Coverage of Compositional Property Sets for Hardware and Hardware-dependent Software in Formal System-on-Chip Verification

Coverageanalyse von kompositionellen Eingenschaftensätzen für Hardware und Hardwarenahe Software bei der formalen System-On-Chip-Verifikation

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

von
Dipl.-Ing. Binghao Bao
geb. in Nei Mongol, VR China

D386

Dekan:	Prof. Dr.-Ing. Ralph Urbansky
Gutachter:	Prof. Dr.-Ing. Wolfgang Kunz Technische Universität Kaiserslautern Prof. Dr.-Ing. Ulrich Heinkel Technische Universität Chemnitz
Datum der Disputation:	6. April 2017

Abstract

Divide-and-Conquer is a common strategy to manage the complexity of system design and verification. In the context of System-on-Chip (SoC) design verification, an SoC system is decomposed into several modules and every module is separately verified. Usually an SoC module is reactive: it interacts with its environmental modules. This interaction is normally modeled by environment constraints, which are applied to verify the SoC module. Environment constraints are assumed to be always true when verifying the individual modules of a system. Therefore the correctness of environment constraints is very important for module verification. Environment constraints are also very important for coverage analysis. Coverage analysis in formal verification measures whether or not the property set fully describes the functional behavior of the design under verification (DuV). if a set of properties describes every functional behavior of a DuV, the set of properties is called *complete*. To verify the correctness of environment constraints, *Assume-Guarantee Reasoning* rules can be employed. However, the state of the art assume-guarantee reasoning rules cannot be applied to the environment constraints specified by using an industrial standard property language such as SystemVerilog Assertions (SVA).

This thesis proposes a new assume-guarantee reasoning rule that can be applied to environment constraints specified by using a property language such as SVA. In addition, this thesis proposes two efficient plausibility checks for constraints that can be conducted without a concrete implementation of the considered environment. Furthermore, this thesis provides a compositional reasoning framework determining that a system is completely verified if all modules are verified with Complete Interval Property Checking (C-IPC) under environment constraints.

At present, there is a trend that more of the functionality in SoCs is shifted from the hardware to the hardware-dependent software (HWDS), which is a crucial component in an SoC, since other software layers, such as the operating systems are built on it. Therefore there is an increasing need to apply formal verification to HWDS, especially for safety-critical systems. The interactions between HW and HWDS are often reactive, and happen in a temporal order. This requires new property languages to specify the reactive behavior at the HW and SW interfaces.

This thesis introduces a new property language, called Reactive Software Property Language (RSPL), to specify the reactive interactions between the HW and the HWDS. Furthermore, a method for checking the completeness of software properties, which are specified by using RSPL, is presented in this thesis. This method is motivated by the approach of checking the completeness of hardware properties.

Acknowledgments

I would like to express my gratitude to all those who helped me during writing of this dissertation. I gratefully acknowledge the help of my supervisor, Prof. Wolfgang Kunz, who has taken me as a PhD student at the University of Kaiserslautern. I thank him for many fruitful advices, for his support and for his motivation.

Many thanks to Prof. Ulrich Heinkel for his interest in reviewing my thesis. I also thank Prof. Gerhard Fohler for chairing my thesis committee.

I am also indebted to Jörg Bormann for many valuable discussions and for his patience. I would like to especially thank Sven Beyer, who supported me to solve many issues when performing the completeness analysis of the property set of a design.

I also owe a special debt of gratitude to Dominik Stoffel for the proofreading of this dissertation in great detail, and he provided me a lot of valuable comments, suggestions and corrections. I would also like to thank the collaboration with him in the past five years.

I am grateful to Markus Wedler, with whom I wrote my first technical paper, and I would like to thank him for many comments and corrections in that work.

I would like to thank Carlos Villarraga and Bernard Schmidt for many discussions and brainstormings about the property language for hardware-dependent software.

Many thanks to Max Thalmaier for his proofreading of this dissertation, and many thanks also to Oliver Marx for his help in proofreading the German summary of this dissertation.

I also would like to thank my colleagues in the electronic design automation group at the University of Kaiserslautern, especially Joakim Urdahl, Shrinidhi Udupi, Sasha Loitz, Andreas Christmann, Matthias Legrom and Carmen Vicente-Fess. With them I had a great time, and many thanks for their support.

Finally, I would like to thank my parents and my wife Xiaojing, they always encouraged me and supported me during the past years.

To my family

Contents

Chapter 1

Introduction

In the past decades, semiconductor technology has evolved to be able to integrate all necessary components required for a computer into a single chip. Such a chip, known as a System-on-Chip (SoC), consists of complex hardware (HW) and embedded software (SW). Like other integrated circuits, the SoC hardware must be extremely reliable and correct, because errors in hardware may require a huge amount of time for re-engineering, and millions of dollars. Although correcting errors in embedded software is usually only a matter of re-compiling the software program and re-refreshing the memory of the targeted hardware, SoC embedded software must be extremely reliable and correct as well, especially for safety-critical systems, because errors in such systems may cost not only money, but also human lives. However, advances in verification techniques have not been able to keep pace with the growing complexity of SoC systems. In today's SoC design and verification flow, about 30% of the effort is put into chip design, and about 70% into the verification of the correctness of a chip. Therefore, to guarantee the correctness of SoC systems and to speed up the verification process, various verification techniques are used, and any emerging technique is also welcome, if it can raise the quality of SoC systems or accelerate the verification process.

When verifying an SoC, the most important and most difficult parts to prove are the interfaces (HW/HW interfaces and HW/SW interfaces). An interface serves as a communicator between two components (HW/HW or HW/SW) of a system, and it could be as simple as a point-to-point connection or as complex as a bus system obeying a communication protocol, where huge amounts of data pass through the bus. In general, the interactions between two components over an interface are reactive. This leads to a situation where, in principle, verification of the behavior at the interface must consider the two components as a whole. If the two components further interact with other components, then in the worst case the entire system must be included to prove the interface. This leads to highly complex verifications that require a huge amount of computation resources and take too much time. Assume-guarantee reasoning in "Divide-and-Conquer" settings is one of the formal methods for validating the interfaces of a system. Methods based on assume-guarantee reasoning conduct the interface verification on the individual components rather than on their sum. In addition, assume-guarantee reasoning needs a

formal description of the interfaces. Formalizing the interfaces yields many advantages: For example, when integrating two Intellectual Property (IP) blocks, a formal description, often provided by IP vendors as a documentation, can eliminate ambiguity about the interface behavior; for design verification, formal interface descriptions can be applied as assumptions or assertions of the components of a system.

The present dissertation presents a new assume-guarantee rule for validating the interfaces of a SoC system (Chapter 4). Derived from it, a set of compositional criteria is introduced for guaranteeing that every functionality of a system is covered by the union of the property sets of the individual components of the system (Chapter 5). Furthermore a property language is contributed, which is used to formally specify the HW/SW interfaces and the hardware-dependent software behavior (Chapter 6). The motivation of these contributions is discussed in greater detail in Section 1.2. The following section overviews the state-of-art approaches applied to verify the functional correctness of SoC systems.

1.1 Design Verification of Systems-on-Chip (SoC)

Due to the high cost of correcting errors in a HW system, a chip must be verified rigorously before being shipped to customers. Design verification is a difficult task, particularly for an SoC, in which many functional units, implemented in HW or in SW, are integrated. Furthermore these functional units usually interact in a very complex manner. At the design phase of an SoC system, functional verification verifies whether a design is implemented as wanted. It includes digital functional verification, analog verification, static timing analysis, etc. Manufacturing verification, also called *test*, examines whether a HW device is manufactured as wanted. This dissertation focuses on digital functional verification. Approaches to functional verification can be classified into static methods, such as formal verification, and dynamic methods, such as simulation. This section reviews their usage for HW verification and SW verification, respectively.

1.1.1 Digital Hardware Verification

Simulation is a common method for verifying a HW design. In general, a (HW) design under verification (DuV) is composed with an environment called a *testbench*, and they all together are synthesized into a simulation model, which is further handled by a simulator. The testbench is responsible for providing the input sequences (test cases) of the DuV and to collect the response of the DuV. For an industrial design, it is not realistic to feed every input sequence into the DuV, therefore how to choose the test cases is especially important for guaranteeing the correctness of SoC systems. For *directed simulation*, the test cases are given by verification engineers manually; every specified test case is meant to cover one aspect of the overall functionality of the DuV, or to show the absence of some erroneous behavior. However, for today's SoC systems it is not feasible to generate every input sequence manually. *Constrained random simulation* uses constraints for the input signals of a DuV. Usually a constraint specifies the interface protocol of the input signals

of the DuV. Randomized input stimuli are generated by a constraint solver, which only produces the input sequences satisfying the constraints. The testbench for constrained random simulation can be reactive. In other words, the response of the design can be used to guide the constraint solver to generate input sequences toward covering untested behavior. Constrained random simulation can automatically create more test cases than directed simulation. However, for complex SoC systems, constrained random simulation still can not examine every input sequence in the limited project time and computation resources. To overcome this problem, formal verification has been adopted in the design flow either as a complement to simulation or an alternative to it. In contrast to simulation, formal verification takes care of every input sequence of the DuV: it is like an exhaustive simulation, but it is much faster than exhaustive simulation. In the SoC design flow, formal verification is mainly applied to perform two tasks: *equivalence checking* and *property checking*.

Equivalence Checking

Equivalence checking is applied to verify the equivalence of two models of a design. One of the models, called the *golden design*, is thought to be correct. The other model is called the *revised design*. Checking the equivalence of two sequential circuits (sequential equivalence) is not an easy task; theoretically it needs to traverse the state space of the product machine of two circuits, which has a high computational complexity. In the last decades many formal equivalence checking techniques for sequential circuits have been developed [1–5], in order to overcome the complexity problem of sequential equivalence. For instance, [1] employs the structural similarity of two circuits to reduce the sequential equivalence checking problem to a combinational equivalence checking problem: if two circuits have the same encoding of states, by relating the state variables of two circuits, then it only needs to check the equivalence of the transition function and output function of two circuits, for example by using methods based on satisfiability solving (SAT) [6] or Boolean decision diagrams (BDDs) [7]. Relating the state variables of two circuits is also called *state matching*. In practice, state matching is conducted either by automatically relating the variables having the same names or by manual inspection. [5] presents a method to verify the equivalence of two circuits, whose state encoding is not fully identical due to design optimizations performed by synthesis tools. Examples of such optimizations are retiming and inverter pushing.

Equivalence checking is already a standard component in today's industrial design flows. It is used to verify the functional equivalence between the Register-Transfer-Level (RTL) design and the synthesized Gate-Level netlist, which may contain bugs introduced by synthesis tools.

Property Checking

Property checking, often also referred to as *model checking*, is an automatic proof method applied to verify whether an implementation of a design satisfies the specification of the

design. In order to perform property checking, the DuV is required to be modeled by a formalism. The known models widely used in model checker are the Kripke model [8] and the finite state machine (FSM) model, the latter will be explained in detail in Section 2.4.

Another input for a property checker is the specification of the design. Usually the specification of a design is non-formal, therefore a formal language is needed to formalize the specification in terms of *properties*. Such a language is called a *property specification language*. In practice, it is not feasible to specify every behavior within a single property, otherwise this property would become too complex to be handled by a property checker, and too complex to be managed by verification engineers. Consequently, a property specifies only an aspect of the design. In addition, a property may describe the design behavior ranging over many clock cycles. Such kinds of behavior must be supported by a *temporal* property language. The most famous temporal languages are computation tree logic (CTL) and linear temporal logic (LTL) [8, 9]. Although CTL and LTL have great expressive power, they are too difficult to use in practice. Hence, standardized languages such as Property Specification Language (PSL) and SystemVerilog Assertion (SVA) have been developed [10, 11], which are user-friendly and easily adopted by the industry.

Once the model of a design and the properties of the design are in hand, it is necessary to have a proof method to verify the properties. *Classical model checking* [12] and *symbolic model checking* [13] are based on algorithms that traverse the state space of a design. Classical model checking explicitly represents the states and the transitions between states. Therefore, classical model checking suffers from the *state explosion* problem [14]. Symbolic model checking, in contrast, applies Boolean functions to symbolically represent the states and the state transitions of a design. In a symbolic model checker, BDDs are used as the data structures to represent Boolean functions. In some cases the size of the BDDs grows exponentially with the number of variables, and the process of manipulating the BDDs requires huge memory as well [15]. This limits the performance of symbolic model checking. One of the methods to handle larger designs is abstraction-refinement [16]. Methods using abstraction-refinement start from the most abstract model of the design to prove a property. In the case of a holding property, it can be concluded that the property is also valid on the concrete model of the design. If a property fails, it needs to simulate the counterexample on the concrete design to determine whether the counterexample is spurious or relates to a real design bug. A spurious counterexample is applied to refine the abstract model. The property then is verified on the refined model of the design. This process repeats until a property is proven or a bug in the design/property is found. Abstract models can also be useful to verify designs by using simulation. For instance, to speed up a simulation, abstract models such as Transaction Level Modeling (TLM) [17] and Electronic System Level (ESL) Model [18] are used to prove the design behavior at a relatively high description level of the design.

Bounded model checking (BMC) [19] and Interval Property Checking (IPC) [20] employ SAT-solvers as their proof engines. BMC and IPC unroll the DuV into a combinational logic model, then the Boolean formulas of this combinational logic model and the property to be verified are handled by a SAT-solver. Note that not the property itself, but its negation is given to the SAT-solver. To put this another way, if the SAT-solver returns

UNSAT, then the property holds. BMC proves the bounded validity of a property starting from the initial state of the design. Therefore, for industrial designs, BMC is mostly applied as a bug hunter, whereas IPC proves the complete validity of a property starting from arbitrary states of the DuV, which may lead to *false negatives*. To overcome the problem of false negatives with IPC, invariants of the design can be applied. Such invariants can be constructed manually by inspecting the RTL code of the design, or can be generated automatically as presented in [20] and [21]. Furthermore, unlike IPC, which uses invariants only as *reachability constraints*, in the context of the SAT-based verification paradigm, invariants can also be used to prove a property. Methods like Property Driven Reachability (PDR) [22] and interpolant-based verification [23] generate invariants stepwise until the generated invariants can prove a property.

1.1.2 Software Verification and Hardware/Software Co-Verification

Methods for ensuring software correctness can be classified into *dynamic program analysis* and *static program analysis*.

Dynamic program analysis examines a software program by executing the program under test on the targeted hardware platform or on a (software/hardware) emulator of it. Most simulative approaches are dynamic. Similar to hardware simulation, for *dynamic testing* an executed program is fed by test cases. The outputs of the program are then collected and examined. In addition, some code coverage criteria are applied to improve the quality of the test cases.

Static methods analyze programs without executing them. The simplest static analysis tools, such as for syntactic analysis, have been integrated into software program compilers already for a long time. They analyze programs at compile time. Unlike the dynamic methods, which analyze the software program by explicitly enumerating input test cases, symbolic execution (a static method) takes symbolic input values instead of concrete input values. Then the symbolic values of the program's output are computed. Formal verification is a static approach. It is used to find functional errors of a software program, which could be something simple like "out of boundary access" or something complex like a wrongly implemented functionality. Since modern software systems are getting more and more complex, finding errors in software only by using simulative methods is no longer sufficient: there is an increasing need to involve formal methods to ensure that the software is free of bugs. There has been a lot of research into software model checking [24–30]. Some of these approaches model software programs as finite-state systems. Such finite-state systems are either represented explicitly, or symbolically in BDDs. Then the algorithms from classical model checking or symbolic model checking are applied to traverse the finite-state systems for the verification of the software programs. SAT-based methods such as [27–29] aim to transform the software programs to Boolean formulas, then the resulting formulas, joined with the formula representing properties, are handled by a SAT-solver. A good summary of software model checking can be found in [31].

However, all the above mentioned research attempts to validate application software written in a high-level programming language such as C/C++. For embedded software, es-

pecially for hardware-dependent software (low-level software), the aforementioned methods are no longer suitable. Hardware-dependent software interacts intensively with its controlled hardware. It is therefore very important to guarantee the correctness of this reactive behavior, and this sophisticated reactive behavior is also very difficult to verify.

In industrial embedded design flows, an *integration test* is responsible for identifying errors at the interfaces between the hardware and the software. Nevertheless, the method of integration test is still dominated by the simulative methods in the industry. There is some research that applies formal methods to co-verify the hardware and the software. The work of [32] convert the software to a hardware module by storing the program in a read-only-memory (ROM). Then the joint model of the processor and the ROM is handled by a symbolic model checker. In contrast, [33] unrolls this joint model to several instances with respect to the time points identified by the clock ticks of the hardware; then the unrolled model is analyzed by a BMC tool. The work of [34] also unrolls this joint model, however, it unrolls the joint hardware model only for a few time points and proves the properties that specify only local behavior on the unrolled model. Such local properties can be seen as an abstraction of the unrolled model. Later, the local properties are used to prove the properties that describe global behavior ranging over hundreds of clock cycles.

Methods based on symbolic execution, such as [35–39], need to explicitly enumerate every execution path of a program to prove a safety property. In the present dissertation, a method similar to symbolic execution is adopted [40]. However, here the symbolic execution is only applied to create the computation model of a software program called *program netlist*. The properties are proven by a SAT-solver, which implicitly enumerates every possible path of the software. Strictly speaking, the method presented in this dissertation is not co-verification of hardware and software, because a program netlist is obtained by using the instruction set architecture (ISA) model of the processor instead of its time-accurate model. The program netlist, however, contains not only the functional behavior of a hardware-dependent software program, but also the input/output (I/O) accesses to the controlled HW peripherals and the correct ordering of these I/O accesses. In this way, it also takes into account how the software interacts with hardware. A detailed explanation of the program netlist can be found in Section 6.2.

1.2 Motivation and Overview

During the simulation of a design, people want to know to what extent the behavior of the DuV is explored. For this purpose, several coverage criteria are introduced that evaluate the quality of the input patterns. One coverage criterion worth mentioning is functional coverage, which evaluates how many times a property, such as an SVA assertion, is triggered by test cases. Note that the coverage criteria for simulation are only applied to evaluate the quality of input test cases. However, formal verification takes care of every possible input scenario, and so the coverage criteria from simulation are not applicable. For formal verification, a verification engineer may be interested in “When should

I stop writing properties?" To this end, a functional coverage criterion for property sets is introduced for formal property checking, which tests whether a property set is *complete*, in other words, it measures whether a property set covers every functionality of the DuV [41]. If the property set of a module is proven to be complete, then the module is said to be proven *completely*.

On the other hand, even with the growing advances in verification techniques, which can handle bigger and bigger designs, Divide-and-Conquer is still a popular and intuitive method to manage the complexity of DuVs, so that the verification engineer can focus on one of the individual design modules at a time. When verifying individual modules of a DuV, in order to save verification time and resources, the module's inputs have to be constrained by using environment constraints. Environment constraints exist implicitly during design simulation, where only valid input traces are fed into the DuV. For constrained-random simulation and for formal verification, environment constraints are given as (temporal) formulas that describe the (reactive) behavior between the DuV and its environment. Similarly, to avoid writing unnecessary properties that cover invalid input scenarios, environment constraints are also employed in the coverage analysis of property sets.

Once the environment constraints are applied to verify a design, one needs to ensure that the constraints are correct. In industry, these constraints are inspected manually, which is error prone and time consuming. On the other hand, since the individual modules of an SoC are usually developed simultaneously, it may not be possible to check the constraints against the environment of a module before integration. Detecting errors in constraints at that late stage of the design process, however, requires a step back into module verification and may compromise the project closure. In order to overcome this bottleneck in the flow of the verification, two efficient plausibility checks for constraints are introduced in this dissertation. These two checks can be conducted on the constraints of a module without a concrete implementation of the module's environment.

Although the presented plausibility checks give verification engineers more confidence in their constraints, it is still necessary to have an automatic method that can precisely verify constraints. Environmental constraints are formal descriptions of interfaces (HW/HW, HW/SW, SW/SW), and most environment constraints are reactive, which means the correctness of the environment constraints depends on their own correctness. This leads to a situation in which proving the constraints needs theoretically to consider the module and its environment as a whole. This violates the intent of applying Divide-and-Conquer approaches. Methods based on assume-guarantee reasoning overcome this issue by reasoning only over the constraints/commitments of individual modules, where the commitment of a module is the interface behavior exposed to the module's environment. Since these constraints/commitments are often abstractions of modules and they are significantly smaller than the modules themselves, assume-guarantee reasoning can be very efficient.

There has been much research into assume-guarantee reasoning for systems modeled either as Moore machines or as Mealy machines [42–47]. An important contribution of this dissertation is that an assume-guarantee reasoning scheme is proposed that allows

for applying one of the standard property specification languages such as SVA or PSL to specify the interface behaviors (constraints/commitments) of the modules. Using standard property languages makes it possible for compact and easy-to-read constraints/commitments of modules to be formulated instead of constructing automata for them. In addition, verification engineers do not need to learn new languages for specifying the interfaces. An interface description written in, e.g., SVA can be re-used either as an assumption or as a commitment for module verification, and the same description can also be re-used as an assertion for chip level verification. Using standard property languages to formulate constraints runs the risk of introducing circular reasoning for assume-guarantee-reasoning. This has not been addressed by research to present, and will be discussed in greater detail in Chapter 4. To resolve the issue of circular reasoning, this dissertation presents a new assume-guarantee rule with the aid of the aforementioned plausibility tests. This rule can guarantee the validity of the environment constraints and the validity of the properties for individual modules of the entire system, where the constraints and properties are written in a standard property specification language like SVA or PSL.

Furthermore, since the coverage analysis of the property sets is conducted module by module, environment constraints are used to avoid writing unnecessary properties. When every module of a system is completely verified, it is all-important to ensure that the union of the property sets of the individual modules completely verifies the system. For this purpose, this dissertation provides a couple of compositional rules derived from the new assume-guarantee rule presented in Chapter 4.

Interface verification is crucial for software verification as well. The HW/SW interface is particularly difficult to prove, because the reactive temporal behavior dominates there. This requires new modeling techniques for hardware-dependent software. Aside from that, it is necessary to have a property language that can specify the reactive temporal behavior at an HW/SW interface, and can be easily combined with the software model for efficient model checking. Given the fact that traditional temporal languages such as LTL and CTL are hard to use and to understand, and run-time assertions like the "assert" statements in C can only specify non-temporal behavior, this dissertation presents a new property language for specifying hardware-dependent software, called the *reactive software property language* (RSPL). This dissertation shows that using RSPL it is very convenient for specifying an HW/SW interface and the properties of software programs based on the input sequences and the output sequences of the programs. For software property checking, it is also interesting to know if a software property set is complete. Because the RSPL properties describe the input/output sequences of a hardware-dependent software program, for the sake of checking the completeness of software properties, methods such as [41, 48] from the hardware domain can be adopted, and these methods verify whether the property set uniquely describes every input sequence and every output sequence.

Considering the characteristics of hardware-dependent software, for software completeness checking this dissertation contributes a simpler method than the one used for hardware properties. This dissertation builds a framework for RSPL, that allows users to specify the interfaces and their reactive behavior of a software program. With further syntax extensions, the contributed assume-guarantee rules in Chapter 4 and compositional

rules in Chapter 5 can be applied to guarantee the results of compositional software model checking using reactive environment constraints.

This thesis is organized as follows.

Chapter 2 introduces the fundamentals of graph theory including algorithms for traversing graphs. It also gives an overview of propositional logic, which is essential for modeling combinational circuits, and of finite state machines, which are essential for modeling sequential circuits. In addition, predicate logic is briefly explained, which is basic for property languages.

Chapter 3 gives a summary of state-of-the-art model checking techniques and property specification languages for hardware designs. Furthermore, it presents a detailed explanation of methods for coverage analysis for simulation-based techniques and for property checking.

Chapter 4 gives an overview of various state-of-the-art assume-guarantee rules which are specialized for different models of systems, and specialized for different application scenarios. In that chapter, a novel assume-guarantee rule is introduced that is more general than previous ones, and it addresses the circular reasoning issues arising when applying reactive constraints. To this end, two plausibility checks for environment constraints are developed. They are parts of the newly presented assume-guarantee rule. However, they also can be used standalone to aid a verification engineer's gaining confidence in the constraints at the early stage of a verification process.

Based on the result of Chapter 4, a set of compositional rules is given in Chapter 5. By using them, one may conclude that the entire system is completely verified when the individual modules of the system are completely verified, without performing a coverage analysis for the entire system. Even though the presented work is explained for the case of applying environment constraints during verification, the scenario that does not use any constraint, which is only a special case of this research, is also covered by the presented compositional rules.

Chapter 6 introduces a novel software property language called RSPL, that allows users to specify the reactive behavior of hardware-dependent software. In addition, this chapter contributes a method to create a joint model composed of properties and the software computation model. Furthermore, an approach to prove the completeness of a property set in RSPL is given.

Chapter 7 summarizes the thesis.

Chapter 2

Fundamentals

In this chapter, the basic theoretical background of this dissertation is introduced. Basic notions of *graphs* and some graph-based algorithms are introduced in Section 2.1. *Propositional logic*, which is essential for digital design and verification, is briefly described in Section 2.2. *Predicate logic* extends propositional logic in terms of expressive power and will be discussed in Section 2.3. Designs with sequential behavior are usually modeled by *Finite State Machines (FSMs)*, which are defined in Section 2.4.

2.1 Graphs

A *graph* is a data model that visualizes binary relations between objects. Many technical problems can be represented by graphs and then be solved by efficient graph-based algorithms. For instance, a Boolean function can be represented by a graph known as a *Reduced Ordered Binary Decision Diagram (ROBDD)* [49–51]. It is a compact representation for Boolean functions and can be found in use in many electronic design automation tools, especially in formal model checking tools. A *control flow graph* of a software program is a graph modeling the control structure of the software program [52]. In this section the fundamentals of graph theory and the algorithms for traversing graphs are introduced.

2.1.1 Fundamentals of Graph Theory

A graph is composed of a set of *vertices* or *nodes* and a set of *edges*. Some graphs may include labels, which are attached to the vertices and edges. This section discusses only graphs without labels.

Definition 1 (Graph)**.** *A graph is defined to be an ordered pair $G = (V, E)$, where V is a non-empty set of vertices and E is a set of edges.*

Every edge is defined by the vertices connected by it. Given two arbitrary vertices v_1, $v_2 \in V$, a directed edge from v_1 to v_2 is defined by a relation $(v_1, v_2) \in E \subseteq V \times V$. An

undirected edge is indicated by a set $\{v_1, v_2\}$ of vertices. A directed graph only contains directed edges, whereas an undirected graph only contains undirected edges.

Definition 2 (Direct predecessors and direct successors)**.** *Suppose that (v_1, v_2) is an edge in a directed graph $G = (V, E)$. The vertex v_1 is defined to be a* direct predecessor *of v_2, and v_2 is said to be a* direct successor *of v_1. An edge that departs from and ends at the same vertex is called a* loop*.*

Definition 3 (Paths)**.** *A path from a vertex x to a vertex y in a directed graph $G = (V, E)$ is a list of vertices $(v_1, v_2, \ldots, v_k)$ such that all vertices are distinct from one another and it is $(v_i, v_{i+1}) \in E$, $i = 1, 2, \ldots, k-1$ with $x = v_1$, $y = v_k$. The length of this path is $k - 1$.*

A *cycle* is a path whose start vertex and end vertex are the same. A directed graph not containing any cycles is said to be *acyclic* and is called a *directed acyclic graph (DAG)*. An example of a DAG is given in Fig. 2.1, which is an ROBDD representing the Boolean function $f(a, b, c) = a \wedge b \vee c$.

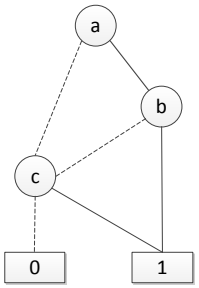


Figure 2.1: An example of a DAG: ROBDD

2.1.2 Algorithms for Traversing Graphs

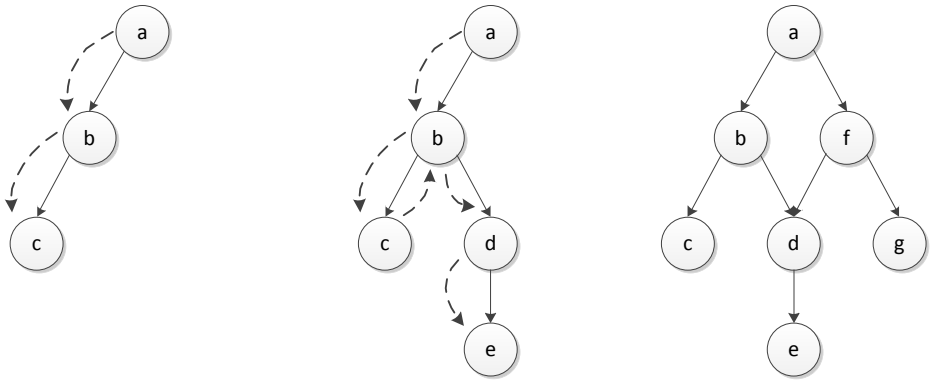
In this section, two basic algorithms for traversing the vertices in a graph are introduced.

Depth-First Search (DFS)

The depth-first search algorithm is a widely used algorithm. For instance, *topological sorting* applies the DFS algorithm to determine the partial order of the vertices in a DAG. Another example is state space exploration for a state machine in model checking by a DFS traversal of the state transition graph of the machine.

Fig. 2.2 shows how the DFS algorithm operates to find all reachable nodes from the root node of a graph. As illustrated in this example, the algorithm tries to find the "farthest" node from the root node as soon as possible, this is the reason why the algorithm is called "depth-first." Intuitively, the algorithm first finds all nodes along an arbitrary path of a graph (Fig. 2.2a). Then it moves back to the last visited node that has unvisited direct

successors. Afterwards it searches an arbitrary unvisited path staring from that node until a terminal node is reached or a visited node is reached (Fig. 2.2b). A terminal node is a node that does not have any direct successors. The process repeats until every node of the graph is found (Fig. 2.2c).



(a) (b) (c)

Figure 2.2: Depth-First Search

The pseudo-code of Algo. 1 specifies how a DFS is implemented with the help of a recursive function. In this function, the set $visited$ indicates the traversed nodes of the graph G and the function $G.E(n)$ computes all direct successors of the current node n. After the program terminates, the set $visited$ contains all reachable nodes from a specified starting node passed by the argument n. Note that the graph G is a DAG. However, with slight modification, this function can be applied to an undirected graph.

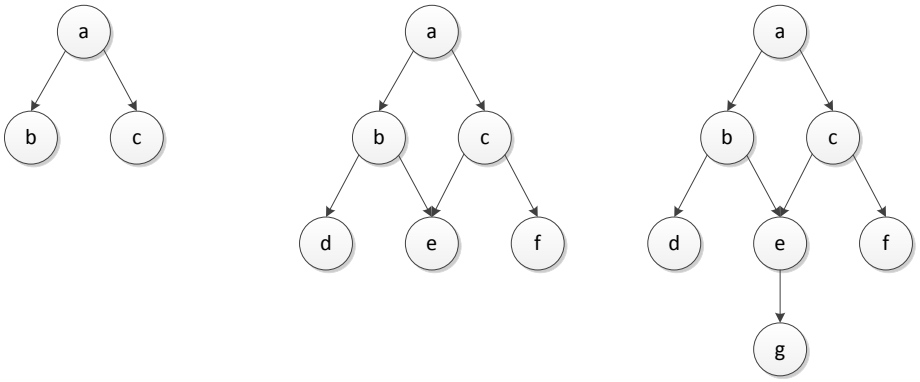
Algorithm 1 recursive DFS function

```
1: visited = ∅
2: function DFS(n, G)
3:     visited = visited ∪ {n}
4:     for all v ∈ G.E(n) do
5:         if v ∉ visited then
6:             DFS(v, G)
7:         end if
8:     end for
9: end function
```

Breadth-First Search (BFS)

In contrast to depth-first search, breadth-first search searches all nodes at step k from a starting node s before it continues to step $k + 1$. It finds its use in many areas, for example, determining the shortest path between two nodes (s, t), or finding classes of connected nodes in a graph.

