Formal Hardware/Firmware Co-Verification of Optimized Embedded Systems

vom
Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von
Michael Schwarz, M.Sc.
geb. in Pirmasens, Deutschland

D 386

Tag der mündlichen Prüfung:	25.06.2021
Dekan:	Prof. Dr.-Ing. Ralph Urbansky
Vorsitzender:	Prof. Dipl.-Ing. Dr. Gerhard Fohler
Berichterstattende:	Prof. Dr.-Ing. Wolfgang Kunz,
	Prof. Dr.-Ing. Daniel Müller-Gritschneder

Abstract

Small embedded devices are highly specialized platforms that integrate several peripherals alongside the CPU core. Embedded devices extensively rely on *Firmware* (FW) to control and access the peripherals as well as other important functionality. Customizing embedded computing platforms to specific application domains often necessitates optimizing the firmware and/or the HW/SW interface under tight resource constraints. Such optimizations frequently alter the communication between the firmware and the peripheral devices, possibly compromising functional correctness of the input/output behavior of the embedded system. This poses challenges to the development and verification of such systems. The system must be adapted and verified to each specific device configuration.

This thesis presents a formal approach to formulate these verification tasks at several levels of abstraction, along with corresponding HW/SW co-equivalence checking techniques for verifying correct I/O behavior of peripherals under a modified firmware. The feasibility of the approach is shown on several case studies, including industrial driver software as well as open-source peripherals. In addition, a subtle bug in one of the peripherals and several undocumented preconditions for correct device behavior were detected by the verification method.

Contents

1 Introduction **1**

1.1 The History of Embedded Systems Design 1

1.2 Motivation and Overview 6

1.3 Publication List 9

2 Background **11**

2.1 Verification of Embedded Systems 11

2.1.1 Integer Linear Programming 11

2.1.2 The Satisfiability Problem 12

2.1.3 Model Checking 13

2.1.3.1 Sequential model 14

2.1.3.2 Interval Property Checking 15

2.1.4 Program Netlist 17

2.2 Timing Analysis 20

2.3 Abstraction Layers 23

2.3.1 Software 24

2.3.1.1 Machine Code & Assembly 24

2.3.1.2 High-level Languages 25

2.3.1.3 Software Concepts and Artifacts 26

2.3.2 Hardware 30

2.3.2.1 Register Transfer Level 31

2.3.2.2 Electronic System Level 32

3 Formal HW/SW-Co-Verification **35**
3.1 Related Work 36
3.2 Verification Model 38
3.3 ACCESS Verification Method 40
3.4 Software Model 46
3.4.1 Binary Level – Program Netlist 49
3.4.2 Source Code Level – C/C++ 50
3.5 Hardware Model 52
3.5.1 Register Transfer Level 53
3.5.1.1 Device Check 56
3.5.2 Electronic System Level – SystemC-PPA 58
3.5.3 Path Predicate Abstraction 60
3.6 Interconnect Models 63
3.6.1 Protocol Emulation 65
3.6.2 Direct Register Access 67

4 Experiments and Case Studies **70**
4.1 Performance Experiments 71
4.1.1 Interconnect: Unrolled vs. Static PN 72
4.1.2 Protocol Emulation vs. Direct Register Access 73
4.1.3 Scalability Experiments 74
4.2 ACCESS device checks 76
4.3 Case Study: Register Interface Optimization of PULPino GPIO . . 78
4.4 Case Study: HW/SW-Optimized Soft-SPI Implementation 80
4.5 Case Study: Industrial LIN 85

5 Cycle-Accurate Timing Extension for Program Netlists **89**
5.1 Cycle-accurate SW modeling 90
5.2 Case Study: Local Interconnect Bus (LIN) 97

6 Conclusion **99**

Bibliography **102**

Kurzfassung in Deutsch **109**

Chapter 1

Introduction

> "Our civilization runs on software."
>
> (Bjarne Stroustrup)

1.1 The History of Embedded Systems Design

Computer applications serve as the basis for modern scientific research, contribute to solving engineering problems, assist in decision making in business, and are often the key factor that differentiates modern products and services. By now, software and small scale computing systems are an integral part in every-day human activities, many times without us giving it much thought. Yet, the fields of computer science and engineering are still very young compared to most other scientific fields.

Every modern computer, as we know it, is based on the principles first described by Alan Turing in his 1936 paper *On Computable Numbers* [1]. The first actual realization of these principles – the 'First Generation' computers – were probably Konrad Zuse's *Z2* (1940), the British *Colossus* (1943) and the *Harvard Mk I* (1944).

For today's standards they were electro-mechanical monstrosities, consisting of electrical relays, vacuum tubes and mechanical memories, such as punch cards; weighing dozens of tons and consuming thousands of watts of electrical power.

The 'Second Generation' dominated the late 1950s and early 1960s and was marked by the invention of the transistor in 1947 at Bell Labs. The transistor proved to be a much smaller and more reliable replacement for the inefficient vacuum tubes. Yet, a computer of the second generation remained to be of considerable size and cost, restricting it still to the domain of universities and governments.

This changed with the invention of the integrated circuit, commencing the 'Third Generation' of computing. While this inherently increased storage and processing capabilities, the integrated circuit allowed the development of small-scale computers that began to bring computing to many smaller businesses. Large-scale integration of circuits finally enabled the development of very small processing units – embedded systems, such as flight data analyzers in the US Navy's F14A 'TomCat' fighter jet.

In 1971, the release of the world's first commercial microprocessor, Intel's *4004*, announced the start of the 'Fourth Generation'. Originally, microprocessors were very limited in their computational ability and speed, and were in no way an attempt to downsize the mainframe computers common at the time, but targeted an entirely different market. Accommodating much of the computer's processing abilities on a single chip, along with other inventions, such as Random Access Memory (RAM), drastically cut down cost. This did not only allow for the development of personal computers that were small and cheap enough to be available to ordinary people, but led to the omnipresence of embedded systems. It is widely regarded that most of today's computers still belong to the fourth generation, as the underlying technology remains fundamentally the same.

Through improvements in architecture design and the level of integration, computers have become extraordinarily more sophisticated, more efficient and faster. Yet,

they haven't become any more "powerful" – after all, they are at most a Universal Turing Machine (UTM), adhering to the same principles and restrictions as their ancestors. One of these principles is the distinction of the physical components of the computing system, i.e., the hardware, and the executed program, the software. So far only the development of the hardware has been portrayed, but the development of software is inseparably connected to it.

In the days of first-generation computers, our understanding of software did not yet exist. Initially, changing the program of those devices meant to manually rewire the hardware. This changed when Claude Shannon [2] explained how binary logic could be used to program a computer. Henceforth, the program could be represented as a long string of bits, the values of which were mechanically represented through holes on punch cards. This was, understandably, a process very prone to errors.

This was rather quickly supplanted by the introduction of assembly languages. The use of mnemonics and the use of an assembler greatly improved software quality by freeing the programmer from remembering numeric codes and calculating addresses. Yet, it didn't free the programmer from needing to have a very close understanding of the underlying hardware because machine languages and the corresponding assembly languages tend to be unique to a particular type of computer architecture.

High-level languages such as Fortran, COBOL and BASIC eventually allowed programmers to specify the software in an abstract way that was independent of the precise details of the hardware architecture of the computer. Since high-level languages are more abstract than assembly language, it is possible to use different compilers to translate the same high-level language program into the machine language of many different types of computers. This further improved programming productivity greatly. Nevertheless, assembly languages remained highly relevant well into the 1990s, as their code was generally of smaller size,

with less overhead and executing at higher speed than the one produced by the high-level equivalents. This can be traced back mainly to the limited performance of high-level language compilers.

Today, assembly code might still be found in device drivers and low-level embedded software, but even there the further usefulness and performance of assembly language relative to high-level languages is still an ongoing debate.

Embedded Software is a short-hand term for "Software of an Embedded System" where an *Embedded System* is a computing system that is "embedded" into a larger system or process. Embedded software is typically designed to perform one specific function, although a single piece of hardware may contain multiple pieces of software embedded in it. Embedded systems often use operating systems tailored to embedded use or they run the application software directly on 'bare metal', i.e., without an operating system, particularly where real-time operating criteria apply. Depending on the specific use case, slight variations in reaction time close to that of real-time approaches might still be acceptable.

The term *firmware* initially referred to the lower-level microcode involved in the implementation of machine instructions. It existed on the boundary between hardware and software; thus the name "firmware". Over time, popular usage extended the meaning of word "firmware" to denote a subset of a system's software that is a 'firm' part of the system. This subset is stored in some form of non-volatile read-only memory (ROM) and takes care of system-critical actions that are tightly linked to hardware, such as processor machine instructions for BIOS, bootstrap loaders, or the control systems for simple electronic devices such as a microwave oven, remote control, or computer peripherals.

The development of easily reprogrammable non-volatile memory, such as flash, enabled easy updates and bug fixes, making firmware less 'firm' and somewhat mutable. On the other hand, embedded software is designed to fulfill a clearly defined task over the course of its lifetime.

Hence, the delineation between the terms embedded firmware and embedded software is blurring.

'Ordinary' software for desktop computer environments can be run on the target platform alongside separate debugger applications monitoring the behavior of the development code as it is executed. The often highly limited resources of an embedded system do not allow to do this on the respective hardware. Running the embedded software directly on the development platform, i.e., a desktop PC, doesn't work, because its behavior often depends on the system's underlying hardware architecture.

One approach is to run the embedded software on the system and use specialized debugging ports, e.g., JTAG to communicate with a debugger on a separate computing platform. Another, frequently used approach is to emulate the physical chip as a virtual prototype (VP), thus making it possible to debug the software as if it was running on the actual hardware. This offers a lot more controllability and observability than debugging over debugging ports, but at the cost of reduced execution speed (due to the overhead of emulation). Furthermore, debugging on a VP might be the only option if the hardware itself is still under development.

Broadly speaking, embedded systems require more attention to testing and debugging because a great number of devices using embedded controls are critical for safety. People often leave their welfare, security, and their decisions in the hands of a software application. The reliability of these devices is, hence, of utmost importance

However, software testing rarely – if ever – eliminates every bug; this gave rise to Lubarsky's Law of Cybernetic Entomology:

> "There is always one more bug."

1.2 Motivation and Overview

Continuously growing demands for computing applications in the domain of *Internet of Things (IoT)* open new opportunities for embedded system technology but, at the same time, increase the already existing pressure of tight resource constraints on embedded computing platforms. Many of the visions of the IoT domain can only become reality if the needed devices have extremely low design and production costs ($<$ 1 $/device), yet, they must be highly efficient for the targeted application. Most nodes in an IoT network consist of rather small devices and provide a well-defined set of very specific functionalities. Hence, their development profile is similar to that of an embedded system.

Designing an embedded system involves the integration of numerous sub-systems in both hardware and software. Modern computing platforms usually include a multitude of peripheral devices through which they communicate with the environment, such as sensors, actuators or more general communication infrastructures.

The embedded system's software itself might again consist of many components, such as operating systems, middleware and device drivers, as well as software specifically developed for the embedded application.

A common strategy to address cost requirements is to use commercial off-the-shelf products and available hardware Intellectual Property (IP), such as microcontroller platforms, which require no or only minimal further hardware design effort. The desired device functionality is then implemented in software by reusing existing software IP and generic libraries. This successfully reduces design effort and costs.

Then, however, the project often fails to meet important non-functional design targets such as low power consumption and short execution time due to the inefficient utilization of the hardware platform resources by the generic software.

A large portion of an embedded system's chip area is consumed by memory, even

more so when relying on software to implement a functionality which otherwise would be implemented by dedicated hardware. One way to reduce the required instruction memory lies in optimizing and unifying the hardware/software interfaces for the peripherals of the system. On the software side, this interface is usually implemented using memory mapped I/O in a number of access methods for each of the system's various peripherals. Unifying these methods, if possible, enables reusing one instance of code for multiple peripheral devices, but may change the I/O behavior of the software to a given device, e.g., by now using a different order or number of load/store accesses. When compared with arithmetic instructions, load/store operations are more expensive w.r.t. the energy budget, as their execution always involves some action by secondary hardware (bus systems, network bridges, memory etc.) and might make energy saving schemes like sleep modes less effective. Therefore, customizing a computing platform usually entails a number of firmware optimizations concerning the access and communication structures between hardware and software.

Interactions between the hardware and software components are often complex. In many cases they require strict sequences of actions that need to follow certain rules and comply with environmental constraints. Often, these access constraints and preconditions are not documented, but nevertheless required by the device hardware or the involved bus systems. On top of possibly leading the application software developer into making wrong assumptions about the usage of already existing IP, it further complicates optimizations of the hardware/software interface.

In general, these optimizations cannot be performed by a compiler because of its lack of detailed knowledge about the system's hardware. Instead, the firmware developer or hardware designer makes such adjustments manually when customizing the platform. Changes to the hardware/software interface, as considered in this thesis, in fact constitute an optimization step that takes place after the regular optimizations performed by the compiler or by other general optimizations methods for firmware.

The class of optimizations considered here typically leads to local modifications of the I/O behavior of the program but does not change its global control flow. Yet, the considered optimizations pose significant challenges to designers and software developers. They may easily compromise the functional correctness of the I/O behavior of the embedded system, for example, by triggering unanticipated side effects in the hardware. As a response, the driver or the hardware device may move into an unexpected or undefined state, which in turn can lead to a failure of the whole system. Hence, even though the modifications are locally contained, their verification remains a challenging task.

It is apparent from the previous discussion that any notion of equivalence which only takes the software into account, even when hardware-dependent software models are used, cannot be appropriate to solve this verification problem. Since programs with different I/O behavior are compared, their equivalence can only be stated by considering, also, their effect on the hardware. In the general case, verification of the considered firmware optimizations would require to prove the *hardware/software co-equivalence modulo latency* over the entirety of the system for as many clock cycles as needed to cover the overall run time of the software. This is an invincibly complex task for realistic applications.

This thesis discusses a new formal approach to this problem which can be feasible in many practical settings. It is based on the observation that the complexity of the proof task can be reduced drastically by exploiting the specific characteristics of the optimization scenario described above.

By utilizing the locality of induced changes, the thesis proposes a formal approach where the cycle-accurate proof for the peripheral can be partitioned. In each local partition the global context of the complete program is still taken into account in a time-abstract fashion. By over-approximating the initial state space of the peripheral for each partition to be considered, we can conduct a series of cycle-accurate local proofs, each spanning over relatively small time intervals, whereas

the equivalence of the entire system follows as a compositional result from the equivalence of all segments examined in this way.

This work is organized as follows: Chap. 2 briefly introduces relevant concepts for the further understanding of this work and discusses research which is related or of interest to the area of HW/SW-Co-verification. It is followed by the presentation of the central verification methodology and its models as proposed by this thesis in Chap. 3. The feasibility of the presented method, along with other performance data, is shown by several experiments and case studies in Chap. 4. The thesis continues in Chap. 5 with a supplementary procedure extending the central verification method with cycle accurate timing information of the software. Finally, Chap. 6 summarizes the findings presented in this work.

1.3 Publication List

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

- M. Schwarz, M. Chaari, B. Tabacaru, W. Ecker: "A Meta-Model-Based Approach for Semantic Fault Modeling on Multiple Abstraction Levels", Proc. Design and Verification Conference & Exhibition Europe 2015 (DVCon), Munich, Germany, November 2015.

- M. Schwarz, C. Villarraga, D. Stoffel, W. Kunz: "Cycle-Accurate Software Modeling for RTL Verification of Embedded Systems", IEEE Design and Diagnostics of Electronic Circuits and Systems (DDECS), Dresden, Germany, 2017.

- M. Schwarz, R. Stahl, D. Müller-Gritschneder, U. Schlichtmann, D. Stoffel, W. Kunz: "ACCESS: HW/SW Co-Equivalence Checking for Firmware Optimization", 56th Annual Design Automation Conference, (DAC'19), Las Vegas, NV, USA, 2019.

- T. Ludwig, M. Schwarz, J. Urdahl, L. Deutschmann, S. Hetalani, D. Stoffel, W. Kunz: "Property-Driven Development of a RISC-V CPU", Design and Verification Conference US (DVCON US), San Jose, CA, USA, Feb. 2019.
- V. Herdt, D. Große, R. Drechsler, C. Gerum, A. Jung, J. Benz, O. Bringmann, M. Schwarz, D. Stoffel, W. Kunz: "Systematic RISC-V based Firmware Design", Forum for Specification and Design Languages (FDL), Southampton, United Kingdom, 2019.

Chapter 2

Background

This chapter gives a broad overview of core concepts employed in the models and the verification approach presented in the subsequent chapters. The goal is to provide sufficient information to enable understanding of the techniques presented in this work without going into the details of the respective background concept.

2.1 Verification of Embedded Systems

This section provides information on mathematical concepts, algorithms and models used in the context of *formal verification*.

2.1.1 Integer Linear Programming

Linear programming (LP) addresses the task of maximizing or minimizing a linear functional over a polyhedron. Integer linear programming (ILP) is a specialized form of LP in which some variables are restricted to integers. It is called pure if the restriction applies to all variables [3]. Formally, an ILP problem is given by:

$$\min \{\mathbf{c}^\intercal \mathbf{x} \mid \mathbf{A}\mathbf{x} \leq \mathbf{b};\ \mathbf{x} \in \mathbb{Z}^{dim(\mathbf{x})}\} \tag{2.1}$$

with $\mathbf{A}$ being the constraint matrix and $\mathbf{b}$, $\mathbf{c}$, $\mathbf{x}$ being vectors determining bounds, cost and variables, respectively.

The general ILP problem is proven to be $\mathcal{NP}$-complete, indicating that for larger problem instances complexity may become a problem. Yet, many powerful solvers exist – both free and commercial – that are capable of handling millions of variables of constraints. This is sufficient for many practical tasks, e.g., the timing computation presented in Chap. 5.

2.1.2 The Satisfiability Problem

Many problems occurring in Electronic Design Automation (EDA) can be mapped to the Boolean Satisfiability Problem (SAT). In this thesis, proofs of properties and equivalence checks are performed by reducing them to instances of the SAT problem. Although solving those problems is an integral part of the approaches presented in this thesis, the technical details on how to solve them is a field of research of its own and well out of the scope of this work. While solving those problems is an integral part of the approaches presented in this thesis, the way in which they are solved is not. Hence, this work relies on standard approaches employed by available solver engines. Therefore, only a brief introduction of SAT is provided. For more in-depth information, please refer to [4].

The goal of SAT is to answer the question whether for a given Boolean formula $f(x_1, ..., x_n)$ a variable assignment $A = x_1, ..., x_n$ exists such that $f(A)$ evaluates to $true$. If so, A is called a *satisfying assignment* and f is *satisfiable*. Otherwise, if there is no satisfying assignment, f is said to be *unsatisfiable*.

SAT was the first problem to be shown to be $\mathcal{NP}$-complete [5]. Even though many of the instances that occur in practical applications can be solved much more quickly, it is still highly advisable to keep the instance of the problem as concise as possible.

Reducing a problem to SAT is widely applied in formal verification. It is one of the basic working principles of *model checking* (see Sec. 2.1.3) and the generation of *program netlists* (see Sec. 2.1.4).

2.1.3 Model Checking

Model checking is a formal method of verifying the correctness of a given system, i.e., whether the system adheres to a given specification. In order to do so, all parts involved in model checking must be translated into an adequate formal representation. The behavior of the design is usually given by a sequential model, e.g., a finite-state machine or Kripke model. How a digital circuit can be formalized as a sequential model is elaborated in Sec. 2.1.3.1. The system's specification is formalized by a set of desired properties expressed in temporal logic. Hence, model checking is also known as property checking. Note that a property is not restricted to asserting the existence of wanted behavior, but can also show the absence of undesired behavior.

Together, the sequential model and the property set, pose the task of proving whether all properties are satisfiable on the model. The combination of a the sequential model with a single property forms a computational model which a model checker attempts to solve using logic reasoning. This differentiates model checking from coverage-driven verification methods such as simulation that employ stimuli and assert an expected response.

Model checking is used for both hardware verification and software verification. Most model checking tools provide front ends for common hardware description languages (e.g., VHDL and Verilog) and common software programming languages. In industry, model checking for software is not quite as popular as for hardware because the inherent possibility of undecidability in software may require further information or involvement of an expert.

The properties are formulated in specific property languages such as the *Property*

Specification Language (PSL) [6], *System Verilog Assertions* (SVA) [7] or the *Interval Temporal Logic Language* (ITL) [8].

Model checking is an umbrella term encompassing various different techniques. Within the scope of this work the concept of Interval Property Checking (IPC) plays an important role. Its basic principles are detailed in Sec. 2.1.3.2.

2.1.3.1 Sequential model

Digital circuits can be categorized as either sequential or combinational. A circuit is combinational if its output is a pure function of its current input. Hence, its functionality can be represented by Boolean logic. In contrast, the output of a sequential circuit additionally depends on the sequence of prior inputs which is encoded by an internal state stored in some form. This cannot be expressed by Boolean logic alone but requires the formalisms described by automata theory, such as state machines or transition systems. In these formalisms, the internal state of the circuit is conceptualized by an abstracted state, often defined by a predicate, which changes according to certain events.

The most common form of visualization is the *state transition graph (STG)*, a directed graph $G = \{V, E\}$, with V being a set of vertices in which each vertex corresponds to a single abstract state, and with $E \subseteq V \times V$ being the set of edges between those vertices, according to the possible transitions in the automaton. Due to this representation as a graph, many terms used in graph theory also apply in the context of automata, such as:

- *Path*: A finite or infinite sequence of edges $P = \{e_1, e_2, ..., e_{n-1}\} \subseteq E$ connecting a sequence of vertices $W = \{v_1, ..., v_n\} \subseteq V$ on a graph $G = \{V, E\}$, such that each vertex in W is distinct.
- *Reachable state*: A state s_R for which a path $P = s_{start} \rightarrow s_R$ beginning at a starting state exists, i.e., it is reachable from the set of initial states.

- *Sequential depth*: The sequential depth of an automaton is equivalent to the maximum *eccentricity* among initial states. In this context, the eccentricity of a node is defined as the maximum *distance* to any other node. The distance between to nodes, in turn, is the minimum number of edges of any path between these nodes. Note that this is not equal to the *diameter* of G, which is the maximum eccentricity among all vertices, not just the ones in the set of initial states.

2.1.3.2 Interval Property Checking

IPC [9] is a SAT-based model checking technique, rooted in the industrial developments of the 1990s and closely related to Bounded Model Checking (BMC) [10].

A BMC problem considers a specific starting state and looks at a time interval of length k, by unrolling the sequential model from the starting state into a combinational circuit for k clock cycles. This problem can be efficiently mapped to SAT and solved by respective solvers. The underlying principle of BMC is to search for a counterexample to the examined property within that bounded time interval. If no bug is found, then one increases k until either a bug is found, the problem becomes intractable, or some preset upper bound is reached.

