u ← head[Q]; {
8:         for each v ∈ Adj[u] {
9:             if marked[v] = false {
10:                ENQUEUE(Q, v);
11:            }
12:        }
13:        DEQUEUE(Q);
14:        marked[u] ← true;
15:    }
16: }
```

Figure A.1: Basic BFS algorithm

A.2.2 Depth-First Search

In contrast to the BFS strategy, the DFS explores as deep as possible the successors of a given node. If a node u is currently examined then the algorithm selects a direct successor of u (which has not been visited yet) and proceeds directly to examine it in a recursive way. Note that unexplored nodes are possibly left in the search after moving to the direct successor of u. This process continues on the current path until the selected node corresponds to an end node or to a node with all successor nodes visited. A node is marked as visited only if all its direct successors are also marked as visited. The algorithm then returns back to the last node which still has pending nodes that are not visited. The search continues recursively until all nodes have been visited.

Figure A.2 shows the basic DFS algorithm. As for the BFS, $marked[u]$ avoids revisiting of node u. As can be observed, the implementation is based on the recursive call of the procedure DFS_VISIT. Non-recursive implementations can be also developed by using a stack structure.

```
1: DFS(V, E) {
2:     for each u ∈ V {
3:         marked[u] ← false;
4:     }
5:     for each u ∈ V {
6:         if marked[u] = false {
7:             DFS_VISIT(V, E, u);
8:         }
9:     }
10: }

12: DFS_VISIT(V, E, u) {
13:     for each v ∈ Adj[u] {
14:         if marked[v] = false {
15:             DFS_VISIT(V, E, u);
16:         }
17:     }
18:     marked[u] ← true;
19: }
```

Figure A.2: Basic DFS algorithm

A.2.3 Topological Sorting

A topological sort orders linearly the nodes of a DAG such that if the graph contains an edge (u, v), then u appears before v in the ordering. A DFS can be employed in order to sort a DAG topologically by executing the following steps.

1. given a DAG $G = (V, E)$ **call** *DFS*(V, E) (cf. Figure A.2)

2. **initialize** a linked list as empty at the beginning of *DFS*(V, E)

3. as a given each node is finished (right after marking it as visited) **insert** it at the head of the linked list

4. **return** the linked list after processing all nodes of G

Figure A.3 shows the modified DFS algorithm implementing the topological sorting. The resulting list contains all DAG nodes in *reverse* order with the last inserted node at the head of the list.

```
1: T_SORT(V, E) {
2:     list ← {}
2:     for each u ∈ V {
3:         marked[u] ← false;
4:     }
5:     for each u ∈ V {
6:         if marked[u] = false {
7:             DFS_VISIT(V, E, u);
8:         }
9:     }
10: }

12: DFS_VISIT(V, E, u) {
13:     for each v ∈ Adj[u] {
14:         if marked[v] = false {
15:             DFS_VISIT(V, E, u);
16:         }
17:     }
18:     marked[u] ← true;
19:     INSERT(list, u);
20: }
```

Figure A.3: Basic topological sorting algorithm

A.3 Boolean Functions

A Boolean variable x can take (be assigned) only one of the values in $\boldsymbol{B}$, where $\boldsymbol{B}$ is the Boolean set $\{0, 1\}$. A *Boolean vector* is a tuple of Boolean variables $X = (x_1, \ldots, x_n)$ which can be assigned an element of $\boldsymbol{B^n}$.

A Boolean function f of n variables $x_1, \ldots, x_n$ is a mapping defined by $f : \boldsymbol{B^n} \mapsto \boldsymbol{B}$. The notation $f(x_1, \ldots, x_n)$ is used to emphasize the dependency on the variables $x_1, \ldots, x_n$. Alternatively, the notation $f(X)$ (with X the Boolean vector $X = (x_1, \ldots, x_n)$) can be also employed. A Boolean function is *satisfiable* if there is at least one input assignment for which the function evaluates (maps) to 1. If such an assignment does not exist then the function is *unsatisfiable*. A *tautology* is a Boolean function that always evaluates to 1.