Fig. 2.3 also illustrates how to find all reachable nodes from the root node of the graph introduced in Fig. 2.2, but this time the breadth-first search method is applied. The algorithm firstly looks for every direct successor of the root node a (Fig. 2.3a), then for every found direct successor j the algorithm seeks every direct successor of j (Fig. 2.3b). The algorithm terminates when every terminal node is reached.



(a) (b) (c)

Figure 2.3: Breadth-First Search

The BFS procedure is described in pseudo-code in Algo. 2 with the help of a fist-in-first-out queue Q. The function $Enqueue(n)$ pushes an element n to the first position of the queue, whereas the function $Dequeue$ pops the element at the first position of the queue.

Algorithm 2 BFS function

```
1:  function BFS(n, G)
2:      Q = ∅
3:      visited = {n}
4:      Q.Enqueue(n)
5:      while Q ≠ ∅ do
6:          v = Q.Dequeue()
7:          for all w ∈ G.E(v) do
8:              if w ∉ visited then
9:                  visited = visited ∪ {w}
10:                 Q.Enqueue(w)
11:             end if
12:         end for
13:     end while
14: end function
```

2.2 Propositional Logic

Propositional logic plays a key role in digital design and verification. As its name suggests, it is a language which people can use to make statements about the relations between propositions. A proposition can have the property that it holds of one of the objects that the language models, or that it is false about that object.

2.2.1 The Syntax of Propositional Logic

Definition 4 (Alphabet)**.** *The alphabet of propositional logic consists of*

- *an infinite set of atomic variables:* $\Phi = v_1, v_2, \ldots,$
- *a set of logical connectives (operators):* $\wedge\,(and), \vee\,(or)\ and\ \neg\,(not),$
- *a pair of parentheses "(" and ")" .*

Definition 5 (Syntax)**.** *The syntax of a* well-formed *formula of propositional logic is inductively defined as follows:*

- *Every atomic variable* $\varphi \in \Phi$ *is a well-formed formula.*
- *If* α *is a well-formed formula, then* $\neg\alpha$ *is a well-formed formula.*
- *If* α *and* β *are well-formed formulas, then so is* $(\alpha \vee \beta)$.
- *If* α *and* β *are well-formed formulas, then so is* $(\alpha \wedge \beta)$.
- *No other formula is a well-formed formula.*

In order to improve the readability of propositional formulas, a precedence of the logic operators is defined, so that some pairs of parentheses can be removed from a propositional formula. The operator $\neg$ has the highest precedence, then comes $\wedge$, then $\vee$. Sometimes a propositional formula contains also the constants 0 and 1; they can be trivially defined by $a \wedge \neg a$ and $a \vee \neg a$, respectively. The formula $\alpha \rightarrow \beta$ symbolizes that a proposition α implies a proposition β, which is formally defined by $\neg\alpha \vee \beta$. Based on this description of implication, the equivalence of two propositions $\alpha \leftrightarrow \beta$ can be defined by $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$.

2.2.2 The Semantics of Propositional Logic

Given a set of *truth values* $\mathbb{B} = \{0, 1\}$, a *valuation* $\mathcal{V} : \Phi \rightarrow \mathbb{B}$ of a well-formed propositional formula φ is a function that assigns to every atomic variable occurring in φ a truth value. The value of the formula φ can be determined by repeatedly applying the following definition:

Definition 6 (The semantics of propositional logic)**.**

- $\mathcal{V}(\neg\alpha) = \begin{cases} 0 & \textit{if } \mathcal{V}(\alpha) = 1 \\ 1 & \textit{if } \mathcal{V}(\alpha) = 0 \end{cases}$
- $\mathcal{V}(\alpha \wedge \beta) = \begin{cases} 1 & \textit{if } \mathcal{V}(\alpha) = 1 \textit{ and } \mathcal{V}(\beta) = 1 \\ 0 & \textit{otherwise} \end{cases}$
- $\mathcal{V}(\alpha \vee \beta) = \begin{cases} 0 & \textit{if } \mathcal{V}(\alpha) = 0 \textit{ and } \mathcal{V}(\beta) = 0 \\ 1 & \textit{otherwise} \end{cases}$

The semantics of an arbitrary propositional formula can be viewed as a function that takes the atomic variables as its arguments and computes the truth value of the whole formula based on the values of arguments. Such a function is also called a *Boolean function*. For example, the transition function of a FSM is a Boolean function. The arguments of a Boolean function are usually called the *inputs* of the function, and the return of the function is called the *output* of the function. A Boolean function can be canonically represented by a truth table. A truth table for a function $f : \mathbb{B}^n \to \mathbb{B}$ lists the value of f for all possible 2^n combinations of input values. Fig. 2.4 shows the truth table of the Boolean function $f = a \wedge \neg b$.

a	b	$a \wedge \neg b$
0	0	0
0	1	0
1	0	1
1	1	0

Figure 2.4: Truth table for $a \wedge \neg b$

The size of a truth table increases exponentially with the input size of the function. For a realistic design it is not possible to build a truth table containing all 2^n rows. Alternatively, a Boolean function can be canonically represented graphically by means of an ROBDD [49–51], which is more compact than the truth table of the function. This section does not give details about ROBDDs: the interested reader is referred to the cited papers. A Boolean function can be implemented by a combinational circuit.

Definition 7 (Combinational circuit)**.** *A combinational circuit C is described by a Boolean network, which is an acyclic directed graph $C(V, E)$. The vertices are called gates, which are the basic building blocks of a combinational circuit. The edge set E describes the connections (wires) between the gates.*

- *The gates of $I \subseteq V$ that do not have direct predecessors constitute the set of primary inputs of the circuit. The edges leaving the gates of I are associated with a set of input variables i_j.*

- *The gates of $O \subseteq V, O \cap I = \emptyset$ that do not have direct successors constitute the set of primary outputs of the circuit.*
- *Each gate $g_j \in G \subseteq V \setminus I$ is associated with a gate type which is one of AND, OR or NOT; they represent logic functions whose semantics are defined according to Def. 6 for the respective logical operators $\wedge$, $\vee$ and $\neg$. The edges leaving the gates of G are associated with variables x_j. A subset of X, whose elements are related to the edges going to the gates of O, is the set of output variables o_j.*
- *The variables x_j are also called the internal variables or the internal signals of the circuit. The output variables (signals) are special internal signals, which represent the circuit's behavior that is visible from outside of the circuit.*

At any given instance in time, the outputs of a combinational circuit are only determined by the present input value of the circuit.

The following terminology will be of relevance in later chapters of this thesis:

Definition 8 (Driver)**.** *Given a vertex $v \in V$ in a circuit C, the driver D of v is the set of all predecessor vertices of v: $D(v) = \{v' | (v', v) \in E\}$; D is also called the* Fan-in *of v.*

Definition 9 (Satisfiability)**.** *A propositional formula φ is satisfiable when there exists a valuation of φ such that φ gets the truth value 1.*

Definition 10 (Tautology)**.** *A propositional formula φ is a tautology when φ yields the truth value 1 for every valuation.*

In other words, when a formula is a tautology, its value is always 1 regardless of its valuations.

2.3 Predicate Logic

Propositional logic is one of the simplest logics. It has limited expressive power. For instance, it is not possible to specify a common property of a set of objects. An object can be anything in the *domain of discourse*. *Predicate logic*, also called *first-order logic*, extends propositional logic by means of *quantifiers*, *function and predicate symbols*.

2.3.1 The Syntax of Predicate Logic

Unlike propositional logic, which reasons with atomic variables as a whole, predicate logic fine-grains the atomic variables from propositional logic by means of quantifiers, functions (with or without arguments) and predicates (with or without arguments). A function returns an object based on its input objects passed by *arguments*. A predicate can be treated as a function that returns a Boolean value. An argument is like a *placeholder* that is replaced by a dedicated object whenever the value of a function or a predicate is computed. A function or a predicate could have zero or more than zero arguments. A

function without arguments is also called a *constant*, referring to a specific object. A predicate with zero arguments has either the Boolean value 0 or 1.

Definition 11 (Alphabet)**.** *The alphabet of predicate logic consists of*

- *an infinite set of function symbols* $F = \{f_k | k \in \mathbb{N}\}$,
- *an infinite set of predicate symbols* $P = \{p_k | k \in \mathbb{N}\}$,
- *an infinite set of variables* $V = \{v_k | k \in \mathbb{N}\}$,
- *the quantifiers* $\exists$ *(existential) and* $\forall$ *(universal),*
- *the Boolean connectives* $\wedge$, $\vee$ *and* $\neg$,
- *auxiliary punctuation marks "(","(" and ",".*

Note that any function could have $n \geq 0$ arguments, hence a function is also called an *n-ary* function. Every function symbol f_k can be reused to represent functions with different arguments, and the function symbol f_k representing a function with n arguments is referred to an *n-ary function symbol*. Similarly for n-ary predicate symbols.

Derived from Def. 11, the syntax of predicate logic is defined stepwise in the rest of this section.

Definition 12 (Syntax of predicate logic: Term)**.** *In predicate logic, a term refers to objects reasoned about by the logic. It is defined recursively by:*

- *every variable* $v_i \in V$ *is a term;*
- *for an n-ary* ($n \geq 0$) *function symbol* $f_i \in F$*, if* $t_0, t_1, \ldots, t_{n-1}$ *are terms, so is* $f_i(t_0, t_1, \ldots, t_{n-1})$*. Here* $t_0, t_1, \ldots, t_{n-1}$ *are the arguments of the function* f_i*;*
- *no other string is a term.*

It is worthwhile to note that Def. 12 includes the case of constants, which are 0-ary functions.

Definition 13 (Syntax of predicate logic: Well-formed formula)**.** *A well-formed formula of predicate logic is recursively constructed as follows:*

- *if* $p_i \in P$ *is an n-ary* ($n \geq 0$) *predicate symbol and* $t_0, t_1, \ldots, t_{n-1}$ *are terms, then* $p_i(t_0, t_1, \ldots, t_{n-1})$ *is a well-formed formula;*
- *if* α *and* β *are well-formed formulas, then so is* $(\alpha \wedge \beta)$*;*
- *if* α *and* β *are well-formed formulas, then so is* $(\alpha \vee \beta)$*;*
- *if* α *is a well-formed formula, then so is* $\neg\alpha$*;*

- *if v_i is a variable and α is a well-formed formula, then $\exists_{v_i}\alpha$ and $\forall_{v_i}\alpha$ are well-formed formulas;*
- *no other string is a well-formed formula.*

Analogously to propositional logic, precedences over logical connectives and quantifiers are introduced to avoid unnecessary parentheses in formulas. $\forall$ and $\exists$ have the lowest precedence of all Boolean connectives and quantifiers. The precedences between logical connectives follow the rule defined in Section 2.2.1 for propositional formulas.

In predicate logic an atomic formula is a predicate together with its arguments. Given a formula $\diamond_{v_i}\alpha$, $\diamond \in \{\forall, \exists\}$, α is said to be the *scope* of the quantifier $\diamond_{v_i}$. The variable v_i is *bound* if it occurs inside of the scope of $\diamond_{v_i}$, otherwise it is called *free*.

2.3.2 The Semantics of Predicate Logic

Unlike propositional logic, in which the value of a formula is determined by the interpretation of the atomic variables and the semantics of the logical operators, the value of a well-formed formula from predicate logic is dependent not only on the value of the predicates and the meaning of the logical connectives, but also on the semantics of the predicate symbols and function symbols. Furthermore, since a formula in predicate logic may contain existential and universal operators, the *universe* or the *domain* of the variables occurring in the formula influences the value of the formula too.

Definition 14 (Interpretation)**.** *An interpretation A (sometimes called a structure) is a pair (U, I), where the universe U is an arbitrary non-empty set, and I is an interpretation function defined as follows:*

- *for every variable v_i, $I(v_i) : V \to U$;*
- *for every n-ary function symbol f_i, $I(f_i) : U^n \to U$;*
- *for every n-ary predicate symbol p_i, $I(p_i) : U^n \to \{0, 1\}$.*

Definition 15. *Given an interpretation $A = (U, I)$ and a term t, the value of t is recursively computed by applying the following rules:*

- *if t is a variable v_i, then $A(t) = I(v_i)$;*
- *if t is an n-ary function symbol f_i with arguments $t_0, t_1, \ldots, t_{n-1}$, then $A(f_i) = I(f_i(I(t_0), I(t_1), \ldots, I(t_{n-1})))$.*

Definition 16. *Given an interpretation $A = (U, I)$ and a formula α, α is evaluated repeatedly as:*

- *if α is an n-ary predicate p_i with arguments $t_0, t_1, \ldots, t_{n-1}$,*

$$\text{then } A(\alpha) = \begin{cases} 1 & \text{if } I(p_i(A(t_0), A(t_1), \ldots, A(t_{n-1}))) = 1 \\ 0 & \text{otherwise} \end{cases}$$

- *if* α *has the form* $p_i \wedge p_j$*, then* $A(\alpha) = \begin{cases} 1 & \text{if } A(p_i) = 1 \text{ and } A(p_j) = 1 \\ 0 & \text{otherwise} \end{cases}$
- *if* α *has the form* $p_i \vee p_j$*, then* $A(\alpha) = \begin{cases} 1 & \text{if } A(p_i) = 1 \text{ or } A(p_j) = 1 \\ 0 & \text{otherwise} \end{cases}$
- *if* α *has the form* $\neg p_i$*, then* $A(\alpha) = \begin{cases} 1 & \text{if } A(p_i) = 0 \\ 0 & \text{otherwise} \end{cases}$
- *if* α *has the form* $\forall_{v_i} p_j$*,*

  then $A(\alpha) = \begin{cases} 1 & \text{if } A(p_j) = 1 \text{ for every } u \in U \text{ and } I(v_i) = u \\ 0 & \text{otherwise} \end{cases}$
- *if* α *has the form* $\exists_{v_i} p_j$*,*

  then $A(\alpha) = \begin{cases} 1 & \text{if } A(p_j) = 1 \text{ for some } u \in U \text{ and } I(v_i) = u \\ 0 & \text{otherwise} \end{cases}$

An interpretation A is a *model* of a formula α when $A(\alpha) = 1$ is valid.

Definition 17 (Satisfiability)**.** *A formula* α *is satisfiable when there exists a model for* α*, otherwise it is unsatisfiable.*

Definition 18 (Tautology)**.** *A formula* α *is a tautology when every interpretation is a model for* α*.*

2.4 Finite State Machines

In today's design flow, models are used intensively to manage the complexity of the system being designed and analyzed. For instance, in a top-down design flow, design engineers usually start from the most abstract model of the system. This model might only contain structural information of the system, e.g., a block diagram, or an abstract behavior that for example only reflects the causality of the objects in the system, or a mix of these. Then, the abstract model is refined in one step or in several steps. At every step, more and more details are added to the design. When analyzing (verifying) a system, different models are required depending on the focus of the verification. These models may constitute different abstraction levels of the system, or be totally irrelevant to each other. But they have at least one thing in common: every model contains only the necessary information for the purpose of the verification, and the irrelevant parts are "abstracted" away.

The behavior of a *sequential* system is not only dependent upon the current inputs to the system, but also on its past inputs. In other words, a sequential system has a memory (register). In this section, a model for sequential systems is presented, namely the *finite*

state machine (FSM). A *state* is a snapshot of the system. A value update in the memory results from how the state of the system changes from the current state s to the next state s'. FSMs fall into two categories: *Moore machines* and *Mealy machines*.

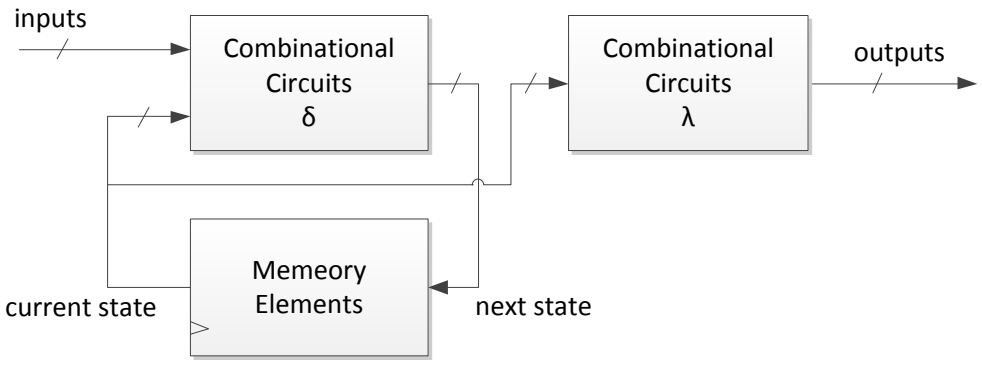


Figure 2.5: The Moore machine

Definition 19 (Moore machine)**.** *A Moore machine* $M = (I, O, S, S_0, \delta, \lambda)$ *consists of*

- *a set of input values* I*,*
- *a set of output values* O*,*
- *a set of states* S*,*
- *a set of initial states* $S_0 \subseteq S$*,*
- *a transition function* $\delta : I \times S \to S$ *that delivers the next state with respect to the current state and input value,*
- *an output function* $\lambda : S \to O$ *that computes the output value based on the current state.*

The only difference between a Moore machine and a Mealy machine lies in the output function. In a Mealy machine, the current input value also influences the current output value (see Fig. 2.5 and Fig. 2.6). Composing two Mealy machines may lead to a situation in which combinational loops are created in the composed system. At a certain point in time, the signal values along a combinational loop may oscillate. This scenario can not be handled by the design and verification tools that use FSMs to model digital systems. The problem brought about by combinational loops will be elaborated in Section 4.2. Composing two Moore machines yields again a Moore machine without combinational loops between the inputs and outputs of the two designs, since the inputs and the outputs of a Moore machine are separated by flip-flops.

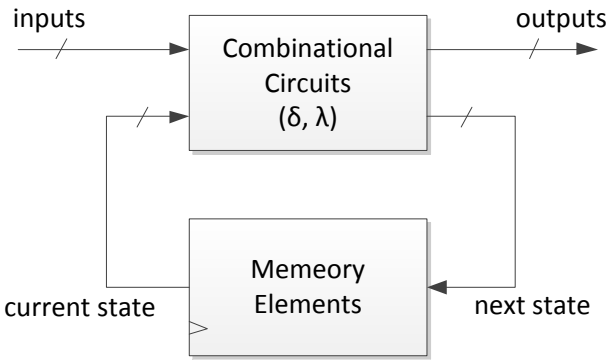


Figure 2.6: The Mealy machine

Definition 20 (Mealy machine)**.** *A Mealy machine $M = (I, O, S, S_0, \delta, \lambda)$ consists of*

- *a set of input values I,*
- *a set of output values O,*
- *a set of states S,*
- *a set of initial states $S_0 \subseteq S$,*
- *a transition function $\delta : I \times S \to S$ that delivers the next state with respect to the current state and input value,*
- *an output function $\lambda : I \times S \to O$ that computes the output value based on the current state and the current input value.*

Definition 21 (Sequential circuit)**.** *A FSM with $I \subseteq \mathbb{B}^n$, $O \subseteq \mathbb{B}^m$, $S \subseteq \mathbb{B}^k$ and $\mathbb{B} = \{0, 1\}$ is called binary encoded, and a FSM like this can be naturally implemented by a sequential circuit with n inputs, m outputs, and k state variables. Such a sequential circuit can be represented by a graph $C(V, E)$. The vertices of the graph can be partitioned into the following groups.*

- *The gates of $X \subseteq V$, $|X| = n$ that do not have direct predecessors, represent the set of primary inputs of the circuit. The edges leaving the gates of X are associated with the set of input variables x_j.*
- *The gates of $Y \subseteq V$, $|Y| = m$, $Y \cap X = \emptyset$ that do not have direct successors, represent the set of primary outputs of the circuit.*
- *The gates of $Z \subseteq V$, $|Z| = k$, represent the set of storage elements, which keep a binary value until a clock tick occurs. The edges leaving the gates of Z are associated with the set of state variables z_j. The value of z_j represents the current*

state of the FSM. Each gate of Z has only one incoming edge denoted by the next state variable z'_j.

- *The gates of $T \subseteq V \setminus (X \cup Z)$ are associated with a set of Boolean functions $t_j : \mathbb{B}^n \times \mathbb{B}^k \to \mathbb{B}$. This set of Boolean functions implements the transition function δ of the FSM. The edges leaving the gates of T must be able to be associated to a next state variable z'_j.*
- *The gates of $F \subseteq V \setminus (X \cup Z)$ are associated with a set of Boolean functions $f_j : \mathbb{B}^n \times \mathbb{B}^k \to \mathbb{B}$. This set of Boolean functions implements the output function λ of the FSM. The edges leaving the gates of F must be able to be associated to an output variable y_j.*

In Def. 21, each gate of T can be implemented by a combinational circuit. All internal signals in these combinational circuits joined with all z'_j make up the internal signals of the sequential circuit.

Definition 22 (Combinational loop)**.** *A combinational loop is a cycle in a sequential circuit $C(V, E)$. Along this cycle there is no gate, which represents a storage element.*

Intuitively, a combinational loop is built by feeding the outputs of a combinational circuit back to its inputs. Many modern design and verification tools experience difficulties in handling combinational loops, because their behavior mostly depends on the relative propagation delays of the gates in the loop. This may lead to issues of stability and reliability for the circuit.

In the remainder of this dissertation, to avoid formalism, an implementation of a FSM $M(I, O, S, \delta, \lambda)$ will be denoted by a sequential circuit $M(i, o, s)$, where i is the set of input variables, o is the set of output variables and s is the set of state variables. The set of internal signals of this circuit is denoted by x.

Chapter 3

Model Checking

Model checking or *formal property checking* is an automatic method to verify designs. Verifying designs means to check if an implementation of the design fulfills the (formal/informal) specification of the design. Theoretically, one can check the equivalence between the implementation and the specification of a design. However, in reality most specifications are informal, therefore verifying the equivalence between the implementation and the specification of a design is almost not possible. Even in the case of formal specifications, it is not possible , because, for example, the computational complexity of such a sequential equivalence check is too high. For these reasons, model checking verifies that the implementation fulfills (satisfies) the specification. The specification of the design is formalized in *properties*.

Model checking can be employed to verify various designs across various application domains, e.g., hardware designs, software designs, embedded systems, hybrid systems. This dissertation considers only (digital) hardware designs, software designs and embedded systems consisting of hardware and software. To use a model checker, the design under verification (DuV) must be modeled by an appropriate mathematical formalism. In addition, since a property usually specifies an aspect of the overall functionality of a design, there is a need for methods to measure how much functionality is covered by a property set.

In this chapter, Section 3.1 gives an overview of the state of the art model checking techniques. The methods based on SAT solvers [53–55] are described in Sec. 3.2, which are the primary model checking methods used in this dissertation. A summary of property specification languages and the methods for the analysis of the coverage of property sets are presented in Section 3.3 and in Section 3.4.

3.1 Introduction to Model Checking

In recent years, formal model checking has been successfully adopted in the semiconductor industry. It has been integrated into design flows either as a technique complementary to traditional simulation techniques, where it is used to verify critical parts or corner cases of a digital circuit, or as a full replacement of simulation techniques.

In general a model checker operates on the finite state machine of a design written in a hardware description language like VHDL or Verilog [56, 57]. The properties can be given as propositional formulas, predicate logic, or temporal logic, the last will be introduced in Section 3.3. The model checker verifies whether a design fulfills the behavior specified in the properties. In case a property fails, a counterexample is produced by the model checker that shows a valuation of a trace of signals leading to the error. An error can be related to a real design bug or a false property. From a practical point of view, simulation is a black-box technique, meaning that verification engineers only care about the input and output behavior of the DuV. In contrast, model checking is *white-box*. Not only the input and output behavior, but also the internal behavior of a design needs to be considered. This requires that verification engineers must have a deep understanding of the design. On the other hand, model checking can help verification engineers to gain knowledge about the design. In recent years, *assertion-based verification* has become popular for hardware design verification. An assertion formalizes the behavior of a design. It is logically equivalent to the "property" in the context of model checking. Therefore an assertion can be proven with a formal model checker too.

Model checking explores the state space of a design. Therefore the scalability and efficiency of a model checker are mostly impacted by how states are represented in the model checker and how effective the algorithm for traversing the state space is. *Image computation* has a central role in state space exploration. It computes the *images* of the states, meaning the direct successor states of the given states. In contrast to "images," a state also has *pre-images*, which are its direct predecessors. *Classical model checking* uses adjacent lists or state transition tables to explicitly represent the states and the transitions between states [12]. As the state space of a design grows exponentially in the number of registers of the design, classical model checking faces the problem of *state explosion*. In other words, a design with a small number of registers could easily exhaust the memory of a computer. The inefficiency of classical model checking is not only due to storing the states, but also due to the computation of the images: at each computation step, only one successor state of a given state is explored.

Symbolic model checking can handle bigger designs than classical model checking [13, 16, 19, 58–60]. In a symbolic model checker, the states and the transition relations of a design are represented implicitly through *characteristic functions*, which identify sets of states and transitions. For example, in a design with four states $\{00, 01, 10, 11\}$, the states can be encoded with two state variables s_0, s_1. The characteristic function to represent the state set $\{00, 01\}$ is a single Boolean function $\xi(s_1) = (s_1 \leftrightarrow 0)$. Given a design modeled by the FSM $M = (I, O, S, S_0, \delta, \lambda)$, where the transition function δ computes the values for the n state variables, the characteristic function τ for the transition relation can be easily obtained by using

$$\tau(s, i, s') = \bigwedge_{j=1}^{n} (s'_j \leftrightarrow \delta_j(s, i)).$$

Symbolic model checking can handle designs with more than 10^{20} states [61]. In

addition, the image computation is performed directly on the characteristic functions of the states and transitions, where the images of a set of states are calculated at each step. Let $C_S(s)$ be the characteristic function of a state set S. The symbolic images of the state set S under the transition relation $\tau(s, i, s')$ can be computed by

$$\mathit{Image}(S) = \exists_{s \in S} \exists_i \, C_S(s) \wedge \tau(s, i, s').$$

In a symbolic model checker, an ROBDD (abbr.. BDD) is used as a data structure to represent the characteristic functions, which exposes the limitations of symbolic model checking, namely the size of the BDDs may grow exponentially in the number of states and input variables. Also symbolic model checking with BDDs often suffers from the state explosion problem, because the representation of state sets and the process of manipulating the BDDs can consume a huge amount of memory, not only because of the size of the DuV [15, 62].

The SAT-based model checking techniques open another door to solve the state explosion problems of classical model checking and symbolic model checking with BDDs. SAT-based model checking applies the conjunctive normal form (CNF) to represent characteristic functions, and the overall verification problem is reduced to a SAT problem [54, 55]. Owing to the rapid advances in SAT solvers, a modern SAT solver can handle CNF formulas with hundreds of thousands of variables [63–65]. Thus a SAT-based model checker can handle big, even huge, industrial designs [66–68]. In Section 3.2 two SAT-based model checking methods are explained in detail.

3.2 SAT-Based Model Checking

3.2.1 Bounded Model Checking (BMC)

Bounded Model Checking (BMC) is a SAT-based model checking method proposed by Biere et al [19]. In contrast to classical model checking and symbolic model checking, which try to "prove" a property of a design, BMC attempts to falsify a property. Put another way, BMC aims at finding design errors as soon as possible. In general, a bounded model checker searches for a counterexample along finite paths of length k starting from the initial states of a design. In case a property holds, the BMC checker increases the step k, until a counterexample is found, or the resources of the underlying computation platform are exhausted, or a so called *completeness threshold* is reached. A completeness threshold is the lowest bound of k needed by BMC to search the entire state space of the design. The property is proven only when the BMC checker reaches the completeness threshold. For industrial designs having large sequential depth, BMC is usually terminated owing to exhaustion of computer resources. In this sense, BMC is not complete, and is only a complementary tool to quickly find bugs in designs.

BMC works on the *iterative circuit model* of a design. Given an FSM $M = (I, O, S, S_0, \delta, \lambda)$, the iterative circuit model is constructed by making (unrolling) k copies of combinational logic of λ and δ, where every copy computes the output value o^j and the next

state value s'^j for the corresponding time instance (Fig. 3.1).

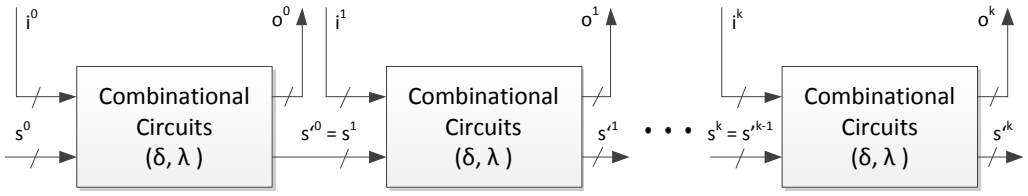


Figure 3.1: Iterative Circuit Model

The characteristic function of the resulting model is represented by

$$\|M\|_k = \xi(s^0) \wedge \bigwedge_{j=0}^{k-1} \tau(s^j, i^j s^{j+1}),$$

which can be further converted to a CNF formula. Here, $\xi(s^0)$ is the characteristic function of the initial states of the design.

Likewise, the (safety) property p can be unrolled into a formula $\|p\|_k$ too. To falsify the property, the CNF formula $\|M\|_k \wedge \neg \|p\|_k$ is checked for satisfiability by a SAT solver. A valuation that satisfies this formula is a counterexample to the property p.

3.2.2 Interval Property Checking (IPC)

As stated in Section 3.2.1, BMC is not complete, it can only be applied to find errors in a design up to certain sequential depths, but not to prove properties. *Interval Property Checking (IPC)* is similar to BMC. However, it provides a complete proof of the properties. A property in IPC describes the design behavior inside of a finite time interval with length j. As with BMC, the model in IPC is also an iterative model. The combinational logic of λ and δ are unrolled j times. The starting state of this model is not the initial state anymore: it starts from an arbitrary state, which might be an unreachable state. This means that IPC proves a property completely, while the property itself is formulated over a bounded time window.

Since the model in IPC starts from an arbitrary state, the disadvantage of this over-approximation is that it may generate *false negative* counterexamples that result from unreachable starting states. To overcome this issue, so called *reachability constraints* are provided to exclude some or all unreachable states. Reachability constraints can be formulated manually or can be obtained automatically by applying reachability analysis algorithms like [20, 21]. The validity of a manually created reachability constraint can be proven by induction over time.