If this upper bound is smaller than the sequential depth of the examined system some part of the reachable state space will not be considered. In this case a proof verifying a property is incomplete and holds true only for the bounded interval. Extending the interval beyond the sequential depth, is, in most cases, not feasible due to the computational complexity of the SAT problem as well as the complexity of calculating the sequential depth in the first place.

Hence, because it is much more efficient in disproving (falsifying) a property, i.e., finding a counterexample that violates a property, rather than in proving (verifying) it, BMC is mostly seen as a formal technique for bug hunting.

In contrast to bounded model checking, IPC is able to produce globally valid, unbounded proofs. It shares the same computational model as BMC, with one significant difference: The starting state is not fixed but is an arbitrary state defined by a predicate P.

If it is shown that an IPC property φ starting from such an arbitrary state described by P will always end in a state satisfying P, then P characterizes an invariant. Hence, a proof verifying φ holds *unbounded*, i.e., it is valid not only for the examined finite time interval, but for an arbitrary large time frame.

Because the property φ describes behavior over a finite time interval it is also known as an *interval property*.

Definition 1 (Interval Property)**.** *An interval property φ is a LTL formula of the form* $\mathsf{G}\ (A \rightarrow C)$ *where both sub-formulas A and C, referred to as* assumption *and* commitment*, respectively, describe behavior over a finite time, i.e., the only temporal operator that may be used is the next operator,* X. □

Fig. 2.1 shows, conceptually, the computational model for an interval property φ for a time interval of length three. The sequential model is unrolled over the length of the interval, i.e., for each timepoint (cycle) in the interval, a new instance of its combinational logic is created and added to the computational model. The registers of the sequential model, which form its state variables, are cut from the combinational model. Instead the values which would be stored are propagated to the following instance of combinational logic. The starting state s_i is determined by the predicate P, which in turn is part of the assumption A. The result is a single combinational circuit representing the behavior of the sequential model over the given time interval.

This is now easily mapped to SAT, which searches for a set of values $X = \{x_i, ..., x_k\}$ that satisfies the negation of the commitment C under the assumption A. If such a set exists, φ is disproved, with X being a counterexample of the property.

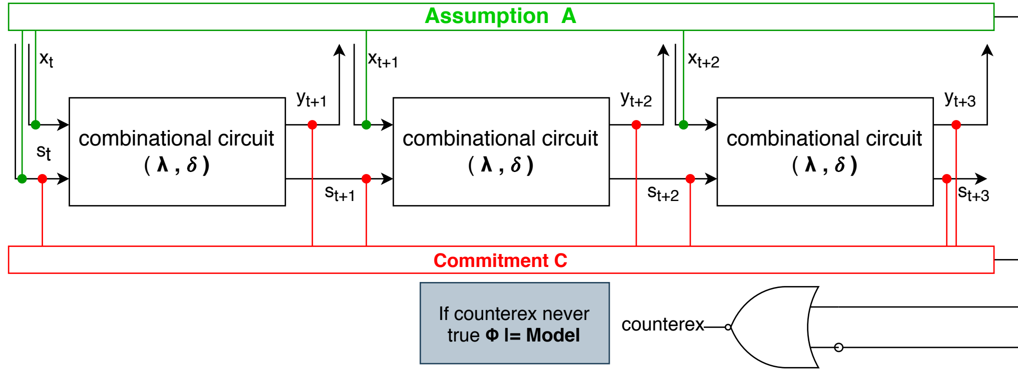


Figure 2.1: Proof computation for interval properties

Otherwise, the property is valid.

Proofs by IPC are *conservative*. The procedure never produces false positives, i.e., the procedure never erroneously reports a a property ϕ to hold if a state in P exists that violates the property . If P over-approximates the actual reachable state space of the sequential model, false negatives may occur and lead to spurious counterexamples. This can be mitigated by providing a strengthened invariant for the design, either by manually inspecting the design and providing additional assertions, or by automatic methods as proposed in [11].

2.1.4 Program Netlist

This work considers low-level driver software interacting with hardware. This calls for a computational model with a path-oriented program view, as it is common in Hardware (HW)-independent software verification. At the same time, the model must be easy to integrate with the models for the interacting hardware and must provide precise information about the effects of the executed software onto the hardware. In light of these requirements we chose to represent the software as a *Program Netlist (PN)* [12, 13]. This model represents all execution paths in a hardware-dependent way. Syntactically, the PN is a combinational circuit, which makes it easy to integrate the model with actual hardware to examine the combined behavior.

The following is a short overview of the PN generation technique; for further details, the reader is referred to the original literature [12, 13].

Starting point of model generation is the extraction of the Control Flow Graph (CFG) of the software from the program's machine code or assembler code, which is defined as a connected graph $G = (V, E)$ with a set of nodes V and a set of edges $E \subseteq V \times V$. While in the usual definition the nodes of a CFG represent basic blocks they, here, represent machine instructions. A machine instruction is a pair (a, w) of an instruction address a and the instruction word w stored at that address in the program memory. There is an edge (v_i, v_j) between two instructions v_i and v_j, if the program can make a transition from v_i to v_j. The CFG is then unrolled into the Execution Graph (EXG) from which, in turn, the PN is synthesized. The PN is a combinatorial circuit representing all possible executions of the program.

For example, a loop is unrolled until the loop end condition is reached. An additional Boolean-valued signal called *active* is propagated by the instructions which allows a SAT solver to keep track of the active execution path in the program under the current set of value assignments in the SAT instance.

The SAT solver is used to check whether there exist executions where the active flag of the loop-back branch can (still) become active. Similarly, branch destination addresses can be computed on the incomplete program netlist to trace the actual flow of control in the program.

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. Fig. 2.2 illustrates this process.

An important component of the model building process is the merging of nodes in the execution graph. Whenever the control flow modeled in an unrolled path segment reaches a previously visited program location, a node merging might be possible. Instead of immediately creating a new node, the last node is tentatively

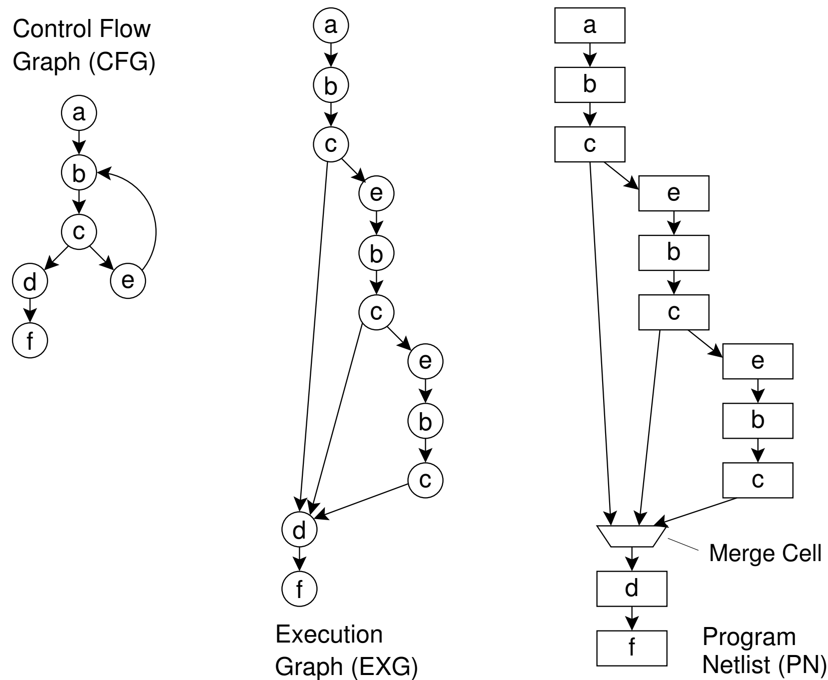


Figure 2.2: Generating a program netlist

connected to the existing one for that location. Provided that this does not introduce a cycle in the graph, the connection is made permanent. Otherwise, the nodes are disconnected and a connection to a newly created node is made. These intermittently performed checks for a possible merging of paths keep the model compact by sharing of sub-graphs.

Hence, the provided EXG is a directed acyclic graph (rather than a tree) containing information about

- all possible execution paths,
- all possible input/output access sequences to peripheral hardware components and to shared memory,
- the address spaces reached by every instruction, and
- all possible effects of the program on the program-visible hardware registers.

2.2 Timing Analysis

In the overarching context of digital circuits, the term *timing analysis* is only loosely defined and can actually refer to a multitude of things. For instance, the probably most common notion associated is that of *static timing analysis* [14, 15], which is used to compute the delay of signals in a combinational circuit. This is important in order to identify the circuit's critical path, i.e., the path between an input and an output with the maximum delay. Only then can it be assured that the circuit is able to operate at the required speed of a specified operating clock signal. Hence, static timing analysis is an essential step in logic synthesis widely used in respective EDA tools.

Another form of timing analysis more relevant to the proposed approaches is that of Worst Case Execution Time (WCET) computation of a particular software. The problem of correctly computing upper/lower bounds of the timing behavior of software is an entire sub-field of research and an extensive body of work already exists on the subject. An overview of various methods and tools can be found in [16]. In the following, only a brief overview of WCET computation and the problems posed by it is given.

The WCET is defined as the maximum time span required by a task to complete its execution on a specific hardware platform. It is typically used in real-time systems, where knowing the WCET of software is fundamental for choosing and realizing a suitable scheduling scheme that ensures correct and timely functional behavior.

The general case for computing a WCET can be mapped to the halting problem and is, therefore, undecidable. Fortunately, most applications for which computing a WCET is required are typically structured in a way that guarantees that they terminate, e.g., by prohibiting the use of recursion and enforcing upper bounds on loop iterations. Broadly, methods of WCET computation can be grouped into two classes: static and measurement-based methods.

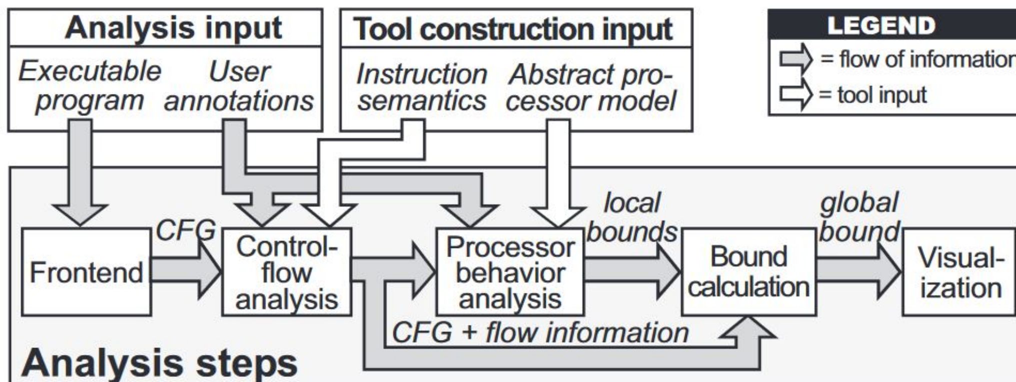


Figure 2.3: Core components of static WCET timing analysis according to [16]

Static methods do not rely on executing code on real hardware or simulating it, but rather combine the task code with an (abstract) model of the system. Bounds are then obtained solely by reasoning on this combination. The necessary core components of such methods are shown in Fig. 2.3. The method proposed later on in Chap. 5 falls in the category of static methods.

Measurement-based methods on the other hand are based on information obtained by observing the timing behavior of the software over a series of executions or simulation runs. This category comes with its own set of advantages, such as no need to formalize the behavior of the executing hardware platform (i.e., creating an abstract model), and disadvantages, e.g., more expensive setups and needing to find suitable input vectors to provoke a WCET response, when compared to static methods.

There are many factors contributing to the complexity in finding the WCET of a program. The ones with the most impact are:

- *Data-Dependent Control Flow*: A program's WCET is determined by the longest possible execution path within it. Usually, this path depends on the right initial conditions of the executing hardware and is only taken if a specific sequence of input values is provided during the execution of

the program. Determining this set of input values is a very hard task and equivalent to solving the WCET problem. Hence, the starting point of most static timing analysis methods is the construction of a CFG. Depending on the size of the analyzed program and the required accuracy this is done either from source code or machine code. A CFG describes a superset of the set of all execution paths, i.e., some paths described by the CFG will never be taken, e.g., due to contradictory consecutive conditions. Furthermore, a CFG may remain incomplete due to dynamic jumps and dynamic calls whose target address is computed during runtime. Eliminating infeasible paths, completing the CFG and determining upper bounds on loops are necessary precursors of a sufficiently precise timing analysis.

- *Context Dependence of Execution Times*: Early approaches to the timing-analysis problem assumed the independence of the timing behavior from prior context, i.e., the execution times in terms of clock cycles for individual instructions were fixed, independent of the execution history, and could usually be found in the processor's specification. However, with the ubiquity of modern architecture features this independence from context is mostly lost. Due to a pipelined execution of instructions in the processor the execution time of individual instructions is a function of possibly all instructions currently processed by the pipeline. Furthermore, different paths taken before the execution of an analyzed code section may lead to different cache states, resulting in different timing due to the cache's hit/miss behavior. Thus, the execution time of a code snippet can depend heavily on the context from which it is executed. Ignoring to consider this context during analysis will result in imprecisely computed timings. Hence, it is necessary to analyze the executing hardware's behavior w.r.t. factors influencing a program's execution time and incorporate them in the abstract model.

- *Timing Anomalies*: Timing anomalies are counter-intuitive effects on the overall execution time caused by timing variations of local sub-task, e.g.,

reducing a sub-task's execution time can actually slow down the task on a global scale. Usually, timing anomalies result from the complexity of the processor stemming from specific architecture features, such as speculative execution, caches and multiple functional units working in parallel.

2.3 Abstraction Layers

The process of abstraction is used pervasively in common sense reasoning and thus, most people have, in all probability, a certain understanding of it. Yet, for the sake of disambiguation and in order to emphasize certain aspects and consequences of the process, a discussion might be necessary. The meaning of the term abstraction varies to some degree, depending on the respective context in which it is utilized. This is mainly determined by the specific field of interest, such as art, mathematics, philosophy or computer science. While the specifics in those might be slightly different, abstraction as a process will further be treated as the action of factoring out unnecessary details, identifying relevant similarities between objects and synthesizing those facts into a concept or class. Such a concept acts then as a super-categorical noun for all subordinate concepts, and connects any related concepts as a group, field, or category. Roughly speaking, one can think of abstraction as the process which allows us to consider what is relevant and to forget many irrelevant details which would get in the way trying to do something.

Ironically, reducing the characteristic of abstraction in this particular way already constitutes an abstraction by itself.

Especially noteworthy in the previous definition are the terms relevant, unnecessary and/or irrelevant. The interpretation of them is the sole guideline of the abstraction process, thus effectively determining its course and outcome. Unfortunately, this interpretation highly depends on the targeted use of the concepts. Therefore, abstraction is a task that can only be performed manually, due to the inability of machines to discern between necessary and negligible information without a

specific, predefined metric.

Particular abstractions have proven to be especially useful during the design and verification of computing systems. The following sub-chapters address those relevant for the understanding of the remainder of this work.

2.3.1 Software

According to [17], the earliest use of the term *software* as we understand it today, goes back to “The Teaching of Concrete Mathematics” [18] from 1958. Therein, software is used to refer to "carefully planned interpretive routines, compilers, and other aspects of automative programming", in contrast to the “hardware of tubes, transistors, wires, tapes, and the like".

Since then, software has become an indispensable layer of abstraction for the processes performed in hardware, without which the complexity of modern tasks and systems could not be managed.

2.3.1.1 Machine Code & Assembly

Machine code, sometimes also called binary code, is a string of bits located in memory and is the most elementary representation of a computer program. By itself, i.e., without the hardware or knowledge about the Instruction Set Architecture (ISA) it was compiled for, machine code carries no intrinsic meaning. The respective ISA defines what sequence of bits form a valid instruction and what each bit or group of bit in this instruction encodes.

While it is possible to write a program directly in machine code, doing so is a tedious and highly error-prone process.

This process is made somewhat more manageable by an ISA’s assembly language, which defines mnemonics for instructions and registers and often allows to define labels in the code. While the mnemonics make the code human-readable and labels

shift some responsibility of computing jump addresses to the compiler, code in assembly is essentially a one-to-one representation of the resulting bit string. There is only a low level of abstraction between the two. Programming in assembly remains, especially in comparison with using high level languages, an arduous task.

Hence, human programmers rarely, if ever, deal directly with machine code anymore, most likely trying to streamline a code section that is especially critical for the performance of the overall system. There is an ongoing debate over the effectiveness of this practice compared to modern compilers.

Nevertheless, binary code (or the more readable assembly) remain an important abstraction layer, as every higher programming language must be translated to it by a compiler or an interpreter prior to execution. This is important in the context of co-verification of hardware/software systems. As established at the beginning of this chapter, an abstract representation of something is stripped of information that is deemed unnecessary for its intended application. High-level languages (see Sec. 2.3.1.2), specify "what" should be done by the processor at the hardware level, but not the "how". The "how" is determined during translation by a compiler and will usually differ between different compilers, their version or the used configuration. Those small differences in program execution at the binary level might trigger different behavior in some part of the hardware, which could not be observed simply by analyzing the high-level code.

2.3.1.2 High-level Languages

A high-level programming language distinguishes itself from a low-level programming language by a strong abstraction from the implementation details of the computer. The underlying idea is to leverage this abstraction to provide constructs, concepts and paradigms which make a program easier to understand. This, in turn, dramatically simplifies the development process of the program when compared to writing it in a low-level languages, i.e., assembly.

Which language qualifies as being a high-level one, depends on the perspective of the user and what it is compared against, because "strong abstraction" is a relative term. For example, one could argue that the C language is not a high-level language, as it does not provide features commonly attributed to one, such as support of object orientation or automatic memory management. While it is not as abstract as its successor/extension C++ or Java, it provides much more abstraction than the previously discussed assembly. Hence, it should be clear that there are different, not clearly separated, levels of "high-level" languages.

For the sake of this thesis, a high-level programming language is any language that may use natural language elements, has a grammar that is defined or at least expressible in extended Backus-Naur form and most importantly has few, if any, language elements that translate directly into a machine's native instructions.

This detachment of high-level source code from actual machine code is one of the major contributors in the productivity gained by using high-level languages, but leads potentially to sub-optimal code or unexpected data movements. This is sometimes known as *abstraction penalty*. To bridge the gap between high-level source code and machine code and reduce the abstraction penalties concepts like abstract machines and compiler optimization are common tools in software development.

2.3.1.3 Software Concepts and Artifacts

Abstract & Concrete Machine To more easily explain the concept of an abstract machine, it is helpful to first understand the term of a real or concrete machine. The term concrete machine is often used to represent the machine code of a given program. Hence, the goal of compilation is to derive an efficient concrete machine from the given high-level source code. An abstract machine is an intermediate concept used in program compilation to bridge the gap between those two levels.

In many cases, e.g., for the languages of C/C++, the abstract machine does not exist.

The machine is literally abstract in the sense of existing in thought only. It is a an imaginary machine which precisely follows the rules of the C language definition standard [19], whereas a concrete machine might have unforeseen semantics, due to features like speculative execution, out-of-order execution and parallelism. The abstract machine serves as reference for the adherence, i.e., correctness, of a concrete machine w.r.t. the standard of the respective high-level language.

Some compilers or compiler frameworks, e.g., LLVM, create an abstract machine as an intermediate step in the compilation process [20]. This abstract machine already resembles the machine code of the program in that it enables a step-by-step execution of the program at the granularity of single instructions and works using a limited set of general purpose registers and a program counter. Yet, the instructions still omit many of the details of the actual hardware and are not particular to any specific ISA [21]. The abstract machine serves as a platform for most hardware-independent optimization performed by the compiler.

Compiler Optimization Software optimization in general is the process of modifying or reorganizing parts of a program in a way that it is more efficient w.r.t. the use of a certain resource. That resource is most often execution time, but also includes a program's memory footprint or energy consumption. Many of these aspects are correlated, i.e., they are not independent from each other. The nature of these correlations is not easily determined. They depend on many facets such as the degree and form of a specific optimization and the underlying hardware system. For example, inline expansion (or simply "inlining"), i.e., explicit insertion of a function's code at the location from where it was called, results generally in faster execution times due to a lower degree of indirection in the code, but increases the program's memory footprint, if the function is also called from elsewhere in the program. Both aspects, in turn, influence the system's energy consumption. The faster execution time might affect it positively, because less instructions need to be executed and there is more potential for power saving schemes. More memory usage usually results in higher energy consumption, but depends largely on the

actual access patterns of the software. The spacial locality principle in combination of a correctly sized cache might actually lead to a positive effect. As a result, it is usually not possible to optimize all aspects of a program at once. Instead, one aspect is given priority.

Optimizations can be classified by their dependency on details of the executing hardware, i.e., as hardware-dependent or hardware-independent, as well as the scope in which the optimization takes place.

The optimization techniques which rely solely on the abstract machine are, per definition, independent of implementation details of the machine targeted by the compiler. Yet, those leave a lot of optimization potential untapped, as many of the most effective optimizations exploit special features of the target platform. For example, using Thumb™mode in many ARM architectures or the compressed instruction format in RISC-V architectures can dramatically reduce code size.

Typical scopes for optimization techniques range from a few adjacent instructions (so called peephole optimizations), over basic blocks, functions, set of functions, up to the whole program.

Memory Barriers A memory barrier is a method of enforcing an ordering of memory accesses by the CPU. It can be realized by a hardware-dependent barrier instruction and is defined by the architecture's memory ordering model and/or by a compiler directive.

A memory barrier usually guarantees that memory operations are performed relative to their position w.r.t. to the memory barrier, i.e., a access issued after the barrier will also be executed after it, and vice versa.

Physical memory barriers, i.e., barrier instructions, are necessary in most modern CPUs that employ out-of-order execution. Reordering of accesses that should go unnoticed from the perspective of a single thread may lead to unexpected behavior

when combined with the concurrent execution of other threads.

Memory barriers in the form of compiler directives are required in order to prevent the compiler from reordering or removing accesses it deems redundant during the compile process. A memory barrier should only be used if necessary, as it is effectively handicapping the compiler in its optimization process. The most common memory barrier directive is realized by the *volatile* keyword in C/C++.

The *volatile* keyword in C/C++ The intent of the volatile keyword was to establish a way for C and C++ programs to directly access memory-mapped I/O.