A Boolean function with m outputs is a vector of Boolean functions $F(X) = (f_1, \ldots, f_m)$, where each function f_i depends on the variables $X = (x_1, \ldots, x_n)$. Note that F corresponds to the mapping $F : \boldsymbol{B^n} \mapsto \boldsymbol{B^m}$.

Boolean functions can be represented using truth tables or formulas. The Boolean functions *AND*, *OR*, *implication* and *equivalence* are represented by the symbols $\wedge$, $\vee$, $\rightarrow$,

and $\leftrightarrow$ respectively. The *NOT* function (or complement) of a variable x is represented as $\neg x$ and defined by $\neg 0 = 1$ and $\neg 1 = 0$.

All possible Boolean functions can be expressed in terms of the AND, OR, and NOT functions, for instance $a \rightarrow b = \neg a \vee b$ and $a \leftrightarrow b = (\neg a \vee b) \wedge (a \vee \neg b)$

The AND and the OR can be respectively expressed as $\bigwedge_{i=1}^{n} x_i$ and $\bigvee_{i=1}^{n} x_i$ in case that they depend on the variables x_i, where $i = 1, 2, \ldots, n$ and $n \geq 2$.

When using formulas to represent logical functions *normal forms* are often employed. The following terminology is introduced when using normal forms. The AND and OR are referred to the *conjunction* and the *disjunction* respectively. A *literal* is a variable or the complement of a variable. A *clause* is a disjunction of literals (e.g. $x_1 \vee \neg x_2 \vee \neg x_3$) and a cube is a conjunction of literals (e.g. $x_1 \wedge x_2 \wedge \neg x_3$). A formula in *conjunctive normal form* (CNF) is a conjunction of clauses. A formula in *disjunctive normal form* (DNF) is a disjunction of cubes.

Different to truth tables, normal forms are not canonical. There are other representations based on *minterms* and *maxterms*, not discussed here, which are canonical [GT96]. Reduced ordered binary diagrams (ROBDDS) [Bry86] are also a canonical representation which can be used to represent Boolean functions more efficiently in a computer.

A.3.1 Characteristic Functions

Boolean sets can be specified by means of characteristic functions. The *characteristic function* of a set $A \subseteq \boldsymbol{B^m}$ is defined as a Boolean function $\chi_A : \boldsymbol{B^m} \mapsto \boldsymbol{B}$ which evaluates to 1 for the elements $a \in A$. Otherwise, for elements $b \notin A$ (and $b \in \boldsymbol{B^m}$) χ_A is equal to 0. Note that if $A = \emptyset$ then $\chi_A = 0$ and if $A = \boldsymbol{B^m}$ then $\chi_A = 1$. Following this definition, the characteristic function of a Boolean relation $R \subseteq \boldsymbol{B^n} \times \boldsymbol{B^m}$ equals to 1 if $(a, b) \in R$ with $a \in \boldsymbol{B^n}$ and $b \in \boldsymbol{B^m}$. For the particular case Boolean functions $f : \boldsymbol{B^n} \mapsto \boldsymbol{B}$, the characteristic function predicates on the valid assignments of the function and can be described by the formula $\chi_f(X, y) = f(X) \leftrightarrow y$, where $X = (x_1, \ldots, x_n)$. Likewise for a function with m outputs $f_1, \ldots, f_m$, the characteristic function can be written as $\bigwedge_{i=1}^{m}(f_i(X) \leftrightarrow y_i)$.

When employing characteristic functions, the set operations $\cap$ and $\cup$ can be replaced by the Boolean functions $\wedge$ and $\vee$ respectively. Likewise, the complement operation can be represented by the NOT function. Therefore, Boolean functions allow to manipulate sets of elements simultaneously and not individually which turns out to be beneficial for implementing efficient algorithms. For example, the union of the sets A and B can done by computing $\chi_A \vee \chi_B$ rather than iterating on each individual element of the sets A, B. This, of course, is valid only if Boolean functions are efficiently represented in a computer.