3.3 Overview of Property Specification Languages

In Chapter 2, some formalisms for modeling hardware systems were introduced. To verify the behavior of an implementation of a system using model checking, the legal behavior of the system must be formalized in *properties* by applying a (formal) property specification language. Usually the behavior of a system, especially of a circuit, is described over time. In other words, the behavior is *temporal* and about the sequences of the inputs, the outputs, and the states of the circuit. This requires that a specification language must be able to access the elements of a sequence, which again model the time instances. In a sequential circuit, a time instance is related to one clock cycle in the system. In an embedded software system, the time instance is further abstracted, which is determined by the causality of two interactions between the software and its environment. Although propositional logic and predicate logic are theoretically able to describe such temporal behavior, they are hardly ever used (they are not intuitive to the user), e.g., temporal behavior in practical cases often has to be expressed with formulas that have huge sizes. These huge formulas are hardly understood and may mislead the reader of the properties. Hardware description languages like VHDL or Verilog may be a choice of property specification languages, however they also lack the ability to compactly and intuitively specify temporal behavior. For example, an arbiter for a communication bus may be required to have a property like "A request is granted exactly in two clock cycles." Fig. 3.2 and Fig. 3.3 list the property specified in Verilog and in a specialized property specification language, "SystemVerilog Assertions" (SVA), respectively. As illustrated, the property written in Verilog is cumbersome and might be error-prone, the property in SVA is more intuitive and compact. Another alternative might be the built-in *assert* statements in VHDL and Verilog. However, the assert statements are used to dynamically test the specified behavior at the special time point at which a simulation run reaches the assert statement. Furthermore the assert statement can only specify non-temporal behavior. Given all this, it is necessary to have a language that is specialized to describe temporal behavior.

Basically, property specification languages can fall into two categories, namely *branching time logic* and *linear time logic*. *Computation Tree Logic (CTL)* is one kind of branching time logic, which describes the properties of the *computation tree* of a design [8]. The computation tree of a design is constructed by unwinding the design in an infinite tree. Given a state s in the computation tree, a CTL formula can specify multiple computational paths leading to different futures of s. Syntactically, CTL extends propositional logic with *path quantifiers* and *temporal operators*. Path quantifiers choose along which path the property shall be valid: A (always) stands for along every path the property shall be valid; E (exists) stands for along at least one path the property shall be valid.

Temporal operators indicate on which state along a path the property shall be valid: G (global) represents the property shall be valid globally on all states of the path; F (future) states that the property shall be valid somewhere in the future of the path; X (next) denotes that from the current state, the property shall be valid in the next state of the path; ϕU φ(until) requires the property ϕ to be true on all states on the path until the property φ is fulfilled. This section does not address the detailed definition of the syntax and

```
always @(posedge clk) begin
      if (req == 1'b1)
          cnt <= 1;
      else if (cnt == 1 && cnt < 2)
          cnt <= cnt + 1;
      else if (cnt == 2) begin
          if (grant == 1'b1)
              $display("request granted");
          else
              $display("request not granted");
          end
      end
end
```

Figure 3.2: Property written in Verilog

```
assert property (@(posedge clk) req |-> ##2 grant)
```

Figure 3.3: Property written in SVA

semantics of property languages, the interested reader is referred to the cited papers.

In contrast to CTL, Linear Time Logic (LTL) models time in a linear fashion [9]. An LTL property specifies the behavior of a system along a single computational path of the system. Therefore LTL does not have path quantifiers. Proving an LTL property on the model of a (hardware/software) system stands implicitly for checking every computational path of the model. LTL also has temporal operators, as in CTL. CTL requires that a path quantifier must be followed by a temporal operator, whereas LTL allows arbitrary combinations of temporal operators. Hence CTL and LTL have different expressive power. Some design behaviors can be expressed in CTL but not in LTL, and vice versa. A summary of the expressive power of these two languages can be found in [69]. CTL*, a superset of CTL and LTL, extends not only the syntax of CTL and LTL, but also their expressive power [70]. It is designed to address the shortcomings of LTL, with which it is not possible to make statements over paths, and of CTL, which requires that every path quantifier A, E must be followed by exactly one of the temporal operators $\mathsf{G}, \mathsf{F}, \mathsf{X}, \mathsf{U}$. In CTL* the path quantifiers and the temporal operators can be combined in any order.

Properties may be classified as *safety properties* or *liveness properties*. A safety property specifies that "something bad must not happen" . It can be expressed in terms of the CTL formula $\mathsf{AG}\, p$. A liveness property indicates that "something good eventually happens" . It can be expressed in terms of the CTL formula $\mathsf{EF}\, p$.

Although CTL, LTL and CTL* have great expressive power to specify a large range

of properties, they are too hard to use in practice. Their syntax and semantics are not intuitive to understand; they don't support the complex data types that enable properties to be described at a high abstraction level. It is almost not possible to write reusable code using these languages. On these grounds, lots of syntax extensions have been invented to make property specification languages easily adopted by engineers from industry. In the rest of this section, some syntactic extensions for these temporal languages will be briefly described. For the detailed syntax and application areas of these syntax extensions (languages), the interested reader is referred to the cited papers. It should be noticed that a syntax extension enhances only the syntax of a base language like CTL or LTL: it does not extend the expressive power, and its statements can always be mapped to statements following the syntactic rules of the base language.

Open Verification Library (OVL)

Strictly speaking, OVL is not a syntax extension. It is a verification library maintained by the Accellera System Initiative, which can be used in simulation, emulation, and formal verification [71]. The library can be used to check hardware designs written in one of VHDL, Verilog, SystemVerilog [10], and SystemC [72]. The library contains a set of (fixed) assertions that need to be checked by a verification engine. These assertions cover typical scenarios that arise in designs, such as one-hot encoding of signals, value increment of signals, toggles of signals, etc. OVL also contains verification IPs for popular building blocks of hardware designs, a FIFO, for example. Usually an OVL checker is instantiated in the HDL source code of the hardware designs. Depending on the parameters configured by the users, an OVL checker can be used as a property, as an environment constraint, or as a coverage measure.

PSL and SVA

PSL stands for *Accellera Property Specification Language*. It is standardized by the industry consortium Accellera and aimed at verifying digital hardware designs [11]. To shorten the learning curve of a new language, it provides different flavors of syntax, which are similar to one of the standard HDL languages, such as VHDL. PSL formulas can be categorized into formulas of the *foundation language (FL)* and formulas of the *optional branching extension (OBE)*; the former is the syntax extension for LTL, the latter has the same syntax as CTL. To manage the structural complexity of the formulas, an FL is usually composed of four layers. The *Boolean layer* is the building block of a formula, atomic formulas and Boolean operations over them belong to this layer. The next layer is the *temporal layer*, which specifies the temporal relations of formulas from the Boolean layer. Its powerful *feature of regular expressions* allows specifying a set of sequential behaviors (sequences) at once. The *verification layer* contains directives instructing verification tools to check a formula (property) or to test if a property is covered by test patterns in simulation. The *modeling layer* makes it possible to construct a helpful auxiliary logic for verification, e.g., environment constraints for the inputs of the design under

verification. In this way, FL retains all the syntax of LTL: this indicates that both safety- and liveness properties are possible to be expressed with PSL.

SystemVerilog Assertion (SVA) is a part of the SystemVerilog standard. It is tightly coupled with other SystemVerilog structures [10]. This means it brings all their strengths into play when it is used in the context of a SystemVerilog design. SVA is derived from PSL, however, it throws away the OBE part of PSL and does not contain any syntax from LTL. Structurally, SVA also has four layers, similar to PSL, especially the *feature of regular expressions* in PSL is denoted by *sequences* in SVA. In contrast to PSL, SVA is more user-friendly, and provides more built-in functions to make the verification code compact. The user can also use the SystemVeriog function feature to create their own functions, so that the readability and re-usability of verification code is improved. In particular, only safety properties are possible to be expressed with SVA. This is however enough for simulation-based techniques. Anyhow, SVA was originally designed for the purpose of simulation.

3.4 Coverage Analysis of Property Sets

Coverage analysis is a necessary component in design verification environments. It answers questions such as "is the DuV simulated enough?" in simulation, and "are the written properties enough?" in property checking. Section 3.4.1 summarizes the state-of-the-art approaches to coverage analysis in hardware design verification. Section 3.4.2 describes a coverage analysis method (complete interval property checking (C-IPC)) for property sets in assertion-based verification and in property checking.

3.4.1 Introduction to Coverage Analysis in Hardware Design Verification

In simulation-based verification, *coverage analysis* measures "how good are the test patterns" fed into the design. Furthermore, the result of the coverage analysis can be used to guide test benches in generating test patterns to achieve full coverage. Different *coverage metrics* are provided for this measurement. Generally speaking, every coverage metric has its advantages and disadvantages compared to other criteria regarding the types of bugs found, its computation complexity, and the degree to which it can be automated. It is hard to judge which one is better. Coverage analysis for hardware design verification borrows some ideas from software testing. The coverage metrics can fall into three categories: code coverage, structural coverage, and functional coverage [73].

Code coverage evaluates whether every statement of the HDL code for a design is executed by the simulator under the specified test patterns. The particularly interesting metric here is *branch coverage*, which measures whether every branch destination is covered by test patterns. Code coverage is easy to compute: it is based on the control flow graph of the design's HDL code. However, due to the concurrent execution of the hardware, it is usually not sufficient. The typical *structural coverages* are toggle coverage and FSM

coverage. Toggle coverage measures whether every signal in a design is triggered from 0 to 1 or vice versa. FSM coverage is a little more complex: it evaluates whether every transition of the FSM of a DuV is covered by test patterns. An FSM can be automatically extracted from HDL code, where some special coding styles, e.g., the application of the "case" statement, must be used to implement designs. An FSM can also be constructed manually. Usually a manually created FSM is more abstract than the one automatically generated by a tool.

The most complex metrics are those for *functional coverage*. Some aspects of the functionality of a design are formulated in, e.g., SVA coverage properties, and a simulator evaluates how many times they are triggered by test patterns. Here the properties could be of any kind, for instance, temporal behavior ranging over many clock cycles, invariants of the DuV or one-hot encoding of FSMs. Obviously the quality and the quantity of property sets have an influence on the quality of the functional metrics.

The coverage metrics for simulation-based verification are not suitable for formal property checking or assertion-based verification with formal techniques since these coverage metrics evaluate the quality of the test patterns of DUVs, whereas formal techniques exhaust all the input patterns. The coverage metrics for formal property checking should answer verification engineer's questions like "Have I written enough properties?" [74] . In other words, the coverage metrics measure how much of the functionality of a DuV is covered by a property set. Note that this kind of functional coverage metrics is interpreted differently than the one in simulation. If a property set covers every behavior of the design, then it is *complete*.

Mutation testing is a common technique to help verification engineers gain confidence in the completeness of a property set [75]. In general, small changes, called *mutations*, are made in the source code of the design. The mutated design is called a *mutant*. Conventional property checking is performed on the mutated design to determine whether these changes are covered by the property set. [76] requires that every signal at every reachable state must be covered by at least one property. In other words, a signal of the mutated design must fail at least one property at some reachable state. This set of "failing" states is recorded for every signal and then it is checked whether this set is a superset of the reachable states of the design. The state exploration of this approach is based on BDDs, which may blow up for industrial designs. Additionally, mutation test cannot *prove* the completeness of a property set: its quality depends heavily on the type of mutants and the density of the inserted mutants. Unfortunately, none of the literature, such as [74–78], addresses how densely should mutants be inserted and what kinds of mutants may potentially prove the completeness of a property set.

The research work presented in [48, 79–81] proves the completeness of a property set by checking to what extent the property set *determines* the input and output sequences of a DuV. The approach from [79] is a design-dependent method in a BMC verification environment and is performed on a concrete implementation of a design. However, since the implementation process is usually iterative in a design flow, the completeness checking has to be run for every change of implementation.

The approaches of [48, 80, 81] are design independent and based on the following

definition of completeness:

Definition 23. *A set of properties P is complete if every two circuits M, M' satisfying the properties in P are sequentially equivalent.*

The work of [48] implements Def. 23 in a straightforward manner as presented in Fig. 3.4, where the inputs, outputs and states of the design are denoted as i, o and s, respectively. A copy of these signals is written as i', o' and s'. The two designs M, M' are empty models only containing signal names and copied signal names. The behavior of the signals is restricted by the property set P and a copy of the property set P', which describes the behavior of the copied signals. Under this prerequisite the output signals of M and M' are checked for equivalence. Obviously this approach takes the whole functionality of the design(as described by the property set) into account at once, which may lead to a high proof complexity.

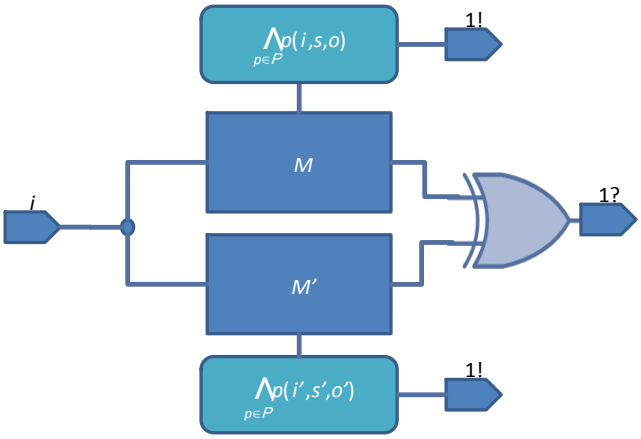


Figure 3.4: Implementation of Checking Completeness of Property Sets

The method of [80, 81], called *complete interval property checking*, overcomes this bottleneck by splitting the overall check into certain subchecks. This method is the starting point for this dissertation, and will be discussed in Section 3.4.2.

3.4.2 Complete Interval Property Checking (C-IPC)

Preliminaries

Before proceeding to explain how to prove the completeness of property sets, this section introduces the formalism to model the properties considered in this dissertation.

A *sequence predicate* describes sets of traces of signal values. As the name suggests, the arguments of a sequence predicate are sequences of signal values, and the predicate itself specifies the finite sequential design behavior constituting the properties. It is

straightforward to apply sequence predicates to formalize timing diagrams in an informal specification.

Given an FSM $M = (I, O, S, S_0, \delta, \lambda)$ consisting of a set of input values I, a set of output values O, a state set S with a subset $S_0 \subseteq S$ of initial states, and two functions $\delta : I \times S \to S$ and $\lambda : I \times S \to O$, called the state transition function and the output function, respectively, a state predicate $\eta(s)$ characterizes a set of states, a state sequence predicate $\sigma_l(s_0, \ldots, s_l)$ of length l characterizes a set of state sequences $\pi_l = (s_0, \ldots, s_l)$. If the length l of the sequence predicate matters it is also called an l-sequence predicate. State predicates $\eta(s)$ can be considered as 0-sequence predicates. It is allowed that every l-sequence predicate π_l can be applied to m-sequences $\pi_m = (s_0, \ldots, s_m)$ with $m \geq l$. In this case the predicate is evaluated on the l-prefix $\pi_l := (s_0, \ldots, s_l)$ of π_m, and the trailing sequence $(s_{l+1}, \ldots, s_m)$ remains unrestricted. This guarantees that the usual Boolean operators $\vee$, $\wedge$, and $\neg$ are also applicable to sequence predicates that may possibly have different lengths. The maximum length l_{max} of the operands to these operators then determines the length of the resulting predicate. Sequence predicates can also be shifted in time using the *next* operator:

$$\begin{aligned} \mathit{next}(\sigma_l, n)((s_0, s_1, \ldots, s_{n-1},\ s_n, s_{n+1}, \ldots, s_{n+l})) \\ := \sigma_l((s_n, s_{n+1}, \ldots, s_{n+l})). \end{aligned}$$

Using this operator one can define a concatenation operation $\odot$ for l-sequence predicates:

$$\sigma_l \odot \sigma_k := \sigma_l \wedge \mathit{next}(\sigma_k, l)$$

The predicate $\sigma_l \odot \sigma_k$ characterizes $(l + k)$-sequences $\pi_{l+k} = (s_0, \ldots, s_l, \ldots, s_{l+k})$ where s_l is the ending state of σ_l and the starting state of σ_k. Non-overlapping concatenation can be expressed using the special l-sequence predicate $\mathit{any}_l(\pi_l)$ that evaluates to true for every sequence π_l:

$$\sigma_l \oplus \sigma_k := \sigma_l \odot \mathit{any}_l \odot \sigma_k$$

If $\hat{v}$ and $\tilde{v}$ are state variables then $\sigma_l[\hat{v} \leftarrow \tilde{v}]$ is the sequence predicate derived from σ_l by substitution of every occurrence of $\hat{v}$ by $\tilde{v}$. Substitution only performed for a particular state s_k of the sequence results in the sequence predicate $\sigma_l[\hat{v}(s_k) \leftarrow \tilde{v}(s_k)]$. Substitution with constants yields the co-factors $\sigma_l(\pi_l)|_{\hat{v}(s_k)} := \sigma_l[\hat{v}(s_k) \leftarrow 1]$ and $\sigma_l(\pi_l)|_{\neg\hat{v}(s_k)} := \sigma_l[\hat{v}(s_k) \leftarrow 0]$. This allows quantifying out particular state variables $\hat{v}$ at a particular timepoint $k \in \{0 \ldots l\}$ from a sequence predicate σ_l. $\forall_{\hat{v}(s_k)} : \sigma_l$ denotes the sequence predicate

$$(\forall_{\hat{v}(s_k)} : \sigma_l)(\pi_l) := \sigma_l(\pi_l)|_{\hat{v}(s_k)} \wedge \sigma_l(\pi_l)|_{\neg\hat{v}(s_k)}.$$

A short notation for quantifying out the state variable $\hat{v}$ at every timepoint $k \in \{0 \ldots l\}$ is described as follows:

$$(\forall_{\hat{v}(\pi_l)} : \sigma_l)(\pi_l) := (\forall_{\hat{v}(s_0)} \ldots \forall_{\hat{v}(s_l)} : \sigma_l)(\pi_l).$$

Similarly, the existential quantifiers $\exists_{\hat{v}(s_k)} : \sigma_l$ and $\exists_{\hat{v}(\pi_l)} : \sigma_l$ can be defined.

Note that the Boolean operations, the operation of substitution, and the quantification operations defined above are not only applicable to the *state sequence predicate*, but also to any kind of sequence predicates. Similarly for an *l-input sequence predicate* $\iota_l(\xi_l)$ and an *l-output sequence predicate* $\varsigma_l(\zeta_l)$ characterizing a set of input sequences $\xi_l = (i_0, i_1, \ldots, i_{l-1})$ and a set of output sequences $\zeta_l = (o_0, o_1, \ldots, i_{l-1})$, respectively.

With the aid of the transition function δ of an FSM, the state sequences that correspond to valid *paths* in the FSM can be determined. Such paths are characterized by the l-sequence predicate *ispath*:

$$\mathit{ispath}_l(\pi_l, \xi_l) := \bigwedge_{j=1}^{l} (\delta(s_{j-1}, i_{j-1}) = s_j).$$

The set of output sequences generated by the FSM can be characterized by an l-sequence predicate *isoutput*:

$$\mathit{isoutput}_l(\pi_l, \xi_l, \zeta_l) := \bigwedge_{j=0}^{l-1} (\lambda(s_j, i_j) = o_j).$$

Note that a sequence predicate can be defined by using the input, state, and output variables of an FSM. Sometimes it is convenient to use $M(i, s, o)$ to denote a design (FSM) M containing the set of input variables $i = \{i_0, i_1, \ldots, i_n\}$, the set of state variables $s = \{s_0, s_1, \ldots, s_j\}$, and the set of output variables $o = \{o_0, o_1, \ldots, o_m\}$. Here the index of the variable names is abused, it shall not be confused with an element for a time instance in a sequence. Similarly, the l-sequence predicate $\sigma(i, s, o)$ expresses that σ is defined in terms of the variables i, s and o. In the rest of this dissertation, this notation will be used when the length of the sequences can be an arbitrary finite number and the discussion merely focuses on the variables rather than individual elements of sequences.

Operational Properties

An operational property P_l viewed as an l-sequence predicate has a cause-effect structure, represented by a pair (A_l, C_l), where $A_l(\pi_l, \xi_l)$ is the *assumption* of P_l, and $C_l(\pi_l, \zeta_l)$ is the *commitment* of P_l. The unbounded validity of the property P_l is proven when

$$(\mathit{ispath}_l(\pi_l, \xi_l) \wedge \mathit{isoutput}_l(\pi_l, \xi_l, \zeta_l)) \Rightarrow (A_l(\pi_l, \xi_l) \Rightarrow C_l(\pi_l, \zeta_l)) \tag{3.1}$$

is a tautology. This check for tautology can be conducted effectively using IPC as described in Section3.2.2. Note that the starting state s_0 of an l-sequence satisfying the *ispath* and *isoutput* predicates is not restricted to the initial states of the FSM. In case a property does not hold, the property checker generates a *counterexample*, which shows

that a state sequence π_l and an input sequence ξ_l falsify Formula 3.1. A counterexample is *false negative* when s_0 of π_l is not reachable from the initial states of the FSM.

In practice, in order to rule out false negative counterexamples, invariants representing over-approximations of the reachable state space are added manually or automatically [20, 21] to strengthen the property, resulting in

$$(\mathit{ispath}_l(\pi_l, \xi_l) \wedge \mathit{isoutput}_l(\pi_l, \xi_l, \zeta_l)) \Rightarrow (\Phi(s_0) \wedge A_l(\pi_l, \xi_l) \Rightarrow C_l(\pi_l, \zeta_l)), \tag{3.2}$$

where the state sequence $\Phi(s_0)$ characterizes such an invariant. The validity of an invariant can be proven by using induction over time.

An operational property P_l can be formulated by using a standard property specification language, such as SVA or PSL. It can then be further converted to formulas in a temporal logic like CTL or LTL. The expressive power of the sequence predicates that model the operational properties is a subset of LTL's expressive power. An operational property P_l corresponds to an LTL safety property $\mathsf{G}\, P_l$.

Definition 24 (Conceptual states)**.** *A conceptual state s_c is an important control state in a design, which may be related to many concrete states in the design. Given a set of conceptual states S_C, a concrete state s can be mapped to a conceptual state s_c by using a surjective function $w : S \rightarrow S_C$. The set of conceptual states in a design can be characterized by a predicate $\Psi(s)$.*

Usually the conceptual states are identified by verification engineers. Every pair of conceptual states is connected by so called *operations* that can be specified by operational properties, i.e., every operational property starts from a conceptual state and ends at a conceptual state. As a verification engineer is able to concentrate on one aspect (operation) of the overall design behavior at every time, such an operational property can be constructed systematically. The assumption part of an operational property regarding conceptual states has the form:

$$A_l(\pi_l, \xi_l) = \Psi(s_0) \wedge \iota_l(\xi_l)$$

in which $\Psi(s_0)$ characterizes the starting conceptual state of the operation along the state sequence $\pi_l = (s_0, s_1, \ldots, s_l)$, and $\iota_l(\xi_l)$ is the characteristic function for the input trigger $\xi_l = (i_0, i_1, \ldots, i_{l-1})$ of the operation.

The commitment part of an operational property has the form:

$$C_l(\pi_l, \zeta_l) = \Psi(s_l) \wedge \varsigma_l(\zeta_l)$$

in which $\Psi(s_l)$ characterizes the ending conceptual state of the operation, and the outputs produced by the operation are characterized by the sequence predicate $\varsigma_l(\zeta_l)$.

Definition 25 (Conceptual state machine (CSM))**.** *A conceptual state machine of a design can be represented by an edge-labeled graph G(V, E, P), where the vertices V are the conceptual states of the design, the edges E are labeled with P, which represent the operations between the conceptual states. Such operations can be formulated by operational properties.*

Fig. 3.5 depicts the FSM and the conceptual state machine of a slave interface for a handshake protocol. (Only the control part is described.) The slave interface may perform two operations $read$ and $write$ in response to the commands given by a master interface. It writes/reads two consecutive bytes of data to/from a memory location. During the operation, the $busy$ flag is set, indicating that the slave interface is processing data. As illustrated in the figure, there is only one conceptual state $idle$, and every operation ($read$ or $write$) starts and ends at this state. Note that a design may have different conceptual state machines, depending on the views of the verification engineers. In a well structured implementation of a design, the conceptual state machine is described explicitly, so that an EDA tool, e.g., Xilinx ISE [82] can extract it automatically. However, in most cases, verification engineers have to manually extract it from a design.

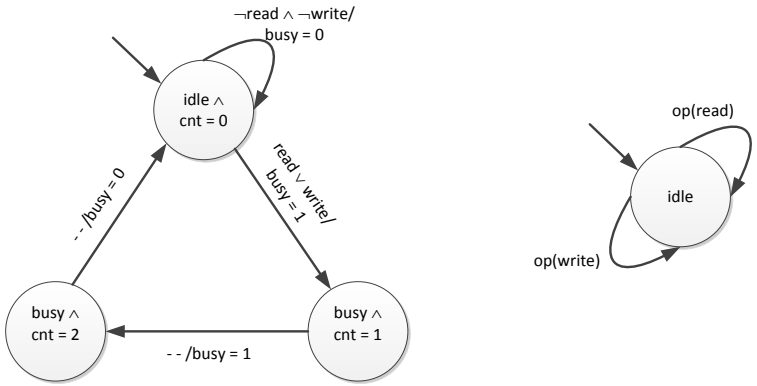


Figure 3.5: FSM and Conceptual state

Completeness of Property Sets

Complete Interval Property Checking is a SAT-based property checking technique [41,81, 83] with the ability to analyze the completeness of property sets. Similar to methods such as those of [48, 84], it checks whether a given property set $P = \{p_1, \ldots, p_n\}$ uniquely determines the input and output behavior of the particular device under verification, where a property p_i is an operational property describing the temporal behavior of the signals (input, output and state variables) $v_1, v_2, \ldots, v_j$ of a device.

Definition 26 (Determination)**.** *A property* p_i determines *a signal* s *if the signal is uniquely defined by* p_i *as a function of other determined signals, such as inputs or state variables. In other words, along every sequence of the design, the property* p_i *specifies exactly one value for the signal* s *at every time point of the sequence.*

The determination of a signal s is checked by considering two instances of the property, p_i and p_i', and two sets, $\{v\}$ and $\{v'\}$, of the variables appearing in the property. The behavior of $\{v\}$ is defined by p_i, and that of $\{v'\}$, by p_i'. Then it is checked whether in all

considered input scenarios the signal s necessarily has the same value in both instances, i.e., whether the formula $p_i \wedge p_i' \Rightarrow (s = s')$ is a tautology. If so, then s is said to be uniquely determined by the property p_i. (Note that the design is not considered in this check.)

In addition, a module also carries so called determination conditions that specify under which circumstances a particular signal in the module has to be uniquely determined. A determination guard d_s for a signal s is a formula that evaluates to true whenever the signal s needs to be determined. For example, a data signal on a bus may be guarded by a valid flag and only needs to be determined if this flag is set. With respect to the determination guard d_s of a signal s, the determination condition for s is defined by the logical formula $D_s := (d_s \vee d_{s'}) \Rightarrow (s = s')$, which is a sequence predicate, i.e., the determination guard may specify a condition spanning over many clock cycles, and may include temporal operators like *next*.

The work of [41] considers determination conditions for inputs and outputs, and refers to them as determination assumptions and determination requirements of the DUV, respectively. (Note that [48] only defines the determination conditions for outputs.) It is assumed that determination assumptions D_i and determination requirements D_o are to be specified for each input i and each output o. The formula $d_i = d_o = \mathit{true}$ defines the default determination guards for all ports i, o of the device, i.e., it is assumed that all inputs are determined at *every* point in time, and likewise for the outputs. However, the verification engineer may overwrite these default values with weaker conditions.

The definition of completeness considering determination conditions is modified as follows:

Definition 3.1 (Completeness (with determination conditions))**.** *A set of properties P completely specifies a module M with respect to determination assumptions $D_{i_1}, \ldots, D_{i_n}$ and determination requirements $D_{o_1}, \ldots, D_{o_m}$ iff the following property is a tautology:*

$$\bigwedge_{j=1}^{n} D_{i_j} \wedge \bigwedge_{p_l \in P} p_l \bigwedge_{p_l' \in P'} p_l' \Rightarrow \bigwedge_{k=1}^{m} D_{o_k}. \tag{3.3}$$

Note that for the default determination guards $d_i = d_o = \mathit{true}$, this definition is equivalent to Def. 23.

As stated in Section 3.4.1, it is highly complex to prove the completeness of a realistic design by directly implementing Formula 3.3. In [41], this check is partitioned into four subchecks. Every subcheck concentrates only on a pair of (operational) properties. The user is required to specify the determination assumption for every input signal and the determination requirement for every output signal. Furthermore, the user needs to extract the conceptual states from the DuV, i.e., identify the key functionality of the design from a high-level abstraction view. With the aid of conceptual states, the user writes properties having the form described in Section 3.4.2. In addition, the user is required to specify a *property graph* $G = (V, E)$, which is tightly coupled with the conceptual state machine of a design. The vertices V of G are a set of properties $\{p_j\}$. Each p_j is a sequence

predicate that has a length l_{p_j} defined by the length of its assumption part A and its commitment part C. An edge $(p_j, p_k) \in E$ indicates that after the operation specified by p_j finishes, the operation specified by p_k may take place immediately. Note that the time point where p_k starts to execute from a conceptual state s_c is defined by the time point at which the previous p_j reaches the conceptual state s_c.

Given a property graph $G = (V, E)$ and the respective property set P, every pair of predecessor/successor in P is denoted by $(p_p, p_s) \in E$. Let (A_p, C_p) and (A_s, C_s) be the assumption parts and the commitment parts of p_p and p_s, respectively. For ease of explanation, the determination assumptions of all input signals are denoted by $D_i = \bigwedge_{j=1}^{n} D_{i_j}$ and the determination requirements of all output signals by $D_o = \bigwedge_{j=1}^{m} D_{o_j}$. During the completeness check, the determination assumptions D_i are assumed by every pair of properties and the determination requirements D_o must be fulfilled by every pair of properties. In addition, every property may have its *local determination requirements*, which only need to be fulfilled by the property containing them. Local determination requirements are of interest to apply on the states that are not related to conceptual states, but they are important to trigger a following operation. For instance, the state of an instruction register in a pipelined processor defines what instruction (operation) is to be executed next, but the operation starts from a conceptual state controlling the pipeline stages. Local determination requirements have the form of the determination conditions stated above. D_p denotes the local determination requirements of p_p and D_s denotes the local determination requirements of p_s. As previously discussed, to check the completeness of a property set, two instances of a property set are considered. Let A'_p, C'_p, A'_s and C'_s be the assumptions and commitments of a copy of each of the properties, p_p and p_s, respectively.

There are four checks performed by the completeness checker on a property graph G: the *determination test*, *successor test*, *case split test* and *reset test*. The validity of these four tests guarantees the completeness of the property set of a design, as formally proven in [41].

1) Successor Test: For every pair (p_p, p_s) of properties, the successor test checks whether the ending states of the predecessor p_p completely determine the starting states of the successor p_s. In other words, it checks whether the assumption A_s of the successor p_s contains only the input signals and the signals determined by the predecessor p_p. This is equivalent to the following tautology proof:

$$D_i \wedge A_p \wedge A'_p \wedge C_p \wedge C'_p \wedge D_p \wedge \mathit{next}(A_s, l_p) \Rightarrow \mathit{next}(A'_s, l_p)$$

Here l_p is the length of the property p_p.

2) Determination Test: The determination test checks whether the outputs of the design under verification are completely and uniquely described (determined) by the property set with regard to the determination requirements stated above. The test is conducted on the pair of predecessor/successor properties as follows:

$$(D_i \wedge A_p \wedge A'_p \wedge C_p \wedge C'_p \wedge \mathit{next}(A_s, l_p) \wedge \mathit{next}(C_s, l_p) \wedge$$
$$\mathit{next}(A'_s, l_p) \wedge \mathit{next}(C'_s, l_p) \wedge D_p) \Rightarrow \mathit{next}(D_o, l_p) \wedge \mathit{next}(D_s, l_p)$$

3) Case Split Test: So far it has only been confirmed there exists a unique sequence of operations for *some* inputs of the design. Obviously it should be shown that this is the case for every input. The case split test checks whether for every important state all input scenarios are covered. Note that the set of important states are given in the assumption parts and the commitment parts of the properties. The complexity of this check is reduced when only pairs of predecessor/successor are considered. In this case, it checks whether every operation starting from the subset of important states given by C_p is covered by at least one of the assumptions $\{A_{s_i}\}$ of all successors $\{p_{s_i}\}$ of p_p. This can be expressed in the following manner:

$$C_p \Rightarrow \mathit{next}((A_{s_1} \vee A_{s_2} \vee \dots), l_p)$$

4) Reset Test: The reset test is similar to the three tests described above. However the reset test is only applied to the reset property, which does not have any predecessors. It checks whether the reset sequence of a design is specified, and the behavior right after reset is determined.