Correct access to a hardware peripheral via memory-mapped I/O generally requires that the order specified in the source code remains unchanged and none of the I/O operations are omitted. A violation of equivalent ordering might break the communication scheme the peripheral expects; omission, in conjunction with other compiler optimization may lead to infinite loops or dead code. An example is shown in Fig. 2.4. Without declaring *status_register* as volatile, the compiler might either deduce that status_register is not changed in the body of the loop and only check the condition once, or that the loop performs no meaningful action at all and can be removed completely. In both cases the resulting machine code would not implement the behavior intended by the source code.

```
1: int polling_read ( ) {
2:     volatile * int data_register = DATA_ADR;
3:     volatile * int status_register = STATUS_ADR;
4:     while ( * status_register == 0) {
5:             - - do nothing
6:     };
7:     return * data_register;
8: }
```

Figure 2.4: Polling read from a hardware peripheral. The CPU will stay in the while-loop until the peripheral is ready.

Hence, a compliant compiler will create, for every access from or to a volatile

variable by the abstract machine, an equivalent access from or to the memory address corresponding to that variable in the concrete machine.

Note that volatile should not be used for inter-thread communication, as it does not enforce cache consistency. As specified in the C standard [19], it should only be used to represent memory-mapped I/O.

2.3.2 Hardware

Electronic hardware consists of one or more interconnected components performing logic operations, often complemented by analog circuits, e.g., to provide a stable power supply or drive an mechanical actuator. The term encompasses everything from individual chips and circuits to distributed information processing systems.

This work is restricted in its scope to synchronous digital circuits, more specifically those processing and interacting with low-level software. Furthermore, the terms *design* and *design process* are used to reflect only the part of the overall fabrication process that is concerned with specifying the logical behavior of the chip. For actual fabrication, many subsequent steps, e.g., place-and-route or packaging, are required.

Designing modern electronics is a highly complicated task, only manageable by employing hierarchical architectures and relying on tool support for established abstraction layers.

The layer with the least abstraction in which a digital circuit is generally represented is called the *transistor level*. On this level, the detailed, non-linear electrical behavior of a transistor is replaced with a simple switching behavior. The transistor is either 'on' or 'off'. The transistor level is the target layer for the design of integrated circuits (ICs).

The next higher abstraction layer is the *gate level*. It uses logic gates that implement common operators of Boolean logic, such as AND, OR and NOT. Hence, it is easy

to establish a direct mapping between a formula in Boolean logic and a circuit on the gate level, and the circuit can be optimized by applying boolean transformations. The gate level is the target layer for designs intended to be instantiated on a Field Programmable Gate Array (FPGA), i.e., no synthesis to the transistor level is required.

Both abstraction layers so far are significantly more abstract than the layer below them. Yet, the abstraction is provided is not sufficient to handle modern designs. The layers explained in the following – the Register Transfer Level (RTL) and the Electronic System Level (ESL) – try to address this problem.

2.3.2.1 Register Transfer Level

The RTL serves as the current entry point of the design process, i.e., it is typical practice to start a modern digital design at this level. It can be compared to what a high-level language like C is to assembly.

It is important to note that a circuit is, in contrast to a software program, a hardwired network of physical components. Hence, its functional behavior is based on synchronous logic and well defined as a Finite State Machine (FSM) by three key elements:

- registers, which refer to a set of bit vectors encoding the state information;
- registers transfers, which define transformations and movement of data between registers in an imperative programming language, resulting in combinatorial logic;
- primary inputs, such as data and clocks which control and trigger the register transfers, and primary outputs, whose value is determined by a function of the registers and primary inputs.

The behavior of a design is defined by the flow of data between hardware registers.

The special requirements of hardware design, especially the inherent concurrency of hardware operations, led to the development of dedicated hardware description languages (HDLs) like Verilog and VHDL. Analogous to the transformation of a high-level source code to assembly by a compiler, high-level design descriptions are transformed into gate level descriptions by the process of hardware synthesis.

However, these Hardware Desciption Languages (HDLs) did not start out with the intention to serve as a design language, but were merely intended for modeling and simulation. Even today, the result of this repurposing are language semantics that are not particularly well suited for synthesis or formal verification, leading to unnecessary complications for the respective tools and users. Nevertheless, until today, VHDL and Verilog are still the standard languages for RTL design entry.

2.3.2.2 Electronic System Level

The challenges imposed by modern circuit design keep growing, up to the point where even the abstraction provided by the RTL is not sufficient anymore to compensate for the complexity of these challenges. The rate in which design complexity increases outpaces our current efforts to increase design productivity. A sensible solution to this might lie in further increasing the abstraction at which designers operate when describing the design.

The ESL aims to enable that by establishing a new abstraction layer above the RTL. One roadblock on that path is the lack of a common, universally agreed definition what exactly constitutes the ESL. This is in stark contrast to the already existing abstraction levels.

[22] provides the following informal definition:

> "[The ESL is defined by the] utilization of appropriate abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a

cost-effective manner, while meeting necessary constraints."

While this definition conveys the intention of the ESL it is still too vague to serve as a valid entry point of hardware design. It is up to the designer to define what constitutes "appropriate abstraction" for his given task. This makes the development of a universally applicable description language for ESL models complicated. As a consequence, the actual semantics expressed by the various types of descriptions at the ESL vary significantly depending on different design philosophies and which aspect of the design is given emphasis — at higher levels of abstraction the descriptions and design methods are, to a large degree, domain-specific.

A major factor impeding widespread application of the ESL for design and verification is the *semantic gap*.

Semantic Gap A abstraction is *sound* w.r.t. something concrete if and only if a well defined formal relationship between the abstraction and the concrete exists. A new abstraction level can only then be used for design and verification when there is a chain of sound abstractions from the new level all the way down to the physical representation of the design. Otherwise, the semantics of the abstract model cannot be understood in terms of the physical circuit. The benefit of establishing such a sound abstraction level is that the verification result of one abstraction level can be leveraged to establish confidence in a model of the same design at the next higher abstraction level. Details verified for a circuit by analysis at an abstraction level should therefore not have to be verified again at the higher abstraction level. The transistor level is a sound abstraction of the physical level due to a defined approximation of the electrical behavior of the transistor, and the gate level, in turn, is sound to the transistor level via Boolean logic. Finally, a formal equivalence check ensures the correctness of the RTL w.r.t. the gate level and thereby with the transistor level.

However, the ESL has no such well defined relationship with the RTL, because it

is, as mentioned before, not clearly defined itself. This breaks the chain of trust, built through the other abstraction levels, which is the reason why verification of RTL models can, with current methods, not be used to establish confidence in ESL models, or vice versa.

This problem is well known and often referred to as the *semantic gap* between the system level and the RT level. Note that the semantic gap does not simply result from a lack of standardization of ESL descriptions, but rather is a theoretical problem of describing a sound relationship between RTL descriptions and profoundly different descriptions which are neither cycle- nor bit-accurate.

Nevertheless, high-level descriptions at the ESL are of great value when the overall functionality of larger systems should be specified and understood. Hence, ESL models are widely adopted in the industry, but their use is currently restricted to design elaboration and virtual prototyping.

A theoretical foundation attempting to bridge the semantic gap was proposed in [23]. It is discussed in more detail in 3.5.3.

Chapter 3

Formal HW/SW-Co-Verification

In the following chapter, a formal approach for HW/SW-co-verification and the involved models are presented. The goal of the proposed approach is to formally verify that two software programs (e.g., an original program and an optimized version) trigger the same behavior in a set of peripheral devices so that the I/O behavior of the combined HW/SW system is identical for the two software versions. The two program variants, by themselves, are not identical in their I/O behavior. Due to the memory barriers in the code associated with the volatile keyword, the optimizations are not performed by the compiler, but come as the result of a manual effort or the use of a specialized optimization tool [43]. In the analysis, the peripheral devices are assumed to be functionally independent of each other, i.e., there exist no hardware side effects of one device on another. This assumption is usually fulfilled in practice.

The proposed approach intends to prove the equivalence of two program variants accessing a peripheral device via a form of equivalence checking based on IPC (see Sec. 2.1.3.2). As usual in equivalence checking, we do not show the correctness of the program variants themselves. Hence, it is assumed that the original variant is already deemed correct w.r.t. the intended functionality and can serve as the

golden model on which the correctness of modifications can be evaluated. Such modifications may comprise any change to internal computation as well as locally restricted changes to accesses of the addressed peripheral, such as their reordering and/or unification.

This chapter formalizes the details and definitions of the required models, followed by experimental evaluation and discussion of the possible configurations.

3.1 Related Work

While there is a plethora of formal verification techniques that separately verify HW or software, only relatively few works consider software at a hardware-dependent level or even attempt a co-verification.

Many of of the approaches that do exist [24, 25], are rooted in and employ languages based on higher order logic. Besides the often substantial manual labor required for the creation of the necessary formal models using such languages requires advanced expertise that is often not available in standard design environments.

In [26] symbolic execution is employed to semi-automatically search for bugs in the hardware/software interface. The method requires to derive a software abstraction of the hardware from the actual implementation or the respective specification. This necessitates further verification w.r.t. to the derived abstraction. [27] presents a hardware/software co-verification tool for designs written in C/C++ and Verilog RTL, along with a property specification framework. There, communication between hardware and software is formalized using transactions to reduce the number of possible interleavings. While certainly effective, these approaches require formulating a set of properties to be matched against a specification. This involves, again, substantial manual effort and requires the verification engineer to have a deep understanding of the correct behavior of the co-design.

In [12] an approach is presented to formally prove the equivalence of two low-level hardware-dependent programs. This approach models the reactive behavior of the software with the hardware and checks if two programs exhibit the same access sequences at the hardware/software interface. However, the hardware representation of the system is restricted to the executing processor core, i.e., the peripheral itself is not taken into account. This disqualifies the approach for the verification of modifications which change the I/O behavior of the software. As motivated in the beginning of this work, this is an important class of optimizations for tightly resource-constrained systems. Similarly, the software equivalence checking approach of [28] as well as the hardware-dependent property checking approaches of [25, 29, 30] consider only the software and are hence not suited to verify this class of modifications, either.

One possible solution to address the limitations of HDLs is to establish the ESL as a reliable design entry point. This requires to first bridge the semantic gap (see 2.3.2.2), a task attempted by Path Predicate Abstraction (PPA). The underlying theory is presented in Sec. 3.5.2 and was first introduced in [31]. It is inspired by the advances in theorem proving and the use of *refinement maps* [32, 33] to model complex relationships between abstract and concrete models in hardware design such as [34, 35, 36, 37, 38].

Notable success was obtained in the hardware domain by the notion of "bisimulation modulo stuttering", as demonstrated in [39] for hardware verification. The difference between bisimulation modulo stuttering and PPA was already discussed in [40].

It should be noted that other methods are loosely related to PPA and the related methodologies. For example, in [41, 42] high-level simulation patterns or test cases are used to derive properties for implementation verification. In contrast to PPA and PPA-based flows, these approaches do not aim at establishing a formal relationship between the system-level model and the RTL implementation, but

instead to maintain coherence between the test cases at the different abstraction levels.

As a consequence, verification results gained using such an ESL model are still not directly transferable to the RTL implementation and therefore do not fulfill the objectives of this thesis.

3.2 Verification Model

The overall verification task at hand is showing that a peripheral controlled by a software variant A exhibits equivalent I/O behavior to a peripheral controlled by a software variant B, assuming A and B are supposed to have equivalent functionality.

A peripheral device communicates with the software through its register interface. In addition, it has inputs and outputs to the environment for providing its service. The notion of *sequential hardware equivalence* compares the inputs and outputs of two digital circuits clock cycle by clock cycle. In most practical cases, this notion is too strict for our problem. The overall I/O behavior of the HW/SW system may vary regarding the timing at the clock cycle scale. If the inputs and outputs of the peripheral device under the control of two versions of a software are functionally equivalent but differ only in timing the common notion of *HW/SW co-equivalence modulo latency* can be applied.

This notion of equivalence leads to prohibitively complex proof problems for the types of systems considered here, when attempting the verification over the full run time of the software. In many practical scenarios, however, the proof problem can be decomposed into manageable subproblems As already elaborated in the motivation of this thesis (see Sec. 1.2), the optimization of the firmware usually changes the program behavior only locally. This is particularly true for the optimizations that target the program memory footprint by increasing code reuse

in the HW/SW interfaces of IoT platforms.

In our approach, we consider each peripheral device individually. We partition both the original software and the revised software (with the optimizations) into *segments*. The segmentation is chosen such that corresponding software segments induce the same behavior in the peripheral device, i.e., corresponding segments are expected to be functionally equivalent w.r.t. to this device.

Such a segmentation is possible if the optimizations are local and can be identified and contained in a software segment. For example, a sequence of device configuration steps may have been replaced by a different sequence after introducing a new API function, however, without changing the effect on the device hardware. Such a sequence would be placed in one segment.

The segmentation of the software allows to decouple individual activities of the peripheral device temporally from each other, reducing the overall complexity of the verification task.

The proposed verification model for one such decoupled activity is illustrated in Fig. 3.1. It consists of three distinct sub-models:

- A **software** model approximating the behavior exhibited by a processor executing the software under verification. Its purpose is to provide stimuli and constraints to the hardware model.
- A **hardware** model representing the peripheral controlled by the software. Not quite intuitively, the equivalence of software variants will be shown on the hardware model. The specifics on how this is done depend on the abstraction of the chosen hardware model.
- A model of the **interconnect** between the processor and the peripheral. It translates the stimuli/constraints provided by the software model to a format that can be processed by the respective hardware model.

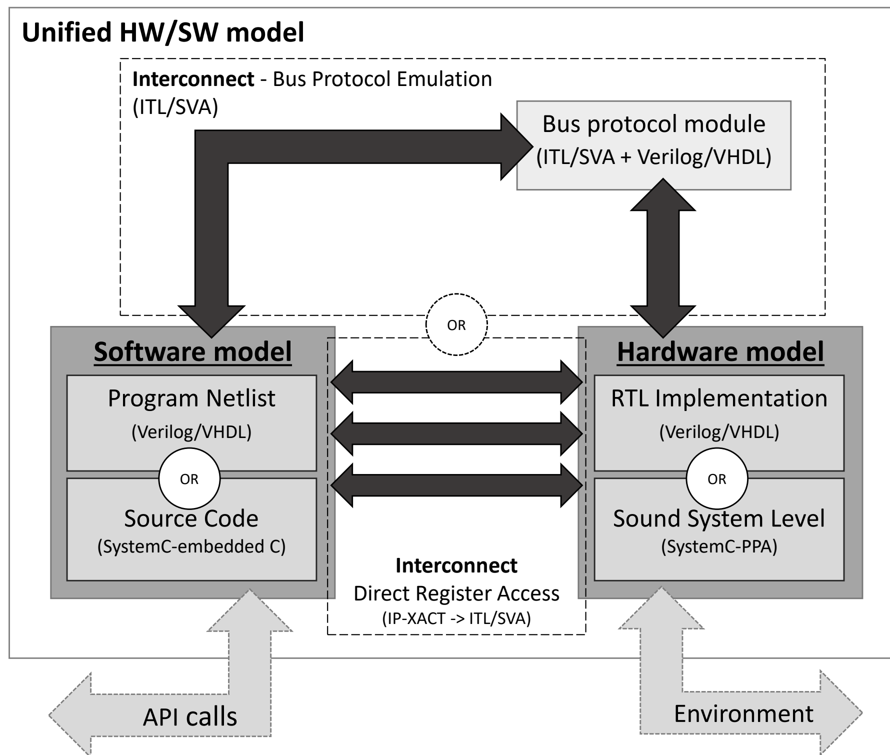


Figure 3.1: Unified HW/SW verification model

Each of these can be modeled on different levels of abstraction. For each component, the chosen abstraction level affects the interactions with the other models and depends on the specific goal of the verification task.

3.3 ACCESS Verification Method

The different variants for each sub-model (software, hardware, interconnect) may use different notions of time. While the timing for each software segment might be modeled with clock cycle accuracy, the time intervals between the segments are modeled abstractly. Hence, our notion of equivalence is *HW/SW co-equivalence modulo latency*, however, *per software segment* and under consideration of the global software context. The exact equivalence condition is provided in the section of the respective model.

Once we have partitioned the software into segments we can build a computational model and formulate an equivalence checking problem for this model.

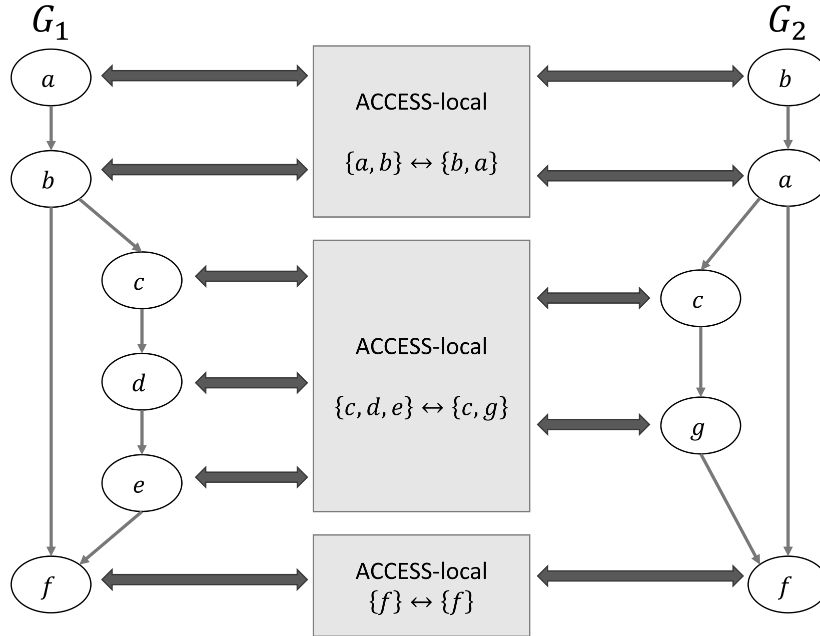
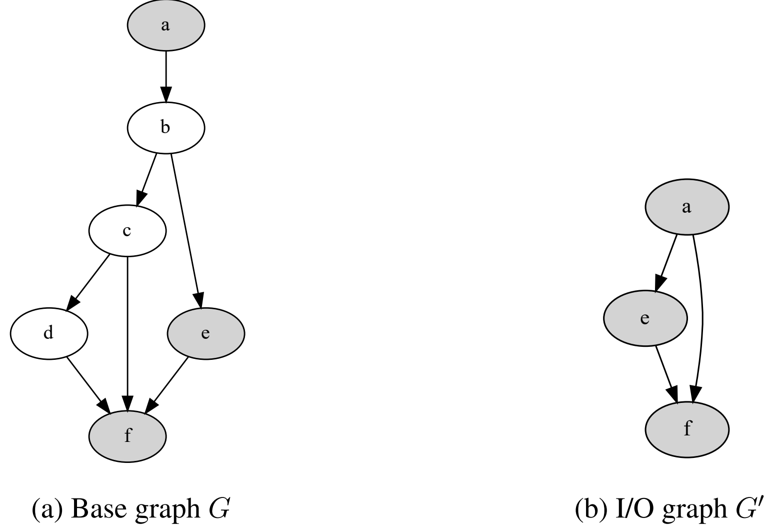


Figure 3.2: ACCESS-global : HW/SW miter. Localized proofs on the hardware constraints by the I/O graph of the software model

Fig. 3.2 shows the overall computational model, which is constructed from several ACCESS-local proofs, i.e., local unrollings over time of the unified HW/SW verification model which try to prove only the equivalence of a part of the program. ACCESS-local proofs, amongst other things, will be presented in more detail below. In the figure, the software models of the original and the modified firmware are each represented by an I/O graph (G_1 and G_2, respectively).

Each vertex in G_1 and G_2 represents an I/O access of the software to an examined peripheral P. Any edge (v, u) in one of the graphs indicates that there exists at least one path from v to u in the software. In case that software contains multiple paths leading from v to u, all of them are represented by a single edge in the I/O graph. The actual sequence of possible transactions represented by this edge is still contained in the software model, but not shown/relevant for the concept of



(a) Base graph G (b) I/O graph G'

Figure 3.3: A graph representing a software's execution and corresponding I/O graph. Gray nodes represent Load/Store accesses to the peripheral P.

ACCESS-global . A simple example of this is illustrated in Fig. 3.3.

Formally an I/O graph is defined as follows:

Definition 2 (I/O graph)**.** *Assume a graph $G = (V, E)$, a peripheral P and an addressing function α. A vertex $v \in V$ is a load/store to P iff $\alpha(v, P) = 1$. A graph $G' = (V', E')$ is called* I/O graph of G to P *iff* $(\forall v \in V' \subseteq V : \alpha(v, P) = 1) \wedge (\forall (v, u) \in E' : \exists \{(v, v_1), (v_1, v_2), ..., (v_{n-1}, u)\} \subseteq E)$

The hardware model of the peripheral is unrolled for a given set of I/O accesses specified by the I/O graph until all accesses in the set are modeled and either the equivalence condition of the hardware model or an upper limit of additional cycles has been reached.

As mentioned in Sec. 3.2, trying to capture the whole behavior of the software in a single, monolithic proof instance quickly results in an infeasible model that cannot be solved in an acceptable time, if at all. To mitigate this escalating proof

complexity, this work proposes partitioning the overall proof into a number of smaller proof instances, each showing the equivalence of a subset of accesses.

We assume that the peripheral device and its respective hardware models are deterministic. Hence, it is not necessary to wait until the complete reaction of the peripheral to a series of accesses is resolved in order to determine whether two access sequences are equivalent. Instead, it is sufficient to show that both sequences induce an equivalent state in the peripheral. This assumes that the peripherals operate under equivalent basic conditions, e.g., identical incoming transmissions/requests from the environment. The setup verifying this for a given sequence is called *ACCESS-local*.

