Path Predicate Abstraction

for Sound System-Level Modeling of Digital Circuits

Vom Fachbereich Elektrotechnik und Informationstechnik
der Technische Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor der Ingenieurswissenschaften (Dr.-Ing.)

genehmigte Dissertation von

Joakim Henrik Urdahl
geboren in Lærdal, Norwegen

D 386

Datum der mündlichen Prüfung:	15. Dezember 2015
Dekan des Fachbereichs:	Prof. Dr.-Ing. Hans D. Schotten
Vorsitzender der Prüfungskommission:	Prof. Dr. Gerhard Fohler
Gutachter:	Prof. Dr.-Ing. Dr. Wolfgang Kunz
	Prof. Dr.-Ing. Kjetil Svarstad, NTNU

Acknowledgments

This thesis documents several years of research carried out while I have been part of the “electronic design automation group” at the University of Kaiserslautern. During this entire period the work has been conducted in the close collaboration of our group.

Specifically, I would like to mention Markus Wedler, Binghao Bao and Shrinidhi Udupi for their contributions to research employed in this thesis, Christian Bartsch for his help with the German summary, and Carmen Vicente-Fess for organizing my life.

Most of all, I would like to especially thank my supervisors, Prof. Wolfgang Kunz and Prof. Dominik Stoffel, who have been as dedicated to this research as myself during all these years. Thank you for all your time and your incredible devotion to our research. I am extremely proud that you have let me work with you.

Many thanks also to Prof. Kjetil Svarstad for his in-depth analysis in the review of this thesis and to Prof. Gerhard Fohler for chairing the examination procedures.

To all friends that have made my time in Germany cheerful.

To my family, thank you for supporting me. I look forward to see more of you.

Kaiserslautern, 14.01 2016
Joakim Urdahl

Contents

Chapter 1

Introduction

In 1938 Claude Shannon showed how any Boolean logic expression can be realized as an electric circuit [50]. This made it possible to distinguish a circuit's digital functional behavior from its physical realization. Since then digital data processing has revolutionized modern technology. The advances in the field have been impressive. At a very high pace, device sizes were shrinking, energy consumption decreased, functional capabilities extended and the performance improved.

Immensely large and complex circuitry can today be placed on a single integrated circuit (IC) which can be acquired at prices where even the simplest of home appliances can take benefit from its use. Several billion transistors can be placed on a single IC who's physical size does not exceed that of a fingernail and which requires only a small battery as power source. The creative potential at disposal for the development of such circuits is tremendous. Realizing this potential, however, means to cope with an equally tremendous complexity. It is the basis for a vigorous field of research where highly complex mathematical theories are brought to life in an actual physical embodiment.

Understanding and reasoning about immensely large networks of electronic components is only possible through a hierarchy of abstractions. A direct analysis of the entire system at the physical implementation layer is far out of scope even for computer-based methods. On the other hand, reasoning about an abstract description is of limited value if it does not also imply that the actual physical realization behaves accordingly.

Based on Shannon's work in [50], Boolean logic augmented with storage elements can be trusted as a "sound abstraction" for the digital functional behavior of an electronic circuit, i.e., any Boolean propositions valid in the Boolean model of the circuit will also hold in its physical realization (if implemented correctly). More recent work on equivalence checking has also established the descriptions at the next higher level of abstraction, the register transfer level (RTL), as sound.

The complexity of today's circuits, however, imposes big challenges even at the RT level. One may resort to descriptions at even higher levels of abstractions, commonly referred to as descriptions at an electronic system level (ESL). This thesis is motivated by the observation that in most pratical cases, however, these ESL descriptions are of ad hoc nature only. A well-defined relationship between the ESL abstraction layer and the underlying RTL does not exist. The proposed thesis attempts to close this gap. Taking today's industrial design styles as starting point, it contributes a theoretical foundation as well as practical methodologies that establish a "soundness" for descriptions at the ESL abstraction layer.

1.1 Abstract Circuit Descriptions

Reasoning in complex circuitry is done through a hierarchy of model descriptions at different levels of abstraction. The level of detail of a description at a lower abstraction level can only be handled when the circuit under consideration is kept small enough. As the size of the considered circuit increases, then, at some point, the amount of detail cannot anymore be coped with and a more abstract description, i.e., a description at a higher abstraction level, is required. An arbitrary abstract description is, however, not sufficient since the reasoning done at the higher abstraction level must be valid also for the lower abstraction levels.

A principle of locality is used to guarantee the soundness of an abstraction level. Only those details that can be ensured for a defined sub-circuit at the lower abstraction level can be ignored in the higher abstraction level. What is considered a single atomic component at one level of abstraction is therefore, at the next lower level of abstraction, itself a system of components.

1.1.1 Transistor Level

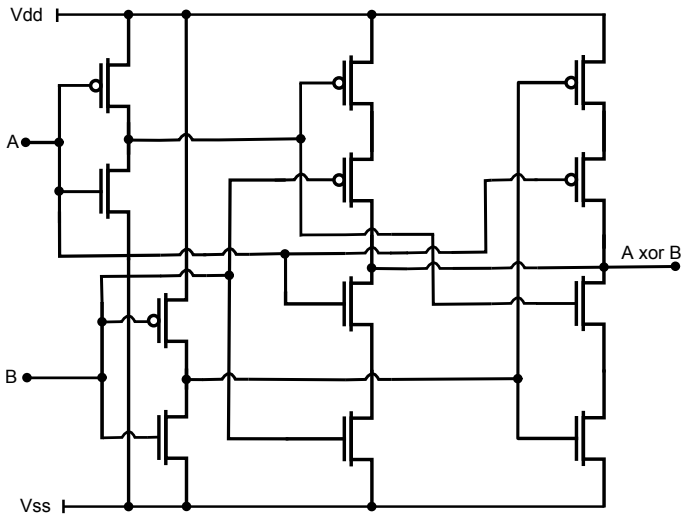
A modern digital circuit is realized as a network of transistors which is integrated on a single piece of semiconductor material, i.e., as an integrated circuit (IC). In a transistor-level description this network of transistors is described explicitly. Using a transistor-level simulator, e.g., SPICE [42], physical characteristics of the transistors, power source, and other possible components can be specified and the behavior of the circuit, in terms of its physical (analog) behavior, can be simulated.

1.1.2 Gate Level

Electrical voltage can be used to represent Boolean values and electrical components can be used to realize Boolean operations. Any Boolean expression may therefore be realized as a system of sub-circuits where each sub-circuit realizes a Boolean operation. This insight is the basis for all modern data processing. It implies that Boolean logic can be considered a description language for electronic circuits which is sound with respect to the digital functional behavior. The same also applies to the realization of binary storage elements, i.e., flip-flops and latches.

We may demonstrate this at the example of the Boolean logic operation *xor* (exclusive or). A transistor-level description of an *xor* realization is shown in Fig. 1.1a. Using a transistor-level simulator most aspects of this circuit can be analyzed, the digital functional behavior would be one such aspect. After appropriate simulations it can safely be assumed that the circuit correctly realizes an *xor* operation. This is a very important result, it means that we may safely use *xor* as an atomic component to describe the digital functional behavior of a circuit. The ANSI standard symbol for an *xor-gate* is shown in Fig. 1.1b.

All Boolean operations and binary storage elements can be verified and represented in a similar way. This brings us to the next higher level of abstraction known as the *gate level*. At the gate level details to how each Boolean operation and each bit-storage element is actually realized are ignored. Still, the digital functional behavior of a realized circuit can be safely analyzed based on its gate-level description.



(a) Transistors netlist of xor operation

(b) ANSI standard xor gate

Figure 1.1: xor operation at transistor level vs. gate level

1.1.3 Register Transfer Level

Progress in semiconductor device fabrication techniques has led to higher integration densities opening the possibility to realize larger circuits and therefore more advanced functionality on an IC. Describing such large and complex functionality at the gate level is, however, impractical. To exploit the possibilities offered by ICs of Very Large Scale Integration (VLSI) the level of abstraction is raised to a level known as the Register Transfer (RT) level.

The RT level has been known as an abstraction level for circuit descriptions since the early 50's, when descriptions inspired by software programming languages were employed to model the functionality of circuits in an easily understandable way [5].

In contrast to a software program, a circuit is a hardwired network of physical components. Its digital functional behavior is well defined as a finite state machine (cf. 2.1.1) with a state space encoded over a set of bit vectors referred to as *registers*. The behavior of such an FSM can be described as *registers transfers*, similar to how the state of a software program is described in an imperative programming language in terms of variables.

Note that languages at the RT level were, for a long time, only considered modeling languages and were not used for the actual realization of the design. Originally, also the RTL description languages commonly used today, i.e., VHDL and Verilog, were merely intended for modeling and simulation.

The first tools to automate the transformation from RTL descriptions to gate-level descriptions, a process known as logic synthesis, came in the mid 80s. Early tools, however, were not trusted to produce well optimized and/or correct gate-level descriptions, i.e., the RTL descriptions were still not considered a sound abstraction level. A manual analysis of gate-level descriptions was therefore still necessary. In other words, the gate level remained the level of abstraction for verification.

The invention of equivalence checking [36, 10] changed this. Using equivalence checking an equivalence could be formally proven between the synthesized gate-level model and the originating RTL model. Today, RTL models are trusted and the RT level is considered a sound level of abstraction.

1.1.4 Electronic System Level

The amount of functionality that is placed on a single chip has continued to grow since RTL was established as a sound abstraction level.

On a single modern IC all the functionality traditionally realized as a system of several communicating ICs can be implemented. Such an IC is therefore sometimes referred to as a *System-on-Chip (SoC)*. Continuous advancements in semiconductor device fabrication has made this increased scale of integration possible. The technology has, however, also reached a critical point where a substantial increase of operation frequency is not anymore possible [51]. This pushes the development towards even more functional complexity since better performance now requires advancements to the architecture, e.g., efficient pipelines and several parallel processor cores.

In the context of a modern digital system, an RTL description is highly detailed and differs much from the abstract view of a system engineer. The bit- and cycle-accurate RTL description may consist of hundreds, possibly thousands of implicitly synchronized processes. This is a difficult challenge with current design and verification methods.

The compositional design approaches supported in hardware description languages, e.g., VHDL and Verilog, ameliorate the problem. RTL modules can either be developed specifically for the project, re-used from other projects, or even be bought as Intellectual Property (IP) from external developers. The design process benefits greatly from such re-use, the system verification, however, does not.

Opposed to the design methods, current verification methods are not compositional, i.e., the verification of each individual module cannot be leveraged for use in a system verification. To handle any further growth in functional complexity it would therefore be highly desirable to raise the level of abstraction both for design and for verification.

A wide range of methods based on various descriptions at abstraction levels higher than RTL exist. These are used to simulate and establish confidence in specific aspects of the system design. The abstraction levels of all of these descriptions are, in common, referred to as the Electronic System Level (ESL). The actual semantics expressed by the various types of descriptions do, however, vary a lot depending on different design philosophies and, more importantly, depending on what aspect of the design is given emphasis — at higher levels the descriptions and methods are, to a large degree, domain specific.

ESL models are useful as virtual prototypes for parallelizing hardware and software development or as intermediate conceptual models while developing a specification. The benefits of this include design exploration, evaluation and verification of the overall system already in an early design phase. The subsequent Register-Transfer-Level (RTL) design phase benefits, however, only little from available ESL descriptions. Even at the presence of ESL descriptions, the RTL is, with few exceptions, developed from natural language specifications. The ESL descriptions merely aid a system design phase resulting in these specifications. In a large majority of applications, the creation of system-level models therefore constitutes extra design and verification efforts that must be added to those required in a conventional design flow. In contrast to the abstraction layers below, i.e., RT level, gate level and transistor level, the developed ESL descriptions are not fully trusted. When verifying system-level functionality, therefore, verifying the ESL model is not sufficient. Instead, chip-wide simulations at the RTL must be performed. This is one of the main bottlenecks in current design flows and accounts for the largest portion of the overall verification costs. Further advances of digital systems will depend much on how

new design and verification methods cope with higher functional complexity, and in particular, how descriptions at higher abstraction levels can be integrated into trusted development steps.

High Level Synthesis

Synthesis from descriptions at abstraction levels higher than the RT level, *high level synthesis*, is a promising approach to raise the level of abstraction. It is a topic that has been given much attention already for many years during which the methods have matured and commercial tools have been made available.

The input languages of these tools are similar. They all accept algorithmic descriptions (in a tool-specific form) written in C or C++. Some tools even accept restricted forms of SystemC. Interface descriptions for specific protocols as well as optimization priorities can be specified within the tool environment. These methods alleviate the weaknesses of the common RTL hardware description languages, i.e., VHDL and Verilog, for describing complex datapath operations. Algorithmic descriptions can be formulated concisely, and from such descriptions the matured tools synthesize well optimized implementations. When combined with high level equivalence checking [34, 33, 12] it is also possible to ensure that the synthesized implementation is functionally correct.

High level synthesis is becoming established as an efficient method for creating strongly datapath-oriented designs such as in certain applications for signal processing. Many typical application areas for hardware are, however, control-oriented, i.e., they deal with complicated protocols and interfaces (and possibly fairly simple datapath operations).

When attempting to eliminate the verification bottleneck in SoC design it is not enough to aid the realization of single modules performing complex data manipulations. Instead, methods are required that aid the verification of systems with complex communication schemes. Current methods for high level synthesis fail to do so [21], since they require a bit- and cycle-accurate description of the control path. While these descriptions offer a concise and efficient representation of complex datapaths they do not offer much advantage for the verification of the overall system when compared with traditional RTL descriptions.

1.1.5 The Semantic Gap

The relationships between abstract circuit descriptions discussed in previous sections are summarized in Fig. 1.2. Note that the reasoning is always bottom-up. A new abstraction level can be used for design and verification only then when it has a well defined relationship with the lower abstraction levels. Otherwise, the semantics of the abstract model cannot be understood in terms of the physical circuit.

The basic idea for establishing a sound abstraction level is that the verification result of one abstraction level should be leveraged to establish confidence in a model at a higher abstraction level. Details verified for a sub-circuit by analysis at a lower abstraction level should therefore not have to be verified again at the next higher abstraction level.

Starting from the lowest level, the transistor level, verification of this level is used to achieve confidence in the elements of the next higher abstraction level, the gate level. After enough transistor-level simulations we are confident that that a certain transistor netlist correctly realizes the intended functionality of a Boolean operator or a binary storage element.