Chapter 4

Assume-Guarantee Reasoning

Divide-and-Conquer is a common strategy to manage the complexity of system design and verification. In the context of SoC design verification, an SoC system is decomposed into several modules and every module is separately verified. Usually an SoC module is reactive: it interacts with its environmental modules. This interaction is normally modeled by environment constraints, which are applied to verify the SoC module. In order to guarantee that an SoC module that has been proven to work correctly under environment constraints also works correctly in the SoC system, several *Assume-Guarantee Reasoning* rules can be used instead of constructing a system model. Section 4.1 gives an overview of state-of-the-art assume-guarantee reasoning rules developed to be applied to various models of systems. This dissertation discusses the problems caused by reactive constraints which are formulated in a standard property specification language. Two plausibility checks for overcoming this problem are described in Section 4.2. Experimental results showing the efficiency of these plausibility checks are presented in Section 4.3. A new assume-guarantee reasoning rule considering the problem introduced by reactive constraints is introduced in Section 4.4. This rule is more general than the rules presented in Section 4.1: it is the fundamental theorem for proving compositional completeness under reactive constraints in Chapter 5.

4.1 Introduction to Assume-Guarantee-Reasoning

In today's industrial work flows, the individual modules of an SoC design are usually developed and verified simultaneously. In practice, it almost never happens that the modules of a system are designed for an unconstrained environment. Instead, designers usually rely on certain assumptions about the behavior of the environment. For example, a communication interface to a bus expects that the other participants of the communication also adhere to the protocol of the bus. In such scenarios, the behavior of a module M_i is only guaranteed if the environment satisfies a particular environment constraint C_i. If the environment does not comply with the constraint C_i, then the behavior of the module M_i may be undetermined. In other words, the module operates correctly only when its environment works correctly. Therefore, the environment constraints have to be selected

carefully, because any error or overconstraining in an environment constraint may cause bugs to remain undetected by property checking. The situation is similar in constrained random simulation.

Hence, the validity of environment constraints is a critical problem needing to be solved for SoC design verification. Furthermore, the correctness of the environment constraints of an SoC module guarantees that the module also works properly in the entire system. Theoretically, to validate the environment constraints, especially reactive environment constraints, one needs to reason over the entire system. This requires that the entire system be modeled appropriately, which is not feasible for industrial sizes of SoC systems. For this reason, instead of reasoning over the entire system at once, *assume-guarantee* rules that are composed of several smaller proof problems are developed and applied.

Assume-guarantee rules are not only useful to guarantee the correctness of environment constraints. They and their derivatives can also be used to guarantee that the properties which are valid for individual SoC modules are also valid for the composed system. In the context of the completeness analysis of property sets, assume-guarantee rules can be used to deduce that the entire system is completely verified when the individual modules have been completely specified. This will be discussed in detail in Chapter 5.

In a bottom-up verification flow, to manage the complexity of an SoC system, the system is usually decomposed into several modules. This can be done by exploiting a *cut* operation described in Def. 27.

Definition 27 (Model Decomposition)**.** *Let X be a set of internal signals of a sequential circuit M with n inputs and m outputs. For each signal $x \in X$, a cut operation is performed, introducing a new input $\hat{x}$ and a new output $\tilde{x}$. The resulting circuit model M' has $n+|X|$ inputs and $m+|X|$ outputs. Wherever the signal x is used in M to define the values of other signals, the free input $\hat{x}$ is used to define the corresponding signal in M'. The model M' is called the* result of the cut *X in M and is denoted by $M' = M/X$.*

Using the appropriate cuts, an SoC S can be decomposed into disjoint sub-circuits M_i. These sub-circuits are called the *modules* of the system. In practice, the boundaries of the modules, e.g., the port definitions of entity descriptions in VHDL, are mostly a good choice for a cut. In contrast, one may build a system by *composing* submodules. Given a set of modules $\{M_i\}$, the composition of a set of modules M_i into a system S is called *well-formed* if in the composition no combinational feedback loops are created.

SoC systems can be categorized in the following way: the sequential systems, whose modules are simply chained together and there are no interactions between modules; and the reactive systems, whose modules interact with each other. Usually most SoC systems are reactive. Similarly, the assume-guarantee rules can be categorized with regard to the systems they are applied to. Furthermore, with respect to the models of reactive systems, some assume-guarantee rules can only be applied to Moore machines, and some of them can be applied to Mealy machines.

Assume-Guarantee rules for sequential systems

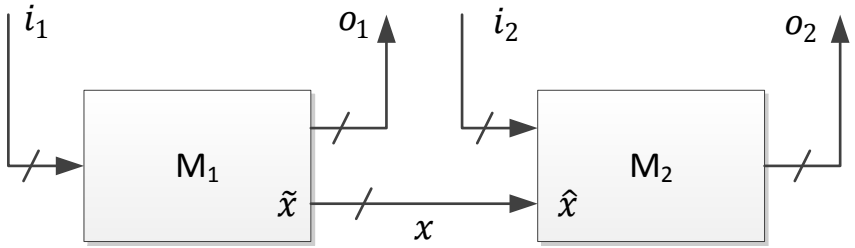


Figure 4.1: Composition Scheme of Sequential Systems

Let $M(i, s, o)$ be a system composed of two modules $M_1(i_1, s_1, o_1, \tilde{x})$ and $M_2(i_2, s_2, o_2, \hat{x})$ that are sequentially chained together (i.e., no interaction happens between them) by internal signals x, as depicted in Fig. 4.1. Here, $i_1 \cup i_2 \subseteq i$, $o_1 \cup o_2 \subseteq o$ and $s_1 \cap s_2 = \emptyset$. As defined above, $\tilde{x}$ and $\hat{x}$ are created by the cut operation. They represent the outputs of the module M_1 and the inputs of the module M_2, respectively. Let $p(i_2, s_2, o_2, x)$ be a property that is to be proven on the system M. Note that the property p depends only on the sequences of the signals in M_2 and the cut signals x. If the system is too complex for verifying the property p, M can be divided into M_1 and M_2. As the next step, one needs to find an assumption (constraint) for the module M_2: this assumption $\alpha(\hat{x})$ restricts the behavior of the signals $\hat{x}$. Then, the overall proof of the property p can be split into two subproofs, which guarantee the validity of p in system M due to the following assume-guarantee rule:

$$\frac{\begin{array}{l}\langle \mathit{true} \rangle\, M_1\, \langle \alpha(\tilde{x}) \rangle\,, \\ \langle \alpha(\hat{x}) \rangle\, M_2\, \langle p(i_2, s_2, o_2, \hat{x}) \rangle\end{array}}{\langle \mathit{true} \rangle\, M\, \langle p(i_2, s_2, o_2, x) \rangle} \tag{4.1}$$

This notation for assume-guarantee rules was firstly introduced by Pnueli [85], and this rule can be interpreted as follows: if M_1 guarantees the predicate $\alpha(\tilde{x})$ in an unconstrained manner, and M_2 guarantees the property p under the environment constraint $\alpha(\hat{x})$, then the system M fulfills the property p. The complexity of every subproof is lower than the complexity of the original proof problem. For proving the property p on M_2, an environment constraint $\alpha(\hat{x})$ is applied: it is an abstraction of M_2's environment, M_1 in this case, and it is much smaller than M_1.

The validity of Rule 4.1 can be proven straightforwardly as follows:

Proof. It is assumed that all the preconditions of this assume-guarantee rule are fulfilled. In order to prove the validity of this rule, one needs to show that every input sequence of

M determines a sequence of internal signals in M, and in particular, for the signals of cut X, so that the property p holds for M.

Because there is no feedback path from M_2 to M_1, the circuit M_2 consumes every sequence of $\tilde{x}$ generated from M_1, i.e., $\tilde{x} = \hat{x} = x$. Due to the second precondition, for each input sequence of i_2 and each sequence of $\hat{x} = x$ that satisfies the assumption $\alpha(\hat{x})$, the circuit M_2 determines a sequence of internal signals and outputs that fulfills the property p. For other sequences of x, the circuit M_2 might not satisfy the property. Therefore, one needs to show that all sequences of x that are generated by the circuit M or M_1 satisfy the assumption α. This is guaranteed by the first precondition. On this account, the property p holds for M. □

The assumption α can be given manually or can be found automatically as presented in [45]. It is straightforward to extend the assume-guarantee Rule 4.1, so that it can be applied to systems with more than two modules, as long as every two modules are sequentially chained. This rule is similar to the Hoare rules/triple [86] in software analysis. For example, in a Hoare triple $\{P\}S\{Q\}$, P denotes the *precondition* of a run/execution of the software S, and Q specifies the *postcondition* of the execution, e.g., which state the system should reach when S finishes its run under the precondition P. The Hoare rules guarantee the soundness of composing several sequentially executed programs that are specified by Hoare triples.

Usually, individual modules M_1 and M_2 are designed and verified by different teams of engineers. The predicates describing the interfaces between M_1 and M_2 might be different too. This results in the following more general assume-guarantee rule:

$$\frac{\begin{array}{l}\langle \mathit{true} \rangle\, M_1\, \langle \alpha(\tilde{x}) \rangle\,, \\ \langle \beta(\hat{x}) \rangle\, M_2\, \langle p(i_2, s_2, o_2, \hat{x}) \rangle\,, \\ \alpha[\tilde{x} \leftarrow \hat{x}] \rightarrow \beta(\hat{x})\end{array}}{M\, \langle p(i_2, s_2, o_2, x) \wedge \alpha[\tilde{x} \leftarrow x] \wedge \beta[\hat{x} \leftarrow x] \rangle} \tag{4.2}$$

Here, the *guarantee* provided by the module M_1 is α and the constraint (assumption) consumed by the module M_2 is β. It should be noted that in the third precondition of Rule 4.2, each occurrence of $\tilde{x}$ in α has to be replaced by the corresponding signal in $\hat{x}$. The by-product of Rule 4.2 is that the predicates α and β are valid in M as well. The proof of the validity of Rule 4.2 is straightforward and similar to the proof of that of Rule 4.1.

Assume-Guarantee rules for reactive systems (Moore machines)

The assume-guarantee rules 4.1 and 4.2 are rarely found in use for today's System-on-Chip designs, since the modules of an SoC usually interact in more complex ways and the environment constraints are often *reactive*. Put another way, the correctness of the environment of a module depends on the module itself. This leads to a situation where the circular reasoning at assume-guarantee-reasoning needs to be broken and where the task of detecting errors in constraints becomes much harder.

Definition 28 (Reactive environment constraints)**.** *A reactive environment constraint models the interaction behavior between the design under verification and its environment.*

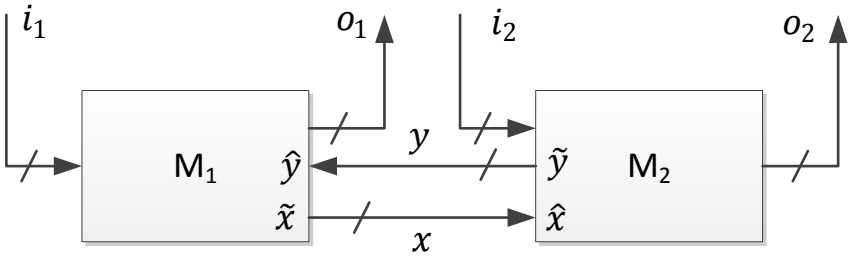


Figure 4.2: Composition Scheme of Reactive Systems

Consider a system M with modules M_1 and M_2 as in Fig. 4.2. The interaction between M_1 and M_2 happens via two groups of signals x and y. After applying a cut operation to signals x and y, the system M is decomposed into submodules M_1 and M_2. The new signals $\hat{x}$ and $\hat{y}$ are inputs to the modules M_2 and M_1, respectively, and the new signals $\tilde{x}$ and $\tilde{y}$ become the outputs of M_1 and M_2, respectively. Suppose that the sequence predicate $\hat{\alpha}(\hat{x}, \tilde{y})$ needs to be assumed by M_2 to verify the sequence predicate $\tilde{\beta}(\hat{x}, \tilde{y})$. Oppositely, the module M_1 needs to guarantee the sequence predicate $\tilde{\alpha} = \hat{\alpha}[\hat{x} \leftarrow \tilde{x}, \tilde{y} \leftarrow \hat{y}]$ under the assumption $\hat{\beta} = \tilde{\beta}[\hat{x} \leftarrow \tilde{x}, \tilde{y} \leftarrow \hat{y}]$. To validate the property $\alpha = \hat{\alpha}[\hat{x} \leftarrow x, \tilde{y} \leftarrow y]$ and the property $\beta = \tilde{\beta}[\hat{x} \leftarrow x, \tilde{y} \leftarrow y]$ in the system M, one may use a preliminary assume-guarantee rule as follows:

$$\frac{\begin{array}{c}\langle\hat{\beta}\rangle M_1 \langle\tilde{\alpha}\rangle, \\ \langle\hat{\alpha}\rangle M_2 \langle\tilde{\beta}\rangle\end{array}}{M\langle\alpha \wedge \beta\rangle} \tag{4.3}$$

At first glance, the assume-guarantee rule 4.3 is circular and not sound, because the correctness of M_1 is assumed to prove the correctness of M_2, and symmetrically, the module M_1 is proven under the assumption that M_2 behaves correctly. However, this rule is not always circular if the temporal behavior of the two modules is considered. In other words, the circular reasoning might be broken with respect to the chosen model of the system.

If every module of a system is modeled as a Moore machine, then the system model turns out to be the product machine of the models of modules. The assume-guarantee rule 4.3 can be viewed as

$$\frac{\begin{array}{c}\langle\hat{\beta}^t\rangle M_1 \langle\tilde{\alpha}^{t+1}\rangle, \\ \langle\hat{\alpha}^t\rangle M_2 \langle\tilde{\beta}^{t+1}\rangle\end{array}}{M\langle\alpha \wedge \beta\rangle} \tag{4.4}$$

The two preconditions of the rule 4.4 can be understood as follows. If the assumption $\hat{\beta}$ holds for any timepoint t (or it holds until any timepoint t), the module M_1 guarantees the commitment $\tilde{\alpha}$ at the next time point $t+1$ (or until the time point $t+1$); and if the assumption $\hat{\alpha}$ holds for any timepoint t, the commitment $\tilde{\beta}$ is valid in M_2. The circularity of the rule 4.4 is then broken inductively over time points, because there is a delay of one time point between the inputs and outputs of a Moore machine. The validity of this rule can be proven by contradiction as follows:

Proof. It needs to be proven that the composite system M fulfills the properties α and β when the two preconditions are valid. In other words, it needs to be shown that the system M behaves correctly when the module M_1 of M behaves correctly under the assumption that the module M_2 of M behaves correctly, and vice versa. Suppose that the system works correctly until a time point t. At time point $t+1$ an error occurs at module M_1, so that the time point $t+1$ is the earliest time point at which the system does not behave correctly. However, the module M_1 behaves incorrectly only when its environment M_2 does not work well. The error in M_2 should be happening at time point t. This contradicts the assumption that the earliest time point for the occurrence of an error is $t+1$. Therefore, the rule 4.4 is correct for Moore machines. □

The assume-guarantee rule 4.4 for Moore machines has been presented in many places in the literature ([42–45], to name a few). In [42], the assume-guarantee rules similar to the rule 4.4 are introduced for compositional symbolic model checking. There, to cope with the complexity of the systems, a large system is decomposed into submodules. Every module is verified separately in a constrained manner. The constraint (assumption) of every module is implemented by a "process" which can be seen as a Moore automaton. However, modeling constraints by using automata is sometimes tedious and error prone, especially for constraints whose behavior ranges over many clock cycles. [43] introduces a set of compositional rules based on assume-guarantee-reasoning, to conjoin the specifications of the modules. In [43], every module is specified by a specification in an assume-guarantee style. By using a set of compositional rules it can be deduced that the system level specification (assume-guarantee style) is fulfilled by the composite system. The compositional rules in [43] are similar to the checks presented in Chapter 5 of this dissertation. They can, however, only be used on Moore machines. Their work does not address the problem of the completeness of the specifications and the problem of potential circular reasoning due to using reactive constraints specified in a standard property specification language.

The research presented in [46] and [47] extends the assume-guarantee rule 4.3 to Mealy machines and requires that there are no combinational loops in the composed system. In this dissertation, systems are also modeled as Mealy machines and are required to be combinationally loop-free. Furthermore, this dissertation provides a check in Section 4.2 to avoid combinational loops existing in the final composite system. This check can be conducted on every module of the system, and it does not need to involve the composite system.

So far, the above presented assume-guarantee rules are applicable to assumptions and commitments that are both safety properties. An assume-guarantee rule for liveness properties is presented by [87], which is outside of the scope of this dissertation.

In fact, the above stated assume-guarantee rule 4.4 for Moore machines and the rules presented by [46] and [47] for Mealy machines are only sound when the constraints of the modules are *implementable*. A constraint is implementable if there exists a circuit implementing the constraint. An example that shows assume-guarantee rules not being sound due to using non-implementable constraints can be found in Section 5.1. Unfortunately, none of [42–47] states explicitly that the constraints must be implementable. Even the argumentation for proving Eq. 4.4 neglects the fact that α and β must be implementable. In [42], the constraints are represented by processes which are already implementations of constraints. The assume-guarantee rules presented by [46] are applied to (hardware/-software) models and their refinements, which are always implementable. In other words, the language used for describing such models can only create implementable models. Similarly, [43–45, 47] are implicitly based on this fact too.

However, in this dissertation, reactive constraints, which are used as assumptions for verification, are described by using a standard property specification language like SVA or PSL. This may lead to a situation where the described constraint can be non-implementable. An extreme case of a non-implementable constraint is that a constraint is always *false*. Using this constraint for property checking causes every property to be trivially true. On the other hand, describing constraints that are safety properties by using a property specification languages such as SVA has huge advantages. As explained at the beginning of Section 3.3, the constraints specified in SVA are more compact and easy to understand. Furthermore, constraints in SVA can be reused for simulation-based tools, because it is a standard language. For these reasons, the method presented in this dissertation can be easily adopted and integrated into existing design verification flows for SoC systems. Therefore, in this application domain, it is crucial to break circular reasoning due to non-implementable constraints. For this purpose, a plausibility test to check the implementability of a constraint is introduced in Section 4.2. And a new and more general assume-guarantee rule for Mealy machines considering all kinds of constraints (reactive or nonreactive, implementable or non-implementable) is presented in Section 4.4. In addition, another plausibility test is also described in Section 4.2, which identifies combinational loops in the system. These two plausibility tests can be used at the early stage of the design process, before integrating the submodules of the system.

4.2 Plausibility Checks for Reactive Environment Constraints

Environment constraints are of major importance for the functional verification of System-On-Chip modules. Regardless of whether a design is verified using formal techniques or whether a classical simulation-based verification approach is chosen, a verification engineer in either case needs to model the environment of the DuV as well.

Classical directed testbenches constrain the behavior of the environment in a rather implicit manner by the set of stimuli generated during the simulation. More advanced techniques such as constrained random simulation or formal assertion-based verification require an explicit specification of the environment constraints [88, 89]. The holy grail of formal verification is its promise to completely exercise the entire input, output and state space of a design in order to prove that there are no bugs. It has been demonstrated that formal property checking, if complemented with a coverage analysis for the property set [41, 48, 75, 81, 84] reaches this goal with reasonable verification effort. The coverage analysis identifies specification gaps in cases where particular input scenarios are not covered by any of the specified properties. Again, environment constraints are used to restrict the analysis to relevant scenarios.

Verifying a design with respect to a constrained environment always runs the risk of masking bugs. Moreover, such overconstraining is more difficult to detect than other specification mistakes, as it does not show up in a counterexample for a failing property. In industrial practice, the review of constraints is thus taken very seriously. However, sometimes a collection of environment constraints may imply subtle consequences that may not be immediately obvious to the verification engineer. In particular, it may be the case that a collection of fairly simple constraints, where each individual constraint has a very reasonable intention, turns out to be problematic when the constraints are applied in their entirety.

Vacuity checking may guarantee that at least one input scenario exists such that the constraint is satisfied. However, it may only identify a case where a constraint cannot be satisfied at all, and leaves many other constraint issues undetected.

This section presents plausibility tests for environment constraints in a SAT-based property checking environment [20,83,90,91]. The vacuity check is a special case covered by the tests presented in this section. It will demonstrate that these checks can identify critical constraint issues and may guide the user to specify consistent constraints. The presented techniques are useful not only in the setting of formal verification, but also in a constrained random simulation environment. The reason to call these tests "plausible" is that these tests do not check whether a constraint specifies an environment exactly as expected by the design engineer. In other words, these tests do not check whether the environment of a module can fulfill the constraint of the module. (Such a test will be provided in Chapter 5.) The plausibility checks presented in this section are mainly intended to find inconspicuous errors in constraints as early as possible, so that verification engineers gain more confidence in the environment constraints used.

In particular, the notions of implementability and of loop-free composability of constraints are introduced. This formalizes the reasonable requirements that there should exist at least one environment that is able to fulfill the constraints in such a way that the composition of the environment and the design is a valid circuit, i.e., the composition does not contain any combinational loops in its logic.

The effectiveness of the presented plausibility tests is shown with a case study conducted in an industrial setting, where the constraints for a verification IP are analyzed with the plausibility tests introduced in this section. The verification IP specifies the pro-

tocol compliance of an Infineon device for the Flexible Peripheral Interface (FPI) bus. The formal property checker OneSpin 360 DV [83] is used to check the properties against the device and to prove the completeness of the verification IP. The entire verification is conducted under the assumption of a set of environment constraints. These constraints describe, for example, the protocol-compliant behavior of the bus signals that are inputs to the DuV. This experimental result is presented in Section 4.3.

Obviously sequence predicates are suitable formalisms for reactive constraints that model the sequential behavior of the interfaces of the DuV. In other words, a reactive constraint only depends on the interface signals of the DuV, and it is a safety property assumed to be true for the DuV. Certainly this safety property needs to be validated by the environment of the DuV. (This problem will be addressed in Chapter 5.) Reactive constraints need to be selected carefully in order to ensure that they can actually be fulfilled by the environment and that they do not overconstrain the design. This is of particular interest when different portions of a design have been developed by different IP providers and the final environment for the design is not known prior to the integration phase. At that stage of the project it is often too late if inconsistent environment constraints for the individual modules are detected and the required changes may cause a project to fail or miss its deadline. Two plausibility checks for reactive constraints of a module will be proposed, that can be checked without a concrete environment at hand.

4.2.1 Implementable Constraints

It is a reasonable requirement for a sequence predicate α_l for it to be used as environment constraint for a circuit model M that at least one environment must exist that can fulfill the constraint. In other words, it needs to be ensured that a model M_E (modeling the environment) exists that satisfies α_l. This environment model computes the inputs to the DuV using the DuV's outputs and possibly also internal signals from the DuV that are used to specify α_l. In addition, it may use additional free inputs to model some non-determinism in the constraint. If such a model M_E exists, the constraint α_l is called *implementable*. For implementable constraints, one may generate a *most-general* implementation that can exhibit every behavior that is not explicitly forbidden by the constraint. In the literature, such an implementation is also known as a *can-do object* [92]. Another way to derive such an implementation is to apply the synthesis techniques of [93].

However, here it is more interesting to focus on the *existence* of an implementation of an environment constraint rather than the actual implementation itself. For the typical constraints encountered in practice, this question can be decided without a full-blown synthesis of can-do objects. Such a test is developed in the sequel.

As an example of a non-implementable environment constraint, consider the sequence predicate $i = \mathit{next}(o)$ for an input i and an output o of M. For the environment M_E, i is an output and o is an input. Unless the output is constant, the constraint models a precognitive, i.e., non-causal, environment that obviously does not exist.

Checking that an arbitrary sequence predicate is implementable is a non-trivial task. To simplify this task, the syntax of the language used to specify the environment con-

straints α_l is restricted in such a way that only causal constraints can be formulated. Furthermore, the constraints used in the context of assume-guarantee reasoning must solely model the interface behavior between modules. More specifically, the constraints can only contain the interface signals of the DuV, i.e., the input signals and the output signals. The reason for this restriction is that if a reactive constraint α of the module M_1 involves M_1's internal behavior, it needs to consider the composite system as a whole to validate α. This violates the intention of the divide-and-conquer approach.

The input value i_l at the current time point l may be restricted by a *basic constraint* of the form

$$c^k(\xi_l, \zeta_l) := i_l \triangleleft \epsilon^k(\xi_{l-1}, \zeta_l), \quad \triangleleft \in \{\rightarrow, \leftarrow\},$$

where $\xi_l = (i_0, i_1, \ldots, i_l)$ is the set of input sequences and $\zeta_l = (o_0, o_1, \ldots, o_l)$ is the set of output sequences. The form of the basic constraints guarantees that the current input i_l is determined by the historic input value and the output value up to the current time point, and it does not depend on their future values. In addition, the current input may be restricted by multiple basic constraints, which constitute the overall constraint $\alpha_l(\xi_l, \zeta_l) = \bigwedge_k c^k(\xi_l, \zeta_l)$ of the module under verification.

The individual constraints c^k are, obviously, implementable and thus causal. However, this does not guarantee that α_l is also implementable. To see this, consider the state predicate $o_1 \rightarrow i \wedge o_2 \rightarrow \neg i$, which places two constraints on the input i depending on two current outputs o_1, o_2 of the module. This restricts o_1, o_2 in such a way that they can not take the value 1 at the same time. However, the outputs of the module are inputs to its environment and cannot be controlled by any circuit implementing the environment. Thus, the constraint is not implementable.

In order to guarantee implementability, it needs to be checked whether the constraint α_l can be fulfilled regardless of the output sequence generated by the module and regardless of the previous inputs that have been asserted. This yields the following QBF formula:

$$\forall_{o_j, j=0,\ldots,l} \forall_{i_h, h=0,\ldots,l-1} \exists_{i_l} : \alpha_l(\xi_l, \zeta_l).$$

The constraint α_l is implementable if the above QBF formula is a tautology. The intuition behind this formula is the following. In order for the environment constraint α_l to be implementable, it is necessary that the outputs of the environment (i.e., the inputs of the DuV) are in a causal functional relationship with the inputs of the environment (i.e., the outputs of the DuV). This is the case if for every combination of DuV output l-sequence and DuV input $l-1$-sequence ("history") there exists a ("current") DuV input complying with the constraint. (No future time references are allowed in the constraint.) To disprove implementability, it is thus sufficient to check the following formula for satisfiability.

$$\exists_{o_j, j=0,\ldots,l} \exists_{i_h, h=0,\ldots,l-1} \forall_{i_l} : \neg\alpha_l(\xi_l, \zeta_l).$$

This formula can be solved by explicit elimination of the inner universal quantifier and a propositional SAT solver. In practice, this is feasible because most often α_l is

merely a conjunction of basic constraints where any input at time point l only depends on a very limited number of other inputs. The negation, $\neg\alpha_l$, is thus a disjunction of the negated basic constraints. Furthermore it is possible to group the inputs with respect to the dependency relation induced by the basic constraints in such a way that no dependency between inputs of different groups exists. The overall check for the implementability of the constraint α_l thus can be split into subchecks, in which the inputs for each group are individually quantified. The constraint α_l is implementable if every group can pass the implementability check.

4.2.2 Loop-free Composability

A design under verification and its environment are modeled as finite state machines. Properties describe the behavior of a system only at discrete synchronous time points (usually with reference to a clock signal) and in steady-state conditions. A property checker for this kind of property cannot verify asynchronous behavior that results from combinational loops. In other words, the oscillating behavior due to combinational loops is not supported by a property checker. In this case, the property checker tries to stabilize the values of the signals inside the combinational loops. The value causing the oscillating behavior is implicitly assumed to not occur in the design. If a constraint builds a combinational loop with its DuV, this may lead to a problem that the constraint unexpectedly overconstrains the DuV. With regard to the proof of assume guarantee rules for a Mealy machine, the oscillating behavior causes the assume-guarantee rules to be unsound, because the proof needs the values of the signals to be stable at every time point. Furthermore, combinational loops also make difficulties for the timing analysis and automatic synthesis of a design. By all accounts, combinational loops between the environment and the DuV need to be excluded. Therefore, the second plausibility test for environment constraints considers the composition of the module with a hypothetical implementation of the constraint and makes sure that no combinational loops are created by this composition. (It is obviously only applicable to implementable constraints.)

At first glance, it seems appealing to require that *every* implementation of an environment constraint could be safely composed with the DuV. However, it turns out that this requirement is too strict. Consider a module with inputs i_1, i_2 and an output o. Assume a combinational dependency between o and i_2 but not between o and i_1. Suppose the environment constraint $i_1 = o$. Every environment connecting i_1 with o implements the constraint. Under all such environments only those are composable that do not connect i_2 and o. Obviously this is not an issue for the constraint that is not talking about i_2 at all.

Instead of considering all implementations of the constraint, one may resort to an existential requirement. In this case, one would require that at least one environment for the module exists such that it implements the constraint and does not produce any combinational loops in the composition with the module M. However, this may cause problems as well. To see this, consider the constraint $x_1 = y$ for a 2-input AND gate with inputs x_1, x_2 and output y. In this case, the constraint can be implemented by connecting the output y with the input x_1. This leads to a combinational loop in the composition.

Certainly this implementation in some sense is the most reasonable implementation of the constraint and we would like to forbid this type of constraint to prevent the resulting combinational loops. However, there are alternative implementations. For example the circuit producing a constant zero output for x_1 also turns out to be a valid implementation of the constraint. Additionally, this implementation does not produce a combinational loop with the AND gate.

The requirement that only a single implementation of a constraint be composable with the module M without combinational loops is, therefore, too weak. To rule out over-constraining implementations like the constant-zero implementation for the above mentioned constraint, the *most-general* implementation of a constraint is considered in this dissertation.

Definition 29. *A most-general implementation of a constraint, also referred to as can-do object [92], is a circuit that can produce every behavior that is not explicitly forbidden by the constraint.*

Note that neither the constant-zero implementation of the previous example nor the implementation connecting o and i_2 in the earlier example are most-general implementations, because they restrict behavior that is definitely allowed by the respective constraints. It turns out that most-general implementations are well suited for the definition of loop-free composable constraints:

Definition 30. *An implementable constraint α is called loop-free composable with a module M if a most-general implementation M_α of α exists that can be composed with M without introducing combinational loops.*

A naive way of checking loop-free composability as defined above would explicitly generate the most-general implementation. (Note that this would be possible with the synthesis techniques of [92].) However, here, the combinational dependencies are more interesting than the exact functionality of these implementations. Therefore, only a much simpler structural analysis is needed, which is conducted on the original constraints. In this analysis, the combinational dependencies are extracted from the basic constraints c^k of the constraint α. Every pair of inputs and outputs that occurs with the same temporal offset of the *next* operator is considered as a potential combinational dependency. (Due to the causality of the basic constraints, there cannot be any other such dependencies.) The dependencies derived from the constraint in this way are added to the signal dependency graph for the inputs and outputs of the circuit model M of the DuV. The resulting dependency graph for the system is then analyzed for loops. Note that in cases where the model of the DuV is not yet available because formal specification and design are developed in parallel, one may refer to the properties of the design and conduct a similar analysis as for the constraint to generate a signal dependency graph. In either case, an acyclic extended dependency graph guarantees the existence of a safe environment.