ACCESS-local contains two instances of the peripheral (as in a classical equivalence checking miter) whose outputs are compared. Each instance of the peripheral is controlled by device accesses from the respective I/O graph. The software models behind both I/O graphs G_1 and G_2 are both connected to the same primary inputs, i.e., they model the original and the revised firmware under the same initial memory content and under the same inputs received from other sources (OS, I/O). For the sake of clearer representation and focus, this constraint is not shown in Fig. 3.2. Both I/O graphs are partitioned and the partitions mapped in a way that assumes that the mapped access sequences are equivalent. For every such pair an ACCESS-local module is created. For each ACCESS-local module, an equivalence check is carried out comparing the two instances of the peripheral under the influence of the original and revised firmware, respectively. For the example in Fig. 3.2, the computational model is constructed for three equivalence checks, each comparing a pair of software segments with regard to the peripheral behavior. The first one checks whether the access sequence (a, b) in G_1 generates the same behavior in the peripheral as the access sequence (b, a) in G_2. The second check evaluates whether sequence (c, d, e) in G_1 produces the same effects in the peripheral as sequence (e, g) in G_2. The last check seems trivial: it compares the instruction sequence (f) in G_1 with (f) in G_2. Even though the accesses (f)

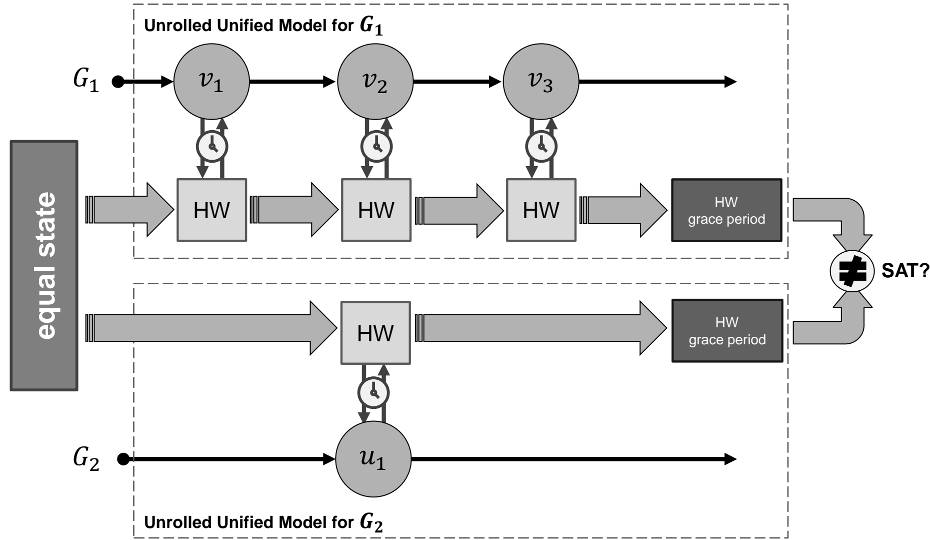


Figure 3.4: ACCESS-local: device miter.

themselves may be identical, the values written depend on the computations of the software model represented by the I/O graph and may be different.

Any difference will be detected because the computational model for an ACCESS-local check includes the full prior execution history (or an over-approximation of it), not just the compared segments, even though only the accesses from a segment are connected to an ACCESS-local module.

Fig. 3.4 illustrates the internals of an ACCESS-local module. Two instances of the peripheral are unrolled for a finite number of time frames. The unrolling begins at an arbitrary state which is, however, identical for both instances. Each instance is controlled by an access sequence from the respective software: one from the original, here called G_1, the other one from the modified variant, here called G_2, as illustrated in the figure. The minimum number of time frames needed depends largely on the length of the examined sequences, but is also influenced by the chosen abstraction level of the hardware model. The maximum number of cycles is determined by a set of user-specified parameters. The outputs of the peripheral device as well as the *ending state* reached at the end of the time frame expansion

are compared for equality.

What does this equivalence check verify? It checks that the behavior of the peripheral device observable from outside of the HW/SW system is the same for both, the segment of the original and the segment of the revised software, modulo timing variations. The proof is conservative because it considers the device behavior from an arbitrary starting state, i.e., the proof includes the state which the device is in when the respective software segment begins execution.

Theorem 1. *If all ACCESS-local equivalence checks (one for each pair of mapped software segments) are successful then the peripheral device behavior is the same for the golden and the revised firmware as a whole.*

Proof. By induction.

Step: ACCESS-local proves the following: If a peripheral device starts from the same state at the beginning of the revised software segment as it does at the beginning of the original software segment, then, after either segment, it arrives in the same ending state.

Base: The peripheral device is in the same state (the *reset* state) at the beginning of the first segment of either software.

□

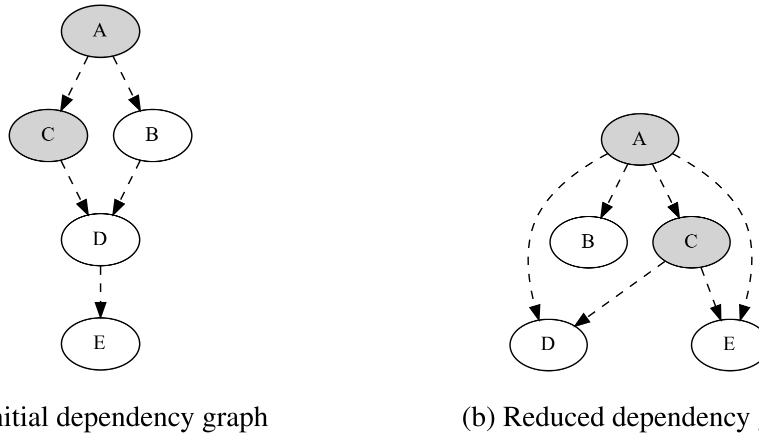
Unfortunately, a completely arbitrary starting state may result in false negatives because one (or both) of the software variants may operate under the (correct) assumption that the peripheral was previously configured in a certain way. The equivalence of two access sequences may hinge on such an assumption. As a result, it might be necessary to propagate certain information from one local proof to another. Only registers whose values may only change by request of the software are eligible for this. Independent of their function in the peripheral, we refer to these further on as *configuration registers*.

An initial dependency graph of this flow of information can be easily extracted from the I/O graphs of the software. In Fig. 3.2, possible information dependencies between instances of ACCESS-local are indicated by a dashed arrows. The emerging information dependency graph imposes an order in which the respective ACCESS-local instances should be processed. Potentially, this can impede the application of parallel computation in the overall verification task, which in turn can be mitigated by reducing the depth of the information dependency graph.

For most peripherals and their respective software drivers, only a minority of accesses target configuration registers. Most likely, these are clustered at the beginning of the program's execution. In the initial dependency graph which is structurally derived from the program, this information is propagated via the transitive property of configuration registers: If a configuration register's value was not changed by the access sequence of a proof instance, the value remains the same for successor proofs in the dependency graph. By backtracking such a path of transitive constraints to proof instances whose access sequence targets a configuration register we can replace the path by a single edge in the dependency graph. Fig. 3.5a conceptually shows a structurally derived dependency graph and Fig. 3.5b the corresponding reduced graph. In the example provided by the figure, in the initial graph the only parallel executable proof instances are B and C. After removing unnecessary constraints, D and E can start computing as soon as the result for C is available.

3.4 Software Model

The software is modeled either on the binary level or on the source code level. The binary model is created by extracting a CFG from the binary of the program and creating a PN (see Sec. 2.1.4). To use a source code model in the unified verification model, the modeled program needs to be written in C/C++. Furthermore, the program's code requires some instrumentation and must be embedded in a SystemC



(a) Initial dependency graph (b) Reduced dependency graph

Figure 3.5: Flattening of the information dependency graph to enable parallel computation of proof instances.

wrapper.

The following definitions are of importance for the verification method, independent of the chosen level of abstraction:

Definition 3 (Value)**.** *A value is the precise meaning of the contents of an object when interpreted as having a specific type [19].* □

For example, a 32-bit object $0x00000001$ interpreted as an integer has the value 1. The same object interpreted as bool yields the value *true*.

Definition 4 (Expression)**.** *An expression is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof [19].*

Definition 5 (Statement)**.** *A statement is a syntactic unit of an imperative programming language L expressing some action to be carried out. A statement may contain internal components, such as expressions or other statements.*

Definition 6 (Program)**.** *A program $P^L(S, E)$ is a collection of one or more*

statements $s \in S$ of language L and $(s_1, s_2) \in E \iff s_2$ is a syntactically valid successor of s_1 in L.

Definition 7 (Code Segment)**.** *A* code segment *is a subset $W \subseteq S$ of statements, $W = \{s_0, s_1, \ldots, s_n\}$, lying on a path from the beginning statement, s_0, to the end statement, s_n, i.e., it is $(s_i, s_{i+1}) \in E$ for $0 \leq i < n$.*

Definition 8 (Access Sequence)**.** *An* access sequence *α of a code segment W is a sequence of accesses $(z_1, z_2, \ldots)$ to the peripheral device registers, corresponding to load/store instructions in W. Each access z_i has a type ("read" or "write"), and an address representing a device register.*

Definition 9 (Access Cover)**.** *A set, $C = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$, of access sequences in a program $P(S, E)$ such that $\bigcup_{i=1}^{m} \alpha_i$ contains every access in P, is called a* cover *of P.*

For equivalence checking, we model both, the original firmware P_1 and the revised firmware P_2, by a decomposition of the *same number m* of access sequences such that we obtain a access cover C_1 for P_1 and a cover C_2 for P_2. The decomposition is chosen such that there is a one-to-one mapping between the elements of C_1 and C_2.

Definition 10 (Sequence Mapping)**.** *Given two code covers, C_1 and C_2, with $|C_1| = |C_2| = m$, a bijection $M : C_1 \mapsto C_2$ is called* sequence mapping.

For every access sequence $\alpha_{1,i}$ of the original program there is a corresponding access sequence $\alpha_{2,i}$ of the revised program. The sequences and a mapping M are defined such that corresponding sequences of the original and the revised software are expected to *induce the same behavior of the peripheral device*.

Sequence mapping according to Def. 10 is mainly a manual task, and is usually straightforward, particularly when the firmware has been optimized by manual steps as well.

Accesses generated by load/store instructions in the software produce read/write transactions at the peripheral device registers. The translation of an access sequence to a format that can be correctly processed by the respective hardware model (see Sec. 3.5) is handled by the interconnect (see Sec. 3.6).

3.4.1 Binary Level – Program Netlist

At the binary level we use PNs as described in Sec. 2.1.4 to model the behavior of the software and possible interactions with the targeted device registers of the hardware model. A PN represents all execution paths between an entry and an exit point in the software. It models the software behavior at the ISA level, including load/store accesses to the peripheral device registers.

A PN is a combinational circuit representing the execution of a sequential circuit, i.e., a processor executing a program, over time. In its base form, it does not consider the time required for performing an execution step, but operates on logical time, i.e., it only captures the ordering of events, without any established relationship to the physical duration between these events. The order of execution is implicitly modeled by the placement of a node (i.e., instruction) in the EXG. If required, the PN can be extended to be cycle-accurate with further analysis, as presented in Chap. 5.

The concept represents software inherently as hardware in a language of choice (VHDL, Verilog, etc.). Hence, integrating it with the hardware model is straightforward.

Due to the unrolling of the CFG into an EXG, each segment is completely contextualized, e.g., the exact iteration of unrolling is known. The low level of abstraction w.r.t. the actual execution of a program running on a processor allows to catch possible effects which might only occur due to characteristics specific to a given ISA.

Only this low level of abstraction enables the analysis of post-compiler optimizations. As a drawback, the size of the analyzed program is limited by the ability to create the respective PN.

As mentioned earlier, a key element of our approach is to partition the overall verification task into smaller, manageable subtasks. Although the program netlist represents all possible executions of the software from entry to exit, we do not consider all accesses to the peripheral along full execution paths. Instead, we decompose the sequence of accesses along an execution path into segments that are considered individually. This is possible due to the local nature of firmware optimizations applied in industrial practice during platform customization and ensures the scalability of the approach.

3.4.2 Source Code Level – C/C++

The proposed approaches utilize proof techniques and tools initially developed solely for the purpose of verifying hardware. Hence, integrating the software of a mixed HW/SW system requires a way to model the software in some compatible manner.

True to its name, the software model at the source code level does this before any compilation, i.e., translation into machine instructions, is performed. This is why the source code model is not able to take the specifics of the ISA of the target platform into account. It is an ISA-agnostic, pure representation of the software behavior as specified by the abstract machine (see Sec. 2.3.1.2).

As a consequence, proofs based on the source code model are unable to capture bugs caused by interactions specific to instructions (or instruction sequences) of a given ISA. Furthermore, the model is unable to represent certain types of optimization, such as peephole optimization.

These are typical shortcomings of a model of a higher level of abstraction; after

all, some details must be dropped in order to have significant abstraction. On the other hand it naturally has the usual advantages of a model of higher abstraction, namely:

- Better knowledge/understanding of the overall control and data flow. It is much easier to extract a complete CFG at the source code level based on function calls and returns than it is at the binary level where the target address of an indirect jump must be computed.
- Operating on variables instead of bit vectors.
- Independence of the restricted capabilities of the hardware platform. Some arithmetic computations are more complex and/or more expensive to implement in hardware than others. Most notable are multiplication and division. If the platform lacks a dedicated functional unit to perform these operations the same result must be achieved by replacing the operation with an equivalent sequence of other computations. For example, multiplication can be replaced by repeated shifting and addition. Division is implemented as either fast (e.g., Goldschmidt division) or slow division (e.g., SRT division), which replace the division with series of multiplications or subtractions, respectively. Hence, multiplications and divisions will produce complex behavior at at the binary level. This significantly increases the effort required by a proof engine when handling this model.

If the trade-off between lost specificity w.r.t. the software behavior and the computational load on the proof engine is justifiable (or necessary) depends on the nature of the targeted application.

In the scope of this thesis only source code in the language of C/C++ (with some restrictions regarding allowed language features) is considered. This is due to practical reasons, not due to some theoretical limitation. Any high-level language meeting the necessary requirements could be used to provide a source code model,

given proper tool support. Given the predominance of the C language for implementing low-level software directly interacting with hardware, e.g., drivers, we see no immediate need to extend the practical aspects of this chapter to other languages.

To integrate source code of a driver into the unified model a hardware representation implementing the behavior of the drivers abstract machine is derived. For this task the source code of the driver is embedded in a wrapper module written in SystemC. This allows to contextualize the software's source code as hardware with the help of commercial tools, e.g., Onespin 360 DV-Verify™ [8].

In the unified model, due to the abstraction of the interconnect between the software and hardware sub-models, there is no need for the two to use the same notion of time. By being ISA-agnostic, the source code model will hardly provide a cycle-accurate timing. When synthesizing the computational model of the target software, its behavior is defined by a set operation in between synchronization points. The synchronization points correspond to accesses of memory mapped I/O by the software. The operations are derived from the possible execution paths between synchronization points. In the model they could be represented by synthesizing them to combinational circuits, transformed into Boolean formulas or other representations of a function, depending on the internal workings of the utilized tool. In any case, all execution paths need to be of finite length.

3.5 Hardware Model

In the scope of this thesis the model of the hardware is either represented by an RTL or a SystemC-PPA ESL description.

Definition 11 (Access Pattern)**.** *An* access pattern *is a sequence of sets of logic values produced at the peripheral device register interface as a result of an access sequence.*

An access pattern is the RTL view of an access sequence of Def. 8, which is the programmer's view of the communication with the device. At the RTL, an access pattern is a clock cycle-accurate "waveform" of read and write transactions at the register interface. At the ESL, it is a series of messages. The individual RTL transactions or ESL messages may vary in timing and may be separated by "idle" periods, i.e., clock cycles of inactivity at the register interface.

A single access sequence therefore corresponds to a multitude of possible access patterns.

3.5.1 Register Transfer Level

The general concept of a hardware RTL has already been presented in Sec. 2.3.2.1. Hence, only those aspects immediately relevant for understanding the proposed verification approach or those playing a major role in distinguishing it from an ESL model are elaborated in this section.

The RTL model is a clock cycle accurate representation of the final hardware circuit. This often results in complex implementations which provide not only the logic necessary for implementing the desired functionality, but also include (intended) logic redundancy, safety mechanisms and considerations for physical problems (e.g., oversampling inputs to mitigate noise on an unstable communication line, using multiple synchronization registers per data bit to avoid metastability).

As a result, the RTL model has a comparatively large state space and/or sequential depth. Together they pose a significant load for the proof engine which quickly becomes prohibitive when trying to verify the effects of an input sequence. For example, a single write access to a Universal Asynchronous Receive Transmitter (UART) peripheral by the CPU can trigger the transmission of data that can last hundreds or thousands of clock cycles.

Hence, in order to include a hardware model at the RTL and still have a feasible

unified model we need to reduce the number of cycles necessary to unroll.

Fortunately, hardware peripherals are deterministic. Hence, assuming equivalent input sequences from the environment, two instances of the peripheral will exhibit equivalent behavior as soon as they are in equivalent states (and will be in the future). This enables us to limit the proof to the time window from the start of one sequence up to the point in time such an equivalent state or a predefined number of cycles is reached. This leads directly to the second point:

A premise of the presented verification approach is the assumption that the optimizations and modifications to the HW/SW system are rooted in the access behavior of the software to the peripheral. The hardware model of the peripheral in each instance of the combined HW/SW system remains largely unchanged. Differences between the instances come only from changes to the addressing scheme, regrouping of registers that allows for a direct one-to-one mapping and removal of functionally redundant registers.

Definition 12 (Register Set)**.** *Assume R_Φ is the set of all registers in a peripheral Φ. $R \subseteq R_\Phi$ is called a register set.*

The following analogy tries to convey the basic idea behind partitioning the set of registers of a design into several register sets: Imagine the RTL design as mechanical contraption. The register sets form the gears, pulleys and levers inside the apparatus. The value of a register set represents the position of a lever or how much a gear is rotated from its base position. To determine whether two such contraptions will perform the same operations in the future we need to verify that the levers controlling the behavior are in the same position. For the gears, however, it is often irrelevant how much they are set off from the base position, as long as we can show that the gears turn in the right direction.

Definition 13 (Register Value)**.** *Consider a register r and a timepoint t. The value of r at t is defined as $r(t) : r \rightarrow \{0,1\}^{|r|}$. Given a register set $R = \{r_0, r_1, ..., r_n\}$*

the value of the set is defined as the set of respective assignments $R(t) : R \to \{r_0(t), r_1(t), ..., r_n, (t)\}$

Def. 13 serves mainly to introduce a notation to differentiate between the value of a register and the register itself, as well as to emphasize its variability over time.

Definition 14 (Register Mapping)**.** *A bijection* $M : R \mapsto \hat{R}$ *between two register sets* R *and* $\hat{R}$ *is called a register mapping.*

A register mapping ensures that every register in a given set has a mapped counterpart in the corresponding set of the modified system. Usually, finding such a mapping is trivial: The registers of the peripheral models should not have changed and can be mapped solely based on their name.

Definition 15 (Peripheral Cover)**.** *A set* $C = \{R_1, R_2, ..., R_m\}$ *of register sets of a peripheral* Φ *such that* $\bigcup_{i=1}^{m} R_i = R_\Phi$*, is called a* cover *of* Φ.

A peripheral cover ensures that every register of a peripheral is grouped in a register set.

Definition 16 (Peripheral Mapping)**.** *A bijection* $M : C \mapsto \hat{C}$ *between two peripheral covers* C *and* $\hat{C}$ *is called a peripheral mapping, if for each register set in the covers a register map exists.*

A peripheral mapping ensures that every register of a peripheral is actually part of a register mapping.

Definition 17 (Equivalent System State)**.** *Assume a peripheral mapping* $M : C \mapsto \hat{C}$*, with* $C = \{R_1, R_2, ..., R_m\}$*,* $\hat{C} = \{\hat{R}_1, \hat{R}_2, ..., \hat{R}_m\}$*. Furthermore, let* $I = [t, t_{max}]$ *be an interval of timepoints. An equivalent system state* Σ *is reached if* $\forall(R, \hat{R}) \in M : \exists(\tau, \hat{\tau}) \in I \times I : R(\tau) = \hat{R}(\hat{\tau})$ *.*

An equivalent system state of a peripheral mapping is reached if for each of its register mappings a pair of timepoints $(\tau, \hat{\tau})$ exists such that both register sets have

the same value in a given interval $[t, t_{max}]$.

Grace Period. The ACCESS-local approach verifies whether two different access sequences can drive a peripheral device into the same ending state, modulo differences in timing. The computational model must accommodate for a number of clock cycles of hardware action in the device after the last access in a sequence has occurred, before the ending state has been reached. We call this the "grace period" of the equivalence check.

The trivial solution for finding a peripheral mapping is having only a single register mapping that contains every register of the design. In the context of the analogy, this would mean that our mechanical machine would contain only a single large gear wheel. For many designs this is a feasible choice. For others, especially those with a lot of internal counters, it can result in an extreme delay of cause and effect in the hardware, beyond the acceptable range for unrolling.

Hence, the first refinement of the peripheral mapping of such a peripheral is to identify counters and place them in their own register map.

A further refinement is to make a distinction between volatile and configuration registers. In the context of this thesis, a *configuration register* is a register of a peripheral that can only be changed by a write access from the CPU / the system bus. Every other register, i.e., registers which can change their value due to events in the environment or internal processes are called *volatile registers* (based on the similar concept in software).

3.5.1.1 Device Check

If an ACCESS-local check is successful the compared software segments are guaranteed to be equivalent, within the global SW context, w.r.t. device behavior. If a firmware optimization is not correct then the ACCESS-local check is guaranteed to fail for the corresponding segments.

However, the opposite is not true, i.e., if an ACCESS-local check fails this doesn't necessarily mean that the compared SW segments are not equivalent. The reason for this lies in the division of the overall verification problem into manageable subproblems. In order for Theorem 1 to be applicable, the ACCESS-local construct must be conservative by analyzing the behavior of the peripheral from an *arbitrary* starting state. This may lead to spurious counterexamples if the starting state of the counterexample is not reachable in the combined HW/SW system.

For example, consider a communication peripheral that needs to be configured and then can be used to transmit data. Normal usage performs these steps separately and it may, in fact, be a requirement that no configuration happens when the device is busy. Let's formulate this requirement as a constraint in the form of an implication $busy \implies \neg config$. Assume now that we are running an ACCESS-local equivalence check to compare two SW segments that configure the peripheral. Since the ACCESS-local computational model begins at an arbitrary state it may return a counterexample in which the constraint is violated, i.e., in which the device begins at a *busy* state and receives a *config* command. This counterexample is, most likely, spurious because the device cannot be in a *busy* state when the SW is still in the device configuration phase. In view of the whole HW/SW system the starting state of the counterexample is unreachable.

When an ACCESS-local check fails, the verification engineer would need to manually analyze the counterexample. If the counterexample is spurious because a constraint was violated, the property instance needs to be strengthened by adding a constraint (in fact, an invariant of the HW/SW system). In our example, the implication $busy \implies \neg config$ would be added.

Usually, a peripheral device imposes several constraints like this on the software using it. Such constraints are a consequence of the device's inner architecture, and, often enough, not all of these constraints are documented for the software programmer. Discovering undocumented constraints through debugging ACCESS-

local checks may be tedious but also very valuable, because the counterexample can also result from plain firmware inequivalence or from an actual bug in the device hardware.

We can use the basic idea of the ACCESS-local approach to specifically search for undocumented device preconditions as well as for bugs. We construct an ACCESS-local model as in Fig. 3.4 but do not connect it to any program netlists. Instead, we control both instances of the device hardware by an access sequence that is *arbitrary* but the same for both instances. For each instance, the ACCESS-local construct models all access patterns (of a given fixed length), i.e., all valid waveforms in all timing variations that represent the abstract access sequence. The problem formulation searches for a pair of access patterns that induce a different ending state in the device. If a solution exists then we have found an access sequence for which the device behavior depends on the actual access timing. Such an access sequence either violates a constraint unknown so far or it exposes a bug in the device hardware.