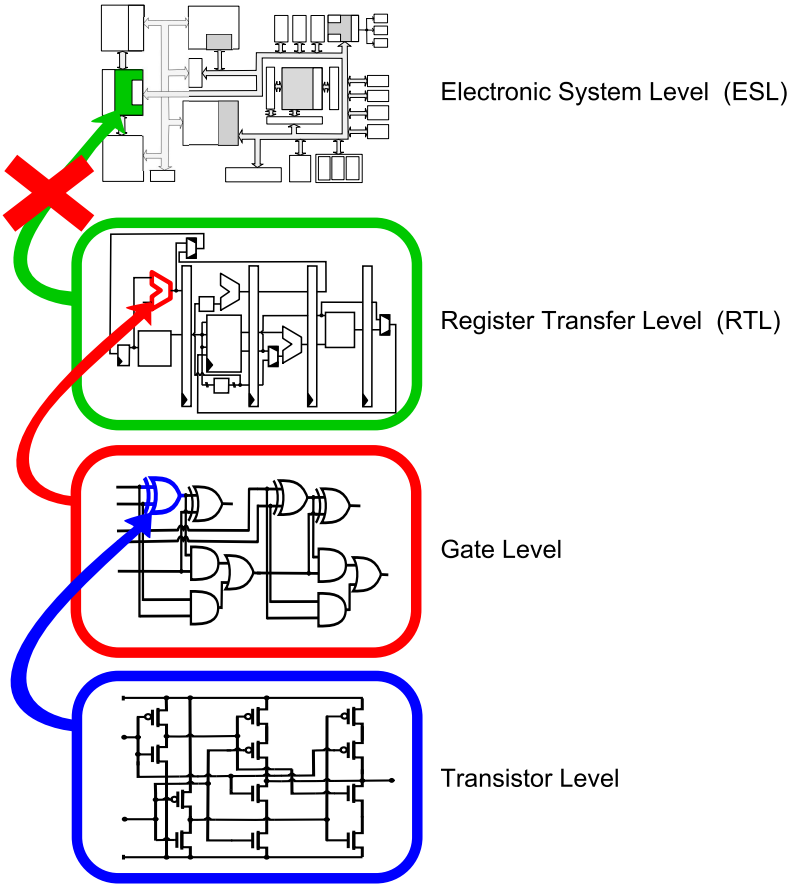


Figure 1.2: Abstraction levels for circuit descriptions

The next higher level, the RT level, also models the digital functional behavior, but at a level which corresponds well with the theoretical model of an FSM, i.e., it models the behavior only in terms of how registers and output change as a function of the input. The details as to how this change is realized as a combinational circuit which is present at the gate level is, to a large extent, abstracted away. The RT level also offers a much more compact representation in terms of register transfers, i.e., word-level information. This way of structuring the description aids a human understanding which is very important in a manual design process. A formal equivalence check ensures the correctness of the RT level with regards to the gate level and thereby also with the transistor level.

Descriptions which are more abstract than the RT level are referred to as ESL descriptions. This is, however, not a well defined level of abstraction. In general, it is most often used to characterize descriptions which are neither cycle- nor bit-accurate. Such high-level descriptions are of great value when the overall functionality of larger systems should be specified and understood. However, the chain of trust, built through the other abstraction levels, is not continued to descriptions at an electronic system level. The verification of RTL models can, with current methods, not be used to establish confidence in ESL models.

The problem is well known and often referred to as the “semantic gap” between the system level and the RT level. The name correctly implies that functional descriptions at such high levels of abstraction do not have well defined semantics at the RT level, and therefore also not as circuits. The reason is not simply a lack of standardization of such descriptions, but rather a theoretical problem of describing a sound relationship between RTL descriptions and these profoundly different descriptions which are neither cycle- nor bit-accurate.

In the view of a system engineer modules are large functional blocks typically modeled with event-driven communication. The overall behavior of a system is easily sketched out and analyzed in this way without worrying about synchronization and details of any specific communication protocol. Implementing such functionality at the RT level is a major and complicated refinement step. A full automation of the process is most likely not even wanted since engineering decisions are yet to be made, e.g., the area requirements of the circuit vs. the performance, which protocols to use, etc.

An equivalence relation, as used to describe the relationship between gate level and RT level, can hardly be used to cover such a large semantic gap. In the work of this thesis, therefore, a new formalism has been developed which is tailored for the purpose of describing such a relationship and thereby to close the semantic gap.

1.2 Verification in Industrial Practice

This section shortly summarizes the methods currently used for functional verification in industry. It will be used as background to discuss a practical employment of the new formal methods introduced in the sequel.

The sign-off verification of large commercial hardware developments is done at the RT level and, with few exceptions, simulation is the main or only used approach. Formal methods are gaining some acceptance, also in industrial practice, but are seldom considered more than a supplement in an otherwise simulation-based work flow.

1.2.1 Simulation

In RTL simulation a computer program, referred to as a simulator, is used to calculate and trace the behavior of an RTL model for a finite length of time when stimulated with a specific sequence of inputs. The resulting simulation trace can be analyzed against expected and/or specified design behavior.

It is common to use simulation in all parts of a design process, e.g., in the design phase where it produces useful feedback while developing the implementation. The discussion here, however, targets the simulation techniques used for comprehensive verification of the design in the verification phase. Note that the verification phase is distinguished from the design phase. In a typically design process it is also executed by a separate engineering team.

In practice, simulation-based verification is done by instantiating the design under verification in a "test bench", a behavioral description accepted by the simulator which is used both for generating stimuli for the design under verification and to aid the analysis of the resulting traces.

In a verification phase the design under verification is treated as a "black box", i.e., the verification is a pure stimulus-response analysis which relies on no or only little knowledge about internal implementation details. This black-box aspect is considered an important virtue of simulation-based approaches because it ensures that the verification is not influenced by specifics of the implementation but is rather based on an independent interpretation of the specification. Having such an independent interpretation is important, especially to identify faulty behavior where the specification is ambiguous or incomplete.

Exhaustive simulation is, for a design of any considerable size, not feasible. Quality of verification therefore depends greatly not only on the analysis methods themselves, but also on the simulation traces that have actually been analyzed. Good strategies are required to choose stimuli exposing potentially faulty behavior. The basis for all such strategies is generalization. An intuitive understanding of the design is used to create an implicit categorization of the behavior into a set of general behavioral scenarios. A subset of traces exposing each scenario is then analyzed, and the validity of the subset is considered representative for the validity of the scenario as a whole.

In an abstract design view it is natural to distinguish between control path and data path. This also explains the categorization of design behavior. The aim can be seen as creating a set of generalized behavioral scenarios which cover all modes and control operations. Simulation traces considered representative for each scenario are analyzed and compared with the expected behavior. This categorization into general scenarios is used for finding good heuristics to identify stimuli for the simulation. Note, however, that it is based on an informal view of the design which itself should be subjected to verification.

Two types of (unverified) assumptions are made when letting a set of simulation traces of fixed length represent the behavior of the design. Assumptions are made about temporal relationships, i.e., what previous behavior affects or, more importantly, does not affect, current behavior. Another type of assumptions are made when separating control path and data path. It is then assumed that a simulation trace with a specific set of data values can serve as representative for a whole range of data values.

For smaller designs, assumptions of the above discussed types can be predicted fairly well from the specification. This is, however, not the case for larger design blocks where composition and intermediate processing steps cannot be anticipated from its global input/output behavior.

To make up for the higher uncertainty it therefore becomes more urgent to simulate larger sets of data values and to analyze the temporal relations in longer simulation traces.

To efficiently simulate larger sets of data values for each scenario a method known as *constrained random simulation* is used. Data values are generated at random, under some constraint, and the analysis of the resulting behavior is automated by also defining the expected output value in terms of this randomly generated value.

Some particularly difficult bugs may still remain in corner cases of the design behavior, i.e., states of the design that can only be reached through specific rare input sequences. Constrained random simulation can be paired with long simulation traces to try to identify also such bugs. An enormous amount of simulation is, however, necessary to achieve a coverage where it can be expected that these potential cases are explored.

A bug in a corner case which is very difficult to reach through simulation may seem like a marginal problem. Such difficult bugs can, however, have critical consequences when they do occur, and in the realized circuit, which is orders of magnitude faster than simulation, they may even occur fairly frequently.

1.2.2 Coverage

An inherent problem to the non-exhaustive nature of simulation-based verification is to decide when enough simulations have been conducted to trust that the implemented behavior works as specified and/or expected. Measures for simulation coverage are used to aid this decision. Two measure types, *code coverage* and *functional coverage*, are in common use.

Code coverage is a simple measure to the portion of the RTL code that has been exercised during simulation, i.e., 100% code coverage means that all RTL statements have been executed by the simulator at least once. Unreachable code, for example due to generics, should be taken into consideration. Code coverage is a useful feedback in the verification process and demands practically no manual effort. As a determination criteria for the verification phase it is, however, of limited value since full code coverage does not imply that all, or even the essential, functionality of the design has been exhibited. Design behavior is typically described by a large set of parallel processes. The complexities of the parallel execution is not considered, but merely the execution of each individual statement. Also note that code coverage is a measure of the exhibited behavior rather than the actually analyzed behavior.

Functional coverage is used to mend the shortcomings of code coverage. An explicit categorization of functionality, referred to as the *coverage model*, is developed along with the test bench as a metric for measuring the functional coverage of a simulation. SystemVerilog [2] supports functional coverage with dedicated statements, but it can be implemented with other means also in other languages.

The coverage model is defined by a set of coverage points and coverage groups. Coverage points are defined in terms of ranges of data values on a set of monitored RTL signals, and coverage groups are hierarchies of such. Coverage points can be added by instrumenting the test bench, and, if this is not considered adequate, a strict black-box approach can be dropped and also the module under verification itself can be instrumented.

Functional coverage is a measure of exhibited combinations of coverage points and coverage groups in the simulation. The quality of functional coverage measurements obviously relies on the quality of the coverage model. Creating a good coverage model is, however, a challenging task and requires considerable manual effort. The model also undergoes many of

the same issues as discussed for the implicit categorization of test bench generation. Pitfalls of the implementation may therefore still not be be analyzed even when full functional coverage has been reached.

1.2.3 Assertions

To ameliorate the discussed problems of black-boxed simulation-based verification, internal information of the implementation considered important to the verification can be forwarded to the otherwise black-boxed verification using statements known as *assertions*. A basic syntax for assertions is part of both VHDL and Verilog. More expressive assertion languages are also available through the standards Property Specification Language (PSL) [1] and SystemVerilog Assertions (SVA) [2]. Both are widely supported by verification tools.

Note that assertions can be placed within the RTL description; they do, however, not describe behavior to be synthesized. Assertions are merely used to monitor certain conditions for verification purposes. The simulator evaluates and responds to these conditions during the simulation. Commonly, writing assertions is part of the design phase. When executing the simulations of the black-boxed verification phase, these assertions may then trigger and reveal problems, even in simulation traces where the unexpected behavior is not propagated to the output, i.e., otherwise not visible in the verification.

Instrumenting the RTL code with many assertions may, however, lower readability and be a serious distraction to the understanding of the actual synthesizable design behavior. A separation of the assertions from the rest of the RTL code has therefore been made possible in the modern assertions languages, i.e., PSL and SVA. These assertions, which are not anymore inserted into the code, are sometimes referred to as *properties*. The separation also enables a shift towards a work flow where such expressions are written as part of a verification phase rather than the design phase.

In recent years, tools using various model checking techniques (cf. Sec. 2.4) for an automatic generation of exhaustive mathematical proofs of assertions/properties, have seen some success in industry. The high computational complexity of the involved algorithms do, however, restrict the set of assertions which can be proven in practice. Such methods are therefore still used only as an addition to traditional simulation.

Formal methods used to prove the validity of assertions integrate seamlessly into a traditional simulation-based verification flow, and exhaustive proofs obviously increase confidence in the verification. However, the results of such verifications, also when augmented with formal proofs, are hard to evaluate since coverage cannot be well measured. Much effort is therefore made early in the design process to develop a verification plan — a plan specifying the verification methods to be used and, for these methods, concrete verification goals to be reached.

A formal solution to the coverage problem is discussed in Sec. 3 where a criterion, first stated in [9], is presented which ensures that a set of assertions/properties fully covers the input/output behavior of a design. An algorithm to check the criterion is implemented in the commercial formal verification tool Onespin 360 DV [44].

1.3 Motivation and Overview

Even after years of progress both in simulation-based and formal verification techniques, System-on-Chip (SoC) verification at the Register-Transfer-Level (RTL) has remained the main bottleneck in the design flow. While more and more sub-systems of increasing size become integrated on a single chip the level of abstraction for state-of-the-art verification techniques has remained unchanged. Even with the advent of new design methodologies based on Electronic System Level (ESL) descriptions, it is still the RTL that has remained the point of reference for creating the "golden" design model.

A comparison with verification practices at lower design levels may illustrate the problem: as a matter of course, when verifying the logic behavior of a large combinational circuit, we consider its gate-level description. Simulating its transistor-level description instead, in principle, is possible but would be considered highly inefficient and inappropriate. Similarly, when verifying the micro-architecture of an SoC design, we base this verification on its RTL description rather than its gate netlist. This is standard procedure, because we can trust the gate netlist and the RTL description: they are considered sound abstractions for the next lower levels.

By contrast, when evaluating the global system behavior of an SoC, this verification is done at the RTL. Verification of high-level models can be useful to build confidence in the specification at an early design phase, but it is not adequate to render a full-blown, system-wide RTL verification superfluous. We only trust the actual implementation of our SoC after chip wide simulations on the RTL have been running for weeks and months on large workstation clusters. Why do we not rely on verifying the global system behavior at the system level? Obviously, we do not trust our system-level models as much as we trust our RTL models and our gate-level models. The reason for this lack of confidence is well known and has often been addressed as the "semantic gap" between the system and the RT level. In fact, establishing a well-defined, formal relationship between these two levels is a difficult problem. It is one of the main hurdles when attempting to integrate system-level models into standard SoC design and verification flows.

The main objective of this thesis is to show how standard property checking techniques and commercially available tools can be used to close the semantic gap. The thesis contributes a theoretical basis as well as a practical methodology to describe a sound relationship between models at the RT level and models at the system level. The methodology can be used both "bottom-up", to create sound abstractions from already existing RTL descriptions, and "top-down", as part of a design flow where RTL descriptions are created from system-level descriptions in a guided refinement process that ensures the same soundness. We envision that, based on the proposed methodology, system-level verification techniques, e.g., based on SystemC [35, 53, 52, 28, 38, 14, 13] can be used to replace chip-level verification at the RTL in the future.

Sound abstraction based on a stack of sound functional models above the RTL has been investigated extensively in the context of formal verification by theorem proving, for example in [48, 39, 47, 30, 29, 7].

An important difference between these previous works and sound abstractions based on *path predicate abstraction* (PPA), as proposed in this thesis, lies in the fact that the methodology developed in our work is entirely based on standardized property languages such as System Verilog Assertions (SVA) and relies exclusively on fully automatic property checking using a bounded circuit model [16, 43]. This does not only support an intuitive formulation of de-

sign properties but also facilitates proof procedures that can handle industrial RTL designs of realistic complexity.

Not many other attempts has been made for creating sound abstract models based on standard property checking. A notable exception is [24]. Their work describes an approach from mathematical logic based on interpretation of theories to relate time-abstract system descriptions to cycle-accurate implementations. In contrast, the approach investigated in this thesis is based on the conventional formalisms of model checking such as Kripke structures. Moreover, the theoretical framework proposed here extends over [24] in that it is fully compositional. The proposed approach can handle the concurrency between RTL modules that are each represented by time-abstract models at the system level [58].

The thesis is structured as follows: Chapter 2 introduces relevant formalisms and summarizes the state of the art in formal verification techniques. Chapter 3 introduces a criterion to the *completeness* of a property set and an algorithm to check this criterion. It ensures that the properties of the set, in sum, describe the full behavior of the considered design. This very important attribute is required in the sound abstraction technique introduced in later chapters and is possessed by any property set generated in the proposed design flow.