List of Figures

List of Tables

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[Acc04] Accellera Organization Inc. Property specification language - reference manual, version 1.1. http://www.eda.org/vfv/docs/PSL-v1.1.pdf, June 2004.

[AEF+05] Tamarah Arons, Elad Elster, Limor Fix, Sela Mador-Haim, Michael Mishaeli, Jonathan Shalev, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Lenore D. Zuck. Formal verification of backward compatibility of microcode. In *Proceedings of the 17th international conference on Computer Aided Verification*, CAV'05, pages 185–198, Berlin, Heidelberg, 2005. Springer-Verlag.

[AEO+08] T. Arons, E. Elster, S. Ozer, J. Shalev, and E. Singerman. Efficient symbolic simulation of low level software. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 825 –830, march 2008.

[Ait03] Thorn Aitch. Aquarius: a pipelined RISC CPU, 2003.

[ARM99] ARM Limited. AMBA specification (rev 2.0). http://www.arm.com, 1999.

[BA15] Armin Biere and Fröhlich Andreas. Evaluating CDCL restart schemes. In *Pragmatics of SAT 2015*, 2015.

[BBM+07] Jörg Bormann, Sven Beyer, Adriana Maggiore, Michael Siegel, Sebastian Skalberg, Tim Blackmore, and Fabio Bruno. Complete formal verification of TriCore2 and other processors. In *Design & Verification Conference & Exhibition (DVCon)*, 2007.

[BC00] P. Bjesse and K. Claessen. SAT-based verification without state space traversal. In *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, FMCAD '00, pages 372–389, London, UK, UK, 2000. Springer-Verlag.

[BC11] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[BCC+99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. International Design Automation Conference (DAC)*, pages 317–320, June 1999.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[BCRZ99] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a power PC-TM microprocessor using symbolic model checking without BDDs. In *Proc. International Conference on Computer Aided Verification (CAV)*, pages 60–71, 1999.

[BH08] Domagoj Babic and Alan J. Hu. Calysto: scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 211–220, New York, NY, USA, 2008. ACM.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast: Applications to Software Engineering. *Int. J. Softw. Tools Technol. Transf.*, 9:505–525, October 2007.

[BHvMW09] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[Bie08] Armin Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.

[BJW04] R. Brinkmann, P. Johannsen, and K. Winkelmann. *Advanced Formal Verification*, chapter Application of Property Checking and Underlying Techniques, pages 125–166. Springer US, Boston, MA, USA, 2004.

[BPR01] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 268–283, London, UK, 2001. Springer-Verlag.

[BPS00] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, June 2000.

[Bra93] Daniel Brand. Verification of large synthesized designs. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 534–537, 1993.

[Bra11] Aaron R. Bradley. Sat-based model checking without unrolling. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer-Verlag.

[BRV+16] Christian Bartsch, Niko Rödel, Carlos Villarraga, Dominik Stoffel Stoffel, and Wolfgang Kunz. A HW-dependent software model for cross-layer fault analysis in embedded systems. In *17th Latin-American Test Symposium (LATS)*, pages 153–158, April 2016.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[BVS+14a] Binghao Bao, Carlos Villarraga, Bernard Schmidt, Dominik Stoffel, and Wolfgang Kunz. A new property language for the specification of hardware-dependent embedded system software. In *Proc. Forum on Specification And Design Languages (FDL)*, Munich, Germany, Oct 2014. (accepted for publication).

[BVS+14b] Christian Bartsch, Carlos Villarraga, Bernard Schmidt, Dominik Stoffel, and Wolfgang Kunz. Efficient SAT/simulation-based model generation for low-level embedded software. In *17. GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 147–157, 2014.

[BVS+16] Binghao Bao, Carlos Villarraga, Bernard Schmidt, Dominik Stoffel, and Wolfgang Kunz. A new property language for the specification of hardware-dependent embedded system software. In Frank Oppenheimer and Julio Luis Medina Pasaje, editors, *Languages, Design Methods, and Tools for Electronic System Design*, volume 361 of *Lecture Notes in Electrical Engineering*, pages 83–100. Springer International Publishing, 2016.