Note that the presented method is based on the syntax analysis of the code of constraints. It provides an overapproximated result. Therefore, false negative counterexamples due to "false" loops may occur. A false loop is a loop that looks like a combinational

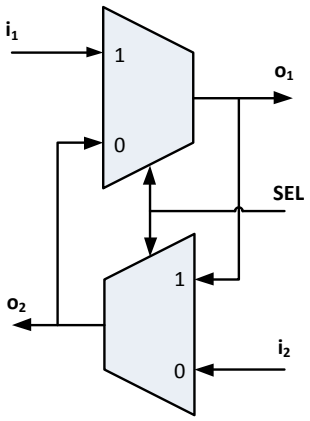


Figure 4.3: An example of false loops

loop (syntactically), but this loop (path) is never closed during a design run. An example of false loops is given in Fig. 4.3. In this example, the suspected combinational loop involving signals o_1 and o_2 can never be logically closed. To overcome the problem of false negatives, one has to synthesize the most general implementations of constraints M_α and rigorously analyze combinational loops in the composite circuit built from M_α and M. However this increases the complexity of the plausibility test for combinational loops.

4.3 Experimental Results

The plausibility checks for environment constraints developed in this dissertation have been successfully evaluated in an industrial setting. A formal specification of a master interface for Infineon's Flexible Peripheral Interconnect (FPI) bus [94] is considered as a case study. The specification ensures compliance of a particular implementation with the FPI bus protocol. This protocol is quite similar to the industry's standard AMBA bus protocol [95]. It includes features like pipelining of transactions to improve the throughput of the bus.

The main task of the considered master interface is to adapt a particular processor interface for reading and writing information from/to peripherals to the particular FPI bus protocol. Each interface transaction of the processor is mapped onto a respective protocol transaction on the bus. Due to the pipelining features of the protocol, the interface is able to handle two concurrent requests from the processor.

An industrial design of such an FPI bus master interface has been completely verified by applying the OneSpin 360 DV interval property checker [83] with its extended features for completeness checking. The completeness of the specification was derived under an environment constraint composed of 72 basic constraints involving 36 signals.

The inputs and outputs of the considered design can be grouped into two categories. The first category is used in the communication with the processor while the other cate-

gory forms the actual interface to the bus.

After thorough review of the environment constraint for the DuV by the verification and design teams the verification engineer was convinced that the constraint precisely captures the required environment behavior under which the DuV was supposed to meet its specification.

However, the plausibility checks developed in this dissertation revealed two serious issues that may easily have masked a bug if they had remained undetected. For the check, the dependency relation between signals referred to by the constraint was used to decompose it into nine groups of basic constraints referring to pairwise disjoint sets of signals. The plausibility checks were individually applied to each group. In one of the groups two bugs were detected. This group still has a source code size of about 30 lines.

This explains why under these circumstances the subtle interdependencies detected by the plausibility checks were overlooked by the verification team. In contrast, the fully automatic plausibility checks presented in this dissertation analyzed each group within 190 ms, using less than 95 MB of memory on an Intel Core i7 CPU 860 at 2.8 GHz with 8 GB of RAM.

In the case of the above-mentioned issues, the code fragments that caused the plausibility tests to fail stem from distant locations within the constraint specification. This is true even though the verification engineer put significant effort into ordering the basic constraints in such a way that expressions referring to similar sets of signals would be located close to each other.

In what follows, a pseudo-code notation is used to illustrate the nature of the identified issues within the constraints of the master interface of FPI bus. For reasons of space, only the relevant subexpressions that actually form the bug are presented here.

The first issue identified by the plausibility checks is related to the *ready_i* signal in the FPI bus. Fig. 4.4 illustrates two subexpressions of the corresponding constraint that introduce an issue regarding implementability. Within these code snippets, the suffix 'i' indicates that a signal is an input of the DuV, whereas the suffix 'o' identifies outputs.

```
if bus_is_idle_i then
        next(ready_i) = '1';
end if and
if next(this_master_is_driving_bus_o) then
        next(ready_i) = next(ready_o);
end if;
```

Figure 4.4: Pseudo-code related to the first bug

The code fragment states that whenever the FPI bus is in the idle state, at the next time point the *ready* signal of the bus should have the Boolean value '1'. The latter indicates that there must be a master or a slave in the system selected to drive the bus signals. The second part of the code fragment considers the case that the DuV itself is selected to drive

the bus. In this case, the *ready_i* signal should actually correspond to the value of *ready_o* provided by the DuV.

It turns out that this constraint fails within the implementability check. A counterexample shows that for the present time point, whenever the previous value of *bus_is_idle_i* is logical '1', and the DuV is driving the bus, and the value of *ready_o* is logical '0', no value for the signal *ready_i* is available to satisfy the constraint. Actually, this constraint has the subtle consequence that if at a previous time point the bus is in the idle state and at present the DuV is selected to drive the bus, the *ready_o* must be '1'.

Note that this implication of the constraint is actually one of the verification goals for the DuV, i.e., the constraint would have masked cases where this requirement is not fulfilled. To resolve this problem, one may use a cascaded if-then-else structure to formulate

```
if next(this_master_is_driving_bus_o) then
       next(ready_i) = next(ready_o);
else if bus_is_idle_i then
       next(ready_i) = '1';
end if;
```

Figure 4.5: Possible solution to the first bug

the constraint, as listed in Fig. 4.5.

The second issue of the constraints of the master interface was discovered by the loop-free composability check specified in 4.2.2. The pseudo-code in Fig. 4.6 illustrates a constraint on the input *grant_i*. It states that whenever the DuV is active and locks the request line, the arbiter should grant this request immediately. The output *active_o* is not a part of the protocol. With this signal the DuV may indicate whether it still needs the bus or whether the bus could be put into a power saving mode. It should be connected to a power management unit in the environment of the DuV. This unit may then determine whether the bus is in use by some other system components and otherwise assert additional signals to set the bus in sleep mode as well.

```
if lock_req_o = '1' and active_o = '1' then
       grant_i = '1';
end if;
```

Figure 4.6: Pseudo-code related to the second bug

Obviously, the above constraint introduces a combinational dependency between *active_o* and *grant_i*. Unfortunately, these two signals are also in a combinational relationship in the circuit model of the DuV. Hence, the constraint causes a combinational loop with the DuV. This combinational loop may overconstrain the design, because the verification tool can only consider the steady-state behavior caused by this loop. The

consequence of this combinational loop is illustrated in Fig. 4.7. Since the signals along the combinational loop (green part of the figure) may oscillate, in order to keep the loop stable (steady state), some signals in the fan-in of *active_o* must be restricted to a subset of the values they should have. This makes the fan-out of these overconstrained signals to be constrained unexpectedly (red part of the figure).

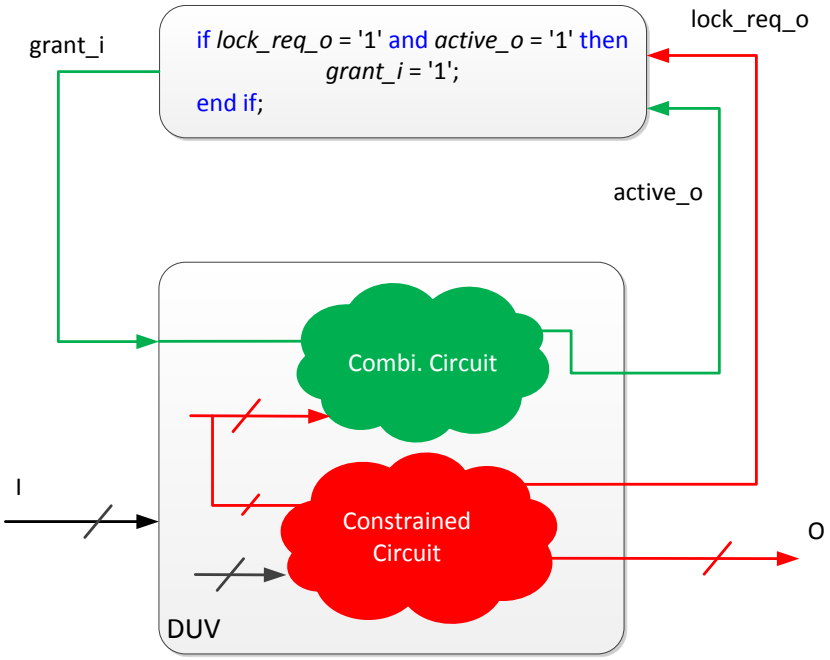


Figure 4.7: Consequence of combinational loops

Further investigation of the constraint reveals that the signal *active_o* is not necessary to formulate the constraint. The request line of the design can only be locked if the design is active at the same time. This relationship between the outputs should be checked by the verification IP of the module and not be implied from the environment constraint. Removing the *active_o* signal from the constraint is therefore a possible solution in this case.

4.4 Assume-Guarantee Reasoning over Cuts

Within the scope of this dissertation, standard property languages, such as SVA or PSL, are used to specify the constraints of a module. The problem with using these languages in the context of plausibility tests for constraints is that the user may unintentionally create non-implementable constraints or constraints that produce combinational loops with the DuV. This may lead to overconstrained designs during verification.

If a non-implementable constraint is used as the assumption for one of the preconditions of an assume-guarantee rule, e.g., Rule 4.4, then the validity of this precondition is no longer trustable. This invalidates the soundness of the assume-guarantee rule.

In this section, a new and more general assume-guarantee rule is presented, which explicitly addresses the above stated problems of constraints, in order to allow using a standard property language, such as SVA, to specify the constraints. Furthermore this assume-guarantee rule is used as the fundamental theory for the compositional completeness of property sets presented in Chapter 5.

Consider a circuit model M with a set of internal signals X. By applying a cut operation on X according to Def. 27, the circuit model is transformed to $M' = M/X$. The cut operation introduces a set of pseudo-inputs $\hat{x}$ of M' and a set of pseudo-outputs $\tilde{x}$ of M'. A constraint $\hat{\alpha}(\hat{x}, \tilde{x})$ is used to restrict the pseudo-inputs $\hat{x}$ in terms of the pseudo-outputs $\tilde{x}$. An assertion $\tilde{\alpha}$ is constructed from the constraint $\hat{\alpha}$ by using $\tilde{\alpha}(\tilde{x}, \hat{x}) = \hat{\alpha}[\hat{x} \leftrightarrow \tilde{x} : x \in X]$. In other words, the assertion $\tilde{\alpha}$ is created by replacing the signals $\tilde{x}$ in $\hat{\alpha}$ with the signals $\hat{x}$, and vice versa. Furthermore, it is assumed that the assertion $\tilde{\alpha}$ is valid for the modified model M' under the constraint $\hat{\alpha}$. Note that this assumption is not circular reasoning, because two sets of variables $\hat{x}$ and $\tilde{x}$ are used to distinguish the roles of the signals as inputs or outputs. In addition, the pseudo-inputs and pseudo-outputs change their role when one considers M' as its environment (constraint restricting pseudo-inputs) to prove the assertion about the pseudo-outputs. From these given conditions, the assertion $\alpha(x) = \hat{\alpha}[\hat{x} \leftarrow x, \tilde{x} \leftarrow x : x \in X]$ is proven in model M using the assume-guarantee rule as follows:

$$\frac{\begin{array}{l}\langle\hat{\alpha}\rangle M'\langle\tilde{\alpha}\rangle,\\ \hat{\alpha} \text{ is implementable},\\ \hat{\alpha} \text{ is loop-free composable with } \boldsymbol{M'}\end{array}}{M\langle\alpha\rangle} \tag{4.5}$$

This rule can be explained with a simple example. Consider a circuit M that consists of a single D-flipflop that reads its own state value s as the next state input, as illustrated in the left part of Fig. 4.8. Further assume that the D-flipflop is initialized with 0 on reset. Consider the cut $\{s\}$ in M. The model M' that results from this cut has an input $\hat{s}$ and and output $\tilde{s}$ and is depicted in the right half of the figure. In this model the property $\tilde{s} = 0$ holds under the constraint $\hat{s} = 0$. The constraint $\hat{s} = 0$ is obviously implementable by the constant signal $\hat{s} = 0$. This is the most general implementation of the constraint. Thus the constraint is loop-free composable with M'. With the assume-guarantee rule 4.5, one may now conclude that $s = 0$ holds in M (which is really true in design M).

The validity of the preconditions of the presented assume-guarantee rule can be proven by the plausibility tests introduced in 4.2 and a property checker. The validity of this assume-guarantee rule is proven in the remainder of this section.

Proof. It is assumed that the preconditions of the assume-guarantee rule are fulfilled. In particular, the property $\tilde{\alpha}$ holds for M' under the constraint $\hat{\alpha}$.

In order to prove the validity of the assume-guarantee rule, it needs to be shown that every input trace for the circuit M determines a trace for the internal signals of M and in

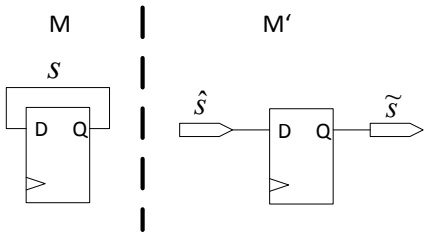


Figure 4.8: Example: D-flipflop

particular for the signals of the cut X such that the property $\mathsf{G}\,\alpha$ is fulfilled. Furthermore, the most general implementation of the constraint $\hat{\alpha}$ is assumed to be composed with M' as illustrated in Fig. 4.10. The existence of the most general implementation of the constraint is guaranteed by the precondition "$\hat{\alpha}$ is implementable." This implementation may depend on a finite number of free inputs $F = \{f_1, \ldots, f_s\}$ with $s \leq |X|$. These free inputs model the indeterministic description of a constraint that will be explained in terms of an example in Fig. 4.9. In this example, the value of the input *rdy_i* is constrained to "1" only when the output *req_o* of the DuV at the previous time point has the value "1". In other cases, the input *rdy_i* may have an arbitrary value, i.e., the value of *rdy_i* is undetermined. Note that this differs from the semantics of an HDL language like VHDL or Verilog, where a signal is determined to keep its value unless it is explicitly altered by an assignment statement.

```
if req_o = '1' then
        next(rdy_i) = '1';
end if;
```

Figure 4.9: Example for indeterministic constraints

Due to the implementability and loop-free composability of the constraint, the composition with M' is a well-formed composition. By the precondition of the assume-guarantee rule this circuit generates traces for the signals $\hat{x}, \tilde{x} : x \in X$ that globally fulfill the constraint $\hat{\alpha}$ and the assertion $\tilde{\alpha}$. However, depending on the choice of the input traces for the signals in F, the traces for the pseudo-inputs $\hat{x}$ may differ from the traces of the corresponding pseudo-outputs $\tilde{x}$.

As the next step, it needs to be shown that the traces for the signals F can be chosen in such a way that this is not the case. Suppose given an arbitrary trace for the signals $f_k \in F$, and consider the first time point t where $\hat{x} \neq \tilde{x}$ for any $x \in X$. There must exist a signal $\hat{x}$ within $X_{\text{diff}} = \{x : \hat{x} \neq \tilde{x}\}$ that does not have a combinatorial dependency on the other signals with this property. Otherwise, a combinational loop could be constructed in the composed circuit of Fig. 4.10. This signal thus only depends on the previous values of the pseudo-inputs that, by the selection of the timepoint, correspond to their respective pseudo-outputs. Moreover, M' produces an output that satisfies $\tilde{\alpha}$. The output value

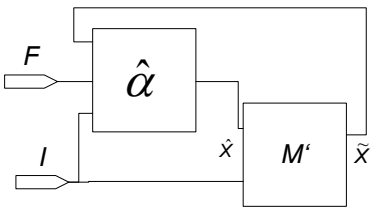


Figure 4.10: Circuit model for generating the initial trace

of $\tilde{x}$ at time t is thus a valid output with respect to $\tilde{\alpha}$. Therefore, this value is a valid output $\hat{x}$ of $\hat{\alpha}$ as well. In addition, since the most general implementation of $\hat{\alpha}$ exposes every permitted behavior of $\hat{\alpha}$, there must be a modified input trace for F' for $\hat{\alpha}$ with the following properties:

1. The output trace up to time $t-1$ remains unchanged.
2. The output for the other signals $x' \notin X_{\text{diff}}$ and the identity $\hat{x}' = \tilde{x}'$ at time t remain unchanged.
3. The difference between $\tilde{x}$ and $\hat{x}$ at time t is fixed.

With the remaining signals differing at time t under F', the corresponding F' is modified in the same manner. By induction over t, a trace F^* is constructed such that the traces for the pseudo-inputs and pseudo-outputs of M' are identical. Under this trace, F^*, the model of Fig. 4.10 produces traces $\hat{x} = \tilde{x}$, i.e., the signals $\hat{x}$ can be simply connected with the corresponding signals $\tilde{x}$ and the composition is equivalent to original model M with $x = \hat{x} = \tilde{x}$. At the same time, the signals $\hat{x} = \tilde{x}$ satisfy the constraint $\hat{\alpha}$ and the assertion $\tilde{\alpha}$. Hence, x satisfies α, i.e., the model M satisfies α. □

Chapter 5

Coverage of Compositional Property Sets

In order to overcome the complexity of the designs and the limitations of verification methodologies, *Divide-and-Conquer* approaches are also exploited for the coverage analysis of property sets. For instance, the verification methodology suggested by [83] can only show its efficiency if the overall SoC can be decomposed into several groups, each group containing only one conceptual state machine. In this case, the completeness of the property sets for the groups (modules) is checked module by module. If every module is verified in an unconstrained manner, and the property set of a module is proven to be complete under this circumstance, then no further efforts are needed to guarantee that the module works correctly in the system, and the completeness of the property sets of the modules implies that the overall system is completely verified. However, since usually modules are not standalone, they often operate in an interactive way. To verify every module and to check the completeness of the property sets for every module with the minimum necessary effort, reactive constraints have to be applied, which saves writing redundant and unnecessary properties. In this situation, the properties are valid and complete with respect to the applied reactive constraints. This chapter addresses questions like "How to ensure that the properties, which hold in every module, are also valid in the system?", "How to guarantee that the complete property sets of all modules cover every behavior of the overall system?".

This chapter presents a compositional reasoning framework for complete property sets under reactive environment constraints. The framework includes the necessary criteria for the constraints of the individual modules and, also, sanity checks of their interplay, so that one can conclude that the property sets of the modules are also valid in the entire system. Moreover, the union of the property sets of the modules completely verifies the entire system. This approach, although presented in the context of C-IPC (see Section3.4.2), can be used with other verification techniques as well.

The contribution in this chapter can be used in practice as follows. A system of RTL modules is verified by property checking as usual, module by module. Every module M_i is independently verified by C-IPC, based only on assumptions about the behavior of its

environment (which includes the other modules in the system). Not only the properties themselves, but also the completeness of the property set relies on the validity of these assumptions. Instead of checking the correctness of the assumptions manually, the software framework implementing the compositional criteria in Section 5.1 is used to check whether the guarantees given by the environment justify the assumptions used in the verification of M_i. The guarantees can be seen as the counterparts of the assumptions. They follow from the properties of the modules that make up the environment of M_i. The software framework only requires the assumptions and guarantees of the modules as input (the design itself is not considered), to check the compositional criteria. If this automatic check succeeds, then the correctness and completeness of the property sets are certified for the entire system. If the check fails, then a counterexample is generated to guide verification engineers in debugging errors or gaps in the assumptions and guarantees. This may lead to modifications of the designs or properties.

This chapter is organized as follows. The design-independent compositional rules are introduced in Section 5.1. These rules are directly implemented in a software tool within a SAT-based verification environment and evaluated on two SoC designs, with the results being presented in Section 5.2.

5.1 Compositional Rules for Property Sets

Before introducing the compositional rules for the compositional completeness of property sets, the definition of the completeness of the property set of a single module needs to be adapted to the situation where reactive constraints are exploited to check the completeness of property sets. Usually, the reactive constraints evaluate the previous and the current outputs of the module to determine the ranges of valid values for the inputs to the module. The guarantees provided by a module for its environment can be modeled in a similar manner, by properties that restrict the outputs of the device in terms of its previous and current inputs. Together with the determination conditions introduced in Section 3.4.2, the environment constraints and the guarantees of a module form the so called integration conditions. For ease of explanation, the determination assumptions are grouped with the environment constraints into the integration assumptions of a module. Likewise, the determination requirements are grouped with the module's guarantees to form the integration commitments of the module. The integration assumptions and commitments together constitute the integration conditions.

By replacing Formula 3.3 in Def. 3.1 with Formula 5.1, the completeness of the property set for a module with respect to its integration conditions can be checked as follows:

$$c \wedge c' \wedge \bigwedge_{j=1}^{n} D_{i_j} \wedge \bigwedge_{p_l \in P} p_l \wedge \bigwedge_{p'_l \in P'} p'_l \Rightarrow \bigwedge_{k=1}^{m} D_{o_k} \wedge g \wedge g'. \tag{5.1}$$

Here, c denotes the conjunction of all environment constraints, and g denotes the conjunction of all module guarantees, that describe the interface behavior between the module and its environment. They are usually very small in comparison to the module

itself. It is necessary to use the predicates c' and g' to restrict and define the interface behavior of the copied set of variables $\{v'\}$ appearing in the property set P'. Based on Formula 5.1, the integration assumption of the module is defined by the logical formula $c \wedge c' \wedge \bigwedge_{j=1}^{n} D_{i_j}$, and the logical formula $\bigwedge_{k=1}^{m} D_{o_k} \wedge g \wedge g'$ indicates the integration commitments of the module.

The complete verification of the system modules with respect to integration conditions can lead to a situation of mutual dependencies between two or more modules of the system. As long as every module is verified completely with respect to the default determination conditions, i.e., all inputs and outputs are determined at all times, and no constraints are applied, this is not an issue. Recall that in this case the completeness results for the modules guarantee the sequential equivalence of alternative implementations. However, as soon as the integration conditions for the modules are weakened, things turn out to be more complicated. This will be illustrated by means of an example.

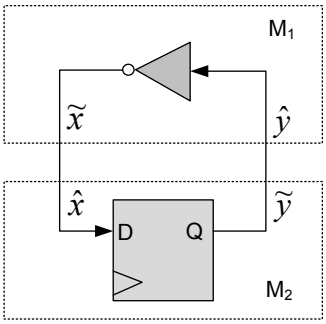


Figure 5.1: D-Flipflop composed with an inverter

Fig. 5.1 depicts a system composed of an inverter M_1 and a D-flipflop M_2, where the inverter M_1 has the input variable $\hat{y}$ and the output variable $\tilde{x}$ and the flipflop M_2 has the input variable $\hat{x}$ and the output variable $\tilde{y}$. The property $p_1 := (\tilde{x} = \neg\hat{y})$ completely verifies the unconstrained module M_1 under the default determination assumption. Then the module guarantee $g_1 := p_1$ of M_1 is used as an environment constraint to verify module M_2, i.e., the constraint $c_2 := (\neg\hat{x} = \tilde{y})$ is applied. With this constraint, the outputs of M_2 are uniquely determined even for the property $p_2 := \mathsf{G}\,(\mathit{true}\,)$, which does not restrict the behavior of M_2 at all. This suggests that p_2 is sufficient to completely specify M_2 under the constraint c_2. Obviously $P = \{p_1, p_2\}$ is not a complete property set for the entire system M, as the behavior of the flipflop remains uncovered. If the assume-guarantee rules presented in [46, 47] are applied to this example, one can deduce that the set $P = \{p_1, p_2\}$ is the complete set for M. Section 5.1.1 shows what is the reason for this false conclusion.

In the remainder of this section, necessary conditions for the integration conditions will be successively introduced, in such a way that the complete verification of the individual modules implies the complete verification of the system. Assume a system has a structure as illustrated in Fig. 5.2, which is general enough to cover every possible scenario of design verification using a divide-and-conquer approach. The system may be

decomposed into several modules M_i by cutting a set of internal signals X, resulting in a set of local inputs $\hat{x}_i$ and a set of local outputs $\tilde{x}_i$. Note that the cut is also applied to the global inputs I, reflecting the fact that some of the ports $\hat{I}$ of a module M_i are driven by the input ports $\tilde{I}$ of the system. The connections of the modules with the output O indicate that each of the modules computes a particular subvector of the global output vector. Some of the outputs y_i drive the interconnections x_i between the modules. A circuit block called *renamed* is responsible for building the relationship between the outputs y_i and the respective signals $\tilde{x}_i$ created by the cut. In contrast, the system can be built by a well-formed composition, i.e., when closing the cut indicated by the dashed lines in Fig. 5.2 by connecting the local inputs $\hat{x}_i$ with their corresponding local outputs $\tilde{x}_i$, the resulting circuit does not contain combinational loops.

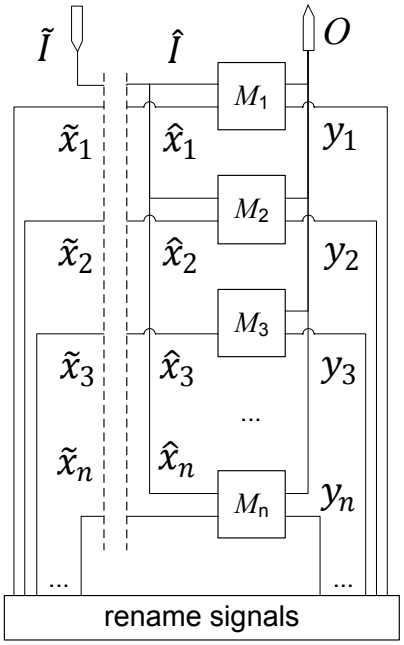


Figure 5.2: General composition scheme for modules

For the system in Fig. 5.2, the global integration assumptions about the interfaces of the system are denoted by $a_M(\tilde{I}, \tilde{I}', O, O')$, and the local integration assumptions of every module are denoted by $a_i(\hat{I}, \hat{I}', \hat{x}_i, \hat{x}_i', y_i, y_i', O, O')$. Similarly the global integration commitments and the local integration commitments are defined by $r_M(\tilde{I}, \tilde{I}', O, O')$ and $r_i(\hat{I}, \hat{I}', \hat{x}_i, \hat{x}_i', y_i, y_i', O, O')$, respectively. The primed signals are the copied variables of the respective signals. Recall that the integration conditions are predicates over two sets of variables (original and copied) of the design, and are used in the context of checking the completeness of property sets.

Note that the bits $\tilde{x}_i'$ of the pseudo-outputs generated by the cut correspond to one of the output bits y_j of one of the modules. In order to indicate this relationship (illustrated by the *renamed* circuit block in Fig. 5.2), the above defined local integration conditions

are rewritten as $a_i(\hat{I}, \hat{I}', \hat{x}, \hat{x}', \tilde{x}, \tilde{x}', O, O')$ and $r_i(\hat{I}, \hat{I}', \hat{x}, \hat{x}', \tilde{x}, \tilde{x}', O, O')$, neglecting that both conditions only refer to a subset of the signals. To lighten the burden of the notation of the signals (original and copied), in the rest of this section the original signal names $\hat{I}$, $\tilde{I}$, $\hat{x}$, $\tilde{x}$, O are reused to represent the unions of the signals $\hat{I} \cup \hat{I}'$, $\tilde{I} \cup \tilde{I}'$, $\hat{x} \cup \hat{x}'$, $\tilde{x} \cup \tilde{x}'$ and $O \cup O'$, respectively, if the meaning is clear from the context.

In the sequel, it is assumed that the modules M_i are completely verified modulo the local integration conditions a_i and r_i. Note that the local integration assumptions are responsible for restricting the local inputs (created by the cut) as well as the global inputs by taking the global outputs and the local outputs as their inputs. In contrast, the global integration assumptions are only responsible for restricting the global inputs in terms of the global outputs.

In the remainder of this section, a set of compositional rules, which can be seen as the derivatives of the assume-guarantee rule presented in Section 4.4, will be introduced, which allows concluding that the entire system is completely verified when the individual modules of a system are completely verified with regard to the reactive constraints. At the same time, the presented method also guarantees that the properties of the individual modules are also valid in the system, and the constraints/commitments used for the module verification are valid in the system as well.

5.1.1 Implementability

For the first requirement to be derived for the integration assumption a_i and a_M, the notion of implementability, as introduced in Section 4.2.1, is used. This criterion requires that at least one circuit fulfilling a particular integration assumption a_i exists. This circuit may continuously observe the outputs of the module and generate valid inputs. Implementability as defined in Section 4.2.1 only focuses on environment constraints. However, in the context of completeness analysis, it is necessary to extend the notion of implementability to include consideration of the determination assumptions.

To see this, the example illustrated in Fig. 5.1 will be reconsidered. Obviously the constraint $c_2 := \mathsf{G}\,(\neg\hat{x} = \tilde{y})$ is implementable, e.g., by the inverter. However, together with the implicitly taken determination guard $d_{\hat{x}} = \mathit{true}$ of module M_2, which implies that the input of M_2 is assumed to be determined at every time, this constraint already determines the output $\tilde{y}$ of the module under verification, M_2. The integration assumption is a conjunction of the environment constraint c_2 and the determination assumption $D_{\hat{x}} = (\hat{x} = \hat{x}')$. As can be easily seen, the formula $c_2' \wedge c_2 \wedge (\hat{x} = \hat{x}')$ implies that $\tilde{y} = \tilde{y}'$ is valid, i.e., the integration assumption necessarily requires determining $\tilde{y}$. In other words, it requires that an implementation of an environment determines the signal $\tilde{y}$. Clearly such an implementation does not exist, because $\tilde{y}$ is a pure *input* to the environment over which the environment has no control.

This problem can be detected by an extension to the implementability test for environment constraints of Section 4.2.1. The test of Section 4.2.1 solves a QBF satisfiability problem that ensures the existence of a current input for every history of the inputs and every history of the output including the current value. Integration conditions can be han-

dled in the same manner except that two versions of the signals need to be considered. In the integration assumption of the aforementioned example, this would require that for every value of $\tilde{y}$ and every value of $\tilde{y}'$, corresponding values of $\hat{x}$ and $\hat{x}'$ need to be found such that the integration assumption

$$c_2(\hat{x}, \tilde{y}) \wedge c_2'(\hat{x}', \tilde{y}') \wedge \hat{x} = \hat{x}' \tag{5.2}$$

is fulfilled. For $\tilde{y} \neq \tilde{y}'$, this is obviously unsatisfiable. Therefore, the integration assumption for the flipflop is not implementable. A reasonable way to resolve this issue would be to remove the assumption that the input is determined, i.e., to allow for using the flipflop in an undetermined environment. In this case, the integration assumption would consist of the two versions of only the constraint c_2 and obviously would be implementable. Note that under this corrected integration assumption, the completeness checker would detect that the trivial property p_2 leaves the output undetermined.

5.1.2 Structural Compatibility of the Integration Assumptions

Composition of circuits that implement Mealy-type state machines, if not carefully conducted, may introduce combinational loops into the overall circuit. It is common practice to avoid such loops when integrating such circuits into a larger system, by conducting a topological analysis that detects combinational loops.

This is because the oscillating behavior caused by a combinational loop can not be handled by a property checker, which models a DuV as a finite state machine. When composing the results of the completeness analysis of the property sets, the combinational dependencies also need to be considered, because the soundness of the compositional rules presented in this chapter needs every value of signals to be stable at every time instance (See Section 4.4).