Own contributions are presented from Chapter 4 and onwards. Chapter 4 presents the developed formalism of a Path Predicate Abstraction (PPA) and shows how it can be applied to an RTL implementation by proving a complete set of properties. It is an abstraction tailored to describe a sound relationship between a non-cycle-accurate and non-bit-accurate system model and an RT-level implementation. First it is defined for directed graphs; this definition eases an intuitive understanding and enables the comparison with other known abstraction techniques in Sec. 4.5.

In Sec. 4.2, the PPA definition is extended for FSMs before it is shown in Sec. 4.3 how C-IPC can be used to create such a PPA from the FSM of an RTL description. The chapter end in Sec. 4.4 where it is shown how the soundness of PPA guarantees a correspondence between important model checking properties of the abstract model and the concrete implementation.

Chapter 5 contributes important extensions to make path predicate abstraction compositional and a theorem for the soundness of a system composed of path predicate abstraction is postulated and proven.

Later chapters are devoted to technical applications of path predicate abstraction. In Chapter 6 it is shown, at the example of a telecommunication module, how a path predicate abstraction is created based on standard property checking on an RTL implementation. The chapter ends with experimental results reported for two case studies where this method was applied to industrial RTL designs. In Chapter 7 a novel design flow is presented where the processes of creating a path predicate abstraction is reversed. In this proposed design flow system-level models are methodologically refined while preserving the relationship of a path predicate abstraction through a set of automatically generated properties. The methodology has been tested in a student project which is reported at the end of the chapter. Note that the verification results obtained by analysis of the abstractions are, for both of the above methods, valid also for the RTL implementation and therefore also for the realized circuit. If such methodologies are followed it would render expensive chip-wide simulations at the RT level superfluous.

At last, in Chapter 8, it is discussed how the above techniques can be exploited for optimizations of the RT level, and for improved power estimates at the system level. A case study has been made where the full behavior of a SONET/SDH framer owned by Alcatel Lucent was re-designed and its energy consumption was compared with the original.

1.4 Publication List

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

1. URDAHL, J., STOFFEL, D., BORMANN, J., WEDLER, M., AND KUNZ, W. Path predicate abstraction by complete interval property checking. In *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)* (2010), pp. 207–215.

2. URDAHL, J., STOFFEL, D., WEDLER, M., AND KUNZ, W. System verification of concurrent RTL modules by compositional path predicate abstraction. In *Proceedings of the 49th Annual Design Automation Conference* (New York, NY, USA, 2012), DAC '12, ACM, pp. 334–343.

3. URDAHL, J., UDUPI, S., STOFFEL, D., AND KUNZ, W. Formal system-on-chip verification: An operation-based methodology and its perspectives in low power design. In *Proc. 23th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)* (2013).

4. URDAHL, J., STOFFEL, D., AND KUNZ, W. Path predicate abstraction for sound system-level models of RT-level circuit designs. *IEEE Transactions On Computer-Aided Design of Integrated Circuits and Systems (TCAD) 33*, 2 (Feb. 2014), 291–304.

5. URDAHL, J., STOFFEL, D., AND KUNZ, W. Architectural system modeling for correct-by-construction RTL design. In *Forum on Specification and Design Languages (FDL)* (2015).

Chapter 2

Reasoning in Circuits

2.1 Modeling Sequential Behavior

Formal reasoning requires a formal representation of the problem at hand. Boolean expressions can, as mentioned in Sec. 1.1, directly be understood as such a representation for combinational logic circuits.

The temporal relationships introduced by storage elements for sequential logic circuits can, however, not be described by Boolean logic. To represent such *sequential behavior* various formalisms for automata (also known as state machines and transition systems) are used. In these formalisms the values of the storage elements are represented as the *state* of the model. This section introduces the automaton formalisms used in the scope of this thesis.

An automaton is often visualized as a *state transition graph (STG)*, a directed graph with a vertex for each state, with an edge for each transition, and with a labeling dependent on the specifics of the automaton formalism under consideration. Terminology from graph theory is therefore used also in the context of automata: a *path* in an automaton is a sequence of states following the transitions of the automaton, an *initialized path* is a path starting from an initial state. Based on these terms we also define *reachable states*, these are states visited on an initialized path.

The computational complexity of the algorithms used for reasoning in circuits depends greatly on the *state space*, i.e., the number of states, of the automaton and on its *sequential depth*. The sequential depth is the length of the longest of the shortest finite initialized paths to any reachable state. In other words, if all finite initialized paths with lengths equal to the sequential depth are analyzed, then all reachable behavior of the automaton has been exhibited.

2.1.1 Finite State Machine

The digital behavior of an electronic circuit can be described by a discrete and deterministic Finite State Machine (FSM). The following definition is for a *Mealy-type* FSM, as used in this thesis. A *Moore-type* FSM only differs in its definition for the output function λ, which for Moore-type FSMs is a function of only the set of states, i.e., $\lambda : S \mapsto Y$.

Definition 1 [Finite State Machine]:
A deterministic Finite State Machine (FSM) is a 6-tuple $M = (S, I, X, Y, \delta, \lambda)$ with:

- a finite set of states S,
- a non-empty set of initial states $I \subseteq S$,

- an input alphabet X (a finite set of input values),
- an output alphabet Y (a finite set of output values),
- a transition function $\delta : S \times X \mapsto S$,
- and an output function $\lambda : S \times X \mapsto Y$.

□

Sequential circuit descriptions, e.g., RTL descriptions, can be interpreted as an FSM where the set of states, the input alphabet, and the output alphabet are encoded by vectors of Boolean values. The transition function, δ, and the output function, λ, are realized in the circuitry as Boolean operations on these vectors.

Definition 2 [Encoded FSM]:
An encoded FSM is an FSM, $M = (S, I, X, Y, \delta, \lambda)$, where:

- The state set S is an encoding over a vector V of Boolean variables referred to as *state variables*, $V = \langle v_1, v_2, \ldots, v_n \rangle$.
- The input alphabet X is an encoding over a vector of Boolean variables referred to as *inputs*, $X = \langle i_1, i_2, \ldots, i_m \rangle$.
- The output alphabet, Y, is an encoding over a vector of Boolean variables referred to as *outputs*, $Y = \langle o_1, o_2, \ldots, o_k \rangle$.
- The transition function $\delta = \langle \delta_1, \delta_2, \ldots, \delta_n \rangle$ is a vector of Boolean functions, where δ_j is a next state function for the state variable v_j.
- The output function $\lambda = \langle \lambda_1, \lambda_2, \ldots, \lambda_k \rangle$ is a vector of Boolean functions, where λ_j is the output function for the output o_j.

□

It follows that for an encoded FSM each unique value of the state vector, input vector and output vector correspond, respectively, to a state, an input symbol and an output symbol. Note that an encoded FSM will have 2^n states (and 2^m input symbols and 2^k output symbols). However, not all of these states are necessarily reachable from an initial state. Finding and representing the set of actually *reachable states* is one of the main concerns when applying formal methods.

2.1.2 Kripke Model

A Kripke model is an automaton formalism much used in computer science, e.g., to define the temporal logic languages introduced in Sec. 2.2.

Definition 3 [Kripke Model]:
A Kripke model is the quintuple $K = (S, I, R, A, L)$ with:

- a finite set of states S,
- a non-empty set of initial states $I \subseteq S$,
- a left-total transition relation $R \subseteq S \times S$,
- a set of Boolean atomic formulas A,
- and a valuation function $L : A \mapsto 2^S$. □

Note that no input or output is defined for a Kripke model. Instead, a valuation function (also referred to as a labeling function) is defined which gives each state a valuation (truth value) to the set of atomic formulas. In a state $s \in S$ the atomic formula $a \in A$ has the value *true* if $s \in L(a)$ and the value *false* if not. Two Kripke models are sequentially equivalent if any

of the initialized paths in one model produces a sequence of labels/valuations which can also be produced by an initialized path of the other, and vice versa.

A Kripke model can be derived from an FSM, and thereby also from an electronic circuit. Let $M = (S_M, I_M, X_M, Y_M, \delta_M, \lambda_M)$ be a Mealy-type FSM. Then, a derived Kripke model K has the following state transition behavior.

- Set of states: $S \subseteq S_M \times X_M \times Y_M$:
  $S = \{(s_M, x_M, y_M) \,|\, y_M = \lambda(s_M, x_M)\}$
- set of initial states: $I \subseteq I_M \times X_M \times Y_M$:
  $I = \{(s_M, x_M, y_M) \,|\, s_M \in I_M \wedge y_M = \lambda(s_M, x_M)\}$
- transition relation: $R = \{((s_M, x_M, y_M), (s'_M, x'_M, y'_M)) \,|$
  $s'_M = \delta_M(s_M, x_M) \wedge y'_M = \lambda_M(s_M, x_M)\}$.

Note that the deterministic behavior of an FSM, due to the next-state function δ, is reflected in the Kripke model by the fact that every state, (s_M, x_M, y_M), has only transitions to next states with a unique FSM state component s'_M; however, it has transitions to *all* states with that FSM state component s'_M, i.e., with any input component x'_M.

The atomic formulas A and the labeling function L of the derived Kripke model need to be chosen such that the properties we would like to prove on the model can actually be formulated. In principle, all states, inputs and outputs of the original FSM can be distinguished by the labeling function of the Kripke model and thus can be reasoned over.

Any automaton derived from an electronic circuit is encoded over the Boolean vectors representing state, input and output of the circuit. In the following we may refer to an *encoded Kripke model*; its definition follows from deriving a Kripke model from an encoded FSM (see Def. 2).

2.1.3 Labeled Transition System

In the field of process algebra labeled transition systems are commonly used automaton formalisms. Although all formalizations contributed in this work are in terms of FSMs or Kripke models, a brief introduction is in order as it is relevant as background for comparing related work.

Definition 4 [Labeled Transition System]:
A Labeled Transition System is the triple $(S, L, \rightarrow)$ with:

- a set of states S,
- a set of labels L,
- a left-total transition relation $\rightarrow \subseteq S \times L \times S$ □

Depending on context, some authors also include a set of initial states.

Opposed to a Kripke model, which has its labeling defined as a function of the states (by the valuation function), a Labeled Transition System has its labeling defined for the transitions (embedded in the transition relation $\rightarrow$). It is however shown in [22] that a mapping exists, both ways, between Kripke models and Labeled Transition Systems.

2.2 Temporal Logic

To describe and reason over logical properties for the sequential behavior of automata, temporal logic languages are used. In the scope of this thesis two commonly used temporal logic

languages are introduced, Computation Tree Logic [17], and Linear Temporal Logic [45].

Note that for industrial application, languages such as PSL (Property Specification Language) and SVA (SystemVerilog Assertions) have been standardized. These provide extended syntax in order to ease the writing of temporal logic in practice. Properties written in these languages can be mapped to the formal temporal logic languages presented here.

2.2.1 Computation Tree Logic

Computation Tree Logic (CTL) was first proposed in [17]. In CTL the usual Boolean logic operators are extended with the two modal operators A ("always") and E ("exists") which are used in combination with one of the *temporal operators*: X ("next"), G ("globally"), F ("finally"), U ("until"), W ("weak until"), to express logical properties quantified over time.

Definition 5 [CTL Syntax]:
The legal syntax of CTL is recursively defined by:

1. Every Boolean atomic formula, $a \in A$, is a CTL formula, ϕ.
2. If ϕ_1 and ϕ_2 are CTL formulas then also: $\neg\phi_1$, $\phi_1 \vee \phi_2$, $\mathsf{EX}\phi_1$, $\mathsf{EG}\phi_1$, $\mathsf{EF}\phi_1$, $\mathsf{E}(\phi_1\mathsf{U}\phi_2)$, $\mathsf{E}(\phi_1\mathsf{W}\phi_2)$, $\mathsf{AX}\phi_1$, $\mathsf{AG}\phi_1$, $\mathsf{AF}\phi_1$, $\mathsf{A}(\phi_1\mathsf{U}\phi_2)$, $\mathsf{A}(\phi_1\mathsf{W}\phi_2)$, are CTL formulas. ($\wedge$, *true* , *false* can be expressed by $\neg$ and $\vee$.) □

A CTL formula is evaluated on the states, $s \in S$, of a Kripke model. The semantics is defined based on the *computation tree* of s, an infinite tree structure obtained by an acyclic unfolding of the Kripke model starting from s. Note that the full expressive power of CTL can be formulated by a subset of the legal CTL syntax. In the following definition such a subset is established and the semantics of the remaining CTL syntax is described based on this subset.

Definition 6 [CTL Semantics]:
Let $M = (S, I, R, A, L)$ be a Kripke model, $s \in S$ be a state, $a \in A$ be a Boolean atomic formula, ϕ_1, and ϕ_2 be CTL formulas over A, and let $s \models \phi$ mean that the CTL formula ϕ is satisfied in the state s of the Kripke model M.

- $s \models a \iff s \in L(a)$
- $s \models \neg\phi_1 \iff s \not\models \phi_1$
- $s \models \phi_1 \vee \phi_2 \iff (s \models \phi_1)$ or $(s \models \phi_2)$
- $s \models \mathsf{EX}\phi_1 \iff$ there exists a state $s' \in S$ such that $(s, s') \in R$ and $s' \models \phi_1$
- $s_0 \models \mathsf{EG}\phi_1 \iff$ there exists an infinite path $(s_0, s_1, s_2, \ldots)$ such that for all $i \geq 0$, $s_i \models \phi_1$
- $s_0 \models \mathsf{E}(\phi_1\mathsf{U}\phi_2) \iff$ there exists an infinite path $(s_0, s_1, s_2, \ldots)$ and an $i \geq 0$ such that for all $0 \leq j < i$, $s_j \models \phi_1$ and $s_i \models \phi_2$
- $\mathsf{EF}\phi_1 \equiv \mathsf{E}(\mathit{true}\,\mathsf{U}\phi_1)$
- $\mathsf{E}(\phi_1\mathsf{W}\phi_2) \equiv \mathsf{E}(\phi_1\mathsf{U}\phi_2) \vee \mathsf{EG}\phi_1$
- $\mathsf{AF}\phi_1 \equiv \neg\mathsf{EG}\neg\phi_1$
- $\mathsf{AX}\phi_1 \equiv \neg\mathsf{EX}\neg\phi_1$
- $\mathsf{AG}\phi_1 \equiv \neg\mathsf{EF}\neg\phi_1$
- $\mathsf{A}(\phi_1\mathsf{U}\phi_2) \equiv \neg\mathsf{E}(\neg\phi_1\mathsf{U}\neg\phi_1 \wedge \neg\phi_2) \wedge \mathsf{AF}\phi_2$
- $\mathsf{A}(\phi_1\mathsf{W}\phi_2) \equiv \mathsf{A}(\phi_1\mathsf{U}\phi_2) \vee \mathsf{AG}\phi_1$ □

2.2.2 Linear Temporal Logic

Linear Temporal Logic (LTL) was first proposed in [45]. It defines the set of temporal operators: X ("next"), G ("globally"), F ("finally"), U ("until"), W ("weak until"), R ("release"), used in combination with the usual logical operators to formalize propositions quantified over time.