We call this check the *ACCESS Device Check*. Every constraint found in this check is iteratively added to a constraint set. This set is used to exclude the same spurious counterexamples to be returned again in further runs of the Device Check. The complete constraint set is then used in the ACCESS-local equivalence checks with the effect that *all* spurious counterexamples are prevented.

3.5.2 Electronic System Level – SystemC-PPA

The computation effort required in the proposed verification approach largely depends on two factors: the complexity of the hardware peripheral (in terms of sequential depth and size of its state space) and the length of a software's access sequences. If the complexity of the peripheral cannot be mitigated by the techniques presented in Sec. 3.5.1 or the sequence mapping (see Def. 10) contains one or more very long access sequence, this can become an issue. In such a case, using the

peripheral's RTL description as the hardware model will most probably result in proofs that are too complex to be solved in an acceptable amount of time, if at all.

A more abstract model at the ESL might provide a solution for this issue. In order for this to have any value, the verification results must be transferable. This requires a formal connection between the respective models at the ESL and the RTL, i.e., the abstraction must be formally sound. The basis for establishing such a connection is laid out in Sec. 3.5.3 where the notion of *Path Predicate Abstraction* is reviewed.

Deriving an PPA from a RTL description costs non-negligible overhead. Hence, it should only be considered if the expected increase in proof performance justifies the cost or if it will not solely be used for the purpose of this verification approach, such as applications as virtual protoype or in unrelated ESL verification approaches.

On the other hand, when following a top-down design approach, similar to the one suggested in [44], the required ESL model is already available at no additional cost. Note, that this does not mean it is always the superior choice. Naturally, using the ESL model restricts proofs to effects and details which are actually modeled by it. Hence, the ESL model must be created with its future use case already in mind. Otherwise, it might lack information that is crucial for a particular application.

The different representation of the peripheral in the hardware model requires a different definition of what constitutes equivalence.

Definition 18 (Equivalence at the ESL)**.** *Let* $\Phi_{\mathfrak{P}}(\sigma) = \{\phi_0, \phi_1, ..., \phi_m\}$ *describe the sequence of operations performed by a peripheral* $\mathfrak{P}$ *under the sequence of I/O accesses* $\sigma = \{\sigma_0, \sigma_1, ..., \sigma_n\}$*. Further, let* $\Omega_{\mathfrak{P}}(\sigma) = \{\omega_0, \omega_1, ..., \omega_k\}$ *describe the sequence of messages sent by* $\mathfrak{P}$ *to its environment. Two combined hardware/-software systems* (P, S) *and* $(\mathfrak{P}, \mathfrak{S})$ *are considered equivalent iff* $\Phi_P(S) = \Phi_{\mathfrak{P}}(\mathfrak{S})$ *and* $\Omega_P(S) = \Omega_{\mathfrak{P}}(\mathfrak{S})$*.*

In simple terms, two systems are equivalent if and only if the access sequences of the software models result in the identical sequence of operations and sent messages in the respective SystemC-PPA model.

3.5.3 Path Predicate Abstraction

The theoretical foundation of the PPA has been published in [40, 45, 46]. It provides a novel take on extending the chain of trust, which winds itself trough the various hardware abstraction levels – currently starting from the transistor level, through the gate level, and ending at the RTL – to also include the ESL.

In the current chain of trust, the correctness of an abstraction w.r.t. to the model it was abstracted from, can be verified by *formal equivalence checking* techniques such as combinational or sequential equivalence checking. The theoretical applicability of such techniques hinges on finding a suitable and verifiable definition or notion of equivalence. Two combinational circuits are equivalent if they produce, for the same input vector x, the output vector y, i.e., both implement the same Boolean function $f(x) = y$. For sequential circuits, equivalency changes in that it is now defined on sequences. For identical same input sequences $X = \{x_0, x_1, ..., x_n\}$ the circuit must produce the identical output sequence $Y = \{y_0, y_1, ..., y_n\}$, i.e., the same functional behavior of the circuits is captured by the same FSM.

Unfortunately, defining such a notion of equivalence between models at the ESL and the RTL proved to be rather difficult. Due to the semantic gap (see Sec. 2.3.2.2) between these two abstraction levels it is not clearly defined how the inputs and outputs of the RTL will be represented in the ESL. In contrast to the RTL, there is no notion of time, in terms of clock ticks, at the ESL. The transfer of data between two hardware modules via a common bus might be modeled as passing a message from the sender to the recipient based on events. The same transfer, at the RTL, can become an arbitrarily complex, cycle-accurate bus protocol. So how can these two be mapped to each other such that it can be shown that the ESL model is a

sound abstraction of the RTL model?

In the following, only the basic ideas necessary for understanding the principles behind the creation of a PPA are presented. For an in-detail explanation of its formal development, please refer to [40].

A necessary core concept is formed by a special graph labeling or coloring called "operational coloring".

Definition 19 (Operational Graph Coloring)**.** *Consider a directed graph* $G = (V, E)$*, a subset* $W \subseteq V$ *of the graph nodes called* colored *nodes, a set of colors* $\hat{W} = \{\hat{w}_1, \hat{w}_2, \ldots\}$ *and a surjective coloring function* $c : W \mapsto \hat{W}$*.*

A path $(v_0, v_1, \ldots, v_n)$ *such that* $v_0, v_n \in W$ *and* $v_1, \ldots, v_{n-1} \in V \backslash W$ *is called an* operational path *in* G*. The set* W *must be chosen and colored such that:*

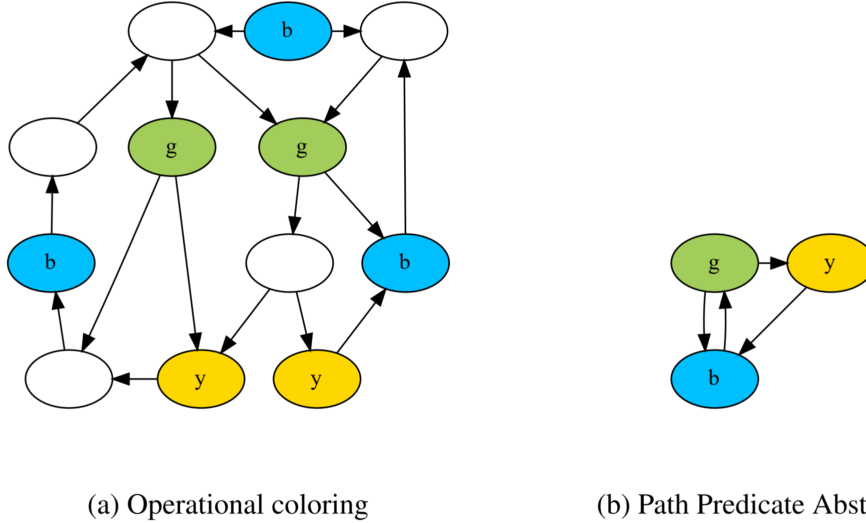
- *every cyclic path in* G *contains at least one node from* W*, i.e., the graph contains no cycles with only uncolored nodes,*
- *for every operational path* $(v_0, v_1, \ldots, v_n)$ *and* $u_0 \in W$*, such that* $c(u_0) = c(v_0)$*, there must exist an operational path* $(u_0, u_1, \ldots, u_m)$ *in* G *with* $c(u_m) = c(v_n)$

We call c *an* operational coloring function *and* G *an* operationally colored *graph.*

□

Fig. 3.6a shows a directed graph abstractly representing a finite state transition diagram, e.g., a Kripke model or an FSM. After inspecting the graph it should be visible that the given example satisfies the requirements above and is indeed an operationally colored graph:

- Its nodes are either colored in blue, green and yellow ($v \in W$ with $\hat{W} = \{b, g, y\}$), or uncolored ($v \notin W$).



(a) Operational coloring

(b) Path Predicate Abstraction

Figure 3.6: Operational Graph Coloring and resulting Path Predicate Abstraction

- The graph contains no path starting from an uncolored node $v_x, ..., v_x$, that does not contain at least one colored node.
- If there is a path from one colored node to another colored node, this is true for every node of the respective colors.

Definition 20 (Path Predicate Abstraction)**.** *We consider a graph $G = (V, E)$ with a set of colored nodes $W \subseteq V$, a set of colors $\hat{W}$ and an operational coloring function $c : W \mapsto \hat{W}$.*

A directed graph $\hat{G} = (\hat{W}, \hat{E})$, such that for any two nodes $u, w \in W$, it is $(c(u), c(w)) \in \hat{E}$ if and only if there is an operational path $(u, \ldots, w)$ in G, is called path predicate abstraction *of G.* □

As a result, an abstracted graph contains exactly one node for each color in the color set $\hat{W}$. Path segments through uncolored nodes in the original graph are replaced by single edges in the abstract graph. Assuming that colored nodes are rare in comparison to uncolored ones, this will yield a dramatic reduction in the

number of nodes, corresponding with an equally powerful abstraction.

Since all nodes of identical color provide transitions to the same set of reachable colors, we can replace the original graph by an abstract graph that has one node for each color and one edge representing each such transition. In the process, white nodes are removed and all operational paths of the same kind are collapsed into a single edge. Fig. 3.6b shows the corresponding abstract graph to the operationally colored graph of Fig. 3.6a.

Both graphs contain the same set of operations: $b \rightarrow g$, $g \rightarrow b$, $g \rightarrow y$, $y \rightarrow b$.

This provides the key to creating a suitable mapping as required in defining a notion of equivalence. The definitions above create a well-defined formal relationship between the abstract graph and the original graph: Any path in the abstract graph can be described by a sequence of colors. The hypothetical bus transaction mentioned earlier is represented by such an operation. The intricacies of a complex bus protocol at the RTL are modeled by uncolored nodes.

3.6 Interconnect Models

The interconnect model is an abstraction of the actual communication structure between the CPU executing the firmware and the targeted peripheral on the chip.

Its main task is to translate the I/O requests generated by the firmware on the CPU into a format that can be processed by the hardware model. Depending on the models for soft- and hardware this can be as easy as passing a message from one model to the other, or demand mimicking the full behavior of bus protocol with correct timing.

In the simple case of both models being able to use message passing, i.e., both being implemented in a variation of SystemC, the interconnect can be as simple as a SystemC channel. In the more complicated combinations, the interconnect

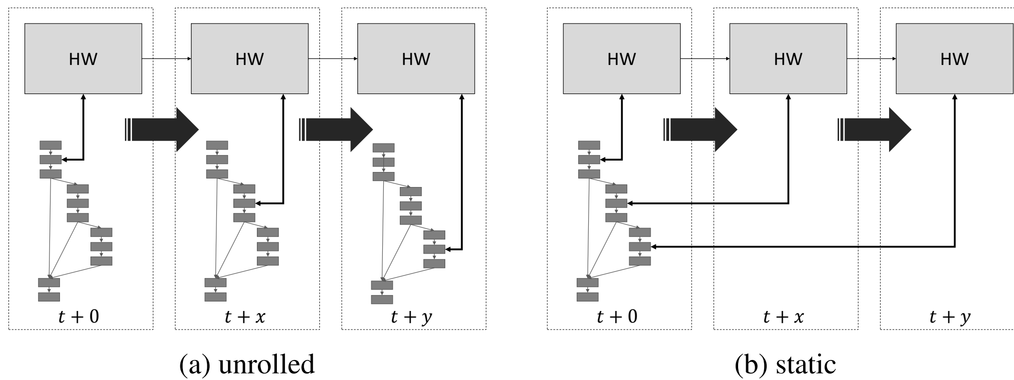
model is realized using assumptions and constraints on the respective interfaces of the hardware and software model. These are implemented using common property specification languages such as ITL and System Verilog Assertions (SVA).

The decision to model the interconnect using constraints instead of a hardware structure has two reasons: First, the model of the hardware is unrolled for the minimal amount of time necessary to prove or disprove equivalence of a single access sequence. Yet, a program consists of many access sequences that need to be verified. Second, if the software is modeled as a PN, the (logical) time of an I/O access is not modeled by unrolling the circuit, but by the placement of the access within the PN.

This either requires many small interconnect models, each forming a single localized proof, resulting in an extensive overhead of re-elaborating and re-compiling the resulting verification model, or a prohibitively large multiplexer structure to handle correct connection.

Furthermore, the PN (see Sec. 3.4) is a combinational representation of a sequential circuit unrolled over time. Integrating an instance of a PN as the software model in the unified verification model will result in the repeated unrolling of this already unrolled representation. Using constraints over actual hardware for implementing the interconnect allows us to connect the unrolled hardware model with the software model at a single point of time. As a result, the proof does not rely on the subsequent unrollings of the software model, which in turn, significantly reduces the size of the model (see Tab. 4.2 in Sec. 4.1.1). The concept is visualized in Fig. 3.7. It shows the unrolling of a hardware device (HW) over three time steps. In the unrolled integration scheme Fig. 3.7a each timestep contains its own instance of the program netlist. In the static integration scheme shown in 3.7b only a single instance of the program netlist is created. All other steps connect back to this instance.

Two types of interconnect models are presented in the following: *Protocol Emulation* and *Direct Register Access*.



(a) unrolled (b) static

Figure 3.7: Model with redundantly unrolled PN and static PN

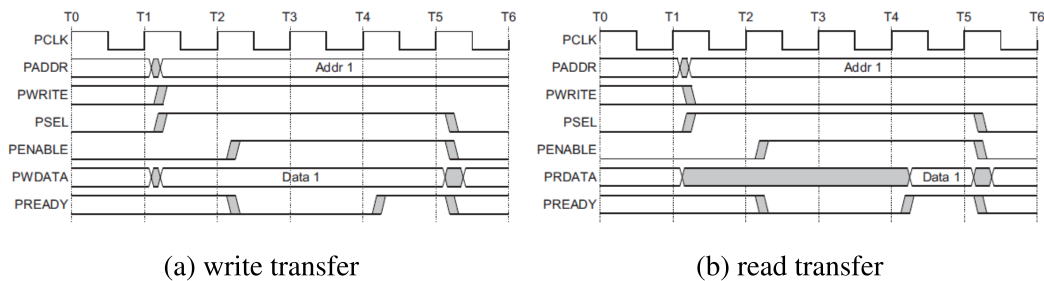
3.6.1 Protocol Emulation

As it name suggests, an interconnect model based on protocol emulation tries to generate behavior as it is generated by a bus system connected to the peripheral's interface. This type of interconnect is especially suited for widely used protocols, such as the ARM Advanced Microcontroller Bus Architecture (AMBA) [47].

The AMBA bus protocols are a set of interconnect specifications from ARM that standardize on-chip communication mechanisms between various functional blocks. A System on Chip (SoC) built with AMBA typically has one or more processors along with several other components, e.g., internal memory and various peripherals like USB, UART, PCIE, I2C etc.

From the various AMBA protocols, the Advanced Peripheral Bus (APB) is of special interest to us, as it is used for connecting low-bandwidth peripherals. It is a simple non-pipelined protocol that can be used to communicate (read or write) from a bridge/master to a number of slaves through the shared bus. The reads and writes share the same set of signals. Fig. 3.8a and Fig. 3.8b show the waveform of these signals as required by a single write or read transfer, respectively.

In the AMBA hierarchy, APB is not designed to directly handle communication requests from a CPU but to serve as a secondary bus to a more complex proto-



(a) write transfer (b) read transfer

Figure 3.8: ARM AMBA APB with wait states [47]

col, such as Advanced eXtensible Interface Bus (AXI), which is useful for high bandwidth and low latency interconnects.

Including the necessary hardware into the verification model would unnecessarily increase the load on the proof engine. In order to avoid this, protocol emulation enforces the correct behavior of the interface signals by a set (or suite) of constraints specific to the protocol, essentially cutting out most of the hardware involved in the SoC's bus system.

Due to it's role as a secondary bus, peripherals implementing an APB interface are only capable of word-size communication. It is expected that half-word or byte-sized accesses are translated to word-size by the AXI/APB bridge.

Because the constraint suite can only enforce behavior which the hardware is inherently capable of, a small hardware attachment in front of the peripherals interface must be added. As its task is only to translate various accesses to word size it is significantly less complex than the actual AXI/APB bridge.

The constraint suite shown in Fig. 3.9, together with the API macro suite in Fig. 3.10 and the aforementioned hardware attachment, form a minimal viable framework to model an APB interconnect model.

The constraints were manually derived from the APB specification document.Due to the relative simplicity of each individual constraint, automated generation tech-

```
 1: macro expr implies(expr condition, consequence) :=
 2:     ! condition || consequence;
 3: end macro;
 4:
 5: constraint reset_enable_when_ready :=
 6:     implies(PENABLE && PREADY, next(! PENABLE));
 7: end constraint;
 8:
 9: constraint leading_select :=
10:     implies(PENABLE, prev(PSEL));
11: end constraint;
12:
13: constraint stable_read_or_write_mode :=
14:     implies(PENABLE , PWRITE = prev(PWRITE));
15: end constraint;
16:
17: constraint stable_data_during_write :=
18:     implies(PENABLE && PWRITE , PWDATA = prev(PWDATA));
19: end constraint;
```

Figure 3.9: Minimal viable constraint suite enforcing APB protocol (in ITL)

niques such as [48] might be applicable.

The signal PSTRB used in the macro STRB (Fig. 3.10 line 9-24) is not part of the APB protocol, but part of the AXI. It indicates which bytes will be actively written or read during a transfer. It is used in the interconnect model to control the hardware attachment. The correct value can be statically evaluated from the provided hardware address and length of data.

3.6.2 Direct Register Access

In contrast to protocol emulation, interconnect based on direct register access completely circumvents the bus interface provided by the peripheral. Instead, the data is directly written or read from the targeted program-visible hardware registers corresponding to the symbolic register used by the program.

The concept is illustrated in Fig. 3.11.

```
 1: macro expr READ :=
 2:     PENABLE && PSEL && ! PWRITE;
 3: end macro;
 4:
 5: macro expr WRITE :=
 6:     PENABLE && PSEL && PWRITE;
 7: end macro;
 8:
 9: macro expr STRB(expr addr, data) :=
10: if ( $length(data) == 8)
11:     case (addr[1:0])
12:         2'b00: PSTRB == 4'b0001;
13:         2'b01: PSTRB == 4'b0010;
14:         2'b10: PSTRB == 4'b0100;
15:         2'b11: PSTRB == 4'b1000;
16:     endcase;
17: elseif ($length(data) == 16)
18:     case (addr[1:0])
19:         2'b00: PSTRB == 4'b0011;
20:         2'b10: PSTRB == 4'b1100;
21:     endcase;
21: elseif ($length(data) == 32)
22:     PSTRB == 4'b1111;
23: endif
24: end macro;
25:
26: macro expr connect_read_port(expr addr, data) :=
27:     READ && STRB(addr, data) &&
28:     addr == PADDR && data == PRDATA;
29: end macro;
30:
31: macro expr connect_write_port(expr addr, data) :=
29:     WRITE && STRB(addr, data) &&
30:     addr == PADDR && data == PWDATA;
31: end macro;
```

Figure 3.10: Macro suite serving as API to construct a interconnect model following the APB protocol (in ITL)

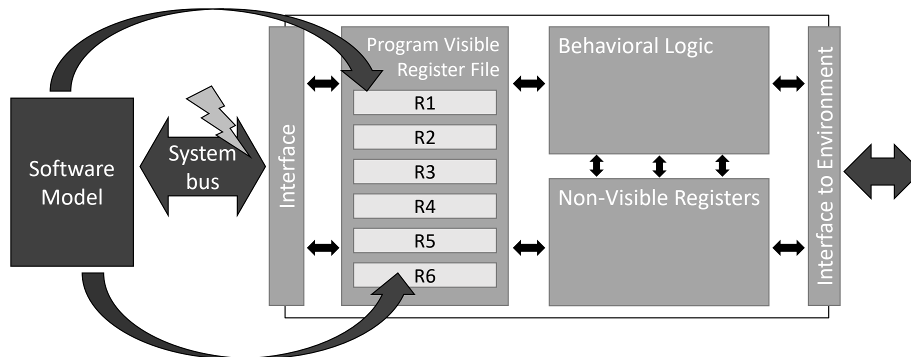


Figure 3.11: Direct Register Access conceptualized

If available, the property suite forming the direct register access can be automatically generated from a description of the programmer's view in the non-functional data of the peripheral's IP-XACT file. Without such a file at hand, creating the properties involves inspection of the peripheral's HDL code, because the naming of the actual registers in the hardware may differ from their symbolic names in the specification.

Implementing the interconnect via direct register access shows no significant benefit in terms of the required time of a proof if the peripheral implementation contains the logic of the bus interface (see Tab. 4.3 in Sec. 4.1.2).

This can be mainly attributed to the fact that the direct access properties merely state what value a target register should have after a write communication or which value should be read. How this value actually gets into the target register is still restricted by the possible behavior of the hardware implementation. This leaves the task of figuring out the correct input sequence at the bus interface up to the proof engine. Hence, there are two main applications of direct register access: The first are peripherals with a non-standard bus interface for which no protocol emulation suite exists and the overhead of creating one is not justifiable due to it being very complex. The second are peripherals whose bus interface logic is clearly separated from the rest of the implementation and, therefore, can be easily and quickly removed.

Chapter 4

Experiments and Case Studies

In order to evaluate the verification method presented in Chap. 3, several experiments and case studies were performed. In this chapter, the setup, goal and results of each such case study are reported.

The first set of experiments explores the general feasibility of the method, as well as the effect of different modeling decisions on the complexity of the resulting proofs. The second set of experiments applies the ACCESS device check (see Sec. 3.5) to several open-source peripherals. Afterwards, the following case studies on the global equivalence check are presented:

1. Driver variants for the PULPino GPIO with modified register interface according to the method presented in [43]. The case study uses program netlists and RTL models.

2. A Soft-SPI, i.e., a software-implemented serial peripheral interconnect (SPI), using the PULPino GPIO. As software model program netlists, as well as source code models were employed. For the hardware only RTL models were used.

Table 4.1: Device data

peripheral	state bits	input bits	output bits
Averaging-Filter	320	34	32
Aquarius UART	161	42	36
PULPino GPIO	320	81	324
PULPino SPI master	668	53	47

3. An industrial local interconnect (LIN) driver implementation using the Aquarius [49] UART. The case study uses program netlists, but uses both RTL and ESL models for the hardware model.

Data on the size of the RTL implementation of the peripherals used in these experiments are shown in Tab. 4.1.

All experiments were performed on an Intel® Core™ i7-6700 CPU @ 3.40 GHz × 8 with 32 GB of RAM running Linux and the OneSpin® 360 Design Verification tool.

4.1 Performance Experiments

The experiments of this section are designed to show how overall proof performance is affected by various decisions in the modeling process as well as configuration parameters of the proof.