[CBM89] Oliver Coudert, C. Berthet, and Jean-Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. *Lecture Notes on Computer Science*, 407:365–373, June 1989.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design*

and Implementation, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[CE81] E. M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *Lecture Notes in Computer Science*, 131, 1981.

[CEP00] L. A. Cortes, P. Eles, and Z. Peng. Formal coverification of embedded systems using model checking. In *Proc. Euromicro Conference*, volume 1, pages 106–113, 2000.

[CFF+06] David Currie, Xiushan Feng, Masahiro Fujita, Alan J. Hu, Mark Kwan, and Sreeranga Rajan. Embedded software verification using symbolic execution and uninterpreted functions. *Int. J. Parallel Program.*, 34(1):61–91, February 2006.

[CGJ+03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[CH00] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, September 2000.

[CHM+96] Hyunwoo Cho, G. D. Hachtel, E. Macii, M. Poncino, and F. Somenzi. Automatic state space decomposition for approximate FSM traversal based on circuit analysis. *IEEE Transactions on Computer-Aided Design*, 15(12):1451–1464, December 1996.

[CJ09] Ken Chapman and Les Jones. SEU Strategies for Virtex-5 Devices, 2009.

[CKY03] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using Bounded Model Checking. In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pages 368–371, New York, NY, USA, 2003. ACM.

[Cla07] Koen Claessen. A coverage analysis for safety property lists. In *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 139–145. IEEE Computer Society, 2007.

[CLM89] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS)*, pages 353–362, Piscataway, NJ, USA, 1989. MIT Press.

[CLM+10] Gianpiero Cabodi, Luciano Lavagno, Marco Murciano, Alex Kondratyev, and Yosinori Watanabe. Speeding-up heuristic allocation, scheduling and binding with SAT-based abstraction/refinement techniques. *ACM Trans. Des. Autom. Electron. Syst.*, 15(2):12:1–12:34, March 2010.

[CLR94] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, 1994.

[CNQ04] Gianpiero Cabodi, Sergio Nocco, and Stefano Quer. Improving SAT-based bounded model checking by means of BDD-based approximate traversals. *Journal of Universal Computer Science*, 10:1693–1730, 2004.

[Coo71] Stephen A. Cook. The complexity of theorem proving procedures. In *Proc. Annual ACM Symposium on the Theory of Computing*, pages 151–158, 1971.

[CS04] Christoph Csallner and Yannis Smaragdakis. Jcrasher: An automatic robustness tester for java. *Softw. Pract. Exper.*, 34(11):1025–1050, September 2004.

[CS05] Christoph Csallner and Yannis Smaragdakis. Check 'N' Crash: Combining static checking and testing. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 422–431, New York, NY, USA, 2005. ACM.

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[EES04] Dr. Wolfgang Ecker, Volkan Esen, and Thomas Steininger. Memory models for the formal verification of assembler code using bounded model checking. In *Proc. IEEE International Symposium on Object-Oriented Real-Time Distributed Computing.*, pages 129 –135, 2004.

[EJ09] Christof Ebert and Capers Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, 2009.

[ES03a] Niklas Een and Niklas Soerensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543 – 560, 2003.

[ES03b] Niklas Eén and Niklas Sörensson. An extensible SAT solver. In *SAT*, pages 502–518, 2003.

[GKD06] Daniel Große, Ulrich Kühne, and Rolf Drechsler. HW/SW co-verification of embedded systems using bounded model checking. In *GLSVLSI '06: Proceedings of the 16th ACM Great Lakes symposium on VLSI*, pages 43–48, 2006.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[God05] Patrice Godefroid. Software model checking the verisoft approach. *Formal Methods in System Design, 2005*, 26:77 – 101, 2005.

[Gru13] Jim Grundy. Firmware validation: challenges and opportunities. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, page 11, 2013.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Proc. International Conference Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pages 72–83, London, UK, 1997. Springer-Verlag.