Definition 7 [LTL Syntax]:
The legal syntax of LTL is recursively defined by:

1. Every Boolean atomic formula, $a \in A$ is an LTL formula, ϕ.
2. If ϕ_1 and ϕ_2 are LTL formulas then: $\neg\phi_1$, $\phi_1 \vee \phi_2$, $\mathsf{X}\phi_1$, $\mathsf{G}\phi_1$, $\mathsf{F}\phi_1$, $(\phi_1 \mathsf{U} \phi_2)$, $(\phi_1 \mathsf{W} \phi_2)$, $(\phi_1 \mathsf{R} \phi_2)$ are also LTL formulas. ($\wedge$, *true* , *false* can be expressed by $\neg$ and $\vee$.) □

The semantics of LTL is defined for infinite paths. An LTL formula characterizes a set of paths (as opposed to a CTL formula which characterizes a set of states) in the considered Kripke model. Note that we may in the following say that an LTL formula holds for a state. This is merely a convenience and means that the LTL formula is satisfied by all paths starting in this state.

Definition 8 [LTL Semantics]:
For a considered Kripke model, let ϕ_1, and ϕ_2 be LTL formulas, $\pi_i = (s_i, s_{i+1}, \ldots)$ be an infinite path from s_i, a be a Boolean atomic formula, and let $\pi \models \phi$ mean that the LTL formula ϕ is satisfied by the path π.

- $\pi_i \models a \iff s_i \in L(a)$
- $\pi_i \models \neg\phi_1 \iff \pi_i \not\models \phi_1$
- $\pi_i \models \phi_1 \vee \phi_2 \iff (\pi_i \models \phi_1)$ or $(\pi_i \models \phi_2)$
- $\pi_i \models \mathsf{X}\phi_1 \iff \pi_{i+1} \models \phi_1$
- $\pi_i \models (\phi_1 \mathsf{U} \phi_2) \iff$ there exists $j \geq i$ such that $\pi_j \models \phi_2$ and for all $i \leq k < j$, $\pi_k \models \phi_1$
- $\mathsf{F}\phi_1 \equiv \mathit{true}\, \mathsf{U} \phi_1$
- $\mathsf{G}\phi_1 \equiv \neg(\mathsf{F}\neg\phi_1)$
- $(\phi_1 \mathsf{W} \phi_2) \equiv (\phi_1 \mathsf{U} \phi_2) \vee \mathsf{G}\phi_1$
- $(\phi_1 \mathsf{R} \phi_2) \equiv \phi_2 \mathsf{W} (\phi_2 \wedge \phi_1)$ □

2.2.3 CTL*

CTL* [23] is an extension of CTL using the same operators but without the requirement that one of the modal operators precede each temporal operator. Its expressive power is a superset of both CTL and LTL.

A formal definition of CTL* is not required in the scope of this thesis. It will only be used in the context of abstraction techniques to express what type of behavior is preserved between the concrete and abstracted model. For this purpose we consider also the universal and the existential fragments of CTL*, referred to as ACTL* and ECTL* respectively. These can be viewed as sub-languages of CTL*. As the name suggests ACTL* defines the fragment of CTL* where only the modal operator A is allowed while only the modal operator E is allowed in ECTL*. For (regular) CTL we may define the corresponding subsets. It is then implied that ACTL is a subset of ACTL* but not of ECTL* and that ECTL is a subset of ECTL* but not of ACTL*.

2.3 Automated Theorem Proving

Automated theorem proving is a subfield of mathematical logic concerning the automation of mathematical proofs. All of the techniques discussed for the formal verification of circuits can therefore be considered automated theorem proving. However, it is useful to distinguish methods based on general purpose *theorem provers*, which are computer programs able to assist a variety of traditional deductive mathematical proofs, from the reasoning techniques more specialized towards formal verification introduced in following sections.

A number of theorem provers are available, differing in their emphasized problem domains and input languages. Most of them are freely available academic developments, e.g., HOL4, Coq, and Isabelle. Complex mathematical theorems have successfully been formalized and proven using theorem provers, including a few previously open problems, e.g., Kepler's conjecture and the four color theorem [26].

A theorem prover requires a formalization of the problem at hand in the syntax of the tool's language. The computation of the proof is automated as part of an interactive process where axioms and reasoning techniques are specified.

Verifying digital systems based on general purpose theorem provers is difficult. It requires expertise for the proofs of classical mathematical theorems and, maybe more critically, to correctly formalize the verification problem in a representation accepted by the tool. The latter also constitutes a considerable verification gap since the proof is obtained for a model which cannot easily be related to the actual implementation.

Due to the complexity and due to the high effort involved the verification methods based on theorem provers have mostly been limited to academic case studies. Commercial implementations have been the subject of some of these studies. After the infamous Pentium FDIV bug [46] especially floating point units have been granted the attention of such rigid formal methods.

An industry setting requires, in general, more automation and support for the standardized description languages than what is offered by general purpose theorem provers. Formal verification tools used in industry today are therefore either tailored for specific niches, e.g., SLAM [4], or they make use of one of the fully automated *model checking* methods described in the following.

2.4 Model Checking

In model checking, also known as property checking, a sequential model's compliance with a temporal logic expression (cf. Sec. 2.2) is evaluated by formal and fully automatic methods. The methods are tailored for the verification of hardware and/or software and tools are available with front-ends for all common hardware description languages, e.g., VHDL and Verilog, as well as for a wide range of common software programming languages.

2.4.1 CTL Model Checking

The temporal logic language CTL was proposed in [17] together with an exhaustive proof algorithm of polynomial complexity with regards to the size of the state set.

In Sec. 2.2.1 we defined CTL and its semantics in terms of the states of a Kripke model. In model checking, however, we are not primarily interested in which states of a model satisfy a

temporal logic formula, but rather whether the formula is satisfied by the model itself.

To apply a CTL formula directly to a Kripke model the semantics of CTL is extended as follows. Let $M = (S, I, R, A, L)$ be a Kripke model, ϕ be a CTL formula, and $S_\phi \subseteq S$ be the set of states satisfying ϕ according to Def. 6; we then say that: $M \models \phi \iff I \subseteq S_\phi$. In other words, a CTL formula is valid for a Kripke model if it is valid for all of its initial states.

The following pseudo-code demonstrates how $\mathsf{EF}\phi$ can be calculated in a fixed-point iteration. Let $pre(R, Z)$ be a function returning the pre-image of a state set Z with respect to the transition relation R.

```
1  calcEF( M, φ ) {
2     Z  := ∅;
3     Z' := S_φ;
4     while (Z' ≠ Z) {
5        Z := Z';
6        Z'  := Z ∪ pre(R,Z);
7     }
8     return Z;
9  }
```

In the case that ϕ is a pure Boolean expression the state set S_ϕ is simply the set of all states where the expression is fulfilled. In all other cases S_ϕ must be calculated by recursive calls to the function implementing the corresponding operator. The implementation of the other operators can be implemented similarly [20].

In the following, for ease of notation, we let ϕ denote both, the CTL formula itself and the set of states, S_ϕ, it characterizes, when the meaning is clear from the context.

Symbolic Model Checking

The computational complexity of model checking is much dependent on how efficiently the pre-image can be calculated. In symbolic model checking [40] this performance is improved by creating a more compact representation of the transition relation. In symbolic model checking, instead of representing each transition explicitly, transitions between sets of states are represented using characteristic functions of the state sets.

As stated in Sec. 2.1, the state set S of automata obtained from functional descriptions is encoded over the vector V of state variables. In such an automaton a state is a unique Boolean assignment to V. Any state set $A_{set} \subseteq S$ can therefore be characterized by a Boolean function over the set of state variables in V.

2.4.2 LTL Model Checking

The semantics of LTL was defined in Sec. 2.2.2 for infinite paths of a Kripke model. For model checking we extend this semantics in order to evaluate LTL formulas directly on the Kripke model. An LTL formula is satisfied by a Kripke model iff the formula is satisfied for all initialized paths of the model.

Let $\sum_M$ be the set of initialized paths of the Kripke model M and let $\sum_\phi$ denote the set of paths characterized by an LTL formula ϕ. The LTL model checking problem is then defined by: $M \models \phi \iff \sum_M \subseteq \sum_\phi$ or equivalently as $\sum_M \cap \sum_{\neg\phi} = \emptyset$.

In the evaluation of the LTL model checking problem the set of paths $\sum_{\neg\phi}$ is characterized, not directly by the LTL formula $\neg\phi$, but by a Büchi automaton.

Definition 9 [Büchi Automaton]:
A Büchi automaton *BA* is a 4-tuple (S, I, δ, F) with:

- a finite set of states S,
- a set of initial states $I \subseteq S$,
- a transition relation $R \subseteq S \times S$,
- and a set of accepting states $F \subseteq S$. □

Temporal logic can be expressed in terms of Büchi automata, i.e., a Büchi automaton may be understood as a temporal logic expression. Its semantics is defined for infinite paths based on the set F of accepting states. An infinite path is said to be accepted iff accepted states $s \in F$ are visited infinitely often on the path.

For model checking, the product machine of the Kripke model M under consideration and a Büchi automaton *BA* characterizing the $\sum_{\neg\phi}$ is built. Any path of this product machine is a counterexample, i.e., if the product is empty the formula holds.

2.4.3 Bounded Model Checking

The number of states in a model generated from a circuit description (or for a software program) grows exponentially with the number of state bits. This "state space explosion" is a problem causing any algorithm requiring the representation of such models to be unfeasible for larger designs, e.g., CTL- and LTL-model checking. Bounded model checking [16] avoids this problem by mapping the model checking problem to a Boolean satisfiability (SAT) problem.

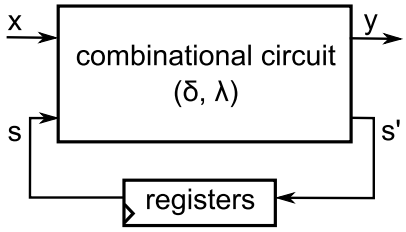


Figure 2.1: Sequential circuit model

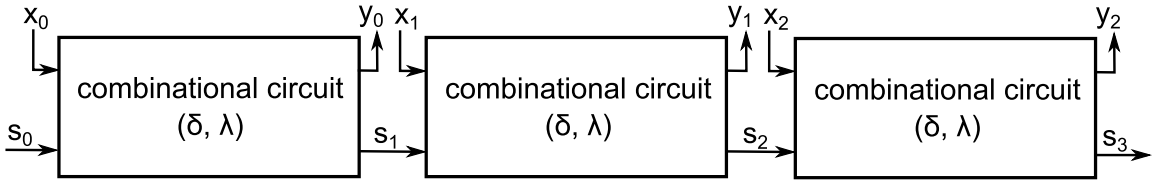


Figure 2.2: Bounded circuit model (for a bound of three clock cycles)

The original model of sequential behavior is commonly depicted as in Fig. 2.1. The box contains the combinational logic of the model which calculates the output and the next state based on input and current state. In discrete time intervals, at the occurrence of a clock event, the state is propagated.

In bounded model checking, properties are restricted to describe behavior over a bounded time interval, i.e., from a specified state (often the initial state) and over a certain finite number of clock periods. For such properties the original sequential model can be mapped to a Boolean

expression by "unrolling" it over a finite time window. This approach is illustrated in Fig. 2.2 for a time window of three clock cycles from the state s_0.

A bounded model is generated by copying the combinational logic (which is a Boolean expression) for as many clock cycles as the property should be proved for. The next state calculated by the combinational logic representing one cycle is then connected to the current state of the logic representing the successor cycle. In this way the entire bounded period is represented as combinational logic in which the state at a particular cycle is represented as a Boolean vector, s_t.

On this bounded model the property can be mapped to a Boolean satisfiability problem where the inputs (at each time point) are taken as free variables restricted only by the assumption of the property itself. The proof of the property succeeds if the negated property cannot be satisfied.

Proofs of bounded properties (or bounded proofs for general properties) can in some cases be useful, for example, as an instrument to find counterexamples. Verification, however, most often requires unbounded proofs, i.e., proofs that are globally valid and not restricted to a certain period of time. A brute force method to conquer this problem would be to ensure that the bounded model covers the entire reachable behavior. This can be done by unrolling from the initial state and for a number of periods equal to (or bigger than) the sequential depth of the original sequential model. However, for larger models this is not feasible due to the computational complexity of the resulting SAT problem and of the sequential depth calculation.

2.4.4 Interval Property Checking

Interval property checking (IPC) [43] is rooted in industrial developments of the 1990s. It is a SAT-based model checking technique producing globally valid unbounded proofs for a restricted type of temporal logic formulas known as *interval properties*.

An interval property is a temporal logic expression formulated in implication form and describing behavior over a finite time interval.

Definition 10 [Interval Property]:
An interval property ϕ is an LTL formula of the form $\mathsf{G}\ (A \rightarrow C)$ where both sub-formulas A and C, referred to as *assumption* and *commitment*, respectively, describe behavior over a finite time, i.e., the only temporal operator that may be used is the next operator, X. □

In practice interval properties are often used to specify "operations" of the model under consideration. To emphasize such use, we may in the following refer to interval properties synonymously as *operation properties*.

The computational model for interval property checking is, with one important exception, the same as for bounded model checking (cf. Sec 2.4.3). The exception is that in IPC no assumption is made on the starting state, i.e., the starting state is left as a free input in the SAT problem. Properties proven on such a model are therefore valid for any starting state. In other words, IPC gives unbounded proofs to properties describing behavior over finite intervals of time.

Fig. 2.3 illustrates the proof computation for a property, ϕ, expressed over an interval of three clock periods, i.e., an interval property where the next operator X is nested at most three times in the property. The starting state, s_i, and the input in each clock period, x_i, x_{i+1}, and x_{i+2}, are considered free input, only restricted by the assumption A of the property under consideration. In other words, any set of values for the starting state and for the input which is not

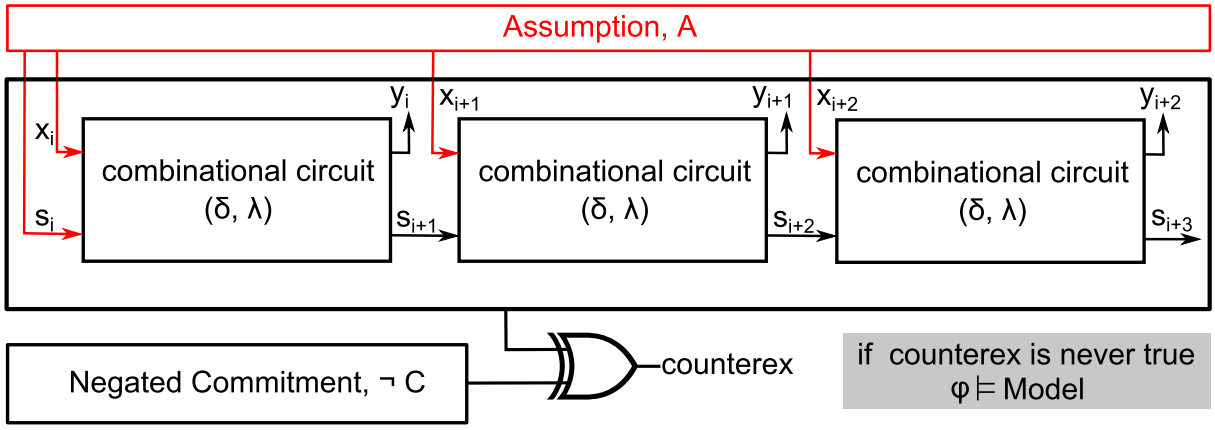


Figure 2.3: Proof computation for interval properties

in contradiction to the assumption will be considered. If any such set of values can fulfill the negation of the commitment *C*, then that set of values is a counterexample of the property. If no counterexample exists the property is valid for the model under consideration.