All tables contain the following metrics: The columns "time" and "memory" show the CPU time and memory required by the solver. As the resources required for checking an assertion depend on the complexity of the design and the assertion, it is sometimes useful to understand this complexity. "Variables" refers to bit-level variables in the cone of influence of the assertion, and "nodes" represent operators used in the internal representation of the assertion. In addition to the operators occurring within the assertion itself, all relevant portions of the design are also taken into account. The size gives only a rough measure for complexity. For

Table 4.2: Proof complexity depending on PN integration scheme

		time (s)			memory (MB)			variables			nodes		
Segment	**cycles**	**static**	**unrolled**	**%**	**static**	**unrolled**	**%**	**static**	**unrolled**	**%**	**static**	**unrolled**	**%**
A-5	24	7	9	22	2373	2728	13	56494	70016	19	310893	332916	7
A-10	39	21	24	13	2940	3868	24	92279	113121	18	514375	550109	6
A-15	54	38	47	19	3759	5750	35	128062	156224	18	717840	767283	6
A-20	69	56	71	21	4218	6236	32	163837	199319	18	921304	984457	6
B-5	44	46	60	23	3885	5011	22	105204	127550	18	587669	626196	6
B-10	79	161	200	20	5656	7255	22	188701	228127	17	1073037	1141089	6
B-15	114	364	698	48	7330	8500	14	272190	328696	17	1558317	1655888	6
B-20	149	1501	1638	8	8637	11709	26	355671	429257	17	2043701	2170792	6

instance, when the nodes include multiplication operators this will normally result in much higher complexity than simple operators like Boolean logic or addition. For one and the same design, though, two assertions with a similar "vars" and "nodes" count can be expected to take similar run times.

4.1.1 Interconnect: Unrolled vs. Static PN

As stated in Sec. 3.6, the PN is a combinational representation of a sequential circuit unrolled over time. Directly integrating it in the unified verification model results in it being unrolled again along with the hardware model. This results in redundant circuitry.

The experiment presented in this section is designed to show the influence of this redundant unrolling on the overall problem size of an ACCESS-local proof instance in comparison to a computational model where this redundancy is being avoided by having a single, static PN. The data for the comparison of the two integration methods is shown in Tab. 4.2.

Two different ACCESS-local proof instances A and B were given to the proof engine. Instance A is located at the very beginning of the PN and contains a total of three hardware accesses in a 2:1 split, i.e., one software variant performs two, the other one a single access. Instance B is at the very end of the PN and contains nine accesses in a 5:4 split. To generate further data points, each instance was

Table 4.3: Proof complexity (measured by required computation time) depending on chosen interconnect type

Interval	5			10			15			20		
#accesses	Protocol	Direct	%	Protocol	Direct	%	Protocol	Direct	%	Protocol	Direct	%
1	3 s	2 s	-50	6 s	5 s	-20	15 s	22 s	32	28 s	22 s	-27
2	4 s	6 s	33	26 s	19 s	-37	43 s	42 s	-2	55 s	95 s	42
3	14 s	10 s	-40	60 s	57 s	-5	90 s	115 s	22	235 s	230 s	-2
4	22 s	24 s	8	120 s	173 s	31	335 s	399 s	16	493 s	894 s	45
5	42 s	44 s	5	328 s	228 s	-44	777 s	475 s	-64	1560 s	901 s	-73

solved with the *interval* parameter set to the values 5, 10, 15 and 20, as denoted in the trailing number in the "segment" column. The column "cycles" shows for how many cycles the model had to be unrolled.

The data clearly shows an improvement across every metric when using the static over the unrolled implementation scheme. On average the static scheme results in a model that has 18% less variables, 6% less nodes, requires 24% less memory and results in 22% faster proof times.

The proof instance's location in the PN has no observable effect on any of the metrics.

4.1.2 Protocol Emulation vs. Direct Register Access

Sec. 3.6 provides two different models on how to connect the software model to the hardware model.

The data presented in Tab. 4.3 comes from an experiment exploring how the chosen interconnect model affects proof performance. In the experiment ACCESS-local checks with an increasing number of accesses (from 1 to 5) are performed. Each check is performed once using protocol emulation and repeated using direct register access. For further data points, the experiment is performed with the interval parameter set to values 5, 10, 15 and 20.

No clear trend can be identified in the data gathered during the experiment. On

Table 4.4: Impact of parameters on device check complexity

sequence	interval	time (h:m:s)	memory (MB)	variables	nodes
2	4	00:00:09	773	804	2118
	8	00:01:10	805	1092	4185
	16	00:06:04	1250	1660	8318
	32	00:28:03	1198	2788	16575
4	4	00:01:50	1059	1340	5408
	8	00:15:58	1253	1916	11391
	16	01:20:48	1749	3052	23356
	32	06:20:49	2078	5308	47277
8	4	00:07:58	986	2428	16275
	8	00:47:50	1661	3580	35802
	16	03:47:20	1500	5852	74855
	32	21:32:47	3650	10364	152952

average, using protocol emulation resulted in 5% faster proofs. Yet, with a standard deviation of $\sigma = 35\%$ and relative proof time varying from -73% to 45%, no predictable effect on proof performance increase by using one interconnect model over the other could be observed. Performancewise there is no benefit of prefering one scheme over the other. Yet, protocol emulation can be more easily adapted in a generalized approach. Hence, all following experiments utilize protocol emulation as its interconnect model.

4.1.3 Scalability Experiments

This set of experiments is directed at showing the impact of modeling uncertainty in the timing of the arrival of an I/O access at the peripheral's hardware interface. The hardware is modeled at the RTL, the software, if applicable, at the binary level by a program netlist.

We performed a series of ACCESS-local checks on a simple averaging-filter peripheral. Its design is based on a shift register of length 10 and some arithmetic logic to compute the average of the 32-bit shift register values. With more than 300 state bits it is reasonably sized to represent peripherals used in IoT applications.

Table 4.5: Impact of parameters on local proof complexity

sequence	interval	time (h:m:s)	memory (MB)	variables	nodes
2	4	00:00:05	744	804	2295
	8	00:00:45	797	1092	4554
	16	00:04:39	983	1660	9071
	32	00:34:18	1215	2788	18096
4	4	00:00:02	343	1340	5955
	8	00:00:08	920	1916	12578
	16	00:00:48	1034	3052	25823
	32	00:05:37	1224	5308	52304
8	4	00:00:14	987	2428	18138
	8	00:02:00	1045	3580	39969
	16	00:11:11	1254	5852	83630
	32	00:55:54	1763	10364	170943

The register interface is generic without any consideration to specific peripheral bus protocols. It imposes very few constraints on access patterns so that the property proof instances grow with the “pure” size of the problem and are not influenced much by the growing complexity of the access constraints.

For each parameter set (interval size, sequence length) the device was subjected once to the ACCESS device check and once to the ACCESS-local verification step. Remember that many such ACCESS-local verification steps compose the ACCESS-global verification of Sec. 3.3. The PNs for ACCESS-local were generated from a simple program which generates a series of input values for the device from a seed value. In this experiment the program is intentionally simple and we use the same program for both PN instances of the device miter of Fig. 3.4. This is because we do not address global equivalence verification, but focus on examining the scalability of the individual ACCESS-local verification steps and the ACCESS device checks. The grace period was set to 5 cycles. Performance data for our method are listed in Tab. 4.4 and Tab. 4.5. The columns “sequence” and “interval” refer to the number of device accesses and to the maximal number of clock cycles between accesses, respectively. As expected, the proof complexity grows the more

combinations of accesses and idle cycles must be enumerated. This is also clearly reflected in the number of created SAT variables and nodes, as reported by the property checker. The column labeled “variables” gives the number of Boolean variables in the proof instance while “nodes” (next column) represents Boolean operators in the proof instance. As opposed to the ACCESS device check, the device miter for the ACCESS-local verification step is constrained by the PNs. This has only minor impact on the size of the SAT instance (see 3.6), but greatly reduces the required proof time. The ACCESS local verification step and the ACCESS device check can be considered the bottleneck for our global approach in terms of computational complexity. The shown results promise that the proposed approach is feasible for realistic device sizes in many IoT applications.

4.2 ACCESS device checks

The ACCESS device check as proposed in Sec. 3.5.1.1 has been applied to three open-source standard peripherals: the General Purpose Input/Output (GPIO) and Serial Peripheral Interface (SPI) master peripherals, as included within the RISC-V-based PULPino [50] platform, and the UART of the Aquarius Open Core SoC [49].

The PULPino peripherals are designed to interface with the AMBA APB (without wait states), the Aquarius peripheral adheres to the open-source Wishbone bus interface. A set of protocol constraints model the compliance of the interface with the respective communication protocol. Theses constraint sets had to be defined manually, which took roughly one hour per set. They can be reused for any peripheral utilizing the respective protocol. The initial setup of the experiment, i.e., generating the wrapper module for unrolling the device hardware and creating the ACCESS property, takes only a couple of seconds as it is fully automated.

Interestingly, in the pursuit of our experiments, we discovered multiple implicit assumptions made by the device w.r.t. how it will be accessed that were not mentioned by the documentation of the hardware device. For the SPI we detected

Table 4.6: ACCESS device check performance

peripheral	interval	time (h:m:s)	memory (MB)
Aquarius UART	2	00:01:17	2316
PULPino SPI	2	00:03:01	6155
PULPino GPIO	4	00:02:28	1248

that any write to the peripheral during an active transmission can possibly lead to divergent behavior of both HW instances of the device miter, i.e., the behavior of the peripheral is different for access patterns generated from the same access sequence. This is valuable information for the firmware developer, because mechanisms must be implemented to ensure that the code avoids creating such a pattern. Furthermore, we detected that the peripheral's software reset is currently not implemented. This is denoted in the HDL code itself, but not in the documentation. The GPIO did behave as described in its documentation.

Additionally, we were able to detect a so far unknown bug in the open-source UART. The UART controls its baud rate with a number of internal counters, which reset to zero if a configurable target value is reached. Due to the UART's asynchronous nature, these counters are always advancing, whether a transmission is currently going on or not. The bug may occur when reconfiguring the baud rate of the UART such that the target value T_{high} is replaced by smaller target value T_{low}. If this change happens while the counter value is in between those two values $T_{low} < V_{counter} < T_{high}$ the internal counter will not be properly reset until it overflows, wraps around and reaches T_{low} again. This renders the UART unresponsive to the environment for up to 2^{18} clock cycles. It is not possible for the Software (SW) to avoid this, as there is no way by which it could infer the current value of the internal counter.

Tab. 4.6 shows the average performance data of our tool for detecting a constraint or bug in each device, i.e., the CPU time spent until a counterexample is found. The sequence length in all cases was set to 2, the grace period was set to 5. Since

no bugs or assumptions were found in the case of the GPIO, the table lists the effort required to prove equivalence of an arbitrary access sequences. Note that it is not necessary to proof the equivalence of arbitrary access sequences with the ACCESS Device Check, but only to identify the constraints required for the ACCESS-local verification step.

Besides avoiding spurious counterexamples in ACCESS-global, our results show that the ACCESS Device Check may also be useful as a stand-alone verification technique for a given peripheral. It identifies undocumented restrictions for SW using this peripheral that may cause bugs if neglected. Moreover, subtle HW bugs of the peripheral may also be identified.

4.3 Case Study: Register Interface Optimization of PULPino GPIO

The case study in this section is the results of a collaborative effort between the author of this thesis and the authors of [43].

In current industrial practice, developers of low-level software (e.g., drivers) stand before the difficult task of implementing the intended driver behavior, while adhering to the peripheral's register layout and simultaneously ensuring good performance and a small memory footprint. To avoid polluting the source code with macros and bit manipulation operations, a Hardware Abstraction Layer (HAL) is introduced. Its purpose is to hide register-layout-specific code like shifting and masking operations behind easily recognizable functions. The drawback of the HAL approach is a possible loss of performance and an increased memory footprint if the chosen register layout and HAL functions are not aligned with the driver behavior.

[43] introduces a DSL-based flow built on top of the C language, with which the developer can define the peripherals register interface in an abstract form as bit

fields and state side effects of the targeted hardware. With this information, the amount of memory barriers introduced in the code via the *volatile* keyword can be reduced, enabling further optimization by the compiler and resulting in a more efficient binary. Additionally, the designer can create and exploit hierarchical bit field declarations and arrays to write high-level code that is free from bit manipulation, effectively reducing his workload.

From this high-level code a register layout optimization of the peripheral can be derived that is based on the actual control and data flow of the software.

Three variants of an application software addressing the systems GPIO were provided by the authors of [43]. These variants are further referred to as $V0$, $V1$ and $V2$. Each variant is based on the same high-level code, but differs from the others in the implementation of its HAL layer as well as the register layout of the GPIO peripheral.

Applying the ACCESS verification method to check equivalence between the different variants initially showed that none of the provided variants were equivalent. Manual inspection of the provided counterexamples of the failed ACCESS-local proofs quickly revealed incorrect read/write alignments in the modified register interface of $V0$ and $V2$. For $V0$ this affected register *INTTYPE1*, for $V2$ the whole *gpio_padcfg* register bank. The authors of [43] confirmed these findings.

Furthermore, a missing side effect annotation in $V0$ introduced a race condition for the *gpio_in* register, which in turn could potentially lead to the peripheral incorrectly raising an interrupt. It was confirmed that this was not an intentionally placed bug, having passed previously employed simulation-based verification efforts.

Finding a counterexample for these bugs took, in all cases, roughly two minutes. After fixing the detected bugs, repeating the ACCESS verification showed the variants to be functionally equivalent. Performance data for the verification runs is

Table 4.7: Proof run times and memory usage for showing equivalence of the provided software variants (post bug fixes). Each row corresponds to an ACCESS-local check.

V0 ⇔ V1			**V0 ⇔ V2**			**V1 ⇔ V2**		
Accesses	Time	Memory	Accesses	Time	Memory	Accesses	Time	Memory
2:1	6 s	2366 MB	2:2	8 s	2154 MB	1:2	7 s	2373 MB
2:1	8 s	2366 MB	2:2	9 s	2154 MB	1:2	8 s	2339 MB
2:1	7 s	2366 MB	2:2	8 s	2282 MB	1:2	7 s	2373 MB
2:1	7 s	2087 MB	2:2	6 s	2039 MB	1:2	6 s	2339 MB
1:1	5 s	2198 MB	1:2	5 s	2234 MB	1:2	7 s	2339 MB
1:1	5 s	2196 MB	1:2	7 s	2271 MB	1:2	10 s	2221 MB
1:1	7 s	2196 MB	1:2	5 s	2299 MB	1:2	6 s	2121 MB
1:1	3 s	1846 MB	1:1	4 s	1940 MB	1:1	9 s	2443 MB
5:4	42 s	2869 MB	5:6	53 s	3995 MB	4:6	49 s	3884 MB
5:4	40 s	3442 MB	5:6	51 s	3954 MB	4:6	41 s	3884 MB
5:4	53 s	2994 MB	5:6	60 s	4138 MB	4:6	57 s	4142 MB
5:4	40 s	2559 MB	5:6	48 s	3736 MB	4:6	46 s	3885 MB

shown in Tab. 4.7.

4.4 Case Study: HW/SW-Optimized Soft-SPI Implementation

This section presents results of a case study in checking HW/SW co-equivalence between variants of an interrupt-driven *Software-implemented Serial Peripheral Interface* (Soft-SPI) slave for the RISC-V based PULPino platform. The case study is repeated at both levels of software abstraction: binary and source-code level.

The Soft-SPI emulates a SPI slave peripheral via “bit banging” on the PULPino *General Purpose Input/Output* (GPIO) peripheral. The protocol requires four dedicated I/O pins: *Synchronous Clock* (SCK), *Slave Select* (SS), *Master-In-Slave-Out* (MISO) and *Master-Out-Slave-In* (MOSI). Fig. 4.1 shows the timing diagram of the protocol for an 8-bit duplex transmission as well as the three phases defined

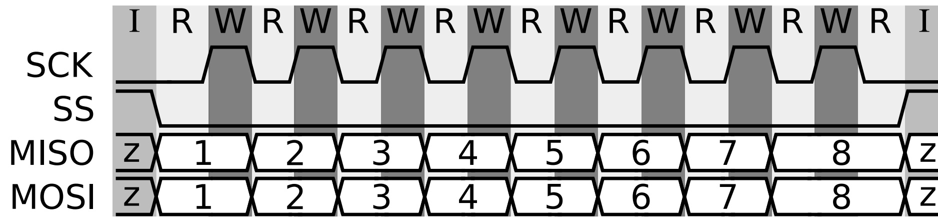


Figure 4.1: SPI Protocol Timing Diagram. Colors highlight different phases as used in the Soft-SPI.

Table 4.8: GPIO pad configurations for the Soft-SPI phases

	Idle (I)			Prepare-for-Read (R)			Prepare-for-Write (W)		
Pad	PADDIR	INTEN	Trigger*	PADDIR	INTEN	Trigger*	PADDIR	INTEN	Trigger*
SCK	IN	disabled	high	IN	enabled	high	IN	enabled	low
SS	IN	enabled	low	IN	enabled	high	IN	enabled	high
MISO	IN	disabled	-	OUT	disabled	-	OUT	disabled	-
MOSI	IN	disabled	-	IN	disabled	-	IN	disabled	-

*high: INTTYPE0=0, INTTYPE1=1, low: INTTYPE0=1, INTTYPE1=1

by the Soft-SPI slave: *Idle* (I), *Prepare-for-Read* (R) and *Prepare-for-Write* (W).

The functions of the GPIO pads are defined by four control registers. Each control register holds four bits that are each associated with one of the four GPIO pads. Each pad can be configured independently by setting the bits associated with it in the four registers. Register *PADDIR* sets a pad's function as input or output, register *INTEN* enables interrupts, and registers *INTTYPE0* and *INTTYPE1* control the interrupt triggering behavior. A GPIO pad needs to be initialized after reset and/or reconfigured according to the current phase of the Soft-SPI. Tab. 4.8 shows the configurations for each pad in each protocol phase.

The Soft-SPI defines one *Interrupt Service Routine* (ISR) for each phase, ISR_{idle}, ISR_R and ISR_W. Besides servicing interrupts, ISR_{idle} is also used for initialization of the peripheral. ISR_R is called for two different interrupt events, depending on the driver state. Fig. 4.2 shows the resulting FW call graph.

We examined three variants of the Soft-SPI: *unpacked*, *SW-packed* and *HW/SW-packed*. All use the same set of GPIO pads. Because the software uses only a

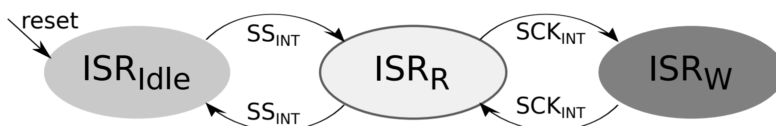


Figure 4.2: Soft-SPI call graph

subset of the available pads, each write access to a register must be preceded by a read access and by creation of an appropriate bit mask for selecting the used pads. This prevents the write accesses from affecting pads that are possibly used by other applications. In the sequel, we will refer to these two steps as a joined *Read-Modify-Write* (RMW) operation. All variants require one RMW to write data to MISO in ISR$_R$, a normal read access to MOSI in ISR$_W$ as well as a read access of the interrupt status register at each interrupt. These are not included in the following descriptions.

The first variant, *unpacked*, uses the GPIO driver provided with the PULPino platform, in which each bit in a GPIO register must be set by its own RMW. This requires at least 12 RMW operations in ISR$_{idle}$, three in ISR$_R$ and one in ISR$_W$.

The next variant *SW-packed* exploits the fact that the FW can simultaneously configure the same control parameter for multiple pads in a single RMW operation. Hence, the RMW operations in ISR$_{idle}$ are reduced to 4 (one for each control register). ISR$_R$ and ISR$_W$ remain unaffected because the accesses are all distributed over different registers, leaving no room for optimizations.

While in variant *SW-packed* only the FW was optimized, both the FW and the HW are modified in variant *HW/SW-packed*. The optimization achieved in variant *HW/SW-packed* is possible due to changes in the GPIO's addressing scheme. In the PULPino SoC the complete address space from 0x1A101000 to 0x1A101ffff is reserved for the GPIO peripheral, but only the address space from 0x1A101000 to 0x1A10103C is actually used. We use this to introduce a new option to the peripheral's hardware register interface at address 0x1A101100. The new option is conceptually presented in Fig. 4.3 and allows all pads that are used by the Soft-SPI

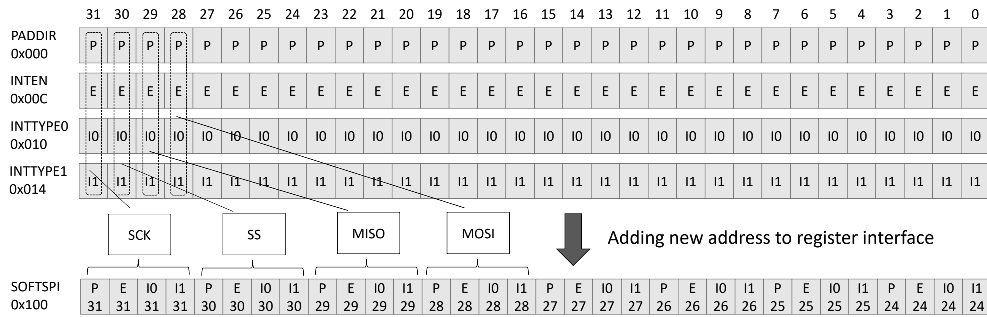


Figure 4.3: New GPIO control register addressing scheme after HW modification.

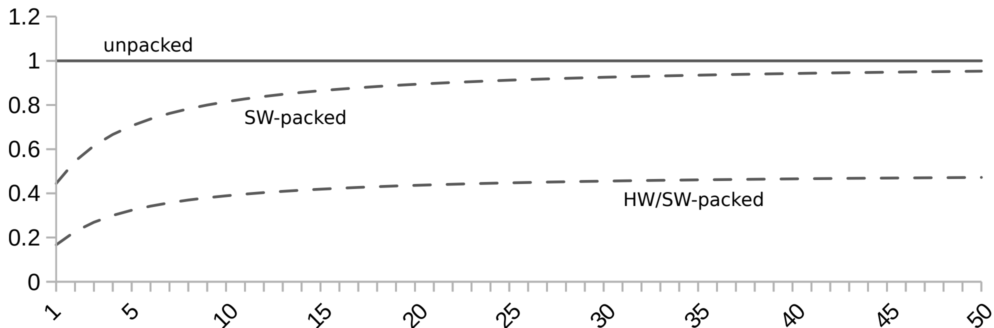


Figure 4.4: Normalized number of configuration accesses to the GPIO by the Soft-SPI variants to transmit x data bits.

to be configured simultaneously with only a single 16-bit write access. Hence, this variant only needs to perform a single write access in each ISR.