[GSY04] Orna Grumberg, Assaf Schuster, and Avi Yadgar. *Formal Methods in Computer-Aided Design: 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004. Proceedings*, chapter Memory Efficient All-Solutions SAT Solver and Its Application for Reachability Analysis, pages 275–289. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[GT96] W. K. Grassmann and J.-P. Tremblay. *Logic and Discrete Mathematics*. Prentice Hall, 1996.

[Har09] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[Hol97] Gerard J. Holzmann. The SPIN model checker. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[Hoo93] J.N. Hooker. Solving the incremental satisfiability problem. *The Journal of Logic Programming*, 15(1):177 – 186, 1993.

[HP00] Klaus Havelund and Thomas Pressburger. Model Checking Java Programs Using Java PathFinder. *International Journal on Software Tools for Technology Transfer STTT*, 2(4):366–381, 2000.

[HS15] Marijn J. H. Heule and Torsten Schaub. What's hot in the sat and asp competitions. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 4322–4323. AAAI Press, 2015.

[HTV+13] A. Horn, M. Tautschnig, C. Val, L. Liang, T. Melham, J. Grundy, and D. Kroening. Formal co-validation of low-level hardware/software interfaces. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 121–128, Oct 2013.

[IBM99] IBM Corporation. CoreConnect Bus Architecture. http://www.ibm.com, 1999.

[Inf] Infineon Technologies AG. TriCore 2 architectural manual, doc v1.0.

[IYG+04] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient SAT-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3):256–274, 2004. International Symposium on Leveraging Applications of Formal Methods (ISoLA 2004).

[JHB12] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. *Automated Reasoning: 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, chapter Inprocessing Rules, pages 355–370. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[JKSC05] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund Clarke. Word level predicate abstraction and refinement for verifying RTL Verilog. In *Proc. International Design Automation Conference (DAC)*, pages 445–450, New York, NY, USA, 2005. ACM Press.

[JLBRS12] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international sat solver competitions. *Artificial Intelligence Magazine*, 33(1), 2012.

[JM09] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41:21:1–21:54, October 2009.

[JMF95] J. Jain, R. Mukherjee, and M. Fujita. Advanced verification techniques based on learning. In *Proc. International Design Automation Conference (DAC)*, pages 420 – 426, 1995.

[JS05] HoonSang Jin and Fabio Somenzi. Prime clauses for fast enumeration of satisfying assignments to boolean circuits. In *Proc. International Design Automation Conference (DAC)*, pages 750 – 753, 2005.

[KG14] Johannes Koesters and Alex Goryachev. Verification of non-mainline functions in today's processor chips. In *Proceedings International Design Automation Conference (DAC)*, DAC'14, pages 1:1–1:3, New York, NY, USA, 2014. ACM.

[KGN+09] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik. Replacing testing with formal verification in intel coretm i7 processor execution engine validation. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 414–429, Berlin, Heidelberg, 2009. Springer-Verlag.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[KK97] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In *Proc. International Design Automation Conference (DAC)*, pages 263–268, November 1997.

[Kri63] Saul A. Kripke. Semantical analysis of modal logic I normal modal propositional calculi. *Mathematical Logic Quarterly*, 9(5-6):67–96, 1963.

[Kro07] Thomas Kropf. Software bugs seen from an industrial perspective or can formal methods help on automotive software development? In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 3–3, Berlin, Heidelberg, 2007. Springer-Verlag.

[KSMS11] Hadi Katebi, Karem A. Sakallah, and Joao P. Marques-Silva. Empirical study of the anatomy of modern sat solvers. In *Proceedings of the 14th International Conference on Theory and Application of Satisfiability Testing*, SAT'11, pages 343–356, Berlin, Heidelberg, 2011. Springer-Verlag.

[Kun93] W. Kunz. HANNIBAL: An efficient tool for logic verification based on recursive learning. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 538–543, November 1993.

[LIN02] LIN Administration. LIN Specification Package Rev. 1.3, 2002.

[LRM06] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Systems*, 34(3):195–227, 2006.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

[McM02] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Proc. International Conference on Computer Aided Verification (CAV)*, pages 250–264. Springer-Verlag, 2002.