Modeling the starting state as a free input effectively means to consider all states as a possible starting state. This is clearly an over-approximation since not all states are necessarily reachable in the design. As a consequence, counterexamples may be found which are actually not reachable in the design; such counterexamples are referred to as "false negatives" or "spurious counterexamples". Note that the opposite, "false positives", cannot occur in IPC, because a valid property proof (a positive) implies the absence of counterexamples in all reachable cases (and possibly some unreachable). We therefore refer to the proof model of IPC as a *conservative* model.

Issues with false negatives are solved in this type of model checking by proving strengthened invariants for the design. This can be done manually by inspecting the counterexamples and formulating assertions for the design, or by automatic methods as proposed in [54].

To a large extent, however, false negatives can be avoided all along by formulating the properties such that they describe "operations" of the design. An operation is a set of sequences of register transfers with a common purpose. An example could be an instruction of a simple processor, or, for a protocol interface of some sort, the functionality caused at the occurrence of a specific bus request. The interdependence among such operations tend to be well manageable. Notice, for example, that the correct execution of an instruction is mostly independent of any previous behavior with the possible exception of datapath variables whose function is immediately related to the operation and whose reachable values at the time of the operation are therefore well known to the verification engineer. Unreachable values of other state variables are unlikely to break the functionality of the operation and therefore usually will not cause false negatives.

Note that the creation of properties can be viewed as a process where an abstract engineering view is formalized. In this context the approach has more resemblance with a design process than with a traditional simulation-based verification approach. From our experience, the overall functionality, even that of complex circuits, can be well understood in terms of operations. This view on formulating properties is common in industry and is well supported by prover technology tailored for this purpose such as in [43, 49, 44]. Establishing the validity of the individual operations therefore leads to high-quality verifications, also for the design as a whole.

2.4.5 *k*-step Induction

k-step induction [54] complements IPC as an alternative proof algorithm for interval properties (see Def. 10). It is a proof by mathematical induction where a bounded proof for k clock periods after reset is used as a base case to generate invariants. Interval properties causing false negatives when applying IPC may be successfully proven using *k*-step induction. The method is therefore useful to reduce manual efforts for generating explicit invariants.

The base case is a proof of $A \rightarrow C$ over the first k periods after reset, i.e., a proof obtained by bounded model checking with a bound of k (see Sec. 2.4.3). The induction step is a proven using IPC, with an additional assumption that over the last k periods $A \rightarrow C$ holds.

2.5 Abstraction Techniques

Exhaustive model checking approaches, i.e., CTL and LTL model checking, and several theorem proving algorithms require a complete analysis of the reachable state space. However, such analysis is often infeasible on the automaton models obtained directly from behavioral descriptions since their state space grows exponentially with the number of state variables, a problem known as state space explosion.

The abstraction techniques presented in this section are employed to create intermediate computational models with a reduced state space which are sound with respect to the problem at hand.

For improved readability of the following discussion we introduce L', a variation of the the labeling function of a Kripke model with the set of states as domain and with a subset of the atomic formulas as codomain. It is defined by $L'(s) = \{a \,|\, s \in L(a)\}$.

2.5.1 Bisimulation

Bisimilarity is a notion of behavioral equivalence for automata. Between bisimilar models [25] the μ-calculus, which is a superset of CTL* and thereby also of CTL and LTL, is preserved.

Definition 11 [Bisimulation Relation]:
Let K_P and K_Q be Kripke models over the same set of atomic formulas, i.e., $A_P = A_Q = A$. Then, a bisimulation relation is a binary relation $B \subseteq S_P \times S_Q$ such that $(s_P, s_Q) \in B$ if and only if:

- $L'_P(s_P) = L'_Q(s_Q)$
- if $(s_P, u_P) \in R_P$ then $\exists u_Q \in S_Q :: (s_Q, u_Q) \in R_Q \wedge (u_P, u_Q) \in B$
- if $(s_Q, u_Q) \in R_Q$ then $\exists u_P \in S_P :: (s_P, u_P) \in R_P \wedge (u_P, u_Q) \in B$ □

In the following we refer to s_P and s_Q as bisimilar states when $(s_P, s_Q) \in B$. The first requirement ensures that two bisimilar states fulfill the same subset of the atomic formulas A. The next two requirements together ensure that the transitions from bisimilar states match each other, i.e., the ending states of the transitions from the one bisimilar state are all bisimilar to one of the ending states of the transitions from the other.

For two Kripke models to be bisimilar, as stated in the following definition, we require for both models that every initial state of the one model is bisimilar with some initial state of the other model.

Definition 12 [Bisimilar Kripke models]:
Two Kripke models K_Q and K_P are bisimilar if a bisimulation relation B exists such that:

- $\forall s_P \in I_P : \quad \exists s_Q \in I_Q :: (s_P, s_Q) \in B$
- $\forall s_Q \in I_Q : \quad \exists s_P \in I_P :: (s_P, s_Q) \in B$

□

2.5.2 Stuttering Bisimulation

A stuttering bisimulation is a weakened form of a bisimulation first proposed in [11] where repetitions of the same label, called "stuttering", are ignored, i.e., only value changes are considered observable.

To support the definition of a stuttering bisimulation relation we define a relation of transitions modulo stuttering.

Definition 13 [Transition Relation modulo Stuttering]:
A transition relation modulo stuttering, R_{stutter}, is defined for a Kripke model K such that $(s_0, s_l) \in R_{\text{stutter}}$ if and only if a path, $(s_0, s_1, \ldots, s_l)$, exists in K such that all states, s_i with $0 \leq i \leq l$, visited along the path fulfill one of the following conditions:

1. $L'(s_i) = L'(s_0)$ for $i < l$ and $L'(s_l) \neq L'(s_0)$ and there exists a state, q, such that $(q, s_0) \in R$ with $L'(q) \neq L'(s_0)$.
2. $s_0 = s_l$ and all $L'(s_i) = L'(s_0)$ and there exists a state, w, such that $(w, s_0) \in R_{\text{stutter}}$.

□

The first condition ensures that a transition exists between two states of different labels if a path exists between the two states such that all but the ending state have the same label. Additionally it is required that the starting state is not part of the stuttering of all other longer paths.

The second condition ensures that the effect of cycles where all states have the same label are modeled. The last part of this requirement, i.e., the existence of a transition $(w, s_0) \in R_{\text{stutter}}$, prohibits that a self-transition is inserted in R_{stutter} for *every* state of a one-labeled loop. Instead, a self-transition is inserted only at each "entry point", w.

A stuttering bisimulation relation is defined analogously to the definition of a (strong) bisimulation relation in Sec. 2.5.1 by replacing R with R_{stutter}.

Definition 14 [Stuttering Bisimulation Relation]:
Let K_P and K_Q be Kripke models over the same set of atomic formulas, i.e., $A_P = A_Q = A$. Then, a stuttering bisimulation relation is a binary relation $B_{stutter} \subseteq S_P \times S_Q$ such that $(s_P, s_Q) \in B_{stutter}$ if and only if:

- $L'_P(s_P) = L'_Q(s_Q)$
- if $(s_P, u_P) \in R_{\text{stutter},P}$ then $\exists u_Q \in S_Q :: (s_Q, u_Q) \in R_{\text{stutter},Q} \wedge (u_P, u_Q) \in B_{stutter}$
- if $(s_Q, u_Q) \in R_{\text{stutter},Q}$ then $\exists u_P \in S_P :: (s_P, u_P) \in R_{\text{stutter},P} \wedge (u_P, u_Q) \in B_{stutter}$

□

Analogously to Def. 12, two Kripke models are stuttering-bisimilar if such a relation exists and all initial states of both models are part of the relation.

It is shown in [11] that the temporal logic language CTL* (which includes CTL and LTL), with exception of the next operator X, is preserved between stuttering-bisimilar models.

2.5.3 Bisimulation modulo Silent Actions

In [41] a special version of a Labeled Transition System, a Transition System with silent actions, is considered. The set of labels is extended with the "silent action", τ (in [41] labels are referred to as actions). Transitions labeled with τ are considered non-observable. A weak bisimulation relation is then proposed in [41] which considers equivalence only in terms of the observable labels.

For the purpose of comparison, the definition is here adapted for a Kripke model. The special silent action, τ, is introduced in the Kripke model by extending the domain of the valuation function to $L : (A \cup \tau) \mapsto 2^S$. We will refer to states of the state set $L(\tau)$ as silent states and we create a weak bisimulation relation which ensures an equivalence of the non-silent states.

With a few important exceptions the definition follows along the lines of the definition for a stuttering bisimulation (cf. Sec. 2.5.2). To support the definition of the weakened bisimulation relation we define a relation of transitions modulo silent actions.

Definition 15 [Transition Relation modulo Silent Actions]:
A transition relation modulo silent actions R_{silent} is defined for a Kripke model K such that $(s_0, s_l) \in R_{\text{silent}}$ if and only if a path, $(s_0, s_1, \ldots, s_l)$, exists in K for which all states, s_i with $0 \leq i \leq l$, visited along the path fulfill one of the following conditions:

1. $s_0 \notin L(\tau)$ and $s_l \notin L(\tau)$ and $s_i \in L(\tau)$ for $0 < i < l$
2. $s_0 = s_l$ and $s_i \in L(\tau)$ for $0 \leq i \leq l$ and there exists a path, $(q, \ldots, s_0)$, in K such that $q \notin L(\tau)$
3. $s_0 \notin L(\tau)$ and $s_i \in L(\tau)$ for $0 < i \leq l$ and $(s_l, s_l) \in R_{\text{silent}}$ □

The first condition ensures that a transition exists between two non-silent states in R_{silent} if a path exists in K between the two states if all intermediate states (if any) are silent states. The second and third condition both only concern the modeling of "silent cycles", i.e., a cycle where only silent states are visited.

Note that if a silent cycle exists in K then condition (1) will add a transition between all non-silent states reachable through this cycle to R_{silent} and (2) will add a silent cycle to R_{silent}, reached from a non-silent by a transition added in (3), which cannot be exited, i.e., it models the possibility to stay silent forever.

A silent action bisimulation relation is defined analogously to the definition of a (strong) bisimulation relation in Sec. 2.5.1 by replacing R with R_{silent}.

Definition 16 [Silent Action Bisimulation Relation]:
Let K_P and K_Q be Kripke models over the same set of atomic formulas, i.e., $A_P = A_Q = A$. Then, a silent action bisimulation relation is a binary relation $B_{silent} \subseteq S_P \times S_Q$ such that $(s_P, s_Q) \in B_{silent}$ if and only if:

- $L'_P(s_P) = L'_Q(s_Q)$
- if $(s_P, u_P) \in R_{\text{silent},P}$ then $\exists u_Q \in S_Q :: (s_Q, u_Q) \in R_{\text{silent},Q} \wedge (u_P, u_Q) \in B_{silent}$
- if $(s_Q, u_Q) \in R_{\text{silent},Q}$ then $\exists u_P \in S_P :: (s_P, u_P) \in R_{\text{silent},P} \wedge (u_P, u_Q) \in B_{silent}$

□

Analogously to Def. 12, two Kripke models are silent-action-bisimilar when such a relation exists and all initial states of both models are part of the relation.

The observable behavior is preserved between silent-action-bisimilar models. Temporal logic formulas proven on one model are therefore valid also for the other model (when X is interpreted as the next observable label).

2.5.4 Localization Reduction

Localization reduction [37] is a computationally tractable and fully automatic abstraction technique for encoded automata where the state set is reduced by ignoring a subset of the state variables. Opposed to a bisimulation, or any of its weakened forms, localization reduction does not preserve the observable behavior. However, it does create an abstract computational model which is "conservative" with respect to a specific property under consideration. Localization reduction is therefore used as an internal computational step to reduce the state space before model checking.

Given a Kripke model K with a set of states S encoded by the vector of state variables V an abstract Kripke model $\hat{K}$ is derived by localization reduction such that the abstract state set $\hat{S}$ is an encoding over a vector $\hat{V}$ of a subset of the state variables in V. The abstraction function, $h : S \mapsto \hat{S}$, projects a concrete state $s \in S$ to the abstract state $\hat{s} \in \hat{S}$ with the same encoding for the state variables in vector $\hat{V}$.

The abstract transition relation $\hat{R}$ is defined in one of two ways depending on the property under consideration. For a safety property, i.e., a property expressed in CTL in the form $\mathsf{AG}p$, *existential abstraction* is used such that $(\hat{s},\hat{s}') \in \hat{R} \Leftrightarrow \exists s,s' :: (s,s') \in R$ and $h(s) = \hat{s}$ and $h(s') = \hat{s}'$. An abstract transition is created between two abstract states if there exists some (concrete) transition between any two concrete states projected by h to the corresponding abstract states. Note that this is an "over-approximation" of the possible behavior.

In [19] it is shown that existential abstraction is sound for the entire universal fragment of CTL*, referred to as ACTL*, i.e., ϕ, $\hat{K} \models \phi \implies K \models \phi$. Note, however, that the opposite does not hold, $K \models \phi \not\Longrightarrow \hat{K} \models \phi$. Model checking on the abstract model may therefore result in spurious counterexamples. Methods to resolve such spurious counterexamples have been investigated, for example in [18].

A *universal abstraction* is the symmetric counterpart of an existential abstraction. It creates a model which is conservative with regards to the existential fragment of CTL*, referred to as ECTL*. The transition relation $\hat{R}$ is defined by $(\hat{s},\hat{s}') \in \hat{R} \Leftrightarrow \forall s \exists s' :: h(s) = \hat{s}$ and $h(s') = \hat{s}'$ and $(s,s') \in R$. An abstract transition is created if for all concrete states projected by h to the corresponding abstract starting state a concrete transition exists to a state projected to the abstract ending state.

2.5.5 Predicate Abstraction

In predicate abstraction [27] a set of Boolean propositions over the concrete state set is used as state variables to encode an abstract state set.

Let P be a set of Boolean propositions over the set of atomic formulas A of a Kripke model, i.e., for each state $s \in S$ of the Kripke model a truth value can be defined for every $p \in P$. Let V be a vector of the elements in P. It defines a partitioning of S such that each value of V characterizes the set of states with the corresponding truth values. Predicate abstraction defines an abstract state set such that each value of V is represented by a unique state. The abstraction function h simply maps a concrete state to the abstract state representing the value for V that characterizes it.