Fig. 4.4 shows the normalized effect of the optimization on the amount of accesses required to reconfigure the GPIO in order to transmit a certain number of bits. We can see that both optimized variants have a significant effect on the number of required I/O accesses, especially for a low number of transmitted bits.

In order to verify the equivalence of the variants with ACCESS, the division of the code into ISRs provides a natural sequence mapping. During PN generation the maximum number of bits per transmission was set to one. This is sufficient to exhaustively explore all transitions in the FW call graph. The HW modifications to the GPIO do not add or remove any of HW ports or registers. Hence, no additional action has to be taken to map HW states.

Table 4.9: Experimental results of equivalence checks between Soft-SPI variants

variant	segment	Program Netlist time	Program Netlist memory	Source Code time	Source Code memory	Relative Change time	Relative Change memory
unpacked	ISR_{idle}	748 s	4881 MB	722 s	4623 MB	3%	5%
⇔	ISR_R	127 s	3148 MB	132 s	2972 MB	-4%	6%
SW-packed	ISR_W	68 s	2347 MB	62 s	2224 MB	9%	5%
unpacked	ISR_{idle}	307 s	3539 MB	321 s	3264 MB	-5%	8%
⇔	ISR_R	88 s	2794 MB	84 s	2621 MB	5%	6%
HW/SW-packed	ISR_W	15 s	2357 MB	14 s	2214 MB	7%	6%
SW-packed	ISR_{idle}	195 s	3939 MB	201 s	3623 MB	-3%	8%
⇔	ISR_R	74 s	2793 MB	78 s	2645 MB	-5%	5%
HW/SW-packed	ISR_W	13 s	2357 MB	13 s	2183 MB	0%	7%

All variants could be proven to be equivalent to each other under the equivalence notion of ACCESS. Proofs were performed assuming a maximum delay of 4 cycles per access and a grace period of 5 cycles.

The experiments of this case study were performed once using a PN to model the software and once more as a source code model as presented in 3.4.

The data on the individual proofs shown in Tab. 4.9 indicates no significant difference in computation time and minor improvement of 5 percent as a result from using the source code model instead of the program netlist.

A possible explanation for this reduced memory footprint could be that the software model is created on the granularity of the API calls, e.g., the source code model for ISR_{idle} contains only the relevant code for this specific code section, whereas the netlist is always included as a whole. On the other hand, the source code model must be constructed from the provided API calls. Due to the way the model is synthesized and integrated in the used verification software, these API calls also form the smallest comparable segment. Hence, it is not possible to further split the ACCESS-local check of ISR_{idle} into smaller ones, which is expected to have significant impact on proof performance (see Tab. 4.5).

4.5 Case Study: Industrial LIN

The Local Interconnect Network (LIN) bus [51] is a serial network protocol that was originally developed for communication between components in vehicles. With an increasing number of devices distributed in modern cars, the CAN bus quickly became too expensive to implement for every such component. LIN was developed as an inexpensive alternative relying on cost-efficient hardware and only a single wire. It is intended as a second hierarchical network layer, usually complementing an existing CAN network.

Data is transferred across the bus as messages in form of a predefined frame. The frame consists of two parts: the header and the response. Both of these parts can further be divided in more specific parts. A LIN frame is shown in Fig. 4.5.

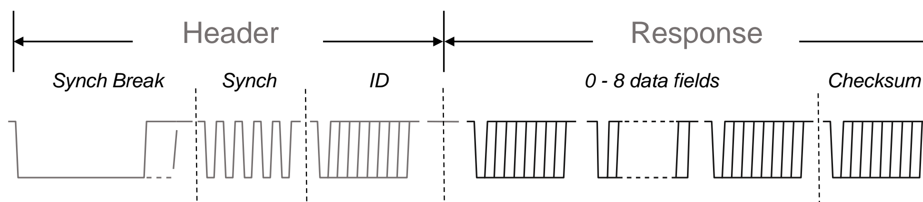


Figure 4.5: LIN frame: Header and Response with respective sub-parts

This case study exercises the verification approach of ACCESS-global on four variants of an industrial Local Interconnect Network (LIN) driver implementation compiled for and run on a model of the Aquarius Open Core SoC and its UART. The variants differ in the way the driver initializes and reconfigures the UART during runtime. Variant LIN-A serves as the golden model. In its initialization sequence, it performs two 8-bit writes, followed by sequentially setting two flag bits. The writing of the bits involves first reading the respective byte, creating a bit mask, modifying the value and writing the modified value back, in order to not override the other bits of the addressed byte. This procedure is further referred to as read-modify-write. All these accesses are to different parts of the same 32-bit configuration register. In an attempt to reduce the number of I/O instructions in the other driver variants, these accesses are combined. Variant LIN-B combines the

Table 4.10: ACCESS-global for industrial LIN

Variant	PN generation time∗ (m:s)	Proof time † (m:s) RTL	PPA	Result
LIN-A / LIN-B	3:38	7:32	0:15	Hold
LIN-A / LIN-C	3:23	5:05	0:10	Fail
LIN-A / LIN-D	3:52	5:51	0:12	Fail

∗ Longest generation time of both PNs
† Proofs are independent and can be parallelized

first two 8-bit writes to a single 16-bit write and the two read-modify-write to a single one. Variant LIN-C combines all accesses to a single 32-bit read-modify-write. Last, LIN-D combines the first two 8-bit writes to a single 16-bit write, just like LIN-B, but performs a 16-bit read-modify-write. This optimization is done so that the initialization and reconfiguration procedures can both utilize the same code segment.

All variants are equivalent from the programmer's view, as each of their modified segments leads to identical memory states of the SW-visible registers of the HW after they have been executed. Yet, as ACCESS-global reveals, variants LIN-C and LIN-D result in erroneous behavior, namely an unintended transmission, due to violating an implicit assumption by the UART. Writing to the lowest byte of the UART configuration register automatically triggers a transmission of this byte, even though its value hasn't changed due to the read-modify-write mechanism. LIN-C and LIN-D fail in considering this side effect of the hardware. The PNs generated for LIN-A, LIN-B and LIN-C contain the initialization segment exactly once and reconfiguration segment four times. The PN of LIN-D contains no distinct initialization, but five reconfiguration segments, as it utilizes the same method for both tasks. This requires ACCESS-global to perform five non-trivial ACCESS-local verification steps for all LIN variants. These can be performed independently, and thus can be parallelized. Tab. 4.10 therefore shows the proof time for the most time-consuming ACCESS-local verification for each LIN variant.

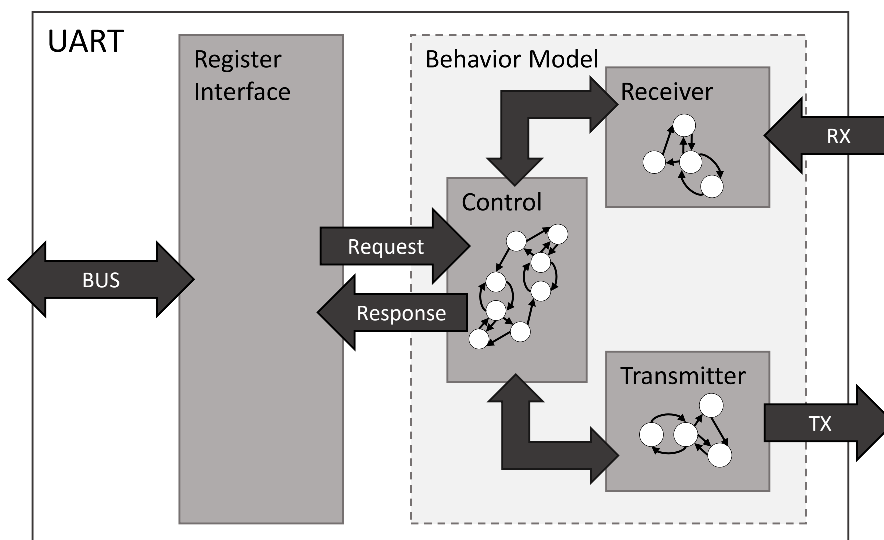


Figure 4.6: Conceptual PPA Model of the Aquarius UART

The experiments of this case study were performed twice: once with the RTL implementation of the Aquarius UART serving as the hardware model and once more with a corresponding PPA model. The PPA model consists of four independent sub-models: one handling the behavior of the receiver, one for the transmitter, a control unit and a model for the register interface. The register interface translates $(address, data)$ pairs into requests for the behavioral part of the model and keeps track of the peripherals configuration. For example, a write access from the bus to the UARTS's TX register will trigger a write request from the interface to the control module with the same data, whereas a write access to any control register will trigger a configuration request. The value of the configuration, e.g., a change of the baud rate, is stored in the register interface but not forwarded in the request. The behavioral model answers in turn with a set of abstract responses.

On this model, two access sequences are equivalent if they result in identical values in the respective register interface and an identical sequence of messages over TX.

The first three form the abstract behavior model which communicates with the register interface via predefined requests and responses.

The results for both versions of ACCESS-global can be found in Tab. 4.10. Thanks to a preceding ACCESS Device Check no spurious counterexamples can occur in the version using the RTL implementation. The same holds true for the one using the PPA, within the granularity of the abstraction. Therefore, with ACCESS we were able to quickly identify the incorrect LIN variants by generating true counterexamples in short time. Similarly, the equivalence of variants LIN-A and LIN-B could be proven without approaching the scalability limits of our approach. The data further shows a significant improvement when using the SystemC-PPA model instead of the RTL model. This can be attributed to the large sequential depth of the UART'S RTL model, which the SystemC-PPA model is able to nearly completely abstract away.

Yet, the SystemC-PPA model comes with another sort of investment. To begin with, as a default the model is not readily available and must first be created. Additionally, not every SystemC-PPA is suitable to be used for every equivalence check. During model creation, special attention and thought must be put into retaining all information necessary in the proof, e.g., in the model above the register file is still crucial. Finally, current tooling has only limited support for SystemC verification (or the way SystemC-PPA is modeled is not well coordinated with the provided support). This leads to additional manual effort for adapting a SystemC-PPA model to work with commercially available tools.

The cost of the SystemC-PPA's manual creation and addaption must first be amortized by the speedup gained by using a SystemC-PPA model over an equivalent RTL model. As can be seen in Tab. 4.10, this speedup is quite significant.

Future efforts in automized model creation and integration will further reduce the initial cost of the SystemC-PPA. As a result, the SystemC-PPA should become the preferred hardware model.

Chapter 5

Cycle-Accurate Timing Extension for Program Netlists

Our experiments (see Sec. 4.1.3) indicate that uncertainty in the timing of each access in an access sequence, modeled by the intervall parameter, strongly affects the complexity of the required local proofs. This is not surprising as an increase in the variability in an access sequences leads to a potentially exponential increase of the modeled access patterns. Hence, eliminating or reducing the uncertainty in the timing of an access should be beneficial to the scalability and performance of the HW/SW-co-verification approach presented in the previous chapters.

This chapter proposes an optional, fully automated approach to generate a clock cycle-accurate view of the software. The generated model is a modified version of a PN (see Sec. 2.1.4). It is fully compositional with RTL hardware and extends the scope of standard techniques for hardware verification to also efficiently examine the software and its interaction with the hardware at a cycle-accurate granularity w.r.t. to the behavior of the processor. This is achieved by interleaving standard PN model generation with existing techniques of Worst-Case-Execution-Time (WCET) analysis to create a cycle-accurate model for programs running on

pipelined processors.

The proposed approach builds upon the previous approaches in [13, 52]. These works present an algorithm to annotate a PN with timing information from which a model for cycle-accurate formal verification of low-level software can be created. The algorithm is only applicable to simple processors that are not susceptible to the problems posed for WCET computation by modern architectures mentioned in Sec. 2.2. In contrast, the approach presented in this chapter is capable of coping with many of these more sophisticated architectures and the complications they cause in WCET computation.

Similar to other WCET approaches, the one proposed in this chapter is based on formulating timing analysis as an ILP problem. The effects of complex microarchitectural features in modern processors on the WCET have also been studied in detail in the literature. For example, [53] considers the influences of caches and speculative execution, and [54] studies pipelined out-of-order processors. The presented approach applies the findings of the rich body of research of WCET computation on the program netlist as computational model. The program netlist already contains a lot of information in an explicit way, which in other approaches must first be computed during the ILP solving process. This significantly reduces the complexity of computing the solution to the ILP problem. The ACCESS verification approach and the proposed WCET computation approach complement each other in that both (can) use the program netlist. Hence, there is no additional cost for creating a program netlist.

5.1 Cycle-accurate SW modeling

In order for timing analysis of software to be precise and feasible, we need an accurate model of the computing hardware as well as the software running on it [53, 55]. Two aspects are essential.

First, any timing analysis needs to consider detailed microarchitectural behavior in order to be sufficiently precise. The computational model must represent the timing-relevant behavior of the targeted microarchitecture. Obtaining such a model can be challenging, depending on the complexity of the target as well as the availability of a sufficiently detailed documentation. Model generation may be a manual or an automated process, e.g., by harvesting simulation traces [56].

Software given at the source level is usually not suitable for a precise static analysis, as its high level of abstraction does not provide the information needed to factor in microarchitectural effects, such as data or structural hazards. Therefore, timing analysis should be performed on the machine code level.

Second, when analyzing a sequence of machine instructions, some notion of context is required. For example, the execution time of a loop or subroutine may differ significantly between iterations or may depend on the background from which it was called.

Given a sufficiently detailed model of the microarchitecture, as well as the software at the binary level, timing analysis can be broken down into two basic steps:

- Restricting the state space of the analysis by pruning infeasible execution paths and deriving upper bounds on the number of loop executions. Usually the Control Flow Graph (CFG) of the program serves as starting point for this step.
- Computing the timing information based on the previous step under consideration of the underlying microarchitectural model.

For the first step, we make use of the EXG which is an intermediate representation produced during PN generation (see Sec. 2.1.4). Generating an EXG from a CFG already requires to check the *liveness* of a given path, effectively exluding infeasible paths from the EXG. For the second step, we can fall back to the classical solution

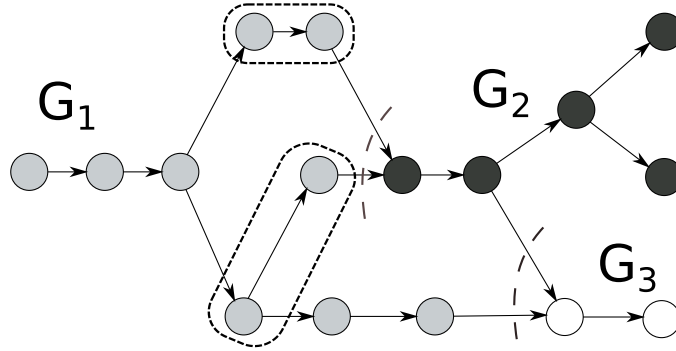


Figure 5.1: Partitioning the EXG for ILP resulting in three trees and showing the context prefixes for G_2 (for length 2)

of formulating the task as an ILP problem (see Sec. 2.1.1). Although current ILP solvers are very powerful it is still sensible to ease the ILP solver's computational load as much as possible.

Fortunately, using the EXG as the computational base model for the analysis allows for several optimizations that keep the problem complexity benign. First, an EXG $G = \{V, E\}$ can be easily partitioned into a minimal, partially ordered set of sub-graphs $\{G_1, \ldots, G_n | G = \cup G_i, \forall i, j, i \neq j : G_i \cap \ G_j = \emptyset\}$, such that each G_i is a tree. This is achieved by cutting G at all incoming edges $e \in E_{in}(v)$ of a vertex $v \in V$ with $|E_{in}(v)| > 1$. As illustrated in the partitioning of Fig. 5.1, cut points correspond to points where one or more paths merge. Instead of creating and solving a single large ILP problem for the whole EXG, we create and solve many small ILP problems for the sub-trees in the EXG. This leads to a substantial boost in performance.

The ILP formulation benefits additionally from the following observation. Nodes in a path in the EXG are strictly ordered. Interdependencies between nodes exist because the corresponding instructions influence each other in the processor pipeline. Direct interdependencies between instructions are, however, limited by the length of the pipeline, and are therefore local. We can model these interdependencies in the ILP formulation using strict inequality constraints. This simplifies the general

ILP problem such that it can be solved in polynomial time [3].

While the EXG model contains the necessary information regarding the software it does not include the effects of the target microarchitecture. We therefore extend the EXG model with a microarchitectural model. This model does not capture any functional behavior but only the interdependencies of instructions in the pipeline. It consists of a set of *instruction constraints definitions* (ICDs) describing the constraints that are inflicted upon the operation of the pipeline by the execution semantics of the instruction. Examples are the ordering of pipeline stages and effects of pipeline hazards. An ICD is a template for a set of inequality constraints written in terms of template variables representing the scheduling of the pipeline stages. When generating the ILP problem we instantiate for each instruction in the EXG a set of pipeline constraints from the corresponding template. Some constraints in the template are conditional. They are instantiated only if the condition is met, e.g., a data hazard exists with another instruction currently in the pipeline.

The following example illustrates the modeling of a small sequence of two instructions as an ILP problem. Consider the following program consisting of two ADD instructions, with a data hazard between them.

```
ADD R1, R1, R2    ; A1
ADD R1, R1, R1    ; A2
```

The ICD of the ADD instruction for the target architecture, assuming a 3-stage pipeline, consists of the following set of constraints.

1: $stages \leftarrow \{IF, EX, WB\}$;
2: *constraints* :
3: $I.IF < I.EX$,
4: $I.EX < I.WB$,
5: $I.IF < (I{+}1).IF$,

6: **if** *I.registers* $\cap$ *(I+1).registers* $\neq \emptyset$ **then**
7: *I.WB* $<$ *(I+1).EX*
8: **end if**

Note that the last constraint is conditional. The condition is used to model a data hazard between the current and the next instruction in the pipeline.

As stated above, we create a new ILP problem instance for each tree in the EXG. This is done by building a directed weighted graph $\mathcal{H} = (S, C)$ called the ***ICD graph***. The nodes S represent the set of variables instantiated by the ICD templates for each instruction in the tree. The instantiated variables represent the time points at which a particular instruction is in a particular stage of the pipeline. The edges C represent the sets of variables corresponding to each instantiated constraint.

The graph $\mathcal{H}$ is weighted so that individual execution times of the pipeline stages can be modeled. The incorporation of the node weights in the generation of the ILP problem is straightforward and is not discussed here.

For generating the ILP problem, each node of a tree is visited and the corresponding ICD template is instantiated. Conditional ICD constraints are instantiated only if the condition is true, e.g., a hazard exists between the current instruction and other instructions in the tree.

After $\mathcal{H}$ has been constructed it is translated to the general ILP formulation (2.1) in a straightforward fashion.

In our running example, the translation process creates the following ILP problem:

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} \mathit{A1.IF} \\ \mathit{A1.EX} \\ \mathit{A1.WB} \\ \mathit{A2.IF} \\ \mathit{A2.EX} \\ \mathit{A2.WB} \end{pmatrix} < \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\text{with } \mathbf{c}^{\intercal} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$$\text{resulting in } \mathbf{x}^{\intercal} = \begin{pmatrix} 1 & 2 & 3 & 2 & 4 & 5 \end{pmatrix}$$

As can be seen in the example, each variable $x \in \mathbf{x}^{\intercal}$ represents a distinct instruction and pipeline stage. The variable's value in the solution is the clock cycle in which the instruction is in the respective pipeline stage. The matrix **A** defines the set of precedence constraints between pipeline stages and instructions. More intricate constraints can be created following well known ILP formulation schemes, see, e.g., [57]. In our example, each stage executes in exactly one cycle, therefore all values in b are simply set to 0. The elements of the cost vector $\mathbf{c}^{\intercal}$ are all set to 1 because there is no priority of one constraint over the other.

The approach is based on the assumption that the internal instruction scheduling of the processor inserts pipeline "bubbles" only if necessary, i.e., an instruction enters a pipeline stage as early as possible.

It is important to retain the context in which a particular section of the program is executed in order to obtain accurate timing information. Instead of considering an ILP formulation obtained from a constraint graph in isolation, apart from the rest of the program, we need to model the context by an additional set of constraints or boundary conditions called the *context prefix* in the sequel.

The context prefix ensures that the timing values and pipeline states of possible entry points from preceding trees are considered when solving the current problem. The set of trees in an EXG is topologically sorted. If we visit the trees in topological order, we can compute timing values based on context prefixes that have already been computed for trees visited earlier. Hence, each context prefix can be represented by the initial concrete constraints associated with the pipeline stages affecting it, but with the bound vector $\mathbf{b}$ updated with timing values already computed for trees in the fan-in. The solution for a given pair of a tree and a context-prefix is used to create a timed graph. This graph is structurally identical to the analyzed tree, however each node is weighted by the delay the instruction produces relative to its predecessor. The overall analysis result $\mathcal{T} = \{\cup \mathcal{T}_i\}$ is pruned by merging isomorphic subgraphs. This is done in an interleaved fashion. The complete analysis can be performed by the following algorithm:

1: **procedure** ANALYSIS($EXG, arch$)
2: $\mathcal{T} \leftarrow$ new $timed\ graph$
3: partition EXG into trees
4: create NOP context prefix for trees with no predecessor
5: **while** some $tree$ not solved **do**
6: **if** $tree$ is ready **then**
7: create constraint graph $\mathcal{H}$ of $tree$
8: **for all** context prefixes P of $tree$ **do**
9: $time \leftarrow$ solve ($P \cup \mathcal{H}$) with ILP-solver
10: create $\mathcal{T}_i$ from $tree + time$
11: merge $\mathcal{T}_i$ with $\mathcal{T}$
12: **end for**
13: **end if**
14: **end while**
15: **return** $\mathcal{T}$
16: **end procedure**

Analyzing a system with advanced microarchitectural features such as caches, out-of-order pipelines and branch prediction, is possible. Branch prediction can be modeled via modifying the EXG to contain paths for either correct and false prediction. For caches, it must be assumed that the pipeline does not progress in the case of a cache miss. Thus, the latency of the LOAD/STORE instruction can be simply incremented by a value provided from an appropriate cache model. Finally, out-of-order pipelines are assumed to still have a fixed order concerning fetch, decode and write-back. The flexible order regarding their execution is modeled by simply omitting the respective precedence constraints.

5.2 Case Study: Local Interconnect Bus (LIN)

Using the proposed technique, we created cycle-accurate program netlists for the industrial LIN of 4.5. As a reminder, LIN is a low-level driver for an automotive communication protocol. We used an industrial implementation of the driver by Infineon Technologies AG, for a LIN master node. The driver was ported to the open-source target Aquarius which implements the SH-2 instruction set architecture. The driver comprises about 1350 lines of C code and 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 by the application. The driver interacts with the LIN bus by means of a UART peripheral providing status, configuration and data registers. The UART is accessible as a memory-mapped I/O device. The driver interacts with the application via shared memory holding receive and transmit data as well as status information. All program netlists generated for the driver support these features.