[McM03] K. L. McMillan. Interpolation and SAT-based model checking. In *Proc. International Conference on Computer Aided Verification (CAV)*, pages 1–13, 2003.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proc. International Design Automation Conference (DAC)*, pages 530 – 535, 2001.

[MSS99] P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[MVSK16] Oliver Marx, Carlos Villarraga, Dominik Stoffel, and Wolfgang Kunz. A computer-algebraic approach to formal verification of data-centric low-level software. In *14. ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE) (to appear)*, 2016.

[NTW+08] Minh D. Nguyen, Max Thalmaier, Markus Wedler, Jörg Bormann, Dominik Stoffel, and Wolfgang Kunz. Unbounded protocol compliance verification using interval property checking with invariants. *IEEE Transactions on Computer-Aided Design*, 27(11):2068–2082, November 2008.

[NWSK11] Minh D. Nguyen, Markus Wedler, Dominik Stoffel, and Wolfgang Kunz. Formal Hardware/Software Co-Verification by Interval Property Checking with Abstraction. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 510–515, New York, NY, USA, 2011. ACM.

[One] Onespin Solutions GmbH. OneSpin 360 DV-Verify. https://www.onespin.com/products/360-dv-verify/.

[PBG05] Mukul R Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology*, 7(2):156–173, apr 2005.

[PE05] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 504–527, Berlin, Heidelberg, 2005. Springer-Verlag.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on Foundations of Computer Science*, pages 46 –57, 31 1977-nov. 2 1977.

[PV09] Corina S. Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353, October 2009.

[Ren05] Renesas Electronics Corporation TYO. SH-1/SH-2/SH-DSP software manual, rev. 5.0. http://www.renesas.com/, 2005.

[RLT+10] Thomas Reps, Junghee Lim, Aditya Thakur, Gogul Balakrishnan, and Akash Lal. There's plenty of room at the bottom: analyzing and verifying machine code. In *Proceedings of the 22nd international conference on Computer Aided Verification*, CAV'10, pages 41–56, Berlin, Heidelberg, 2010. Springer-Verlag.

[Ros11] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Education, 2011.

[Sch10] Bastian Schlich. Model checking of software for microcontrollers. *ACM Trans. Embed. Comput. Syst.*, 9(4):36:1–36:27, April 2010.

[Sht01] Ofer Shtrichman. *Correct Hardware Design and Verification Methods: 11th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2001 Livingston, Scotland, UK, September 4–7, 2001 Proceedings*, chapter Pruning Techniques for the SAT-Based Bounded Model Checking Problem, pages 58–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[Sht02] Ofer Shtrichman. Sharing information between instances of a propositional satisfiability (SAT) problem, September 5 2002. US Patent App. 09/990,390.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, September 2005.

[Spe08] Chris Spear. *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer, 2008.

[SSS00] M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a SAT-solver. In *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2000.

[SVB+13] Bernard Schmidt, Carlos Villarraga, Jörg Bormann, Dominik Stoffel, Markus Wedler, and Wolfgang Kunz. A computational model for SAT-based verification of hardware-dependent low-level embedded system software. In *Proceedings of the 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 711–716, 2013.

[SVF+13a] Bernard Schmidt, Carlos Villarraga, Thomas Fehmel, Jörg Bormann, Dominik Stoffel, and Wolfgang Kunz. A hardware-dependent model for SAT-based verification of interrupt-driven low-level embedded system software. In *16. GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, 2013.

[SVF+13b] Bernard Schmidt, Carlos Villarraga, Thomas Fehmel, Jörg Bormann, Markus Wedler, Minh Nguyen, Dominik Stoffel, and Wolfgang Kunz. A new formal verification approach for hardware-dependent embedded system software. *IPSJ Transactions on System LSI Design Methodology (Special Issue on ASPDAC-2013)*, 6:135–145, 2013.

[SWWK04] D. Stoffel, M. Wedler, P. Warkentin, and W. Kunz. Structural FSM-traversal. *IEEE Transactions on Computer-Aided Design*, 23(5):598–619, May 2004.