The abstract transition relation $\hat{R}$ is created as an existential abstraction or as an universal abstraction (cf. Sec. 2.5.4) such that the resulting abstract model becomes a conservative model with respect to the property under consideration.

Chapter 3

Complete Interval Property Checking

In Sec. 1.2.2 coverage measures were discussed in the context of simulation based methods. In the context of formal methods the situation is somewhat different. Clearly, a formal proof of a property implies that it is valid for the design behavior for all allowed input stimuli. However, a measure reflecting what portion of the design behavior is actually described by the proven properties is still needed.

In this chapter an absolute cover measure is presented. It is a formal criterion for the *completeness* of a set of interval properties and was first developed in [9, 8]; later a similar result was obtained independently in [15]. For a property set fulfilling this criterion it is ensured that the set of properties, in sum, completely describes the output behavior of the design in terms of the design's input.

This chapter briefly summarizes Complete Interval Property Checking in the terminology and notations of this thesis. For a more comprehensive and illustrated treatment of this technique the reader may refer to [8]. The chapter is structured as follows: First we will introduce some important notions related to completeness. These notions are also important prerequisites for the formalization of path predicate abstraction — the main theoretical contribution of this thesis. In Sec. 3.2 the completeness criterion will be formally defined, before an algorithm to check the fulfillment of this criterion is presented in Sec. 3.3. The completeness check, in practice, is computationally tractable even for large designs and it is commercially available [44].

3.1 Terminology

To characterize behavior over a finite time we introduce the notion of a *sequence predicate*.

Definition 17 [Sequence Predicate]:
A sequence predicate is an LTL formula where the only temporal operator used is the next operator, X. □

Note that the assumption A and the commitment C of an interval property are examples of sequence predicates.

For ease of notation we also define a generalized next operator.

Definition 18 [Generalized Next Operator]:
The generalized next operator denotes a finite nesting of the next operator, X. Let ϕ be an LTL formula. The generalized next operator X^k is defined by: $\mathsf{X}^0(\phi) := \phi$, and $\mathsf{X}^i(\phi) := \mathsf{X}(\mathsf{X}^{i-1}(\phi))$ when $i > 0$. □

The completeness check is closely linked with a method of formulating properties that describe design behavior in terms of *operations*. The basic idea is that a set of properties is complete if every input sequence and every output sequence of the *finite state machine (FSM)* under verification is fully described by a sequence of operations. An operation describes finite sequences of behavior of the model under consideration, i.e., a set of path segments in the considered Kripke model. For the following discussion, operations are formalized by means of interval properties augmented with a length.

Definition 19 [Operation]:
An operation O is a set of finite path segments of length l in a Kripke model K characterized by the pair (P,l) of an interval property, $P := \mathsf{G}(A \rightarrow C)$, with $P \models K$, and the operation length l. A path segment, $(s_0,\ldots,s_l)$, is element of O if and only if for any path $\pi = (s_0,\ldots,s_l,\ldots)$ it holds that $\pi \models A$. □

In other words, the pair (P,l) characterizes the path segments containing l transitions (between $l+1$ states) which are prefixes of one or more paths on which the assumption A (and, hence, the commitment C) holds. The pair (P,l) can be understood as the *specification* of an operation. For simplicity of terminology we may refer to both, (P,l), and O, as "operation" when it is clear from the context whether the specification or the actual design behavior is meant.

In most practical cases, l is simply the length of the finite behavior associated with an operation, i.e., the number of transitions needed to check an assumption A and to produce an output sequence fulfilling the commitment C of the property. In certain cases, however, a shorter value for l needs to be chosen. For example, in pipelined designs, the computed results may be only visible at the output several cycles after the operation was issued. A new operation may already be in progress while the results of an earlier operation are still being produced at the outputs. The length l is chosen equal to the time between the start of the current operation, O, and the start of the next one, as will be explained below when considering chains of operations.

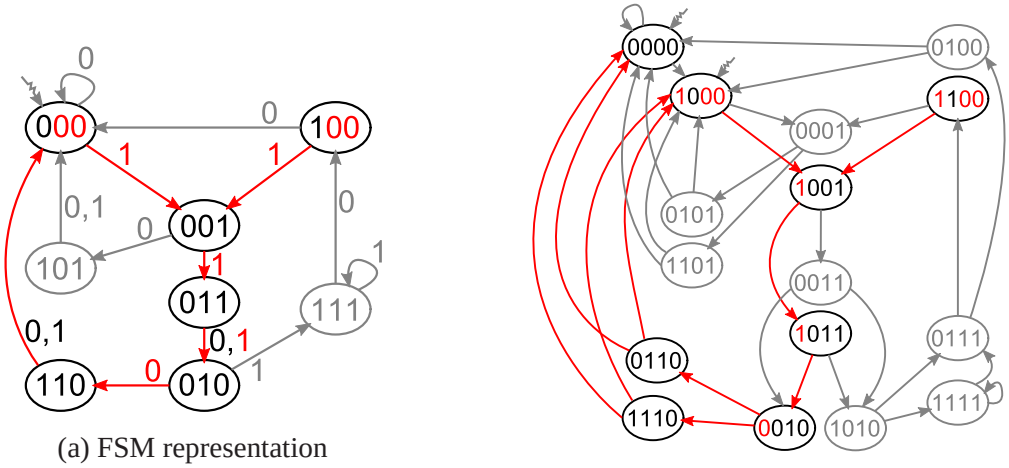
Example 3.1 An example operation can be created for the model in Fig. 3.1 in such a way that all inputs, outputs and FSM states can be distinguished through the labeling of the Kripke model.

We consider the operation $(\mathtt{op},5)$ characterized by the property op and the length $l = 5$ for the model of Fig. 3.1. Let $\mathtt{op} := G(A \rightarrow C)$ where $A := \neg b \wedge \neg c \wedge i \wedge \mathsf{X}^1(i) \wedge \mathsf{X}^2(i) \wedge \mathsf{X}^3(\neg i)$ and $C := \mathsf{X}^1(\neg a \wedge \neg b \wedge c) \wedge \mathsf{X}^2(\neg a \wedge b \wedge c) \wedge \mathsf{X}^3(\neg a \wedge b \wedge \neg c) \wedge \mathsf{X}^4(a \wedge b \wedge \neg c)$.

The reader may verify by inspection of Fig. 3.1 that the property op holds. We can identify all paths satisfying A, i.e., all paths with a prefix matching the state codes $(\mathtt{1x00}, \mathtt{1xxx}, \mathtt{1xxx}, \mathtt{0xxx})$. In the model, the paths with prefix $(1000, 1001, 1011, 0010)$ and the paths with prefix $(1100, 1001, 1011, 0010)$ fulfill this condition. On these paths also the commitment C, given by $(\mathtt{xxxx}, \mathtt{x001}, \mathtt{x011}, \mathtt{x010}, \mathtt{x110})$, is fulfilled and, hence, the interval property op holds.

The operation op comprises the following set of path segments, each of which are prefixes with length $l = 5$ of paths fulfilling A:

```
{(1000,1001,1011,0010,0110,0000), (1000,1001,1011,0010,0110,1000),
 (1000,1001,1011,0010,1110,0000), (1000,1001,1011,0010,1110,1000),
 (1100,1001,1011,0010,0110,0000), (1100,1001,1011,0010,0110,1000),
 (1100,1001,1011,0010,1110,0000), (1100,1001,1011,0010,1110,1000)}
```



(a) FSM representation
State Encoding: (abc)
Input i on edges

(b) Kripke model representation
State Encoding: $(iabc)$

Figure 3.1: Example model to illustrate formalisms

The check of completeness (to be introduced in Sec. 3.3) for a set of operations relies on a sequencing of the operations. For a complete set it is implied that the end of any operation is the start of another. The state sets reachable at the end of an operation, i.e., the set of all ending states in the characterized path segments, are therefore of special interest and will be referred to as *important states*.

Definition 20 [Important Ending State]:
The set of important ending states of an operation O of length l is the set of states $\{s \mid \exists (s_0, s_1, \ldots, s_l) \in O : s = s_l\}$. □

In other words, the set of important ending states is the set of states reachable at the end of an operation. For example, in the operation $(\mathtt{op}, 5)$, the important ending states are the states 0000 and 1000.

Definition 21 [Important State]:
The *important states* in a set of operations is the union of all *important ending states* of all operations in the set. □

By fulfilling the completeness criterion, as stated in the following section, it is ensured that for any operation, all important ending states of that operation are also a starting state for at least one of its successor operations.

In practice a complete set of operation properties is created by identifying important modes of the system and by partitioning the design behavior into operations transitioning between these modes. Modes can be associated with important states in our formalism and can be specified as macros in terms of expressions over RTL state variables. Operation properties can then be written by referring to these macros. The starting mode is specified in the assumption A and the ending mode in the commitment C of the property. Following such a procedure, completeness can be ensured by writing the properties in such a way that the ending mode of all properties is the starting mode of a set of "successor" properties and that this set of "successor" properties in sum covers all possible input scenarios and describes the resulting output sequences.

3.2 Completeness Criterion

For a property set to be complete the properties are required to describe the output sequences according to certain *determination requirements*. The determination requirements are a specification of which signals in the design need to be *determined* and under what circumstances.

A signal is determined at a specific time point if and only if its value is described through the property set as a function of the inputs at current and earlier time points. (The expression for determining a signal's value must be in terms of inputs and/or of other determined signal values.)

As an example of a determination requirement, consider a data bus with a valid flag. In the absence of the valid flag, "datavalid", the value of the data bus signals are irrelevant; the determination requirement for the data bus signals, "data", could therefore be written as "if (datavalid = true) then determined(data)". Assuming that the valid flag is determined unconditionally, the determination requirement specifies a set of time points for all the operation properties in the set considered for completeness, for which the value of the data bus signals should be determined.

In general, a determination requirement is a pair (s, σ_s) for a signal s (ex.: data) and a condition σ_s defining when the signal s is to be determined. The determination requirement is fulfilled if the signal s is determined for any given time point characterized by σ_s.

Definition 22 [Complete Property Set]:
A property set $V = \{P_1, P_2, \ldots, P_n\}$ is *complete* if two arbitrary finite state machines satisfying all properties in the set are sequentially equivalent in the signals specified in the determination requirements at the time points characterized by the guards of the determination requirements. □

In our theory the properties reason about the labeling of a Kripke model. A labeling must therefore be chosen such that the signals can be identified. An obvious choice would be to let each RTL signal be represented by its own Boolean atomic formula. (A vector of signals, e.g., the above data bus, would be represented by a set of atomic formulas, one for each bit.)

3.3 Completeness Check

The completeness of a property set can be checked formally by a proof algorithm which is experienced to be computationally feasible in industrial practice. It is a proof by induction starting from the initial states (in practice defined by reset). Beginning from an initial state, it is proven that every input sequence drives the design through a sequence of operations, each of which is described by an operation property. The idea is that each operation begins in a "mode" determined by the history of the system, i.e., by the previous inputs since reset. In practice, a mode is described in terms of expressions over the state variables of the system; this corresponds to a state predicate in our formalisms.

The special reset property is a base case; it determines the state assumed by the system in a finite number of cycles after reset. Together, all other operation properties of the complete property suite constitute the inductive step. It is then ensured that for any input sequence received in any ending state of a previous operation, an operation property exists which determines an ending state of the current operation. Checking that an operation ends in a determined state amounts to checking whether the operation is a *function* of only inputs and its starting state.

In *Complete Interval Property Checking (C-IPC)* this basic idea is implemented in a collection of tests that can be checked automatically on the set of properties. It is important to note that the checks are performed solely on the property suite — the design is not taken into account. The checks reason exclusively on signals and behavior as given by the properties.

The user needs to specify what signals are inputs and what signals should be determined according to determination requirements mentioned above. In addition to the outputs, also state variables can be declared as determined. Signals may be expressed in terms of state variables within the operation property only if that state variable itself is declared to be determined at the time point of reference. Typically the number of state variables to be declared as determined is small since most state information is implicit to a single operation, i.e., the output can be described in terms of input sequences within the temporal scope of the operation without any explicit reference to state variables.

The user is also required to specify a sequencing of operations. This is done by specifying a *property graph* $G = (V,E)$ where the nodes $V = \{P_i\}$ are the operation properties and the edges describe their sequencing. There is an edge $(P_j,P_k) \in E$ if the operation specified by (P_k,l_k) can take place immediately after the operation specified by (P_j,l_j). (This is the case if operation (P_k,l_k) starts in a state that is reached by operation (P_j,l_j).) Note that, in principle, the property graph could be determined automatically from the set of operations. However, in case the property suite is incomplete or incorrect, an automatic completeness checker can give useful debugging information if the user has supplied an *intended* property graph which, anyways, involves only a small extra effort.

The proof of completeness for a set of properties is obtained by conducting four checks that are each performed on the property graph G: a *case split test*, a *successor test*, a *determination test* and a *reset test*, all described below.

Example 3.1 Continued A complete set of properties is created for the model shown in Fig. 3.1. to demonstrate how the individual tests for establishing completeness are applied. The example should also provide an intuitive understanding how the four tests together ensure the overall completeness of the property set.

As determination requirement we list the atomic formula c unconditionally, i.e., our only determination requirement is (c, true). Atomic formulas a and b are considered internal state only and do not need to be determined. We consider the set of operations $\{(\texttt{long},5)$, $(\texttt{short},3)$, $(\texttt{idle},1)$, $(\texttt{wrong},4)$, $(\texttt{readErr},1)$,, $(\texttt{keepErr},1)$, $(\texttt{reset},0)\}$, where the interval properties are defined by:

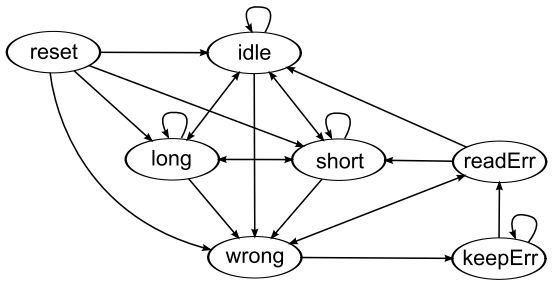


Figure 3.2: Property Graph

$$
\begin{array}{rl}
A_{\texttt{long}} := & \neg b \wedge \neg c \wedge i \wedge \mathsf{X}^1(i) \wedge \mathsf{X}^3(\neg i) \\
C_{\texttt{long}} := & \mathsf{X}^1(c) \wedge \mathsf{X}^2(c) \wedge \mathsf{X}^3(\neg c) \wedge \mathsf{X}^4(\neg c) \wedge \mathsf{X}^5(\neg b \wedge \neg c) \\
A_{\texttt{short}} := & \neg b \wedge \neg c \wedge i \wedge \mathsf{X}^1(\neg i) \\
C_{\texttt{short}} := & \mathsf{X}^1(c) \wedge \mathsf{X}^2(c) \wedge \mathsf{X}^3(\neg b \wedge \neg c) \\
A_{\texttt{idle}} := & \neg b \wedge \neg c \wedge \neg i \\
C_{\texttt{idle}} := & \mathsf{X}^1(\neg b \wedge \neg c) \\
A_{\texttt{wrong}} := & \neg b \wedge \neg c \wedge i \wedge \mathsf{X}^1(i) \wedge \mathsf{X}^3(i) \\
C_{\texttt{wrong}} := & \mathsf{X}^1(c) \wedge \mathsf{X}^2(c) \wedge \mathsf{X}^3(\neg c) \wedge \mathsf{X}^4(a \wedge b \wedge c) \\
A_{\texttt{readErr}} := & a \wedge b \wedge c \wedge \neg i \\
C_{\texttt{readErr}} := & \mathsf{X}^1(\neg b \wedge \neg c) \\
A_{\texttt{keepErr}} := & a \wedge b \wedge c \wedge i \\
C_{\texttt{keepErr}} := & \mathsf{X}^1(a \wedge b \wedge c) \\
A_{\texttt{reset}} := & \texttt{reset} \\
C_{\texttt{reset}} := & \neg b \wedge \neg c
\end{array}
$$

Table 3.1: Operation properties for example model

3.3.1 Case Split Test

The case split test checks that for any input sequence there exists a chain of operation properties such that the assumption of each property in the chain is fulfilled. In other words for an arbitrary sequence of inputs there will be a sequence of operations which are triggered, one after the other.