We compiled the software into two binaries by using two different compiler options for optimization (O0 and O1).

Table 5.1 shows the results for model generation for each of the two binaries. The first three columns show the sizes of the CFG and EXG by number of machine

Table 5.1: Performance evaluation data

SW	CFG size	EXG size	path count	MG time (s)	TA time (s)
lin-o0	1168	4835	44538	26.8	3.9
lin-o1	703	2408	17406	12.4	0.9

TA: timing analysis, MG: model generation

instructions. The third column gives the number of execution paths modeled by the EXG. The last two columns show the CPU times needed for generating the EXG model and computing the cycle-accurate timing information, respectively.

For ILP solving we used the *GNU linear programming kit* (glpk), with default settings. As can be seen, the computational overhead of computing cycle-accurate timing information for the program netlists is small.

Integrating cycle-accurate PN in the HW/SW system is usually not necessary because the uncertainty in timing introduced by the bus system generally supersedes that of the CPU. Yet, the additional timing information can possibly be used in the ACCESS verification method to narrow down the timing variability between two accesses. Our experiments (see. 4.1.3) have already shown that this variability greatly affects the overall proof complexity. The extent to which we may benefit from a cycle-accurate PN strongly depends on the SW application and the predictability of the timing of the bus system.

Finally, it is worthwhile noting that a cycle accurate PN, besides its potential performance benefits, can form the basis for extending firmware verification to properties which rely on cycle-accurate timing.

Chapter 6

Conclusion

This thesis presented *ACCESS*, a novel method for proving the correctness of code transformations which modify a firmware I/O sequence accessing a peripheral device.

ACCESS achieves this by accepting that neither the software nor the peripheral of each instance of the HW/SW-system needs to be equivalent on its own as long as the observable behavior of the whole system is.

Showing this equivalence is rather complex; for the general case too complex to be feasible for realistic designs. ACCESS mitigates this complexity partly by partitioning the proof of global equivalence into a set of localized proofs on the hardware, while including the global context into the analysis using a sound abstraction of the software. The computational complexity of the localized proofs is further reduced by only unrolling until an equivalent hardware state (module timing) is reached in the hardware instead of showing the equivalence of the whole hardware operation. ACCESS defines a unifying verification model which integrates hardware, software and the connecting bus into a single model. The unifying model is designed to allow each component to be modeled at different levels of abstraction. The

supported (sub-)models were discussed and experimentally evaluated.

No significant difference in proof performance could be shown between the two proposed models for abstracting the interconnect. The same holds true for the available software models, i.e., program netlists and source code, with the software model showing only a minor improvement in memory consumption. This improvement is bought with more constraints w.r.t. to the sequence mappings, possibly leading to overall decreased performance. Future work will explore whether this effect becomes more pronounced with increasing program size. In contrast to the other sub-models, having a suitable SystemC-PPA model available for the examined peripheral dramatically reduces the required time for proofing equivalence between two systems. Unfortunately, these models are not (yet) readily available. Defining one and integrating it into the verification environment currently requires considerable effort by the verification engineer, due to lack of compatibility between the available tools.

The potential of the PPA – not only as an effective verification model, but also as a possible bridge of the semantic gap and, hence, as a new design entry point – was recently recognized by the German Federal Ministry for Economic Affairs and Energy (BMWi). This recognition resulted in us receiving the 'EXIST Transfer of Research' grant for the development of a commercial PPA tool and the potential founding of a spin-off.

Our results show that the method scales well for realistic components of small and mid-size computing platforms, as they have gained great popularity in IoT systems. In addition, the proposed method proved effective in identifying subtle bugs in the peripheral HW and undocumented constraints relevant for firmware development. The method would strongly benefit from improved tool support w.r.t. to debugging counterexamples and the integration of SystemC-PPA models.

We have further shown that the availability of an EXG, as defined in previous work [13], allows for a new and efficient formulation of ILP-based timing analysis.

By a combination of EXG generation with ILP-based timing analysis a substantial part of the complexity of timing analysis for software is shifted from ILP solving to SAT solving. In our combined approach, the power of modern SAT solvers is utilized to obtain a precise and compact model of the reachable program paths. We show how this can be used to simplify timing analysis substantially. Augmenting program netlists by precise timing information allows for a more accurate estimate of the software's access behavior as seen by the peripheral. This can potentially reduce the complexity of ACCESS-local equivalence checks or even pave the way towards cycle-accurate formal co-verification by equivalence checking and property checking.

How much this can affect the overall run time of the proposed co-verification has to be determined in future work.

Performing any kind of optimization with confidence requires a suitable method to verify the result. With ACCESS, this thesis provides such a method, enabling to verify the correctness of equivalence transformations defined across the boundaries of hardware and software.

Bibliography

[1] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936.

[2] C. E. Shannon, "A mathematical theory of communication," *Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[3] A. Schrijver, *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., 1986.

[4] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009, vol. 185.

[5] S. A. Cook, "The Complexity of Theorem Proving Procedures," in *Proc. Annual ACM Symposium on the Theory of Computing*, 1971, pp. 151–158.

[6] "IEEE Standard for Property Specification Language (PSL)," *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pp. 1–182, 2010.

[7] "IEEE Standard for System Verilog – Unified Hardware Design, Specification, and Verification Language," *IEEE Std. 1800-2017*, 2018.

[8] Onespin Solutions GmbH, "OneSpin 360 DV-Verify," 2020. [Online]. Avaiable: https://www.onespin.com/products/360-dv-verify, [Acessed: 2020-09-15].

[9] M. D. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz, “Unbounded protocol compliance verification using interval property checking with invariants,” *IEEE Transactions on Computer-Aided Design*, vol. 27, no. 11, pp. 2068–2082, November 2008.

[10] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu, *Bounded Model Checking, Advances In Computers Volume 58*. Academic Press, 2003.

[11] M. Thalmaier, M. D. 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)*, 2010, pp. 176 –181.

[12] C. Villarraga, B. Schmidt, C. Bartsch, J. Bormann, D. Stoffel, and W. Kunz, “An equivalence checker for hardware-dependent software,” in *11. ACM-IEEE International Conference on Formal Methods and Models for Codesign*, 2013, pp. 119–128.

[13] B. Schmidt, C. Villarraga, T. Fehmel, J. Bormann, M. Wedler, M. D. Nguyen, D. Stoffel, and W. Kunz, “A new formal verification approach for hardware-dependent embedded system software,” *IPSJ Transactions on System LSI Design Methodology*, vol. 6, pp. 135–145, 2013.

[14] T. M. McWilliams, “Verification of timing constraints on large digital systems,” in *Proceedings of the 17th Design Automation Conference*, 1980, pp. 139–147.

[15] J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*. Springer US, 2011. [Online]. Available: https://books.google.de/books?id=N9ROkgEACAAJ

[16] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, “The worst-case execution-time problem—overview of methods and survey of tools,” *ACM*

Transactions on Embedded Computing Systems (TECS), vol. 7, no. 3, pp. 1–53, 2008.

[17] F. R. Shapiro, "Origin of the term software: Evidence from the JSTOR electronic journal archive," 2000.

[18] J. W. Tukey, "The teaching of concrete mathematics," *The American Mathematical Monthly*, vol. 65, no. 1, pp. 1–9, 1958.

[19] *ISO / IEC 9899:2018 Information technology – Programming languages – C*, 4th ed., 2018.

[20] LLVM Project, "LLVM Language Reference Manual," 2020. [Online]. Available: https://llvm.org/docs/LangRef.html, [Accessed: 2020-09-15].

[21] S. Diehl, P. Hartel, and P. Sestoft, "Abstract machines for programming language implementation," *Future Generation Computer Systems*, vol. 16, no. 7, pp. 739–751, 2000.

[22] G. M. B. Bailey and A. Piziali, *ESL Design and Verification - A Prescription for Electronic System-Level Methodology*. Systems on Silicon, 2007.

[23] J. Urdahl, D. Stoffel, and W. Kunz, "Architectural system modeling for correct-by-construction RTL design," in *2015 Forum on Specification and Design Languages (FDL)*, Sept 2015, pp. 1–8.

[24] T. Arons, E. Elster, L. Fix, S. Mador-Haim, M. Mishaeli, J. Shalev, E. Singerman, A. Tiemeyer, M. Y. Vardi, and L. D. Zuck, "Formal verification of backward compatibility of microcode," in *Proceedings of the 17th international conference on Computer Aided Verification*, ser. CAV'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 185–198.

[25] 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*, march 2008, pp. 825 –830.

[26] 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*, Oct 2013, pp. 121–128.

[27] R. Mukherjee, M. Purandare, R. Polig, and D. Kroening, "Formal techniques for effective co-verification of hardware/software co-designs," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 35.

[28] D. Currie, X. Feng, M. Fujita, A. J. Hu, M. Kwan, and S. Rajan, "Embedded software verification using symbolic execution and uninterpreted functions," *Int. J. Parallel Program.*, vol. 34, no. 1, pp. 61–91, Feb. 2006.

[29] E. Clarke, D. Kroening, and K. Yorav, "Behavioral consistency of C and Verilog programs using Bounded Model Checking," in *Proceedings of the 40th annual Design Automation Conference*, ser. DAC '03. New York, NY, USA: ACM, 2003, pp. 368–371.

[30] B. Schlich, "Model checking of software for microcontrollers," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 4, pp. 36:1–36:27, Apr. 2010.

[31] J. Urdahl, "Path Predicate Abstraction for Sound System-Level Modeling of Digital Circuits," Ph.D. dissertation, Technische Universität Kaiserslautern, December 2015.

[32] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, 1991.

[33] R. Milner, "Operational and algebraic semantics of concurrent processes," in *Handbook of theoretical computer science (vol. B)*, J. van Leeuwen, Ed. Cambridge, MA, USA: MIT Press, 1990, pp. 1201–1242.

[34] S. Ray and W. A. Hunt, Jr., "Deductive verification of pipelined machines using first-order quantification," in *Proceedings of the* 16*th International*

Conference on Computer-Aided Verification (CAV 2004). Boston, MA: Springer, 2004, pp. 31–43.

[35] P. Manolios and S. K. Srinivasan, "A refinement-based compositional reasoning framework for pipelined machine verification," *IEEE Transactions on VLSI Systems*, vol. 16, pp. 353–364, 2008.

[36] R. Gentilini, C. Piazza, and A. Policriti, "From bisimulation to simulation: Coarsest partition problems," *J. Autom. Reasoning*, vol. 31, no. 1, pp. 73–103, 2003.

[37] K. Hao, S. Ray, and F. Xie, "Equivalence checking for behaviorally synthesized pipelines," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 344–349.

[38] E. Cohen, W. Paul, and S. Schmaltz, "Theory of multi-core hypervisor verification," in *SOFSEM 2013: Theory and Practice of Computer Science*, ser. Lecture Notes in Computer Science, P. van Emde Boas, F. C. A. Groen, G. F. Italiano, J. Nawrocki, and H. Sack, Eds., vol. 7741. Springer, 2013, pp. 1–27.

[39] P. Manolios and S. K. Srinivasan, "Verification of executable pipelined machines with bit-level interfaces," in *Proc. International Conference on Computer-Aided Design (ICCAD)*, 2005, pp. 855 – 862.

[40] J. Urdahl, D. Stoffel, and W. Kunz, "Path Predicate Abstraction for Sound System-Level Models of RT-Level Circuit Designs," *IEEE Transaction On Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2014.

[41] M. Diepenbeck, M. Soeken, D. Große, and R. Drechsler, "Behavior-driven development for circuit design and verification," in *IEEE Intl. High-Level Design Validation and Test Workshop (HLDVT)*, 2012.

[42] R. Drechsler, M. Diepenbeck, D. Große, U. Kühne, H. M. Le, J. Seiter, M. Soeken, and R. Wille, “Completeness-driven development,” in *Graph Transformations*. Springer, 2012, pp. 38–50.

[43] R. Stahl, D. Mueller-Gritschneder, and U. Schlichtmann, “Driver Generation for IoT Nodes With Optimization of the Hardware/Software Interface,” *IEEE Embedded Systems Letters*, vol. 12, no. 2, pp. 66–69, 2020.

[44] T. Ludwig, M. Schwarz, J. Urdahl, L. Deutschmann, S. Hetalani, D. Stoffel, and W. Kunz, “Property-Driven Development of a RISC-V CPU,” in *Proc. Design and Verification Conference United States (DVCON-US)*, 2019.

[45] J. Urdahl, S. Udupi, T. Ludwig, D. Stoffel, and W. Kunz, “Properties first? a new design methodology for hardware, and its perspectives in safety analysis,” in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2016, pp. 1–8.

[46] T. Ludwig, J. Urdahl, D. Stoffel, and W. Kunz, “Properties First – Correct-By-Construction RTL Design in System-Level Design Flows,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, June 2019.

[47] ARM Ltd., “AMBA Overview,” 2020. [Online]. Avaiable: https://developer.arm.com/architectures/system-architectures/amba, 2020, [Acessed: 2020-09-15].

[48] O. Keszocze and I. G. Harris, “Chatbot-based assertion generation from natural language specifications,” in *2019 Forum for Specification and Design Languages (FDL)*. IEEE, 2019, pp. 1–6.

[49] T. Aitch, “Aquarius: a pipelined RISC CPU,” 2003. [Online]. Available: http://opencores.org/project,aquarius, [Accessed: 2020-09-15].

[50] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse, and L. Benini, “PULP: A parallel ultra low power

platform for next generation IoT applications," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, Aug 2015, pp. 1–39.

[51] *ISO 17987-1: Road vehicles - Local Interconnect Network (LIN) - Part 1: General information and use case definition*, 1st ed., 2016.

[52] C. Villarraga, B. Schmidt, B. Bao, R. Raman, C. Bartsch, T. Fehmel, D. Stoffel, and W. Kunz, "Software in a hardware view: New models for HW-dependent software in SoC verification and test," in *2014 International Test Conference*, Oct 2014, pp. 1–9.

[53] X. Li, T. Mitra, and A. Roychoudhury, "Accurate timing analysis by modeling caches, speculation and their interaction," in *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*, June 2003, pp. 466–471.

[54] X. Li, A. Roychoudhury, and T. Mitra, "Modeling out-of-order processors for software timing analysis," in *25th IEEE International Real-Time Systems Symposium*, Dec 2004, pp. 92–103.

[55] J. Reineke and R. Wilhelm, *Static Timing Analysis – What is Special?* Springer International Publishing, 2016, pp. 74–87.

[56] H. M. Nyew, N. Onder, S. Onder, and Z. Wang, "Verifying micro-architecture simulators using event traces," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ser. ICS '14. New York, NY, USA: ACM, 2014, pp. 323–332.

[57] S. P. Bradley, A. C. Hax, and T. L. Magnanti, *Applied Mathematical Programming*. Addison-Wesley, 1977, ch. Integer Programming, pp. 272 – 319.

Kurzfassung in Deutsch

Durch kontinuierlich steigende Anforderungen an Rechneranwendungen im Bereich des Internet of Things (IoT) ergeben sich neue Möglichkeiten im Technologiefeld eingebetteter Systeme. Gleichzeitig steigt hierdurch jedoch der bereits bestehende Anforderungsdruck im Entwurfs- und Produktionsprozess. Eine Vielzahl der durch IoT-Technologien angestrebten Ziele sind nur dann realisierbar, wenn die zur Umsetzung notwendigen Geräte günstige Entwurfs- und Produktionskosten mit hoher Effizienz vereinen.

Eine verbreitete und bewährte Strategie zur Reduktion der Entwurfskosten liegt in der Wiederverwendung bereits existierender Hardware- und Softwarekomponenten. Dies führt jedoch häufig dazu, dass die engen Ressourcenbeschränkungen des Projektes aufgrund nicht optimal aufeinander abgestimmter Komponenten nicht eingehalten werden können.

Ein signifikanter Anteil der Chipfläche eines einebetteten Systems wird von dessen Speicher eingenommen. Desto mehr das System auf Software zur Implementierung seiner eigentlichen Funktion setzt, desto größer ist die benötigte Chipfläche für Speicher in Relation zum Rest des Systems. Dies hat somit unmittelbaren Einfluss auf die Produktionskosten sowie den Energieverbrauch des Systems. Eine Möglichkeit zur Reduktion des benötigen Programmspeichers eines Systems liegt darin, die Schnittstellen zwischen Hard- und Software aufeinander abzustimmen, zu optimieren und, falls möglich, zu vereinheitlichen. Solche Schnittstellen werden

i.d.R. in der Software durch Memory Mapped I/O realisiert und auf dedizierte Treiber für das jeweilige Peripheriegerät verteilt.

Eine Vereinheitlichung der Zugriffsmethoden erhöht die Möglichkeit einzelne Teile eines Treibers zur Ansteuerung mehrerer Peripheriegeräte wiederzuverwenden. Dies wirkt sich positiv auf die Größe der Software und somit direkt auf die Größe des benötigten Programmspeichers aus. Weiterhin kann eine Optimierung der verwendeten Zugriffssequenzen zu einer Reduktion des Energiebedarfs führen.

Allerdings gilt es hierbei auszuschließen, dass durch solche Veränderungen das funktionale Verhalten des Systems beeinflusst wird. Veränderungen an der Kommunikationsschnittstelle zwischen Hardware und Software führen per Definition dazu, dass sich, in Isolation betrachtet, weder Hard- noch Software äquivalent zur Originalversion verhalten müssen. Berücksichtigt man nun zusätzlich, dass Kommunikation zwischen Hard- und Softwarekomponenten komplexe zeitliche Abfolgen, Protokolle und mögliche Seiteneffekte beachten muss, so erkennt man die Notwendigkeit automatisierter Prozesse zur Umsetzung und Verifikation besagter Optimierungen.

Eine Methode zur Optimierung der Peripherieschnittstellen wird in [43] präsentiert. Die dort vorgestellten Prozesse führen typischerweise zu einer lokalen Veränderung des Ein-/Ausgabeverhaltens der Software, jedoch ohne dabei deren globalen Kontrollfluss zu verändern.

Aus der vorangegangenen Diskussion sollte jedoch ersichtlich sein, dass gängige Verifikationsverfahren, welche Hardware und Software separat betrachten, den Anforderungen zur Verifikation dieser Veränderungen nicht genügen. Es wird ein Verifikationsverfahren benötigt, dass nicht nur die Software, sondern auch deren detaillierten Effekt auf die angesteuerte Hardware berücksichtigt. Für den allgemeinen Fall bedeuted dies einen Beweis der *Hardware/Software Co-Äquivalenz modulo Latenzzeit* über die gesamte Laufzeit der Software. Für realistische industrielle Anwendungen stellt dies jedoch eine nicht zu bewältigende Aufgabe

dar.

In dieser Arbeit wird ein neuer formaler Ansatz zur Bewältigung dieses Problems für Fälle, wie sie durch die Klasse der Optimierungen wie sie von [43] erzeugt werden, diskutiert. Der vorgeschlagene Ansatz nutzt die Lokalität der eingebrachten Änderungen zur Partitionierung des Gesamtproblems in handhabbare Teilprobleme aus. Jedes Teilproblem der Partitionierung wird unter dem Kontext der bis dahin möglichen Programmhistorie gelöst. Durch eine Überapproximation des möglichen Zustandsraums des Peripheriegeräts zu Beginn eines lokalen Beweises, können diese Einzelbeweise verkettet und die Äquivalenz des Gesamtsystems als Komposition der Äquivalenz der Einzelbeweise geschlossen werden.

In ACCESS wird hierfür ein HW/SW-Modell des zu verifizierenden Systems vorgestellt, welches jeweils ein Modell der Hardware, der Software und eine Abstraktion des zwischengeschalteten Bussystems integriert. Für jede dieser integrierten Komponenten werden mögliche Modellierungen auf unterschiedlichen Abstraktionsebenen vorgestellt. Der jeweilige Effekt dieser Modellierungen auf ACCESS, sowie die allgemeine Durchführbarkeit und Leistungsfähigkeit der Verifikationsmethode, wird anhand dreier Fallstudien sowie mehrerer Leistungstests untersucht.

Die Ergebnisse zeigen die Anwendbarkeit von ACCESS für realistische Anwendungen im üblichen Größenbereich von IoT-Anwendungen. Zusätzlich hat sich ACCESS als besonders effektiv im Auffinden subtiler Bugs der Peripheriehardware erwiesen.

Es konnte kein signifikanter Effekt der vorgestellten Modellierungsebenen des Bussystems, sowie der Software, auf die Skalierbarkeit und Leistungsfähigkeit der Methode nachgewiesen werden. Die Verwendung eines SystemC-PPA Models gegenüber des RTL Models der Hardware zeigte hingegen deutliche Verbesserung in der benötigten Zeit zur Durchführung der Beweise.

Die Methode würde somit stark von einer besseren Integration von SystemC-PPA in kommerzielle Software-Tools, bzw. eine besser auf diese abgestimmte Modellierung der SystemC-PPA Modelle profitieren.

Abschließend wird eine optionale Methode zu zyklenakkuraten Timing-Analyse auf Basis von Programmnetzlisten vorgestellt. Die Ergebnisse dieser Methode können zu einer verbesserten Konfiguration der Beweisparameter beitragen und somit den für ACCESS benötigten Rechenaufwand positiv beeinflussen.

Lebenslauf

Persönliche Daten

Name:	Michael Schwarz
Geburtsort:	Pirmasens, Rheinland-Pfalz, Deutschland
Staatsangehörigkeit:	deutsch
Familienstand:	ledig

Berufserfahrung

06/2021 – heute	Mitgründer und CTO LUBIS EDA GmbH
02/2015 – heute	Wissenschaftlicher Mitarbeiter Lehrstuhl „Entwurf informationstechnischer Systeme“ Technische Universität Kaiserslautern
02/2013 – 11/2014	Wissenschaftliche Hilfskraft Lehrstuhl „Entwurf informationstechnischer Systeme“ Technische Universität Kaiserslautern
07/2010 – 04/2013	Werksstudent Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI) GmbH Abteilung „Innovative Fabriksysteme

Ausbildung

10/2012 – 11/2014	Studium der Elektro- und Informationstechnik (Master) an der Technischen Universität Kaiserslautern Schwerpunkt Embedded Systems / Eingebettete Systeme
09/2011 – 12/2011	Auslandssemester an der School of Informatics der University of Edinburgh, Schottland
04/2009 – 11/2012	Studium der Elektro- und Informationstechnik (Bachelor) an der Technischen Universität Kaiserslautern Schwerpunkt Embedded Systems / Eingebettete Systeme
08/1999 – 03/2008	Immanuel-Kant-Gymnasium in Pirmasens

Wehrdienst

04/2008 – 12/2008	Grundwehrdienst in 6./292 JgBtl in Messstetten / Baden Württemberg