In addition to the combinational dependencies present in the circuit under verification, it is necessary to consider the additional dependencies that are introduced by the integration assumptions. Combinational loops introduced by these additional dependencies must be avoided because if a reactive integration assumption of a module builds a combinational loop with the DuV, this may lead to the overconstraining of the DuV (see Fig. 4.7 in Section 4.3). Under this circumstance, the verification results are no longer trustable (viz., the declaration of the completeness of the property sets and the validity of the property sets). This kind of problem of overconstraining can not be detected by just checking that the environment's commitments imply the assumptions of the DuV, since the environment is assumed to work properly only when the DuV works correctly: in spite of the trustworthiness of the property sets, the DuV may function wrongly.

In the following, let $K(M)$ denote the set of pairwise combinational dependencies between the signals in the circuit M. Likewise, let $K(a)$ be the set of pairwise combinational dependencies between signals induced by the integration assumption a.

For integration assumptions, unlike general constraints, it may be unclear what precisely a combinational dependency should be, as they describe relations between signals

rather than functional dependencies, i.e., one signal is a function of another one. However, as only implementable integration assumptions are considered here, one may resort to a most general implementation of the conditions and take the dependencies from there. In the ideal case, the true semantic dependencies would be considered, i.e., one would synthesize the most general implementation of the integration assumptions and combine it with the DuV to analyze the combinational loops. However, this might be hard to compute for large circuits. In practice, this can be done as shown in Section 4.2.2 by structurally extracting syntactic dependencies that over-approximate the semantic dependency relation.

With the above notation, the structural compatibility of the integration assumptions with a DUV M is defined as follows:

Definition 5.1. *A system M with modules $M_1, \ldots, M_n$ is structurally compatible with integration assumptions a_M, a_1, ..., a_n if the signal dependency graph representing the combinational dependencies $K(M), K(a_M), K(a_1), \ldots, K(a_n)$ is acyclic.*

Structural compatibility is a necessary condition to obtain a well-formed composition during system integration. Moreover, it is a prerequisite that the system itself can be integrated with its environment in the same manner.

5.1.3 Conditions for Local Integration Assumptions

The next necessary condition for the integration of the modules states the following: given that the global integration assumption is valid, it must be possible to find values for these local signals such that all the local integration assumptions are simultaneously satisfied.

Definition 5.2. *The local integration assumptions a_i weaken the global assumption a_M if every sequence of values for the global input/output signals $\hat{I}, O$ that satisfies $a_M(\tilde{I} \leftarrow \hat{I}, O)$ can be extended by suitable values $\hat{x}, \tilde{x}$ for the local inputs/outputs of the modules such that all the local assumptions $a_i(\hat{I}, \hat{x}, \tilde{x}, O)$ are simultaneously satisfied.*

In the global integration assumption a_M, the replacement of the signals $\tilde{I}$ by the corresponding signals $\hat{I}$ is necessary, because they are two separate variable sets and a design-independent analysis does not know their relationships without their connectivity information. The intuition of Def. 5.2 is that the local integration assumptions of the individual modules will allow every sequence of values for the global inputs that is generated by the global integration assumptions.

The next criterion to be developed is to ensure that extensions of the input sequences of $\hat{I}, O$ with local values $\hat{x}, \tilde{x}$ that do not fulfill the local integration assumptions of a module may not occur in a verified system. For this purpose, the integration commitments of other modules are used. In general, these commitments depend on the corresponding local assumptions. This circular reasoning must be broken, which is not a trivial task for reactive systems with circular dependencies between modules. To break circular reasoning, one must make sure that the commitments of the other modules that are used to verify an integration assumption of a given module, do not require themselves in their proof. At

this point, the cut in the verification model of Fig. 5.2 comes into play. The cut operation breaks the circular dependencies of the modules and the signals, by introducing two independent sets of variables $\hat{x}$ and $\tilde{x}$ for the internal signals x of the system as well as $\hat{I}$ and $\tilde{I}$ for the global inputs I.

In order to deduce the local integration assumptions from the global assumptions, one must assume the validity of these global assumptions for both $\hat{I}$ and $\tilde{I}$ under the same output behavior. In addition to the global assumptions, it is supposed that every local commitment of the modules restricts some of the local outputs $\tilde{x}_i$ and some of the global outputs O depending on the local inputs $\hat{x}_i$ and the global inputs $\hat{I}$. Under these circumstances, it must be analyzed whether the local assumptions are satisfied by the independent global inputs $\tilde{I}$ together with the local signals $\hat{x}_i$ and $\tilde{x}_i$ and the global outputs O.

Definition 5.3. *The local integration assumptions a_i are justified by the global assumption a_M and the local commitments r_i if the following implication holds:*

$$a_M(\hat{I}, O) \wedge a_M(\tilde{I}, O) \wedge \bigwedge_{i=1}^{n} r_i(\hat{I}, \hat{x}, \tilde{x}, O) \Rightarrow \bigwedge_{i=1}^{n} a_i(\tilde{I}, \tilde{x}, \hat{x}, O).$$

Note that in Def. 5.3 every occurrence of the signals $\hat{I}$, $\hat{x}$ and $\tilde{x}$ in the local integration assumptions a_i is replaced by the corresponding counterpart signals $\tilde{I}$, $\tilde{x}$ and $\hat{x}$ that are created by the cut operation. In brief, Def. 5.3 states that the integration assumptions of every module must be *guaranteed* by the module's environment.

The last criterion for compositional completeness considers the global integration commitments. These commitments should follow from the global integration assumptions together with the local integration commitments, so that the global integration commitments need to be sure that they specify every behavior that the modules of the system can produce.

Definition 5.4. *The local integration commitments r_i imply the global integration commitments r_M under the global integration assumptions a_M if the following implication holds:*

$$a_M(I, O) \wedge \bigwedge_{i=1}^{n} r_i(I, x, x, O) \Rightarrow r_M(I, O).$$

Note that in Def. 5.4 every occurrence of signals $\tilde{I}$, $\hat{x}$ and $\tilde{x}$ in a_M, r_i and r_M is substituted by the corresponding signals I and x, respectively.

So far, a set of criteria has been presented that allows determining whether the collection of property sets for all the modules form a complete property set for the composed system.

Theorem 1. *Let M be a system that is configured in a well-formed composition from $M_1, M_2, \ldots, M_n$ as indicated in Fig. 5.2 and that fulfills the following requirements:*

1. *Every module M_i is completely verified by a property set P_i with respect to local integration assumptions a_i and local integration commitments r_i on the system model $\hat{M}$ that is obtained by performing the cut in Fig. 5.2.*
2. *The local integration assumptions a_i as well as the global integration assumptions a_M are implementable (Section 5.1.1).*
3. *The integration assumptions are structurally compatible with the system (Def. 5.1).*
4. *The local assumptions a_i weaken the global assumptions a_M (Def. 5.2).*
5. *The local assumptions a_i are justified by a_M and the local integration commitments r_i (Def. 5.3).*
6. *The global integration commitments r_M follow from a_M and the r_i (Def. 5.4).*

Then, the property set $P = \bigcup_i P_i$ completely verifies the system M with respect to the integration assumptions a_M and commitments r_M.

Proof. In order to prove the completeness of the union of property sets, P, it is necessary to show that

$$a_M(I,O) \wedge \bigwedge_{p \in P} p(I,x,x,O) \Rightarrow r_M(I,O)$$

is a tautology. For each module M_i it holds due to precondition 1 of Theorem 1:

$$a_i(\hat{I},\hat{x},\tilde{x},O) \wedge \bigwedge_{p \in P_i} p(\hat{I},\hat{x},\tilde{x},O) \Rightarrow r_i(\hat{I},\hat{x},\tilde{x},O).$$

It follows that

$$\bigwedge_i a_i(\hat{I},\hat{x},\tilde{x},O) \wedge \bigwedge_{p \in P} p(\hat{I},\hat{x},\tilde{x},O) \Rightarrow \bigwedge_i r_i(\hat{I},\hat{x},\tilde{x},O)$$

and further

$$(a_M(\hat{I},O) \wedge a_M(\tilde{I},O) \wedge \bigwedge_i a_i(\hat{I},\hat{x},\tilde{x},O) \wedge \bigwedge_{p \in P} p(\hat{I},\hat{x},\tilde{x},O)) \Rightarrow$$
$$(a_M(\hat{I},O) \wedge a_M(\tilde{I},O) \wedge \bigwedge_i r_i(\hat{I},\hat{x},\tilde{x},O)).$$

With condition 5, this implies

$$(a_M(\hat{I},O) \wedge a_M(\tilde{I},O) \wedge \bigwedge_i a_i(\hat{I},\hat{x},\tilde{x},O) \wedge \bigwedge_{p\in P} p(\hat{I},\hat{x},\tilde{x},O)) \Rightarrow$$
$$a_M(\tilde{I},O) \wedge \bigwedge_i a_i(\tilde{I},\tilde{x},\hat{x},O)). \qquad (5.3)$$

To close the dashed line created by the cut operation(see Fig. 5.2), the assume-guarantee rule presented in Section 4.4 is applied. At the first glance, this assume-guarantee rule can not be used directly here, because this rule is developed for systems without global reactive constraints ("integration assumptions" in the context of completeness analysis) for the inputs of the systems. In order to apply this rule, the model of the system can be extended by the most general implementation of the global reactive constraints, i.e., as circuit model the composition M' (represented by p_i) together with the most general implementation of $a_M(\tilde{I},O)$ is considered. In this circuit model, Formula 5.3 guarantees that the assertions $(a_M(\tilde{I},O) \wedge \bigwedge_i a_i(\tilde{I},\tilde{x},\hat{x},O))$ are valid under the reactive constraints $(a_M(\hat{I},O) \wedge \bigwedge_i a_i(\hat{I},\hat{x},\tilde{x},O))$.

Another precondition for applying the assume-guarantee rule is that the constraints $(a_M(\hat{I},O) \wedge \bigwedge_i a_i(\hat{I},\hat{x},\tilde{x},O))$ must be implementable. Although the constraints a_M and a_i are implementable individually (condition 2), it can not conclude that the constraints $a_M \wedge \bigwedge_i a_i$ are also implementable. For instance, Suppose $a_M := (i = o_1)$ and $a_i := (i = o_2)$, every constraint can be trivially implemented by a wire, however, the constraints $a_M \wedge a_i$ are not implementable because they require their implementation's inputs must be equal for any environment, which is actually impossible for a circuit.

Therefore, as the next step, it needs to be shown that $(a_M(\hat{I},O) \wedge \bigwedge_i a_i(\hat{I},\hat{x},\tilde{x},O))$ is implementable. Here the implementability test presented in Section 4.2 can be applied. This test checks whether for every sequence of the signals $\tilde{x}$ and O, the constraint $(a_M(\hat{I},O) \wedge \bigwedge_i a_i(\hat{I},\hat{x},\tilde{x},O))$ can generate an sequence of the signals $\hat{x}$ and $\hat{I}$. Obviously this is the case for the local inputs $\hat{x}$, since every a_i is implementable and every a_i restricts different sets of the local input variables. For the global inputs $\hat{I}$, since a_M and a_i both generate values for $\hat{I}$, it needs to ensure that every of them can generate a set of input value for every O and $\tilde{x}$, this is valid due to the implementability of a_M and a_i. It also requires that for every O and $\tilde{x}$, the set of value $\{I^{a_M}\}$ for $\hat{I}$ generated by a_M and the sets of value $\{I^{a_i}\}$ generated by every a_i fulfill the condition $\{I^{a_M}\} \subseteq \bigcup_i \{I^{a_i}\}$, and it is valid due to precondition 2, which guarantees that every input sequence generated by the global integration assumptions must be allowed by every local input integration commitment. In summary, the constraints $(a_M(\hat{I},O) \wedge \bigwedge_i a_i(\hat{I},\hat{x},\tilde{x},O))$ are implementable.

The structural compatibility of the assumptions with the circuit model stated above is assured by the precondition 3. Given all that, the assume-guarantee rule can be applied to Formula 5.3, which proves the unbounded validity of the assertion $\bigwedge_i a_i(I,x,x,O)$ in every such system model M' without cut, and in particular in M with the reactive constraint $a_M(I,O)$.

Due to condition 1, also every $r_i(I,x,x,O)$ is valid in M and with condition 6, $r_M(I,O)$ follows. □

5.2 Experimental Results

In order to evaluate the usefulness of the completeness criteria developed in this research, a software tool has been implemented within a SAT-based verification environment. This tool has been successfully evaluated on two designs. The first is a system composed of industrial IP blocks and based on Infineon's Flexible Peripheral Interconnect (FPI) bus. The second example is the "Minimal OpenRISC System on Chip (MinSoC)" from *www.opencores.org*. Every criterion presented in the previous section has been fully checked automatically and within 300 ms using less than 95 MB of memory on an Intel Core i7 CPU 860 at 2.8 GHz with 8 GB of RAM. The computational complexity is small because only sets of integration conditions are checked against each other. The design is not considered in these proofs. However, the conceptual complexity of these sets of constraints can be quite daunting for a verification engineer and bugs in these constraints may be easily missed by manual review, as will be demonstrated below. Since the computational resources required by the presented approach are low, the following focuses only on examples of environment constraints and what problems can possibly be detected by the approach.

5.2.1 The FPI Bus System

The FPI bus protocol is quite similar to the industry's standard AMBA AHB bus protocol. It also supports pipelined transactions to increase the throughput of the bus. The FPI bus system considered here is illustrated in Fig. 5.3. It is composed of a master, a slave agent and a master agent. The master is a Graphics Processing Unit (GPU). Its bus interface is not compatible with the FPI bus protocol, therefore it requires using a master agent to adapt the GPU's interface for reading and writing information from/to peripherals to the particular FPI bus protocol. Each interface transaction of the GPU is translated into the corresponding protocol transaction on the bus. Due to the pipelining features of the protocol, the master agent may handle two requests from the GPU simultaneously. The slave agent operates similarly to the master agent, however, it does not initiate bus transactions.

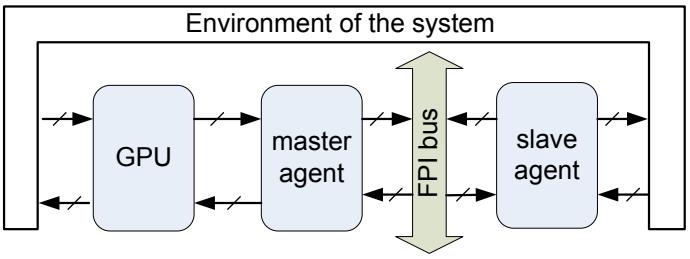


Figure 5.3: Example: FPI bus

The environment of this system is modeled by global integration assumptions. The guarantees provided by the system are modeled by the global integration commitments.

It is worth noting that the errors in the constraints of the master agent that were found by the plausibility checks presented in Section 4.3, have been corrected for this experiment. All modules have been completely verified by applying the OneSpin 360 DV property checker [83] with its completeness analyzer under the global integration conditions and the respective local integration conditions. As a rule of thumb, the effort of applying C-IPC to an SoC module such as the ones considered here is about 2000 lines of code per person month. The modules used in this case study were verified before the compositional technique proposed here was available. Therefore, all integration conditions were initially only inspected manually.

It was therefore interesting to apply the here presented automated checks to these property suites, which were considered finished. In fact, it turned out that one local integration assumption of the integration conditions failed with respect to criterion 5 of Theorem 1 in Section 5.1. The problematic integration assumption is related to the master agent module and has about 400 lines of code. Therefore, due to space limitations, by using a pseudo-code notation, only the relevant subexpressions that turned out to form the bug are presented. It must be mentioned that these relevant subexpressions stem from distant locations in the original source code. This explains why under these circumstances the issue detected by these new automatic checks had been overlooked by the verification team.

```
if read = '1' or write = '1' then
        opcode /= NOP;
end if and
if request = '1' then
        opcode /= NOP;
else
        opcode = NOP;
end if;
```

Figure 5.4: Pseudo code related to the bug about *opcode*

The code snippet in Fig. 5.4 shows a fragment of the environment constraint for the input *opcode* of the master agent. All variables in this code fragment of the constraint are inputs to the DUV. The GPU asserts the *request* signal to issue a transaction. The *read* and *write* inputs indicate a read or write transaction. The input signal *opcode* specifies the transfer mode of a transaction. The constant value *NOP* for the signal *opcode* indicates that the interface does not execute any transaction on the bus. At first glance, the above-listed constraint assumes two restrictions for the *opcode*. First, it states that whenever the processor asserts a *request*, the *opcode* must not be *NOP*, i.e., a read or write transaction has to be started. Second, whenever the processor asserts the *read* or the *write* line to indicate a read or a write transaction, the *opcode* must not be a *NOP* either. Note that this constraint implies that $(read \vee write) \Rightarrow request$ is valid. This integration assumption of

the master agent violates the condition 5 of Theorem 1, because the guarantee given by the GPU allows for both, *request* to be asserted or not asserted when the *read* or *write* line is set. In other words, the property set of the master agent covers fewer input sequences than the GPU can provide to it, hence, in the composed system, the property sets do not cover every behavior of the system. Assuming that the behavior of the GPU is reasonable, since *read* and *write* signals are only evaluated when $request = 1$, if $request = 0$, then *read* and *write* can be chosen to be either 0 or 1, whichever is suitable for circuit optimization.

```
if read = '1' or write = '1' then
        opcode /= NOP;
end if and
if request = '1' then
        opcode /= NOP;
end if;
```

Figure 5.5: Possible fix for the bug related to *opcode*

This problem can be resolved by releasing the constraint a bit. A possible solution is listed in Fig. 5.5. With this set of constraints, all the properties of the master agent can be verified, and all the completeness criteria are fulfilled.

5.2.2 MinSoC

The technique presented in this dissertation was also evaluated on a MinSoC system from *www.opencores.org*. The MinSoC system has a CPU core (or1200) and several peripherals, including SPI, JTAG, UART and ETH [96]. Here, a subset of this system is considered that is composed of the or1200 core, SPI, UART, RAM, ROM, and a startup module which provides the initialization sequence for CPU after power-on. The or1200 core communicates with the rest of the modules via a Wishbone [96] bus.

Every module has been completely verified under global/local integration conditions. The bugs found during the property checking process have been corrected and reported back to the designers. The global/local integration conditions used fulfill all the criteria of Theorem 1 in Section 5.1. Therefore, one can conclude that the system is completely verified based on the verification results of each module.

Due to space limitations, only one example of an integration assumption of the SPI module is shown here. The integration assumption is related to the behavior of the Wishbone interfaces.

In the reactive constraint illustrated in Fig. 5.7, the suffix *_i* denotes the inputs to the SPI module and the suffix *_o* denotes the outputs. This constraint states that if a request from the CPU is not yet acknowledged by the SPI module, then the CPU must keep requesting. The determination conditions to the signals in this constraint are the default determination conditions introduced in Section 3.4.2. As mentioned above, the SPI

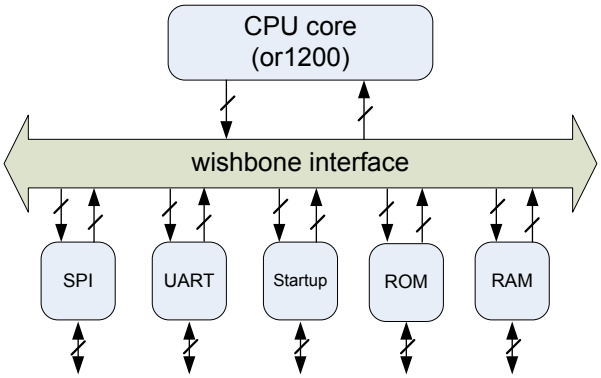


Figure 5.6: Example: MinSoC

```
if wb_cyc_i = 1 and wb_stb_i = 1 and
   wb_ack_o = 0 then
      next(wb_cyc_i) = 1 and
      next(wb_stb_i) = 1;
end if;
```

Figure 5.7: Pseudo code for wishbone protocol constraints

is completely verified under this integration assumption. The same behavior specified in this integration assumption is also employed as an integration assumption for the other modules, except for the CPU. In terms of Section 5.1.1, the integration assumption obviously is implementable. It can be implemented using a few multiplexers and registers. Clearly, this integration assumption is also structurally compatible with the SPI module because all the outputs of the most general implementation of this integration assumption are fed to registers, i.e., there are no combinational loops when composing the most general implementation with the SPI module. The integration commitments of the SPI module's environment, namely the CPU, imply this integration assumption as well, since syntactically it has the same structure as illustrated in Fig. 5.7, except that the input signals are replaced by the output signals of the CPU, and the output signals are replaced by the input signals of the CPU. Therefore, it can be concluded that the system is completely verified (considering a closed system composed of only the CPU and the SPI).

Chapter 6

Software Property Language

At present, there is a trend that more of the functionality in SoCs is shifted from the HW to the SW, also to the hardware-dependent software. Hardware-dependent software, directly interacting with its hardware environment, is a crucial component in an SoC that needs to be tested rigorously, since other software layers, such as the operating systems and the application software are built on it. Therefore there is an increasing need to apply formal verification to hardware-dependent software, especially for safety-critical systems. The interactions between HW and hardware-dependent software are often reactive, and happen in a temporal order. This requires new techniques to model the hardware-dependent software, and it requires new property languages to specify the reactive behavior at the HW and SW interfaces. In more detail, the computational model for hardware-dependent software must not only include the functional behavior in terms of instruction computations, but also the addresses and data of all input/output (I/O) accesses and the ordering of the I/O accesses for all program executions. At the same time, the property language for such a computation model must support specifying the functional behavior of a software program, and the temporal behavior at the HW and SW interface as well.

This chapter introduces a new property language to specify the reactive interactions between the HW and the hardware-dependent software based on the software model presented in Section 6.2. The presented sequence-based property language makes it possible to apply the method of checking the completeness of hardware properties, explained in Chapter 3 and Chapter 5, to check the completeness of the software properties. The software property language is introduced in Section 6.3. The criteria for checking the completeness of the software properties are given in Section 6.4. A case study to illustrate the usage of the presented property language is shown in Section 6.5.

6.1 Introduction

Unlike the wide adoption of formal model checking for industrial hardware designs, functional verification for software designs by formal methods is still mostly restricted to the academic world. There, a lot of of research effort has been put into formal verification of software designs, and many model checking tools have been developed. Like many model

checkers for hardware, these tools usually model the software program by using (finite) state transition systems. Some tools [24,97] are based on the explicit representation of the states and the transitions between states, and suffer from the well-known *state explosion* problem. In contrast, approaches like [60,98] represent the states and state transitions of a software design implicitly by using BDDs. Then, model checking is conducted by applying the state exploration algorithms often used in hardware verification tools. SAT-based approaches like [27–29] represent software programs in Boolean formulas and transform the verification problems to SAT problems. All these approaches are aimed at verifying software programs written in a high level programming language like C, which is *independent* of the target hardware platforms. A good survey about the standard approaches to software model checking tools can be found in [31].

However, the present dissertation focuses on the verification of hardware-dependent software, which is the part of the software in an embedded system that interacts directly with the surrounding hardware. Examples of such software are, e.g., drivers of peripherals, firmware to bring up a system, etc. This kind of software is a critical component in embedded systems since all the other software layers (e.g., the operating system, the application software, etc.) are built on top of it. Additionally, hardware-dependent software in embedded systems performs control-intensive tasks with complex interactions with the hardware and with other software layers, making its development error prone and its systems difficult to test. Therefore, it is particularly important to guarantee the correctness of hardware-dependent software.

Because of the reactive behavior of HW/SW interactions, the specification languages and validation methods, as they have been developed for application-level software, are in many cases not suitable for hardware-dependent software. In [32] a joint model of hardware-dependent software and the underlying microcontroller is created, then the states and state transitions are represented and explored explicitly, which is not feasible for microcontroller and software programs with industrial sizes. In [33, 34] a hardware-dependent software program is unrolled by representing every execution step of the program by an instance of the processor hardware, and then the properties are proved by using SAT-based bounded model checking. The advantages of these approaches are that full details both of the hardware and software are modeled, at the same time. Since the software is represented as binary code stored in read-only memory (ROM), the interactions between the processor and the ROM are modeled, and can be verified too. However, these approaches create a copy of the whole system for every HW clock cycle of a software program. A small program may easily exhaust the memory of a computer. In [40] a combinational Boolean computational model, called *program netlist*, is also created by unrolling the program. Instead of representing every step of the program execution by a detailed copy of the hardware, the program netlist applies an abstracted hardware, which is relevant to the instructions being executed. This significantly reduces the size of the software model.

This chapter introduces a new property specification language for describing the behavior of hardware-dependent software programs, and it presents how the proposed verification language can be used to perform formal verification based on the program netlist of

a program. For generating a program netlist, path-oriented techniques related to *symbolic execution* [99] are used. There has been much research on software testing/simulation based on symbolic execution [35–39]. The idea is to apply symbolic values, e.g., the variables's names, to compute the outputs of a program. In this case, the output of a program is a Boolean function of symbolic input values. Software testing (simulation) tools based on symbolic execution do not need a complete model of a software program: they generate and analyze the execution path of the software program one by one. In contrast, model checking needs a model that fully captures the behavior of the design. The work of [40] introduces a way to generate such a model, and Section 6.2 gives details on what this model, i.e., program netlist, looks like.

Furthermore, this dissertation will present how the language presented in Section 6.3 can be used in conjunction with the program netlist in order to perform formal property checking. Unlike methods based on symbolic execution, in which the properties are proven by explicitly traversing the possible execution paths of a given program, this dissertation adopts the approach of [40], which employs a SAT solver in order to perform this traversal. A SAT proof benefits from the control logic in the program netlist by being able to focus on the execution paths that are important for the particular problem instance and to prune at once entire execution paths that are not relevant. The effectiveness of this approach has been shown in [40]. In order to use a SAT solver for path traversal, it is necessary to create a combined model containing the logic for the property and the program netlist, so that the SAT solver has "a global view of the verification problem" (instead of having only a view of the problem for individual execution paths). For a global view, a model of the input/output sequences of the software is synthesized and integrated to the model (see Section 6.3).

State of the art software property languages

Besides continuous advances in methods and algorithms for formal property checking of hardware designs, an important role in the adoption of formal verification techniques in industry in the last few years has also been played by the languages for formulating properties. For instance, SVA and PSL allow concisely specifying the behavior of the hardware, which is typically described at the RTL. While being founded in a strictly defined mathematical framework, these property languages include various syntactic enhancements offering a natural and easy way to capture the temporal behavior of the design. Current commercial technology allows checking assertions using simulation or formal verification engines.

There are currently a number of different approaches to formalizing the properties of embedded software. In the most simple case, *static analysis* tools [100,101] automatically check that the software is free of common errors such as null pointer dereference, memory leaks, out of boundary access to arrays, etc. In these proofs no functional properties of the software are proved and therefore the verification engineer isn't required to specify any properties. Because of the reactive behavior of embedded software, it is not sufficient to only specify the pre/post conditions [86] for a program: the interaction between the hard-

ware and software must also be specified. This interaction is often a temporal behavior, which requires a property specification language that can support formulating it.

Run-time assertions are being used widely for the testing [102] and formal verification [27, 28] of embedded software. For example, high-level programming languages like C provide the *assert()* statement for specifying predicates over the values of the program variables. The main use of run-time assertions is to describe properties that are valid locally. More specifically, this kind of property is evaluated only when a program run reaches the location where the involved *assert()* statement has been placed. While run-time assertions have the advantage that the user is not required to learn a new language in order to specify the properties, their main limitation is in what can be expressed. For the case of application software or for simple transformational code, run-time assertions may be sufficient. However, for hardware-dependent software, it is necessary to be able to describe reactive behavior relating to the inputs, outputs and states of the software and hardware *at different points in time*. For specifying temporal behavior, temporal logics such as CTL and LTL can be used. There exist verification tools such as [24, 32] that accept temporal formulas directly as a property specification language. However, although CTL and LTL are powerful in formulating temporal relationships, they are hard to understand and use in practice.

Other tools, such as [103, 104], in a similar way, employ automata in order to specify temporal properties. The use of automata can be convenient in many cases since they are easier to understand by a designer or verification engineer than the temporal formulas in CTL or LTL. However, except for simple cases, the process of modeling a property using an automaton is cumbersome and error prone.

Unlike the approaches mentioned previously, this dissertation presents a new verification language for hardware-dependent embedded software, which allows specifying the temporal behavior of the interactions between the software and hardware. The proposed language is intuitive and easy for the verification engineer to use. It allows referring to the interfaces of the software and explicitly describing the sequences of the input/output operations at these interfaces. It adopts many syntactic elements from the C language, which makes learning the new language easy for software engineers. The language facilitates the re-use of verification code by providing features like macros and functions. To the best of my knowledge, this is the first research on a property language for hardware-dependent embedded software with the characteristics mentioned above. The property language proposed here can be employed for simulation or verification purposes.

Since formal verification examines every possible input scenario, the usual coverage criteria evaluating the quality of the test cases for software are not suitable for formal property checking. In the case of software property checking, verification engineers face the same problem as engineers in hardware verification: “Does my property set cover every aspect of the design?”. In Chapter 3 and Chapter 5 a number of methods for coverage analysis attempting to prove that a hardware property set is “complete” were introduced. In these approaches, a set of safety properties is called a *complete specification* or simply *complete* if it uniquely describes the behavior of a design. More precisely, these methods prove the completeness of a property set by means of checking to what extent

the property set *uniquely* specifies the input and output behavior of a hardware design. Based on these results, Section 6.4 will present a method for proving the completeness of software properties specified in the software property language presented in Section 6.3. Such a completeness check is of particular importance for software property sets, because a complete set of properties can, at least in principle, fully replace classical software tests.

A typical source of error when writing software is that the programmer simply forgets to treat certain input sequences in the program, causing undefined behavior when these inputs occur at runtime. Also, a verification engineer may forget to specify tests (or, in our case, properties) for such missing input sequences, so that the bug can escape verification. The completeness check presented in this chapter removes such gaps in the verification. Because of the similarity between hardware and hardware-dependent software, the presented method for proving the completeness of software property sets checks whether every input/output sequence is specified by a property in the set. The checking algorithm leverages the property language presented in this dissertation, which allows referring to the interfaces of the software program in order to describe its reactive and sequential behavior.

6.2 Hardware-Dependent Software Model

In this section, the basic characteristics of the HW-dependent software model underlying the proposed language is briefly reviewed. The model is called *program netlist*. A complete description of it and how it is generated can be found in [40].

A program netlist is a combinational circuit that compactly represents the software that is executed on the underlying hardware. In order to generate a program netlist, the control flow graph (CFG) is extracted from a low-level description of the software program, such as assembly or machine code. Every node in the CFG represents an instruction of the program and the associated *program state* (PS). The PS includes the contents of data memory associated with the variables used in the program, and the *architecture state* (AS), defining the state of the processor's registers that are visible to the programmer. An edge between two CFG nodes indicates a possible execution from one instruction to another one.

An additional Boolean signal, called *active*, is attached to PS in order to model the control flow of the program. This signal is propagated alongside the nodes in the program netlist and helps the SAT solver to efficiently explore the possible execution paths of the program. The *active* signal, when set to 1, indicates that a given node (instruction) belongs to the active execution path. In the case that a node has more than one successor (e.g., nodes related to jump/branch instructions), exactly one branch is active at any time.