This can be ensured by checking, for every operation (P, l_P), whether the disjunction of the assumptions $\{A_{Q_j}\}$ of all properties Q_j of successor operations covers the commitment C_P of P after l_P transitions. This implies that, if these properties are all satisfied by the considered model, then for every path starting in an important ending state of (P, l_P) there exists an operation property Q_j whose assumption A_{Q_j} describes the path. Let $\{A_{Q_1}, A_{Q_2}, \ldots\}$ be the set of assumptions of the successor operations, then the case split test checks if

$$C_P \Rightarrow \mathsf{X}^{l_P}(A_{Q_1} \vee A_{Q_2} \vee \ldots)$$

The assumptions A_{Q_i} of the properties Q_i are shifted in time to the end of property P so that the first state of A_{Q_i} coincides with the last state of C_P.

Example 3.1 Continued Consider the case split test applied to operation $(\texttt{wrong}, 4)$. The successor properties of this operation are defined by the property graph of Fig. 3.2 to be `readErr` and `keepErr`. For `wrong` the case split test checks that the commitment $C_{\texttt{wrong}}$ implies that either the assumption $A_{\texttt{readErr}}$ or the assumption $A_{\texttt{keepErr}}$ holds at the end of the operation. That is, $\mathsf{X}^1(c) \wedge \mathsf{X}^2(c) \wedge \mathsf{X}^3(\neg c) \wedge \mathsf{X}^4(a \wedge b \wedge c) \Rightarrow \mathsf{X}^4((a \wedge b \wedge c \wedge \neg i) \vee (a \wedge b \wedge c \wedge i))$. Clearly, the case split test holds for $(\texttt{wrong}, 4)$.

If the case split test succeeds (for all operations), this means that for every possible input trace of the system there exists a chain of operations that is executed. However, this chain may not be uniquely determined. Therefore, the following successor test is performed.

3.3.2 Successor Test

The successor test checks whether the execution of an operation (Q, l_Q) is completely determined by every predecessor operation (P, l_P). For every predecessor/successor pair $(P, Q) \in E$ it is checked whether the assumption A_Q of operation property Q depends solely on inputs and

on signals determined by the predecessor P. This is checked in a SAT instance created in the following way. The set of signals mentioned in the properties P and Q are duplicated. The first set of signals is used to describe executions of an operation (P, l_P) followed by operation (Q, l_Q). The second set describes operation (P, l_P) followed by an operation not being (Q, l_Q). In both executions, the same input sequences are applied. Also, the state variables that are assumed to be determined are given the same values in both executions (in the time points specified by the guards of the determination requirements.) Let A'_P, C'_P and A'_Q be the assumption and commitment of property P and the assumption of property Q, respectively, expressed in the copied signals. Further, let $D = \bigwedge d_i$ be the set of determination requirements expressed as a conjunction of terms d_i. Each d_i represents one requirement (s, σ_s) expressed as $d_i = (\sigma_s \wedge \sigma'_s \Rightarrow s = s')$. Intuitively, this expression states that whenever the guard of signal s is true in one of the executions the signal s must have the same value in both executions.

The successor test checks the following implication on the SAT instance (with the same input values in each time frame):

$$A_P \wedge C_P \wedge A'_P \wedge C'_P \wedge D \wedge \mathsf{X}^{l_P}(A_Q) \Rightarrow \mathsf{X}^{l_P}(A'_Q)$$

Example 3.1 Continued Consider the predecessor/successor pair (`wrong`, `keepErr`) in the example. For this pair the successor check becomes: $A_{\texttt{wrong}} \wedge C_{\texttt{wrong}} \wedge A'_{\texttt{wrong}} \wedge C'_{\texttt{wrong}} \wedge (c \Leftrightarrow c') \wedge \mathsf{X}^4(A_{\texttt{keepErr}}) \Rightarrow \mathsf{X}^4(A'_{\texttt{keepErr}})$.

Since the commitment of `wrong` fully specifies the FSM state at X^4 this formula clearly holds for the example.

For further illustration of the test, consider a modification of `wrong` in which the state at X^4 was specified only for b and c. If we also added a successor property to the property graph such that the case $a = 0$ was part of its assumption then the case split test for `wrong` would still hold. However, since a would not be determined at the end of `wrong` it would not be possible to know if the new property, or one of the two original successor properties would trigger at X^4. The successor test would fail and would correctly flag such a set of properties as incomplete.

If this implication does not hold, then there exists an input sequence such that operation property P is executed and the assumption of operation property Q may hold or may not hold, depending on the other signals mentioned in the properties. This is the case if the assumption A_Q was written such that it depends on some state variables other than inputs and variables determined by P. If the implication does hold, then the assumption A_Q uniquely determines for any input sequence applied after completion of operation (P, l_P) whether operation (Q, l_Q) will follow or not (hence the name, "successor test".)

Having established that there exists a unique chain of operations for every input trace it remains to be shown that these operations determine the output signals as stated in the determination requirements.

3.3.3 Determination Test

The determination test checks whether each operation (Q, l_Q) fulfills its determination requirements provided the predecessor operation (P, l_P), in turn, fulfilled its determination requirements. The test creates a SAT instance that is satisfied if a determination requirement is violated, i.e., if a variable required to be determined by the operation (Q, l_Q) is actually not a

function of the variables determined by (P, l_P) and/or of inputs during the operation (Q, l_Q). Similar to the successor test, the set of signals mentioned in the operation properties P and Q is duplicated in order to describe two executions. In both executions, Q is followed by P and the same input sequences are applied. Again, also, the state variables that are assumed to be determined are given the same values in both executions in the time points specified by the guards of the determination requirements. Let D_P and D_Q be the determination requirements of property P and Q, respectively. The determination test checks the following implication on the SAT instance (with the same input sequences applied in both executions):

$$A_P \wedge C_P \wedge A'_P \wedge C'_P \wedge D_P \Rightarrow \mathsf{X}^{l_P}(D_Q)$$

If this implication does not hold, then there exists an input sequence and sequences of other signals mentioned in the operation properties such that Q is executed after P but signals that are supposed to be determined may have different values in different executions.

Example 3.1 Continued In our example we only have a single and unconditional determination requirement — c should be determined at any time, i.e, $D_P = (c \Leftrightarrow c')$. In our rudimentary example the value of c is explicit at all "time points" of all operations. The determination test clearly holds for the example set.

In more realistic examples the signals to be determined may be vectors whose values are expressed by functions dependent on input and/or other signals at earlier time points. In such cases the determination test will be non-trivial.

3.3.4 Reset Test

The three tests explained above prove the hypothesis of an inductive proof that states the following: assume that an operation $(P, l_P) \in V$ has uniquely determined its ending state then an operation $(Q, l_Q) \in V$ exists that uniquely determines the ending state as well as the output sequence of (Q, l_Q) solely from the ending state of (P, l_P) and from the input sequence applied during the operation (Q, l_Q). The inductive proof is rooted at the reset state.

A *reset property* is a special property describing the behavior after reset, i.e., the assumption contains only the reset condition. The *reset test* checks whether reset can always be applied deterministically and whether reset fulfills all determination requirements. It is constructed similarly to the above tests, however for the reset property and without any predecessor property.

Example 3.1 Continued The reset operation $(\texttt{reset}, 0)$ specifies the state after reset without referring to the state before the reset condition, hence, it guarantees that reset can be applied deterministically. Also the determination requirement is clearly fulfilled; $C_{\texttt{reset}} := \neg b \wedge \neg c$ determines the value of c from the fulfillment of the reset condition until one of the successor operations determines its value (in the next clock cycle).

Chapter 4

Path Predicate Abstraction

Two RTL designs are sequentially equivalent, if their input/output behavior is the same clock cycle by clock cycle. Such a notion of equivalence is, however, too strict to describe the relationship between a system model and its corresponding implementation at the RT level.

The degree of refinement when making the transition from a system model to an RTL model is considerable. At system level the functional intent of a system is described as a network of modules. Individual modules are thereby only describing the details required to specify their obligations to the system. At the RT level this functionality is given a cycle- and bit-accurate realization. Such a refinement typically involves encoding the system-level objects over a number of clock cycles.

In our experience, the major part of the complexity in RTL descriptions originates from such sequentially defined objects being processed in parallel by functionality distributed over several sub-FSMs. In practice, this is realized as several (concurrent) VHDL or Verilog processes communicating without the overhead of any explicit synchronization mechanisms. This distributed implementation style is used to describe parallelism and is needed to achieve the wanted processing efficiency.

This thesis contributes *path predicate abstraction (PPA)*, a formal abstraction technique tailored to describing the relationships between time-abstracted system-level models and their cycle-accurate implementation at the RT level.

For a pair of an abstract model and a concrete model in such a relationship, the formalization guarantees a specific soundness, i.e., properties of the one model will be reflected by corresponding properties in the other. This soundness can be exploited to prove the validity of properties of the concrete implementation based on model checking on the greatly simplified abstract model. As a result, it becomes possible to verify properties of the circuit which are far out of scope for automatic reasoning approaches operating directly on the RTL model.

Industry standard property checking techniques, as they are available today, are sufficient to formulate a methodology to ensure the PPA relationship. Two methodologies have been explored and will be presented in later chapters, one for creating a path predicate abstraction "bottom-up", from an already existing RTL implementation, and one for establishing it from a system-level description as part of a "top-down" design flow.

This chapter is structured as follows: In Sec. 4.1 *operational coloring* is defined for directed graphs and it is shown how an abstraction with a specific soundness is derived from these graphs based on such a coloring. In [56] it was shown that this PPA definition can be fulfilled for

the state transition graph of a Kripke model derived from an RTL design by creating a three-component coloring, with components based on input sequences, output sequences and FSM states, induced through C-IPC.

A slight variation of this approach, based on an extended PPA definition for finite state machines, is taken in this thesis to better clarify how the input/output alphabet can be changed and to better support an intuitive understanding of the formalism when applied to RTL descriptions. This is presented in Sec. 4.2. Then, in Sec. 4.3, it is shown how an operational coloring can be created on an FSM based on C-IPC.

Further, in Sec. 4.4, it is shown how temporal logic properties are translated soundly between the concrete model and its PPA, allowing properties to be proven for a complex concrete model based on its much simpler path predicate abstraction.

Finally, in Sec. 4.5, the technique is compared against other known abstraction relations, in particular the ones discussed in Sec. 2.5.

4.1 Path Predicate Abstraction for Directed Graphs

Path predicate abstraction is defined based on an *operational coloring* of the concrete model. In order to provide a good intuition of how an operational coloring can create a sound abstraction, this section will formulate the basic idea for directed graphs. In later sections, this formalization will be extended from graphs to FSMs so that the notion of path predicate abstraction will become applicable to practical RTL designs.

Definition 23 [Operational Graph Coloring]:
Consider a directed graph $G = (V,E)$, a subset $W \subseteq V$ of the graph vertices called *colored* nodes, a set of colors $\hat{W} = \{\hat{w}_1, \hat{w}_2, \ldots\}$ and a surjective coloring function $\mathtt{f_C} : W \mapsto \hat{W}$. A path segment $(v_0, v_1, \ldots, v_n)$ such that $v_0, v_n \in W$ and $v_1, \ldots, v_{n-1} \in V \backslash W$ is called *operational path segment* in G. The set W must be chosen and colored such that:

1. every cyclic path in G contains at least one node from W (no cycles with only uncolored nodes in the graph),
2. for every operational path segment $(v_0, v_1, \ldots, v_n)$ and $u_0 \in W$ such that $\mathtt{f_C}(u_0) = \mathtt{f_C}(v_0)$ there must exist an operational path segment $(u_0, u_1, \ldots, u_m)$ in G with $\mathtt{f_C}(u_m) = \mathtt{f_C}(v_n)$

We call $\mathtt{f_C}$ an *operational coloring function* and G an *operationally colored* graph. □

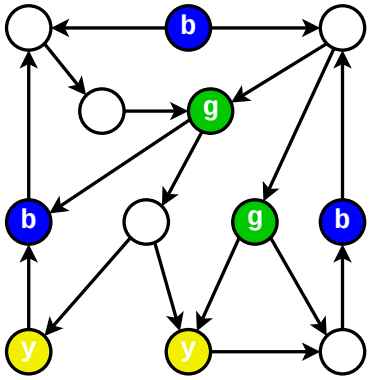


Figure 4.1: Example of an operational coloring

Fig. 4.1 shows an example of an operationally colored graph. The blue (b), green (g) and

yellow (y) vertices are elements of $W \subseteq V$. The uncolored (shown here as white) nodes do not belong to W. As can be seen, the first condition of Def. 23 is fulfilled as there is no cyclic path along only uncolored nodes. Also the second condition can be checked easily. For example, from *every* green node there are operational path segments to blue nodes as well as to yellow nodes, but there are no operational path segments to any other color. Intuitively, the graph describes the "operations": green→yellow, green→blue, blue→green, yellow→blue.

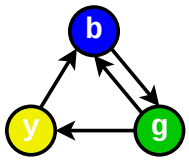


Figure 4.2: Path predicate abstraction for the operational coloring in Fig 4.1

For the operationally colored graph of Fig. 4.1, we can draw an abstract graph as shown in Fig. 4.2. It has only a single node for each color. Every operational path segment in the original graph is represented by an edge in the abstract graph. Such an abstraction is called a *path predicate abstraction*.

Definition 24 [Graph Path Predicate Abstraction]:
We consider a graph $G = (V,E)$ with a set of colored nodes $W \subseteq V$, a set of colors $\hat{W}$ and an operational coloring function $\mathtt{f_C} : W \mapsto \hat{W}$.

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

The following soundness theorem connects sets of paths in the concrete graph with paths in the abstract graph in terms of *color sequences*.