[TNW+10] M. Thalmaier, M. Nguyen, M. Wedler, D. Stoffel, J. Bormann, and W. Kunz. Analyzing k-step induction to compute invariants for SAT-based property checking. In *Proc. International Design Automation Conference (DAC)*, pages 176 –181, 2010.

[Tse68] G. S. Tseitin. *Studies in constructive mathematics and mathematical logic. Part II*, volume 8, chapter On the complexity of proof in propositional calculus, pages 234–259. "Nauka", Leningrad. Otdel., Leningrad, 1968.

[vE00] C. A. J. van Eijk. Sequential equivalence checking based on structural similarities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(7):814–819, Jul 2000.

[vEJ95] C.A.J. van Eijk and J. A. G. Jess. Detection of equivalent state variables in finite state machine verification. In *In Proc of the International Workshop on Logic Synthesis*, pages 3–35, 1995.

[VSB+13] Carlos Villarraga, Bernard Schmidt, Christian Bartsch, Jörg Bormann, Dominik Stoffel, and Wolfgang Kunz. An equivalence checker for hardware-dependent software. In *11. ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 119–128, 2013.

[VSB+14] Carlos Villarraga, Bernard Schmidt, Binghao Bao, Rakesh Raman, Christian Bartsch, Thomas Fehmel, Dominik Stoffel, and Wolfgang Kunz. Software in a hardware view: New models for HW-dependent software in SoC verification and test (invited paper). In *Proc. International Test Conference (ITC'14)*, 2014.

[VSK16] Carlos Villarraga, Dominik Stoffel, and Wolfgang Kunz. *Formal System Verification*, chapter Software in a Hardware View: New Models for HW-dependent Software in SoC Verification. Springer (to appear), 2016.

[WCGP12] M. Wedler, E. Cabrill, S. Graham, and L. Patrick. Using formal verfication for HW/SW co-verification of an FPGA IP core. *Xcell Journal (Xilinx, Inc.)*, 0:56–61, 2012.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-case Execution-time Problem-Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[WPL+12] T. Webel, T. Pflueger, R. Ludewig, C. Lichtenau, W. Niklaus, and R. Schaufler. Scalable and modular pervasive logic/firmware design. *IBM J. Res. Dev.*, 56(1):54–63, January 2012.

[Xil11a] Xilinx, Inc. Logicore ip soft error mitigation controller v3.1, 2011.

[Xil11b] Xilinx, Inc. PicoBlaze 8-bit embedded microcontroller user guide, 2011.

[YSTM14] Yinlei Yu, Pramod Subramanyan, Nestan Tsiskaridze, and Sharad Malik. All-sat using minimal blocking clauses. In *Proceedings of the 2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, VLSID '14, pages 86–91, Washington, DC, USA, 2014. IEEE Computer Society.

[ZES13] R. Zahir, M. Ewert, and H. Seshadri. The medfield smartphone: Intel architecture in a handheld form factor. *IEEE Micro*, 33(6):38–46, Nov 2013.

Lebenslauf

Name:	Villarraga Pinzón, Carlos Ernesto

Ausbildung

1996-2002	Bachelor Elektro- und Eletronikingenieur an der Universidad de los Andes (dt. Universität der Anden) Bogota, Kolumbien
2003-2006	Master der Ingenieurwissenschaft mit dem Schwerpunkt Mikroelektronik an der Universidad de los Andes

Stipendien

2009-2013	Stipendium des DAAD für Promotionstudium

Berufstätigkeit

2003-2006	Wissenschaftliche Hilfskraft Forschungsgruppe für Mikroelektronische Systeme an der Universidad de los Andes
2013-2015	Wissenschaftliche Hilfskraft (als Doktorand) an der Technische Universität Kaiserslautern am Lehrstuhl für Entwurf informationstechnischer Systeme Kaiserslautern, Deutschland
seit 2015	Wissenschaftlicher Mitarbeiter am Lehrstuhl Entwurf Informationstechnischer Systeme Technischen Universität Kaiserslautern