The CFG is fully unrolled into an *execution graph* (EXG). An EXG is a directed acyclic graph containing all possible execution paths of the program. An execution path always begins at a start state of the program and ends at an end state. The CFG is unrolled by unwinding the loops of the program. Figure 6.1 illustrates an example of unrolling. In order to reduce the complexity of the model, only branches that are part of at least

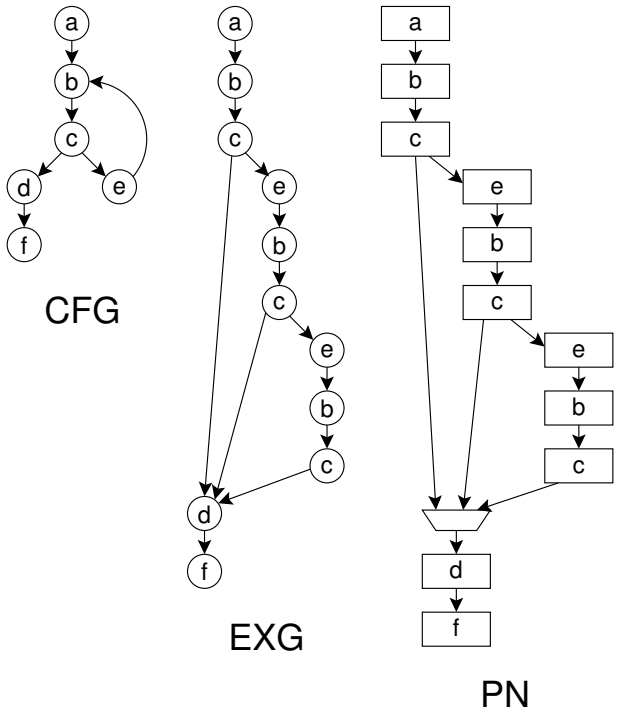


Figure 6.1: Generating the Program Netlist (PN)

one possible execution path are processed. A SAT solver can be used to identify such branches. Alternatively, this process of identifying branches can be accelerated by using the simulation technique presented in [105]. The unrolling ends when all *active* branches have been processed and the end of the program has been reached. In addition, in order to minimize the size of the model, nodes belonging to identical program locations are merged. In this manner, an EXG is obtained, in which a single node may be shared by different execution paths. Merging is only allowed if it does not insert loops into the EXG. A program netlist is then obtained from the EXG by replacing every node by its corresponding *instruction cell*. An instruction cell is a piece of combinational logic circuitry describing the functional behavior of an ISA instruction according to the specific CPU architecture at hand. Consecutive instruction cells are connected by buses representing the program state. It is worth mentioning that instruction cells need to be defined for every kind of processor. However, this should not burden the verification engineers when they verify the processor (hardware) by writing properties with the format introduced in [106]. As described in [106], every property applied to verify the processor is an instruction cell, which can be reused for generating the program netlist.

A kind of instruction that is especially relevant to this work are load/store instructions which are used to communicate with the program's environment, e.g., the hardware periphery or other software layers. Instruction cells corresponding to such instructions are equipped with additional input and output ports, as shown in Fig. 6.2. These ports are called *pdata*, *ploc* and *pact* and represent respectively the data value, the accessed loca-

tion and the *active* flag indicating the activeness of the related instruction cell. Depending on whether the instruction cell reads or writes, *pdata* is an input or output signal. These three signals of an instruction cell constitute an *access port* for I/O memory locations.

In the sequel, the term *program location* indicates a memory location storing an instruction, and the term *memory location* is used to indicate an address corresponding to a location of the hardware periphery or a memory variable.

In the program netlist, instructions that access data memory require additional constraints so that the behavior of the data memory is also modeled [107]. Therefore, for each instruction cell that reads from data memory, there is a multiplexer structure that selects the last valid value written to the memory location being read by the instruction. In the case that a program depends on external events, e.g., by means of shared variables/channels, additional access ports of the respective instruction cell are left open or unconstrained as shown in Fig. 6.2. These access ports serve as the interfaces of the program, as will be further explained in the next section.

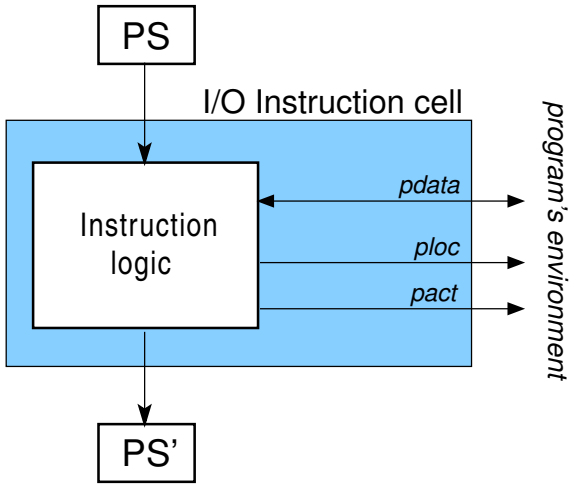


Figure 6.2: Instruction cell with ports for accessing the environment

6.3 Software Property Language

This section presents how to describe the interaction between the hardware and software in terms of I/O sequences, and how the model of the sequences can be synthesized and combined with the underlying model of the software in order to perform formal property checking. As explained earlier, this is necessary in order to capture the reactive behavior exhibited by hardware-dependent software. An additional advantage is that a model of the sequences allows mapping the elements of the language to the elements of the underlying software model in a straightforward way. Subsequently, in terms of such sequences, a property language can be developed. The current working name for this language is RSPL (Reactive Software Property Language).

In the following, the main syntactic elements of RSPL are introduced. Since the programming language C is widely used for embedded software, the presented property language adopts many operators and syntax elements from C. For example, RSPL inherits from C the standard arithmetic, Boolean and comparison operators, which are summarized in Fig. 6.3.

```
// comparators
== (equal), != (not equal)
>= (greater than or equal)
<= (less than or equal)
> (greater than), < (less than)

// Boolean operators
&& (logical and), || (logical or)
! (logical not)

// arithmetic operators
+ (addition), - (substraction)
* (multiplication), / (division)
% (modulo)

// bitwise operators
& (bitwise and), | (bitwise or)
^ (bitwise xor), ~ (ones complement)
<< (shift left), >> (shift right)
```

Figure 6.3: Comparators and Operators

6.3.1 The Interfaces of a Hardware-Dependent Program

A property language for hardware-dependent software needs to provide a means for referring to the interfaces of a given program. In contrast to hardware description languages, software programs in high-level languages such as C do not explicitly capture their interfaces in a separate entity. For hardware-dependent software, the elements of the interface correspond to a set of addresses identifying, for example, registers inside hardware peripherals or shared memory locations used for communication with the operating system or with the application code. In view of the program netlist, such interface elements are modeled by means of access ports belonging to input/output instruction cells as explained in Section 6.2. In RSPL each of these addresses is assigned a name. In the case of compiled machine code, these names can be automatically obtained from the symbol table dumped by a cross-compiler. Otherwise, variable names can be defined manually by the user to enhance the readability of the verification code. In order to distinguish the action of reading a variable (as input) from the action of writing a variable (as output), two variable attributes are introduced to the property language, namely *read* and *write*, as depicted in Fig. 6.4. This is the basis for referring to all input/output operations. Note

```
⟨Name of variable⟩'read
⟨Name of variable⟩'write
```

Figure 6.4: read/write attributes

that an address can be read and written several times. How this can be handled by the property language and how the verification engineer can refer to the different read and write instances is described below in Section 6.3.2.

There are also cases in which it is necessary to refer to the programmer-visible registers, for example when separately verifying a subroutine of a software driver. In such cases the content of a register can be expressed using the following syntax.

```
$⟨Name of register⟩'start
$⟨Name of register⟩'end
```

The attributes "*start*" and "*end*" indicate the start and the end of the program, respectively.

6.3.2 Sequences of Variables

The *sequence* is the key concept of RSPL; it is inspired by sequences in hardware property languages like SVA [89]. A sequence in SVA is constructed using the delay operator #, which specifies the relative clock cycles (delay) between two events. However, the semantics of sequences from SVA can not be directly imported to RSPL, since a sequence in SVA is defined over cycles which are relative to a global time reference such as a hardware clock. Models used for software (and in particular the program netlist) are not accurate with respect to hardware clock cycles, but rather instruction-accurate. Therefore, sequences are defined relative to the ordering of instruction executions. As illustrated in Fig. 6.5, RSPL provides users with a way to define the individual elements of a sequence. Several such elements may be combined using Boolean operators in order to form sequences. The symbol # is defined as the *element accessor* for sequences and the natural number n represents the n-th element of a sequence. Since not every instruction accesses the interface of the program, the element n is the n-th occurrence of the associated interface variable along an execution path of a program (as opposed to the n-th instruction along that path). A software tool evaluating properties written in RSPL needs to map the

```
⟨Name of variable⟩'read #⟨n⟩
⟨Name of variable⟩'write #⟨n⟩
```

Figure 6.5: Element Accessor

elements of a sequence to the respective access ports in the program netlist. Because of the merging mechanism used to generate a program netlist, an input/output instruction cell can belong to several different execution paths. In other words, along an execution path

an access port might be the i-th sequence element of a variable, whereas along another path, it might correspond to the j-th ($j \neq i$) sequence element of the same variable.

In the following, an algorithm is presented to map elements of sequences to the corresponding access ports in the program netlist. This algorithm is the basis for building the property logic for a SAT-based proof engine. To simplify the presentation, in the sequel, only the memory locations of the input/output variables are considered, not their symbol names, since this relationship can be established easily through the symbol table. The term *memory location* is used to refer to both an input or output variable, as defined in Section 6.3.1, if the use is clear from the context.

The first step in the algorithm is a topological sorting of the nodes in the execution graph. Every node is assigned a unique index m, with $m \in \mathbb{N}$, such that along every execution path the (instruction) node indexed with i is executed earlier than the node indexed with j, if $i < j$. In the sequel, for ease of explanation, a node is referred to by its index in the topological order. Each memory location Loc_k is associated with a set of nodes accessing this location, i.e., $W = \{i_1, i_2, \ldots, i_{|W|}\}$ with $i_j < i_{j+1}$ and $1 \leq j < |W|$. The access port AP_{i_j} for a node i_j is composed of $pdata_{i_j}$, $pact_{i_j}$, and $ploc_{i_j} = Loc_k$, corresponding to the signal names in Fig. 6.2.

Algorithm 3 compute index of the node associated with n-th sequence element

```
1:  function COMP_INDEX(n)
2:      if n = 1 then
3:          if pact_{i_1} = true then
4:              return i_1
5:          else if pact_{i_2} = true then
6:              return i_2
7:              ...

8:          else if pact_{i_|W|} = true then
9:              return i_|W|
10:         else
11:             return 0
12:         end if
13:     else
14:         if pact_{i_n} = true ∧ comp_index(n − 1) < i_n then
15:             return i_n
16:         else if pact_{i_{n+1}} = true ∧ comp_index(n − 1) < i_{n+1} then
17:             return i_{n+1}
18:             ...

19:         else if pact_{i_|W|} = true ∧ comp_index(n − 1) < i_|W| then
20:             return i_|W|
21:         else
22:             return 0
23:         end if
24:     end if
25: end function
```

Given a memory location Loc_k, to map the n-th element ($1 \leq n \leq |W|$) of Loc_k's

sequence to an access port AP_{i_j}, the challenging task is to find the index i_j of the node related to this element. The function *comp_index(n)*, depicted in pseudo-code notation in Algo. 3, performs this task. It is generated for every memory location Loc_k. The function *port_mapping()* of Algo. 4 is based on *comp_index()*. It connects the n-th element of the sequence to an access port in the program netlist.

The formulation of *comp_index()* is based on the fact that at any time, exactly one execution path of a program is active. An active path is characterized by the nodes in the program netlist whose *active* flags are asserted. In summary, *comp_index()* works as follows: To determine the n-th element of a sequence along any execution path, it first checks the *active* flag of the node with index i_n: this is the very first node that could be the n-th element of a sequence. It also examines whether the $(n-1)$-th element along that path exists already, by checking whether the index of the node associated with $(n-1)$-th element is smaller than the index of the current node. If both conditions are met, then the n-th element of the sequence is known to exist and the corresponding index can be returned. Otherwise the function moves on to the next candidate until a node related to the being searched element is found or does not exist. The *comp_index* function can be used to test whether an element of a sequence exists on a given path (by testing whether the result of *comp_index* is zero). This function is also used for verifying the execution order (cf. Section 6.3.3) of the sequence elements that are related to different memory locations (variables).

With the ability to obtain the index i_j of the node related to the n-th element of a sequence, it is straightforward to map the n-th element of the sequence to the access port AP_{i_j}. The function *port_mapping* depicted in Algo. 4 performs this task.

Algorithm 4 Map sequence to access port

```
1:  function PORT_MAPPING(n)
2:      if comp_index(n) = i_1 then
3:          return pdata_{i_1}
4:      else if comp_index(n) = i_2 then
5:          return pdata_{i_2}
6:          ...

7:      else if comp_index(n) = i_{|W|} then
8:          return pdata_{i_{|W|}}
9:      else
10:         return UNDEFINED
11:     end if
12: end function
```

In the remainder of this dissertation, for simplicity, the term *variable* is used for both an element of an input/output sequence and for the state of a register at the start node/end node of the program.

6.3.3 Execution Order

Besides being able to relate software accesses to the same location at different points in time, in many cases it is also important to specify a temporal order of accesses to different memory locations. For instance, in order to issue a new transaction, a peripheral device may require that its driver first write the configuration/data register of the device at memory location Loc_1, and then set the start flag at memory location Loc_2; not maintaining this order could result in undefined behavior of the device. Obviously, the property specification "**Loc_1'write#1 == Config_Data && Loc_2'write#1 == Start**" is insufficient for this requirement, since this statement does not define which of the two accesses, "**Loc_1'write#1**" and "**Loc_2'write#1**", is to be executed first by the software driver.

In RSPL, the temporal ordering of accesses to different locations can be specified using the *execution order* section in a property. A user may specify an execution order between an input and an output, two inputs related to two different memory locations and two outputs related to two different memory locations. Checking the execution order is implemented by comparing the results of the functions *comp_index* for the respective sequence elements. Taking the example from above, if the returned value of the function *comp_index* for "**Loc_1'write#1**" is smaller than the returned value of this function for "**Loc_2'write#1**", then "**Loc_1'write#1**" is executed first. The syntax definition and an example of an *execution_order* specification are given in Fig. 6.6.

```
// execution order definition
execution_order:
⟨sequence element⟩ > ⟨sequence element⟩;

// examples
execution_order:
config'write#1 > Start'write#1;
config'write#2 > Start'write#1;
config'read#1 > config'write#1;
```

Figure 6.6: Execution Order

6.3.4 Safety and Liveness Properties

So far, the basic concepts and building blocks of the property language have been introduced. This section will present how to use them to build a property, and it will show what kinds of properties a verification engineer can specify with RSPL.

A property begins with the keyword *property*, followed by a valid identifier. The general structure of the property body follows an assumption/guarantee style. The body consists of two optional sections, *execution_order* and *assume*, and one mandatory section, *prove*. The *assume* part specifies the circumstances under which the assertion as specified in the *execution_order* and *prove* parts is to be checked. If the assumption part

is denoted by a predicate a, the prove part by c and the execution order part by o, then a property p is translated to the Boolean formula $p := a \rightarrow (c \wedge o)$.

Given a property p, a verification engineer can instruct the property checker to check it as a safety property or as a liveness property. As illustrated in Fig. 6.7, a safety property is indicated by the keyword "*always*", and a liveness property is indicated by the keyword "*eventually*" .

```
// Example of property

property example1;

// assumption part
assume:
$R5'start == 2;

// prove part
prove:
data_out'write#1 == data_in'read#1 + 4;
$R4'end == 0;

// execution order part
execution_order:
data_out'write#1 > data_in'read#1;
endproperty

// safety property
inst1: always example1;
// liveness property
inst2: eventually example1;
```

Figure 6.7: Safety- /Liveness- Property

The semantics of "safety/liveness" is defined by evaluating the execution paths of a program. In contrast to the Kripke models used in LTL or CTL model checking, the program netlist contains a finite number of paths of finite length. This greatly simplifies the evaluation of safety and liveness properties. A safety property "*always* p" means that on every execution path (from a start state to an end state of the program), the property p holds. This is similar to the LTL property $\mathsf{G}\ p$, however, applied to a finite-length path. A liveness property "*eventually* p" means that there exists at least one execution path on which the property p holds. This is similar to the meaning of the CTL property $\mathsf{EG}\, p$. It is straightforward to check the safety property using an SAT solver. In order to check a liveness property, one only needs to check the safety property "*always* $\neg p$". In case this property holds, one may conclude that the corresponding liveness property fails.

6.3.5 Syntax Extensions

With the language elements presented so far, a user of RSPL is able to capture the reactive behavior of the software programs considered in this dissertation. In this section, a number of extensions to the syntax are introduced that do not increase its expressive power but make the property notation easier and more compact.

In the following, a variable *var* represents either an input (with attribute *read*) or an output (with attribute *write*). The symbol $\bowtie$ represents an arbitrary comparison operator, and *expr* represents any valid expression at either side of a comparison operator. The accessors depicted in Fig. 6.8 can be used to access a range of sequence elements related to a variable *var*. Every element is compared with the expression *expr*. If all comparison operations evaluate to true, then the result of this statement is true, otherwise it evaluates to false.

```
var#[m:n] ⋈ expr :=
(var#m ⋈ expr) && (var#(m+1) ⋈ expr) && ... && (var#n ⋈ expr)
with  m ≤ n and m,n ∈ ℕ
```

Figure 6.8: Access a range of elements (Universal)

The dual case is handled by the accessor depicted in Fig. 6.9: it evaluates to true if the comparison operations return true at least N times in sequence.

```
var#EN[m:n] ⋈ expr :=
(var#m ⋈ expr) || (var#(m+1) ⋈ expr) || ... || (var#n ⋈ expr)

with  m ≤ n, 0 < N < m − n, K ≥ N, and m,n,N ∈ ℕ,
where  K represents the number of terms evaluated true
```

Figure 6.9: Access a range of elements (Existential)

The function **exists** tests whether a sequence element exists on an execution path. In a safety property **exists** (**var#1**) checks whether **var#1** exists for every execution path, whereas in a liveness property it checks whether this element can be generated/consumed at least once during the execution of the program. This function can also be used to check whether the *assume* part of a property always evaluates to false due to non-existing sequence elements. Again, a SAT-based property checker can implement these checks using *comp_index()*.

An If-Then-Else statement is applicable to Boolean expressions. It specifies a conditional requirement for a program. Its semantics is defined by Boolean implication $p \rightarrow q$, which is equivalent to the Boolean expression $\neg p \vee q$. The "else" part is optional.

```
if (blexpr1)
    blexpr2;
else
    blexpr3;
endif;
```
Its semantics is equivalent to
$(blexpr_1 \rightarrow blexpr_2) \wedge (\neg blexpr1 \rightarrow blexpr_3)$

6.4 Completeness of Property Sets

The completeness of a set of properties for a hardware design can be proven by using design-independent approaches such as [48, 80]. The presented approach here is closely related to [80], which is explained in more detail in Section 3.4.2. This method proves that two models of a design satisfying a set of properties $\{p_i\}$ are sequentially equivalent in terms of input/output sequences. What input and output values are considered and at what time points (clock cycles) is specified by the user in terms of so-called *determination conditions*. For example, a "data" signal needs to be uniquely determined by the design whenever a corresponding "valid" signal is asserted. A complete property set fulfills this determination condition if every property specifies the expected "data" value at the time points when the "valid" signal becomes asserted. Note that a property set can be checked for completeness independently of any design implementation for which this set of properties holds, because only the relationships between the signal names in the properties are checked. This idea can also be transferred to software properties written in RSPL and results in the following definition for the completeness of a set of properties:

Definition 31. *A set of RSPL properties is called complete, iff*

1. *there exists a property with a matching assume part for every possible input sequence applied to the program, and*

2. *every property uniquely specifies every output sequence produced by the program under the input sequences specified in the assume part.*

Testing the two conditions of Def. 31 can be directly implemented in two checks, called the *Determination Test* and the *Case Split Test*.

6.4.1 Determination Test

Unlike hardware that generates output sequences for every time point (clock cycle), hardware-dependent software may produce output sequences of varying length, depending on the input sequences applied to the program. For instance, depending on the configuration data given by the program's environment, a software driver may perform burst write operations with 2 or 4 beats of data transfer, causing sequences with 2 or 4 elements, respectively. If design-independent methods are used, the completeness checker needs to know how many elements of sequences should every property at least specify. Denoting

these values for every output in every property is tedious and error-prone. Therefore, for checking the completeness of RSPL property sets, one has to give up on the design independence of a completeness criterion. Instead, the method presented here makes use of the software model to determine how long the checked sequences are on each program execution.

In order to ensure that every output signal is uniquely determined by the property set, two steps are performed. The first step ensures that every property describes every element of all the output sequences that are produced under all matching input sequences specified in the assumption part of the property. This problem is resolved with the help of the design under verification, M, and the *comp_index* function. For simplicity, in the following it will be assumed that the properties are written in a *causal* form, expressed as an implication between a cause (property assumption) and an effect (property commitment). This causal form is given if the input sequences are specified in the *assume* part of the property and the expected output behavior is specified in the *prove* and *execution_order* parts. Given a property $p := a \rightarrow (c \wedge o)$, by syntactic analysis, it must be trivial to identify the maximum sequence length k of any output sequence specified in the property. Then it has to be checked whether k is the maximum element generated by the software model M under the assumption a. For this purpose, the *comp_index(k+1)* function with respect to the assumption a is applied: if this function returns a non-zero value, this means that the program can output a sequence with at least $k + 1$ elements. The property under consideration does not specify this output sequence element, hence, a verification gap has been detected. Similarly, if a property does not mention at all some output produced by the program, this kind of gap can be detected by checking for a non-zero return of *comp_index(1)* for that output. A list of all possible outputs of a program can be easily obtained when synthesizing the model of the program.

Once it is certified that every property describes every possible element of all output sequences, the next step checks whether these output sequences are determined *uniquely*, i.e., whether, along any execution path, the property specifies exactly one value for every element of the output sequence. This can be done for every property, independently of the software model. Let p be a property containing a set of signals $\{v_i\}$ (composed of a set of inputs $\{x_j : j \in \mathbb{N}, j \leq m\}$ and a set of outputs $\{o_n : n \in \mathbb{N}, n \leq l\}$) and the corresponding sequence elements $\{v_i^{k_{v_i}} : k_{v_i} \in \mathbb{N}_{\neq 0}, k_{v_i} \leq t_{v_i}\}$. A copy p' of p is created by considering a copied set $\{v_i'\}$ of the variables appearing in the property and imposing the property on the copied variable set. The property p determines the outputs $\{o_n\}$ uniquely iff the following formula is a tautology

$$(p \wedge p' \wedge \bigwedge_{j=0}^{m} \bigwedge_{k_{x_j}=1}^{t_{x_j}} (x_j^{k_{x_j}} = {x_j'}^{k_{x_j}})) \rightarrow \bigwedge_{n=0}^{l} \bigwedge_{k_{o_n}=1}^{t_{o_n}} (o_n^{k_{o_n}} = {o_n'}^{k_{o_n}})$$

Note that in this formula, every element of each input sequence of the software program is assumed to be determined, and every element of each output sequence needs to be proven for determination. It is straightforward to adapt this formula to weaker determination conditions that allow some of the sequence elements to not be determined.

6.4.2 Case Split Test

The case split test checks whether the property set covers every possible input sequence. Given a set of properties p_i with their respective assumption parts a_i, the case split test is conducted by proving that the formula $\bigvee_i a_i$ is a tautology.

6.4.3 The Completeness Criterion

Theorem 2. *If and only if a set of RSPL properties $\{p_i\}$ passes both the Determination Test and the Case Split Test, then the property set is complete according to Def. 31.*

Proof: The theorem is true by Def. 31 because the Case Split Test checks for the fulfillment of condition 1 of Def. 31 and the Determination Test checks for the fulfillment of condition 2. □

6.5 Case Study

The property language developed in this dissertation has been successfully applied to specifying properties for an industrial software driver for a *Local Interconnect Network* (LIN) master node [108]. The software was developed by Infineon Technology AG. Note that the focus of this chapter is on the challenges of specifying complete sets of properties for this type of software, not on the proving techniques.

The properties shown in this case study have already been proven earlier [40], based on a manual construction of checker automata which were added to a program netlist model of the software. This section presents the formulation of these properties in RSPL. The hardware peripheral controlled by the software driver is a UART (Universal Asynchronous Receiver/Transmitter), connected to physical LIN bus lines. A LIN bus is composed of one master node and several slave nodes. Data is transmitted on the LIN bus in so-called *frames*. A frame is composed of several fields: a header, up to 8 bytes of data, and a checksum. The master node is responsible for sending the header field, which is composed of a *break* field indicating the start of a new frame, a *sync byte* field used for synchronization, and an *identifier* (ID) field. Slave nodes evaluate the identifier field and, if there is a match, then the corresponding slave node either sends or receives the data. The LIN driver code under consideration implements a master node. It supports six fixed-valued IDs. It can send or receive 2, 4 or 8 bytes of data for each of the six IDs. Data is communicated with the application software through shared memory locations, which serve as the interface of the LIN driver.

The first example of the property in Fig. 6.10 specifies the transmission of a frame, according to the protocol specification for the LIN bus. For reasons of space, only the case for data length of 2 bytes is shown. Furthermore, for readability, the names of variables and constants are used, instead of their memory addresses. The variables *data1* and *data2* store the payload data provided by the application software. The *s_id* are shared variables,

```
property lin_master_transmits_2_bytes;

assume:
s_id'read#1 == C_ID0;

prove:
// master task
uart'write#1 == C_BREAK;
uart'write#2 == C_SYNC;
uart'write#3 == C_ID0;

// slave task

// an example for undetermined uart'write#4 would be
// uart'write#4 == 1 || uart'write#4 == 0;
uart'write#4 == data1'read#1;
uart'write#5 == data2'read#1;
uart'write#6 == CHECKSUM;

execution_order:
data1'read#1 > uart'write#4;
data2'read#1 > uart'write#5;
endproperty;

// safety property
inst1: always lin_master_transmits_2_bytes;
```

Figure 6.10: LIN_TX_Frame_2_Bytes

storing the ID that needs to be transferred to the slave task. The symbol *uart* refers to the Tx/Rx buffer of the UART.

In the following, the prefix "C_" indicates a constant value. *C_ID0* identifies a 2-byte transmission. *CHECKSUM* abstracts the "checksum" computation.

It is also necessary to define an *execution_order* section in the property in order to specify that the data must be available before it is transmitted.

Note that a program that does not support *C_ID0* at all may nevertheless fulfill the property in Fig. 6.10. Therefore it needs to check the liveness property in Fig. 6.11 in order to make sure that at some point in time a *C_ID* = *frame* is indeed sent to the UART.

Fig.6.12 shows the use of the *exists* function in a property that checks whether the driver is capable of transmitting 8-byte data frames.

Obviously, the safety property in Fig. 6.10 does not completely specify the entire program. It specifies only the case that the LIN master transmits 2 bytes of data to a slave. The case split test presented in Section 6.4 identifies the missing cases by checking whether there exists a corresponding property for every value of *s_id*. The comments section of this property shows an example of a failing determination test for a sequence element, where this statement states that the value of "**uart'write#4**" could be either 0 or

```
property lin_test_C_ID0;

prove:
uart'write#3 == C_ID0;
endproperty;

// liveness property
inst2: eventually lin_test_C_ID0;
```

Figure 6.11: LIN Liveness

```
property lin_test_support_8_bytes;

prove:
exists(uart'write#11);
endproperty;

inst3: eventually lin_test_support_8_bytes;
```

Figure 6.12: LIN Liveness 2

1. Thus, the value of this variable is not *unique*: Suppose that "**uart'write#4**" is a Boolean variable. Then the expression in the statement evaluates to *true*. Hence, the property proves nothing about this variable.

Chapter 7

Summary

Nowadays, simulation is still the dominant technique applied to verify System-on-Chip designs. A testbench is responsible for generating test patterns for the design under verification. Such test patterns must be carefully selected so that as many of the design's functional aspects as possible are simulated. In order to evaluate the quality of test patterns, several coverage criteria are used. In contrast, formal property checking takes care of every possible input of a design, and it proves the properties of the design. Usually every property describes one aspect of the overall functionality of the design. Hence, for property checking, the coverage criteria from simulation are no longer applicable. However, a verification engineer still wants to know how many functionalities have been covered by the properties he/she wrote. In order to evaluate the coverage of properties, this dissertation adopted the coverage criterion from [80], which proves that a property set is complete, i.e., a complete property set uniquely covers every input sequence and every output sequence of the DuV. In general, checking the completeness of a property set for a DuV is as complex as the problem of checking the sequential equivalence of two circuits. The work of [41] considers an abstract state machine of the DuV and splits this overall proof into several subproofs by only considering a pair of properties at once. Every pair of properties indicates two possible consecutive transitions in the abstract state machine of the DuV. The limitation of this approach is that one has to manually extract the abstract state machine, and to make sure that the DuV contains exactly only one such state machine.

Divide-and-Conquer is one of the most intuitive ways to handle design complexity and to overcome the limitation of design/verification methodologies like the example presented above. Usually a system is divided into serval modules. Every module is designed/verified separately. However, for today's SoC systems, a module never works standalone, it works correctly under the assumption that its environment behaves correctly. When a design engineer implements such a module, he/she makes an implicit assumption about the correct input sequences for the module. For example, for a module in a communication protocol, the engineer assumes that the inputs of the module obey the protocol. For verifying such a module and to save time and resources of verification, environment constraints are needed that restrict the inputs of the DuV to only have the

values that are produced by the DuV's environment. Since the SoC modules are mostly reactive, the environment constraints applied to verify every module are also mostly reactive. For modules related to communication protocols, the reactive constraints can be obtained from the specifications of the protocols. For other modules, the verification engineers have to contact the design engineer or analyze the implementation code to get the environment constraints. Environment constraints are important for coverage analysis too, and save writing unnecessary and redundant test cases or properties for simulation or for property checking respectively.

On these grounds, selecting and formalizing environment constraints must be done very carefully, because any errors in the environment constraints could overconstrain the DuV, which could lead to an invalid verification result. In industry, the environment constraints are inspected manually, which is tedious and error-prone, especially for reactive constraints that specify sophisticated interactions between modules. On the other hand, since the correctness of a reactive constraint is dependent on the DuV itself, to validate reactive environment constraints, this cycle reasoning must be broken. Assume-guarantee reasoning is a formal method for validating environment constraints. This work presented a new assume-guarantee reasoning rule that can be applied to systems modeled as Mealy machines, and it addresses the problem of circular reasoning coming from the reactive constraints which are formulated in a property specification language such as PSL or SVA. For this purpose, two plausibility checks for reactive constraints have been introduced, which can identify the potential errors in the constraints when verifying the individual modules of a system. The presented implementability check aims to check whether there exists an implementation for a constraint. Obviously if a constraint is not implementable, then the constraint itself must be invalid. The vacuity check, that checks whether a constraint is always false, is only a special case of the implementability check presented in this dissertation. Since the oscillatory behavior caused by combinational loops can not be handled by a property checker, another plausibility check "Loop-free composibility" checks whether an implementation of the constraint builds a combinational loop with the DuV. The presented method makes an over-approximation analysis by doing a syntactic analysis, which may have false negative counterexamples due to false loops.

In addition, in this dissertation, some compositional rules for compositional coverage analysis have been derived from the presented novel assume-guarantee rule. By applying these rules, verification engineers can automatically validate the used reactive constraints for property checking and coverage analysis. On the other hand, obeying these rules guarantees that the property sets for individual modules are also valid for the system, and the composition of the complete property sets for individual modules also completely verifies the entire system. Although the compositional rules and the assume-guarantee rule are presented in a setting of SAT-based property checking, they can be applied to simulation based verification too.

The correctness of hardware-dependent software is critical for embedded systems, because other software layers, such as the application software and the operating systems, are highly dependent upon the hardware-dependent software. Testing hardware-dependent software is especially difficult, because one needs to verify the highly complex

interactions between the software and the hardware. This is a good reason to apply formal property checking to verify hardware-dependent software, and it is necessary to have a complete set of properties too. Also one faces the same above mentioned problem as with hardware if the divide-and-conquer approach is used to verify software systems.

Hardware-dependent software is similar to reactive hardware systems, which assume sequences of inputs and generate sequences of outputs. Therefore, the methods presented in Chapter 4 and Chapter 5 can be adopted for guaranteeing the completeness of the software properties, and can be used for compositional software verification. This can not be realized without a software property language that supports formulating the temporal reactive behavior between the software and hardware. Unfortunately, there has been very little research on property languages for software, not to mention for hardware-dependent software. In Chapter 6, a new reactive software property language RSPL was presented. RSPL facilitates the access of input/output interfaces; it simplifies specifying the input/output sequences between the hardware and software; and it is easy to use. Furthermore, Chapter 6 also presented a method to prove the completeness of the property sets written in RSPL. The presented method checks the completeness of the software property sets by proving that every input sequence and every output sequence are covered by the property sets. Such a completeness check is of particular importance for software property sets because a complete set of properties can, at least in principle, fully replace classical software tests. A typical source of errors when writing software is that the programmer simply forgets to treat certain input sequences in the program, causing undefined behavior when these inputs occur at runtime. Also, a verification engineer may forget to specify tests (or, in our case, properties) for such missing input sequences so that bugs can escape verification. The completeness check presented in this dissertation removes such verification gaps. Chapter 6 presented the fundamental framework of the RSPL, which contains the basic building block of a software property language. Based on them, by extending the syntax, the method for compositional verification as presented in Chapter 5 can be adopted for software verification. In addition, Chapter 6 presented an approach to combining the property logic with the software model, which was then further analyzed by a SAT-based model checker.

Chapter 8

Zusammenfassung

Im heutigem Verifikationsablauf für System-on-Chip(SoC) Designs spielen auf Simulation basierte Methoden immer noch eine große Rolle. Für die Simulation wird normalweise eine Testbench aufgestellt, die zuständig für das Erzeugen von Eingabestimuli für das Design unter Beweis ist. Solche Eingabestimuli müssen sorgfältig ausgewählt werden, damit möglichst viele Aspekte der gesamten Funktionalität des Designs simuliert werden. Um die Qualität der Eingabestimuli einzuschätzen, werden verschiedene Arten von Coveragekriterien für den Abdeckungstest (Engl. coverage analysis) verwendet. Coveragekriterien sind üblicherweise eine Menge von Regeln, welche durch Eingabestimuli erfüllt werden sollen. Eine Coverage von 100% bedeutet aber nicht, dass ein Design fehlerfrei ist, sondern gibt dem Verifikationsingenieuren mehr Vertrauen bzgl. der Fehlerfreiheit des Designs. Der Grund dafür ist, dass es fast unmöglich ist, alle Eingabestimuli eines Designs durchzusimulieren. Außerdem verlangen die modernen Coveragekriterien,

1. dass jedes Statement des HDL-Codes simuliert wird,
2. dass jedes Signal hin und her geschaltet wird,
3. dass jede Transition eines Zustandsautomaten durch Eingabestimuli überdeckt wird.

Solche Coveragekriterien sind leider ungenügend für die Hardwareverifikation. Beispielweise ist ein Fehler, der durch ein Verhalten verursacht wird, welches sich über viele Transitionen eines Automaten erstreckt, schwer zu überdecken. Formale Verifikation ist eine mathematische Methode um die funktionale Korrektheit eines Designs zu verifizieren. In der formalen Verifikation wird keine Testbench benötigt, da die formale Verifikation verifiziert, ob eine Implementierung eines Designs die sogenannten “Eigenschaften” (Engl. properties) des Designs erfüllt. Eine Eigenschaft, die üblicherweise in einer Beschreibungssprache für Eigenschaften (Engl. property specification language) formuliert wird, ist eine formale Beschreibung des Verhaltens eines Designs. Da formale Verifikation alle Eingabestimuli berücksichtigt, sind die Coveragekriterien der Simulation nicht mehr verwendbar. Im Allgemeinen beschreibt eine Eigenschaft nur Teile der gesamten Funktionalität eines Designs. Deshalb ist es für einen Verifikationsingenieur sehr wichtig, wann er keine weiteren Eigenschaften zu schreiben braucht.

Daher übernimmt diese Dissertation das in [80] eingeführte Coveragekriterium, welches die Vollständigkeit eines Eigenschaftensatzes überprüft. Eine vollständige Menge von Eigenschaften muss jede Eingabefolge des Designs abdecken und jede Ausgabefolge des Designs eindeutig determinieren. Die Komplexität für die Überprüfung der Vollständigkeit eines Eigenschaftensatzes ist prinzipiell gleich der Prüfung der sequenziellen Äquivalenz zweier Schaltungen. Die Methode von [41] verwendet einen abstrakten Automaten eines Designs und zerlegt damit die komplexe Prüfung der Vollständigkeit in mehrere kleine Tests. Jeder kleiner Test wird auf einem Paar Eigenschaften durchgeführt anstatt auf allen Eigenschaften gleichzeitig. Das Paar von Eigenschaften sind zwei mögliche Transitionen in einem abstrakten Automaten eines Designs. Die Einschränkung der Methode von [41] ist, dass ein Verifikationsingenieur den abstrakten Automaten aus dem Design von Hand extrahieren muss. Weiterhin muss er in der Lage sein ein komplexes System so zu spalten, dass jedes Modul nur genau einen derartigen abstrakten Automaten beinhaltet. Außerdem gibt es leider keine systematische Herangehensweise, den abstrakten Automaten eines Designs zu ermitteln.

"Teile und Herrsche" (Engl. Divide-and-Conquer) ist einer der einfachsten Ansätze, um die Komplexität eines Designs und die Einschränkungen der Methodik für den Entwurf und die Verifikation zu überwinden. Generell wird ein SoC-System in viele kleinere Module aufgeteilt. Jedes Modul wird separat entworfen und verifiziert. Allerdings sind einzelne Module eines heutigen SoC-Systems nicht alleine für sich lauffähig. Diese arbeiten nur unter der Annahme richtig, dass ihre Umgebung ebenfalls richtig arbeitet. Beim Entwurf eines solchen Moduls wird angenommen, dass das Modul nur gültige Eingabefolgen aus seiner Umwelt erhält. Zum Beispiel setzt ein Ingenieur während der Implementierung eines Moduls eines Kommunikationsprotokolls immer voraus, dass die Eingabe des Moduls mit dem Protokoll übereinstimmt. Zur Verifikation dieses Moduls werden oft sogenannte "(Environment) Constraints" benutzt, um die Verifikation nur auf die relevanten Szenarien zu beschränken. Dadurch können Kosten und Ressourcen zur Verifikation eingespart werden. Solche Constraints, welche mit Hilfe einer Verifikationssprache wie SystemVerilog Assertions (SVA) formuliert werden, beschreiben das korrekte Verhalten der Umgebung eines Moduls. Weil die Module eines SoC-Designs miteinander wirken, sind die Constraints der Module meistens auch reaktiv. Für die Module eines Kommunikationsbusses können reaktive Constraints einfach aus der Spezifikation des Kommunikationsprotokolls bezogen werden. Andererseits ist für andere Arten von Designs häufig eine Zusammenarbeit zwischen Designingenieuren und Verifikationsingenieuren erforderlich, um die korrekten Constraints der Module zu formulieren. Constraints sind auch bedeutsam für den Abdeckungstest, denn die Anwendung der Constraints spart die Zeit für das Schreiben von überflüssigen Testfällen der Simulation beziehungsweise von überflüssigen Eigenschaften der formalen Verifikation.

Aus diesen Gründen ist es besonders wichtig sicherzustellen, dass die Constraints fehlerfrei sind. Irrtümliche Constraints können das Designverhalten mehr als erwartet beschränken, sodass es zu falschen Ergebnissen der Verifikation führen kann. In der Halbleiterindustrie werden solche Constraints manuell geprüft. Das ist mühsam und fehleranfällig insbesonders für reaktive Constraints, die komplexe Interaktionen zwischen Modulen spe-

zifizieren. Da die Korrektheit der Constraints für das Design unter Verifikation (DuV) auch vom DuV selbst abhängt, muss dieser Zirkelschluss unterbrochen werden, um die Constraints formal zu validieren.

"Assume-Guarantee Reasoning" ist eine formale Methode für die Validierung von Constraints. Diese Dissertation präsentiert eine neue Regel für Assume-Guarantee Reasoning, welche auf Systeme angewendet werden kann, die als Mealy-Automat modelliert werden. Der neue Beitrag dieser Regel ist, dass man sie auf reaktive Constraints, die in einer Beschreibungssprache für Eigenschaften, wie SVA, formuliert sind, einsetzen kann. Dazu werden zwei Plausibilitätstests eingeführt. Einer davon, Implementierbarkeitstest genannt, prüft ob ein Constraint als Schaltung implementiert werden kann. Offensichtlich gilt, dass wenn ein Constraint nicht implementierbar ist, es falsch sein muss. Der andere Test garantiert, dass keine kombinatorische Schleifen zwischen den Constraints und dem DuV gebaut werden. Wie in Kapitel 4 erläutet, kann eine kombinatorische Schleife unerwartet zu starke Constraints verursachen. Dies kann zu Lücken in der Verifikation führen.

Außerdem wird im Rahmen dieser Dissertation eine neue Methodik für kompositionale Verifikation erforscht. Die präsentierten kompositionalen Regeln, hergeleitet von der in Kapitel 4 neu vorgestellten Regel für Assume-Guarantee-Reasoning, ermöglichen, die Validierung der Constraints automatisch durchzuführen. Weiterhin kann man unter Verwendung dieser Regeln folgern, dass das gesamte System vollständig verifiziert ist, wenn jedes einzelne Modul vollständig verifiziert ist. (Daher wird ein Modell des gesamten Systems zur Verifikation nicht benötigt.) Obwohl die neue Regel für Assume-Guarantee-Reasoning und die kompositionale Regeln im Rahmen des SAT-basierten Property Checking vorgestellt wurden, können sie trotzdem für simulationsbasierte Verifikation eingesetzt werden.

Ein Trend beim Entwurf von heutigen SoC-Designs ist, dass immer mehr Funktionalität, die früher in Hardware implementiert wurde, jetzt in Software, insbesondere in hardwarenaher Software, realisiert wird. Eine Garantie der Korrektheit solcher hardwarenahen Softwareprogramme ist äußerst wichtig, da diese die Grundlage aller anderen Ebenen von Software, wie z.B. Anwendungssoftware und Betriebssysteme, ist. Allein mit Softwaretests ist es sehr schwer, Fehler in hardwarenahe Software zu erkennen, da diese mit der Hardware zusammenspielt. Dieses Zusammenspiel ist ein reaktives Verhalten der Schnittstelle zwischen der Hardware (HW) und der Software (SW). Deswegen ist es sehr bedeutsam, die Fehlerfreiheit der Kommunikation bei HW/SW Schnittstellen sicherzustellen. Normalerweise wird das Verhalten der HW/SW Schnittstelle wie auch bei Hardwaremodulen durch Sequenzen von Signalen der HW/SW Schnittstelle repräsentiert. Deshalb sind die in Kapitel 4 und Kapitel 5 präsentierten Methoden sehr interessant und können leicht angepasst werden für die Verifikation hardwarenaher Software, besonders für die Validierung deren HW/SW Schnittstelle. Die Methoden in Kapitel 4 und in Kapitel 5 verlangen eine formale Beschreibung des Verhaltens der Schnittstelle zwischen den Modulen. Diese Beschreibung wird in einer Beschreibungssprache für Eigenschaften, wie SVA oder PSL, formuliert. Zum Spezifizieren hardwarenaher Softwareprogramme sind SVA und PSL nicht verwendbar, weil die beiden Sprachen für die Verifikation von Hard-

ware gedacht sind. “Run-time Assertions” von einer Programmiersprache können nur “non-temporal” Verhalten beschreiben. Daher sind sie ungenügend für das Spezifizieren von hardwarenaher Software, deren Verhalten auch “temporal” sein könnte. Traditionelle temporale Sprachen wie “Computation Tree Logic” (CTL) und “Linear Time Logic” (LTL) sind für die Verifikation der hardwarenahen Software nicht leicht zu verwenden. Demzufolge präsentiert Kapitel 6 eine Beschreibungssprache, “Reactive Property Specification Language (RSPL)” genannt, für hardwarenahe Software. Mit dieser Sprache kann das Verhalten der HW/SW Schnittstelle leicht definiert werden. Ferner erlaubt die Sprache die Anwendung der in Kapitel 4 und in Kapitel 5 präsentierten Methoden für kompositionale Verifikation. Weiterhin wird in Kapitel 6 eine Methode für die Prüfung der Vollständigkeit einer Menge von RSPL-Eingenschaften vorgestellt. Dieser Ansatz lässt sich in der Softwareverifikation leichter umsetzen als in der Hardwareverifikation. Die Gründe dafür sind:

1. Mit RSPL kann man das globale Verhalten einer hardwarenahen Software über die HW/SW Schnittstelle spezifizieren.

2. Das benutzte Modell für hardwarenahe Software kann nur terminierende Software modellieren. Anders formuliert, die modellierte Software kann nur endlich lange Eingabesequenzen bearbeiten, welche nur endlich lange Ausgabesequenzen erzeugen können.

Deshalb sind als Vollständigkeitskriterium für RSPL-Eigenschaften nur “determination test” und “case split test” notwendig. Sie garantieren, dass alle Eingabefolgen und alle Ausgabefolgen eines Softwareprogrammes eindeutig determiniert sind. Eine solche Vollständigkeitprüfung für Software ist auch sehr interessant für Softwaretester, weil ein häufiger Fehler beim Schreiben der Testcases ist, dass einfach ein Testcase vergessen wird, was eine Verifikationslücke verursacht. Die vorgeschlagenen Vollständigkeitskriterien können diese Verifikationslücke schließen.

Appendix A

Backus-Naur-Form for Reactive Software Property Language

⟨*PropertyDefinition*⟩ ::= ‘`property`’ ⟨*PropertyIdentifier*⟩ ‘`;`’
⟨*PropertyBody*⟩
‘`endproperty`’ ‘`;`’

⟨*PropertyIdentifier*⟩ ::= ⟨*Identifier*⟩

⟨*Identifier*⟩ ::= ⟨*Letter*⟩, {⟨*Letter*⟩|⟨*Digit*⟩|‘_’}

⟨*Letter*⟩ ::= ⟨*UppercaseLetter*⟩|⟨*LowercaseLetter*⟩

⟨*Digit*⟩ ::= ⟨*DigitwithoutZero*⟩|‘`0`’

⟨*PropertyBody*⟩ ::= [⟨*AssumePart*⟩], ⟨*ProvePart*⟩, [⟨*ExecutionOrderPart*⟩]

⟨*AssumePart*⟩ ::= ‘`assume`’ ‘`:`’ ⟨*BooleanExpressions*⟩ ‘`;`’

⟨*BooleanExpressions*⟩ ::= ⟨*BooleanExpression*⟩, {‘`;`’ , ⟨*BooleanExpression*⟩}

⟨*BooleanExpression*⟩ ::= ⟨*AndBooleanExpression*⟩
| ⟨*BooleanExpression*⟩ ‘`||`’ ⟨*AndBooleanExpression*⟩

⟨*AndBooleanExpression*⟩ ::= ⟨*AtomicBooleanExpression*⟩
| ⟨*AndBooleanExpression*⟩ ‘`&&`’ ⟨*AtomicBooleanExpression*⟩

⟨*AtomicBooleanExpression*⟩ ::= ‘(’⟨*BooleanExpression*⟩‘)’
| ‘!’ ‘(’ ⟨*BooleanExpression*⟩ ‘)’
| ⟨*EqualExpression*⟩

⟨*EqualExpression*⟩ ::= ⟨*RelationalExpression*⟩
| ⟨*EqualExpression*⟩ ‘==’ ⟨*RelationalExpression*⟩
| ⟨*EqualExpression*⟩ ‘!=’ ⟨*RelationalExpression*⟩

⟨*RelationalExpression*⟩ ::= ⟨*NumericalExpression*⟩
| ⟨*RelationalExpression*⟩ ‘>=’ ⟨*NumericalExpression*⟩
| ⟨*RelationalExpression*⟩ ‘<=’ ⟨*NumericalExpression*⟩
| ⟨*RelationalExpression*⟩ ‘>’ ⟨*NumericalExpression*⟩
| ⟨*RelationalExpression*⟩ ‘<’ ⟨*NumericalExpression*⟩

⟨*NumericalExpression*⟩ ::= ⟨*BitwiseXOR*⟩
| ⟨*NumericalExpression*⟩ ‘|’ ⟨*BitwiseXOR*⟩

⟨*BitwiseXOR*⟩ ::= ⟨*BitwiseAND*⟩
| ⟨*BitwiseXOR*⟩ ‘^’ ⟨*BitwiseAND*⟩

⟨*BitwiseAND*⟩ ::= ⟨*ShiftOperation*⟩
| ⟨*BitwiseAND*⟩ ‘&’ ⟨*ShiftOperation*⟩

⟨*ShiftOperation*⟩ ::= ⟨*AdditiveExpression*⟩
| ⟨*ShiftOperation*⟩ ‘<<’ ⟨*AdditiveExpression*⟩
| ⟨*ShiftOperation*⟩ ‘>>’ ⟨*AdditiveExpression*⟩

⟨*AdditiveExpression*⟩ ::= ⟨*MultExpression*⟩
| ⟨*AdditiveExpression*⟩ ‘+’ ⟨*MultExpression*⟩
| ⟨*AdditiveExpression*⟩ ‘−’ ⟨*MultExpression*⟩

⟨*MultExpression*⟩ ::= ⟨*UnaryExpression*⟩
| ⟨*MultExpression*⟩ ‘⋆’ ⟨*UnaryExpression*⟩
| ⟨*MultExpression*⟩ ‘/’ ⟨*UnaryExpression*⟩
| ⟨*MultExpression*⟩ ‘%’ ⟨*UnaryExpression*⟩

⟨*UnaryExpression*⟩ ::= ⟨*AtomicNumericalExpression*⟩
| ‘+’ ⟨*AtomicNumericalExpression*⟩
| ‘−’ ⟨*AtomicNumericalExpression*⟩

⟨*AtomicNumericalExpression*⟩ ::= ‘(’ ⟨*NumericalExpression*⟩ ‘)’
| ⟨*SequenceElement*⟩
| ⟨*Integer*⟩

⟨*SequenceElement*⟩ ::= ⟨*CIdentifier*⟩“’ ⟨*WriteReadAttribute*⟩ ‘#’ ⟨*NaturalNumber*⟩
| ⟨*CIdentifier*⟩“’ ⟨*StartEndAttribute*⟩

⟨*CIdentifier*⟩ ::= ⟨*Identifier*⟩

⟨*WriteReadAttribute*⟩ ::= ‘`read`’|‘`write`’

⟨*StartEndAttribute*⟩ ::= ‘`start`’|‘`end`’

⟨*NaturalNumber*⟩ ::= ⟨*DigitwithoutZero*⟩, {⟨*Digit*⟩}

⟨*Integer*⟩ ::= [‘−’], ⟨*DigitwithoutZero*⟩, {⟨*Digit*⟩} | ‘`0`’

⟨*ProvePart*⟩ ::= ‘`prove`’ ‘`:`’ ⟨*BooleanExpressions*⟩ ‘`;`’

⟨*ExecutionOrderPart*⟩ ::= ‘`execution_order`’ ‘`:`’ ⟨*ExecutionOrders*⟩ ‘`;`’

⟨*ExecutionOrders*⟩ ::= ⟨*ExecutionOrder*⟩, {‘`;`’, ⟨*ExecutionOrder*⟩}

⟨*ExecutionOrder*⟩ ::= ⟨*SequenceElement*⟩ ‘>’ ⟨*SequenceElement*⟩

⟨*PropertyFile*⟩ ::= ⟨*PropertyDefinitions*⟩
| ⟨*Directives*⟩

⟨*PropertyDefinitions*⟩ ::= ⟨*PropertyDefinition*⟩, {⟨*PropertyDefinition*⟩}

⟨*Directives*⟩ ::= {⟨*Directive*⟩}

⟨*Directive*⟩ ::= ⟨*Identifier*⟩‘`:`’⟨*SafetyLiveness*⟩ ⟨*PropertyIdentifier*⟩‘`;`’

⟨*SafetyLiveness*⟩ ::= ‘`always`’ | ‘`eventually`’

⟨*DigitwithoutZero*⟩ ::= ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

⟨*UppercaseLetter*⟩ ::= ‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’ | ‘G’ | ‘H’ | ‘I’ | ‘J’ | ‘K’ | ‘L’ | ‘M’ | ‘N’ | ‘O’
| ‘P’ | ‘Q’ | ‘R’ | ‘S’ | ‘T’ | ‘U’ | ‘V’ | ‘W’ | ‘X’ | ‘Y’ | ‘Z’

⟨*LowercaseLetter*⟩ ::= ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’ | ‘h’ | ‘i’ | ‘j’ | ‘k’ | ‘l’ | ‘m’ | ‘n’ | ‘o’
| ‘p’ | ‘q’ | ‘r’ | ‘s’ | ‘t’ | ‘u’ | ‘v’ | ‘w’ | ‘x’ | ‘y’ | ‘z’

List of Figures

Bibliography

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

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

[3] Y. Matsunaga. An efficient equivalence checker for combinational circuits. In *Proc. International Design Automation Conference (DAC)*, pages 629–634, June 1996.

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

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

[6] Armin Biere and W. Kunz. SAT and ATPG: Boolean engines for formal hardware verification. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 782 – 785, San Jose, November 2002.

[7] Alan J. Hu. Formal hardware verification with BDDs: An introduction. In *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM)*, volume 2, pages 677 –682, aug 1997.

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

[9] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes – The Automata-Theoretic Approach*. Princeton University Press, Princeton, New Jersey, 1994.

[10] IEEE standard for system verilog – unified hardware design, specification, and verification language. *IEEE Std. 1800-2009*, 2009.

[11] IEEE Standard for Property Specification Language (PSL). *IEEE Std 1850-2005*, 2005.

[12] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, London, England, 1999.

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

[14] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 176–194, London, UK, UK, 2001. Springer-Verlag.

[15] Kavita Ravi and Fabio Somenzi. Hints to accelerate symbolic traversal. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 250–266. Springer Berlin Heidelberg, 1999.

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

[17] Adam Rose, Stuart Swan, John Pierce, and Jean-Michel Fernandez. Transaction level modeling in systemc.

[18] Grant Martin, Brian Bailey, and Andrew Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

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

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

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

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

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

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

[25] David L. Dill. The Murphi Verification System. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 390–393, London, UK, 1996. Springer-Verlag.

[26] Patrice Godefroid. Verisoft: A tool for the automatic analysis of concurrent reactive software. In Orna Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 476–479. Springer Berlin / Heidelberg, 1997.

[27] F.Ivancic Z. Yang, M.K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. FSoft software verification platform. In *Proc. International Conference Computer Aided Verification (CAV)*, pages 301–306. Springer, 2005.

[28] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi c programs. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[29] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.

[30] W. Visser, K. Havelund, G. Brat, and Seungjoon Park. Model checking programs. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, pages 3–11, 2000.

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

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

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

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

[35] Lori A. Clarke. A program testing system. In *Proceedings of the 1976 Annual Conference*, ACM '76, pages 488–491, New York, NY, USA, 1976. ACM.

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

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

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

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

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

[41] J. Bormann. *Vollständige Verifikation*. Dissertation, Technische Universität Kaiserslautern, 2009.

[42] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 353–362, Piscataway, NJ, USA, 1989. IEEE Press.

[43] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–535, May 1995.

[44] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16, 1991.

[45] Anubhav Gupta, KennethL. McMillan, and Zhaohui Fu. Automated assumption generation for compositional verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 420–432. Springer Berlin Heidelberg, 2007.

[46] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Form. Methods Syst. Des.*, 15(1):7–48, July 1999.

[47] Kenneth L. McMillan. A compositional rule for hardware design refinement. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 24–35, London, UK, UK, 1997. Springer-Verlag.

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

[49] C. Lee. Representation of switching circuits by binary-decision programs. *Bell Systems Technical Journal*, 38:985–999, July 1959.

[50] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

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

[52] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

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

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

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

[56] J. Bhasker. *A Verilog HDL Primer, Second Edition*. Star Galaxy Publishing, 1999.

[57] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[58] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proc. ACM/IEEE International Design Automation Conference (DAC)*, pages 403–407. ACM Press, 1991.

[59] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design*, 13(4):401–424, April 1994.

[60] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model verifier. In N. Halbwachs and D. Peled, editors, *Proc. International Conference on Computer Aided Verification (CAV)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer.

[61] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[62] Jiazhao Xu, M. Williams, H. Mony, and J. Baumgartner. Enhanced reachability analysis via automated dynamic netlist-based hint generation. In *Formal Methods in Computer-Aided Design (FMCAD), 2012*, pages 157–164, Oct 2012.

[63] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = lazy clause generation. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, LNCS*. Springer-Verlag, 2007.

[64] M. Wedler, D. Stoffel, and W. Kunz. Frontend model generation for SAT-based property checking. In *Proc. International Conference On ASIC (ASICON)*, pages 1017 – 1022, Shanghai, China, October 2005 (invited paper).

[65] M. Wedler, D. Stoffel, and W. Kunz. Arithmetik reasoning in DPLL-based SAT solving. Technical Report EIS-11-03-1, Dept. of Electrical and Computer Engineering, University of Kaiserslautern, Germany, 2003.

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

[67] S. Loitz, M. Wedler, C. Brehm, T. Vogt, N. Wehn, and W. Kunz. Proving functional correctness of weakly programmable IPs - a case study with formal property checking. In *Proc. 6th IEEE Symposium on Application Specific Processors (SASP)*, pages 48 –54, Anaheim, CA, USA, June 2008.

[68] Alexander Thomas, Jürgen Becker, Ulrich Heinkel, Klaus Winkelmann, and Jörg Bormann. Formale verifikation eines sonet/sdh framers. In *MBMV*, pages 280–288, 2004.

[69] MosheY. Vardi. Branching vs. linear time: Final showdown. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer Berlin Heidelberg, 2001.

[70] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, January 1986.

[71] Open verification library (ovl). http://www.accellera.org/activities/committees/ovl.

[72] Standard SystemC language reference manual. *IEEE Std 1666-2011*, 2011.

[73] Andrew Piziali. *Functional Verification Coverage Measurement and Analysis*. Springer US, 2008.

[74] Sagi Katz, Orna Grumberg, and Daniel Geist. "have i written enough properties?" - a method of comparison between specification and implementation. In *Proc. Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 280–297, London, UK, 1999. Springer-Verlag.

[75] Te-Chang Lee and Pao-Ann Hsiung. Mutation coverage estimation for model checking. In Farn Wang, editor, *Automated Technology for Verification and Analysis (ATVA)*, volume 3299 of *Lecture Notes in Computer Science*, pages 354–368. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-30476-0_29.

[76] Yatin Hoskote, Timothy Kam, Pei-Hsin Ho, and Xudong Zhao. Coverage estimation for symbolic model checking. In *Proc. International Design Automation Conference (DAC)*, pages 300–305, New York, NY, USA, 1999. ACM.

[77] Serdar Tasiran and Kurt Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design and Test of Computers*, 18(4):36 –45, jul/aug 2001.

[78] P. Basu, S. Das, A.Banerjee, P. Dasgupta, P.P. Chakrabarti, C.R. Mohan, L. Fix, and R. Armoni. Design-intent coverage-a new paradigm for formal property verification. *IEEE Transactions on Computer-Aided Design*, 25(10):1922–1934, October 2006.

[79] Daniel Grosse, Ulrich Kühne, and Rolf Drechsler. Estimating functional coverage in bounded model checking. In *Proc. International Conference on Design, Automation and Test in Europe (DATE)*, pages 1176–1181, 2007.

[80] Joerg Bormann and Holger Busch. Verfahren zur Bestimmung der Güte einer Menge von Eigenschaften (Method for determining the quality of a set of properties). European Patent Application, Publication Number EP1764715, 09 2005.

[81] Joakim Urdahl, Dominik Stoffel, and Wolfgang 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.

[82] Xilinx. Xilinx ise. http://www.xilinx.com/products/design-tools/ise-design-suite/.

[83] Onespin Solutions GmbH. Germany. OneSpin 360MV. http://www.onespin-solutions.com.

[84] Finn Haedicke, Daniel Große, and Rolf Drechsler. A guiding coverage metric for formal verification. In *DATE*, pages 617–622, 2012.

[85] A. Pnueli. In transition from global to modular temporal reasoning about programs. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 123–144. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[86] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[87] K.L. McMillan. Circular compositional reasoning about liveness. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 342–346. Springer Berlin Heidelberg, 1999.

[88] J. Yuan, C. Pixley, and A. Aziz. *Constraint-based verification*. Springer, 2006.

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

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

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

[92] M. Schickel, V. Nimbler, M. Braun, and H. Eveking. On consistency and completeness of property sets: Exploiting the property-based design process. In *Proc. Forum on Design Languages*, 2006.

[93] Jan Langer and Ulrich Heinkel. High level synthesis using operation properties. In *Proc. of Forum on Specification Design Languages (FDL 2009)*, pages 1–6, Sep. 2009.

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

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

[96] opencores.org. `www.opencores.org`.

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

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

[99] R.E. Bryant. Symbolic simulation-techniques and applications. In *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pages 517–521, June 1990.

[100] AbsInt Angewandte Informatik GmbH. Germany. Astrée. http://www.absint.com.

[101] Coverty, Inc. USA. Coverty Code Advisor. http://www.coverity.com/.

[102] The MathWorks, Inc. USA. Polyspace - Static Analysis Tools. http://www.mathworks.com/products/polyspace/.

[103] Thomas Ball and Sriram K. Rajamani. Slic: A specification language for interface checking (of c). Technical Report MSR-TR-2001-21, Microsoft Research, January 2002.

[104] Dirk Beyer, AdamJ. Chlipala, ThomasA. Henzinger, Ranjit Jhala, and Rupak Majumdar. The blast query language for software verification. In Roberto Giacobazzi, editor, *Static Analysis*, volume 3148 of *Lecture Notes in Computer Science*, pages 2–18. Springer Berlin Heidelberg, 2004.

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

[106] Shrinidhi Udupi. Generation of abstract cells for aquarius architecture via gapfree verification. Master's thesis, TU Kaiserslautern, 2011.

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

[108] LIN Administration. LIN Specification Package Rev. 2.0, 2003.

Lebenslauf

Name:	Binghao Bao

Schulbesuch und Studium

1986-1991	Grundschule in Chifeng, VR China
1991-1994	Middle School in Chifeng, VR China
1994-1997	High School in Chifeng, VR China
1997-2001	Bachelorstudium Elektrotechnik an der Technischen Universität Liaoning
2002-2007	Diplomstudium Elektrotechnik an der Technischen Universität Kaiserslautern

Berufstätigkeit

2007-2008	Application Engineer bei der Firma OneSpin Solutions GmbH in München
2009-2014	Wissenschaftlicher Mitarbeiter und Promotionsstudent am Lehrstuhl Entwurf Informationstechnischer Systeme des Fachbereichs Elektrotechnik und Informationstechnik der Technischen Universität Kaiserslautern
2015-jetzt	R&D Engineer bei der Firma Advantest Europe GmbH in Böblingen