Definition 25 [Color Sequence]:
Consider a graph $G = (V,E)$ with a set of colored nodes $W \subseteq V$, a set of colors $\hat{W}$ and an operational coloring function $\mathtt{f_C} : W \mapsto \hat{W}$. A *color sequence* produced on a path $(s_0,s_1,s_2,\ldots)$ in G is the sequence of colors $(\hat{w}_0,\hat{w}_1,\ldots)$ of the colored nodes on the path, i.e., the i-th color in the sequence is $\hat{w}_i = \mathtt{f_C}(s_j)$ if and only if $s_j \in W$ is the i-th colored node on the path. (Note: it is $i \leq j$ since uncolored nodes $s \notin W$ on the path do not contribute to the color sequence.) □

Theorem 1 [Soundness w.r.t. color sequences]:
Let $G = (V,E)$ be a graph and $\hat{G}$ be the path predicate abstraction induced by an operational graph coloring $\mathtt{f_C} : W \mapsto \hat{W}$ with a set of colored nodes $W \subseteq V$.

(1) Given an arbitrary finite (infinite) concrete path $(v_0,v_1,v_2,\ldots)$ in G with $v_0 \in W$. There exists a finite (infinite) abstract path $(\hat{w}_0,\hat{w}_1,\ldots)$ in $\hat{G}$ that represents the color sequence produced on the concrete path.

(2) Given an arbitrary finite (infinite) abstract path $(\hat{w}_0,\hat{w}_1,\ldots)$ in $\hat{G}$. For every node $v_0 \in V$ in G such that $\mathtt{f_C}(v_0) = \hat{w}_0$ there exists a finite (infinite) concrete path $(v_0,v_1,v_2,\ldots)$ such that the color sequence produced on that path is $(\hat{w}_0,\hat{w}_1,\ldots)$. □

Proof. (1) follows immediately from the construction: every colored node $v \in W$ is mapped to the color $\hat{w} = \mathtt{f_C}(v)$ by the coloring function.

(2) We consider the edges traversed on the abstract path. According to requirement 2 of Def. 23, for an edge $(\hat{w}_i,\hat{w}_{i+1})$ there exists an operational path segment from *every* colored node v_i with $\mathtt{f_C}(v_i) = \hat{w}_i$ to some colored node v_j with $\mathtt{f_C}(v_j) = \hat{w}_{i+1}$ along uncolored nodes

$v_{i,1},\ldots,v_{i,l-1}$. Hence, for any abstract path we can find a correspondingly colored concrete path by concatenating the concrete path fragments. Again, according to requirement 2 of Def. 23, such a path can be found for *every* starting node v_0 colored with $\mathtt{f_C}(v_0)=\hat{w}_0$. □

The theorem states that a path predicate abstraction is sound with respect to sequences of colors on paths. This is easy to see in the example of Fig. 4.1 and Fig. 4.2. In other words, every chain of operations in the concrete graph has its correspondence in the abstract graph, and vice versa.

In the following, we will extend our notions from graphs to FSMs and Kripke models that can be modeled by labeled STGs. Similarly like the abstract graph of Fig. 4.2 represents the same color sequences as the concrete graph in Fig. 4.1, we will consider abstract FSMs and Kripke models that preserve sequences of labels in their STGs when compared with their concrete counterparts. This can be used to establish specific soundness properties of the abstracted models.

When considering FSMs derived from RTL descriptions, as will be shown, our problem of finding an operational graph coloring, and thus a path predicate abstraction, can be formulated as the task of specifying and proving a complete set of interval properties.

4.2 Path Predicate Abstraction for FSMs

In this section path predicate abstraction (PPA) is defined for FSMs based on an *operationally colored FSM*. The definition of an operationally colored FSM extends over that for a directed graph and has an individual coloring for states, input sequences and output sequences. The states are colored according to the requirements of Def. 23 where the graph is the state transition graph (STG) of the FSM. Additionally a coloring will be defined to characterize the I/O sequences observed between colored states, i.e., on *operational path segments* (cf. Def. 23).

A deterministic FSM has a state transition function and an output function defined for the pair of current state and current input. It follows that any path and any output sequence of the FSM is uniquely determined for a specified starting state and input sequence. In other words, starting from a state $s_0 \in S$, a sequence of k input symbols $\rho_k=(x_0,\ldots,x_{k-1})$ makes the FSM transition along the path segment $(s_0,\ldots,s_k)$ where $s_i=\delta(s_{i-1},x_{i-1})$ for $i>0$, making it output the sequence of k output symbols, $\gamma_k=(y_1,\ldots,y_k)$ where $y_i=\lambda(s_{i-1},x_{i-1})$. We say, the sequence ρ_k *causes* the FSM to follow the taken path segment. The FSM, in turn, *produces* the resulting output sequence γ_k.

In the following, we consider an FSM with an STG that has been operationally colored according Def. 23. Then, every path from a colored state has a finite prefix which is an operational path segment. It follows that any (infinite) input sequence has a finite prefix that from any colored state *causes* the FSM to follow an operational path segment.

Definition 26 [Operational Input Prefix]:
Let $s \in W$ be a colored FSM state, and let ρ be an infinite input sequence. The *operational input prefix* of $\rho=(x_0,x_1,\ldots)$ for the colored state $s_0 \in W$ is the finite prefix $(x_0,\ldots,x_{l-1})$ of ρ *causing* an operational path segment from s_0. □

For an FSM with an operationally colored STG it follows that all input sequences have an operational prefix for all colored states. To simplify notations below we let the operational path segment from a colored state $s \in W$ *caused* by the operational input prefix of an input sequence

ρ be denoted by $\mathtt{opath}(s,\rho)$.

To aid the following discussion we also define *color-equivalent* operational path segments.

Definition 27 [Color-Equivalent Operational Path Segments]:
Let $\mathtt{f_C}$ be an operational coloring function according to Def. 23, applied to the STG of an FSM, coloring a subset W of the set of states S with colors from the set of colors $\hat{W}$. Also let $\pi_n = (s_n,\ldots,s_{n+k})$ and $\pi_m = (s_m,\ldots,s_{m+l})$ be two operational path segments, i.e., $s_n, s_{n+k} \in W$ and $s_i \notin W$ for $n < i < n+k$ and $s_m, s_{m+l} \in W$ and $s_j \notin W$ for $m < j < m+l$. Then, the two operational path segments are *color-equivalent*, denoted $\pi_n \overset{\text{color}}{=} \pi_m$, iff both start in equally colored states and end in equally colored states, i.e., iff $\mathtt{f_C}(s_n) = \mathtt{f_C}(s_m)$ and $\mathtt{f_C}(s_{n+k}) = \mathtt{f_C}(s_{m+l})$. □

In other words, adopting the terminology of Sec. 4.1, color-equivalent operational path segments are path segments that belong to the same "operation".

Definition 28 [Operational FSM Coloring]:
Let $\{\rho_i\}$ denote the set of all input sequences and $\{\gamma_i\}$ the set of all output sequences that can be produced by the FSM. Then, an FSM, $M = (S, I, X, Y, \delta, \lambda)$, is operationally colored with a set of *state colors*, $\hat{S}$, a set of *input colors*, $\hat{X}$, and a set of *output colors*, $\hat{Y}$, iff all of the following requirements are fulfilled:

1. An operational graph coloring according to Def. 23 is defined on the state transition graph of the FSM such that a function, $\mathtt{f_{SC}}$, assigns a *state color* $\hat{s} \in \hat{S}$ to a subset W of the states S of the FSM. (Compared to Def. 23, V becomes S, $\mathtt{f_C}$ becomes $\mathtt{f_{SC}}$ and $\hat{W}$ becomes $\hat{S}$.)
2. All initial states must be colored states and have the same state color, i.e., $s \in I \implies s \in W \wedge \mathtt{f_{SC}}(s) = \hat{s}_{reset}$.
3. A surjective function, $\mathtt{f_{XC}} : W \times \{\rho_i\} \mapsto \hat{X}$, exists which maps all pairs, (s,ρ), of a colored state $s \in W$ and an input sequence $\rho \in \{\rho_i\}$, to an input color $\hat{x} \in \hat{X}$, such that:

$$\begin{aligned} &\forall s_a, s_b \in W, \quad \forall \rho_a, \rho_b \in \{\rho_i\} \quad : \\ (\mathtt{f_{SC}}(s_a) = \mathtt{f_{SC}}(s_b) &\implies \mathtt{f_{XC}}(s_a,\rho_a) = \mathtt{f_{XC}}(s_b,\rho_a)) \quad \wedge \\ (\mathtt{f_{SC}}(s_a) = \mathtt{f_{SC}}(s_b) \quad \wedge \quad &\mathtt{opath}(s_a,\rho_a) \overset{\text{color}}{\neq} \mathtt{opath}(s_b,\rho_b) \\ &\implies \mathtt{f_{XC}}(s_a,\rho_a) \neq \mathtt{f_{XC}}(s_b,\rho_b)) \end{aligned}$$

4. A surjective function, $\mathtt{f_{YC}} : W \times \{\gamma_i\} \mapsto \hat{Y}$, exists which maps all pairs, (s,γ), of a colored state $s \in W$ and an output sequence $\gamma \in \{\gamma_i\}$, to an output color $\hat{y} \in \hat{Y}$. Let γ_i denote the output sequence produced by the input sequence ρ_i when applied to s_i, then, for an operational coloring, $\mathtt{f_{YC}}$ must fulfill:

$$\begin{aligned} &\forall s_a, s_b \in W, \quad \forall \rho_a, \rho_b \in \{\rho_i\} \quad : \\ \mathtt{f_{SC}}(s_a) = \mathtt{f_{SC}}(s_b) \quad &\wedge \quad \mathtt{f_{XC}}(s_a,\rho_a) = \mathtt{f_{XC}}(s_b,\rho_b) \\ &\implies \mathtt{f_{YC}}(s_a,\gamma_a) = \mathtt{f_{YC}}(s_b,\gamma_b) \end{aligned}$$

□

The above definition makes sure that the entire reachable behavior is described by sequences of state colors for which the FSM produces sequences of output colors, and that this "color behavior" is a function of the sequences of input colors.

Requirement 1 and requirement 2 together ensure that any initialized path is split into a sequence of operational path segments and thereby describes a unique sequence of state colors.

This means that a coloring of states exists such that all cyclic paths include at least one colored state, and that if a path exists from a state colored "blue" to a state colored "green" (without visiting any other colored states in between) then such a path exists from all states colored "blue" to some state colored "green".

Requirement 3 extends the definition for input sequences. The first part ensures that a single input sequence is given one and the same color for all states of the same state color it is applied to. The second part of requirement 3 ensures that any two input sequences which from states of the same state color cause operational path segments ending in states of different state color are given different input colors. In other words, it ensures that if an operational path segment exists from a state colored "blue" to a state colored "green" which is caused by an input sequence colored "red", then for all states colored "blue" in combination with all input sequences colored "red" a path is followed such that the next colored state is "green". (It is not forbidden that an input sequence with a different color may also cause an operational path segment from "blue" to "green".)

Note that the color of the input sequence is uniquely defined only with respect to the state color of the starting state. In other words, we allow the same input sequence to be mapped to different input colors when used for different starting state colors. This construction reflects how prefixes of the same sequence can be given a context-dependent meaning in the concrete implementation. This flexibility will be required in Chapter 5 in order to make path predicate abstraction compositional.

Finally, requirement 4 for an operationally colored FSM ensures that the color of the output sequence produced by any operational path segment is a function of the current state color and the color of the input sequence.

All requirements together now ensure that if from *some* state colored "blue" *some* input sequence colored "red" causes an operational path to a state colored "green" and produces an output sequence colored "yellow" then for *all* states colored "blue" in combination with *any* input sequence colored "red" the next colored state is "green" and the produced output sequence is colored "yellow".

For an operationally colored FSM a path predicate abstraction can be created where the states correspond to state colors, the input symbols to input colors, and the output symbols to output colors. To support the definition we first define a *color transition relation* and an *output color relation*. It will be shown that these relations are, in fact, functions, and they will be used to define the transition function and the output function of the FSM path predicate abstraction.

Definition 29 [Color Transition Relation]:
For an FSM, $M = (S, I, X, Y, \delta, \lambda)$, operationally colored by $(\hat{S}, \hat{X}, \hat{Y})$ the *color transition relation* is a relation $R \subseteq \hat{S} \times \hat{X} \times \hat{S}$ where: $(\hat{s}, \hat{x}, \hat{s}') \in R$ iff an *operational path segment*, $\pi = (s_n, \ldots, s_{n+l})$, exists such that $\mathtt{f_{SC}}(s_n) = \hat{s}$ and $\mathtt{f_{SC}}(s_{n+l}) = \hat{s}'$, which is *caused* by the operational input prefix of ρ from s_n such that $\mathtt{f_{XC}}(s_n, \rho) = \hat{x}$. □

From Def. 28-3 it follows that any two color-equivalent operational paths are triggered by input sequences of equal input color. As a consequence, the color transition relation is right-unique. The relation is, however, not necessarily left-total since it is not guaranteed that all input colors are used for all colored states. This is a result of the flexibility introduced with Def. 28-3 which, in effect, allows a separate input alphabet for each state color. It follows that the relation is a partial function $\hat{\delta} : \hat{S} \times \hat{X} \mapsto \hat{S}$. In the following, we refer to it as the *color transition function*.

Definition 30 [Output Color Relation]:
For an FSM, $M = (S, I, X, Y, \delta, \lambda)$, operationally colored by $(\hat{S}, \hat{X}, \hat{Y})$ the *output color relation* is a relation $R \subseteq \hat{S} \times \hat{X} \times \hat{Y}$ where: $(\hat{s}, \hat{x}, \hat{y}) \in R$ iff an *operational path segment* $\pi = (s_n, \ldots, s_{n+l})$ exists such that $\mathtt{f}_{\mathtt{SC}}(s_n) = \hat{s}$, that is *caused* by the operational input prefix of ρ from s_n, such that $\mathtt{f}_{\mathtt{XC}}(s_n, \rho) = \hat{x}$, and that *produces* an output sequence γ, such that $\mathtt{f}_{\mathtt{YC}}(s_n, \gamma) = \hat{y}$. □

It follows from Def. 28-4 that the output color relation is a partial function, $\hat{\lambda} : \hat{S} \times \hat{X} \mapsto \hat{Y}$, defined for the same set of state color and input color combinations as the color transition function. In the following we refer to this partial function as the *output color function*.

Definition 31 [FSM Path Predicate Abstraction]:
Let M be an operationally colored FSM according to Def. 28. A path predicate abstraction of M is the partially defined FSM $\hat{M} = (\hat{S}, \hat{s}_{reset}, \hat{X}, \hat{Y}, \hat{\delta}, \hat{\lambda})$. □

A detailed example of how a path predicate abstracted FSM is constructed for its concrete RTL implementation will be presented in Sec. 6.2 of this thesis. In order to simplify the presentation of how a practical design and verification methodology can be based on our theoretical framework, path predicate abstraction is here defined for FSMs rather than for Kripke models. Obviously, in a similar way as Kripke models can be derived from FSMs, also the definitions for FSMs, as they were just presented, can be translated to Kripke models derived from FSMs. This can be done in such a way that the STG of the concrete Kripke model becomes operationally colored (with three color components) according to Def. 23 and the STG of the abstract Kripke model is a PPA with regard to the concrete STG according to Def. 24. This is further detailed in [55].

In the following, we will show the soundness of path predicate abstraction. This soundness can be understood intuitively as a preservation of "color behavior". Through the operational coloring, the behavior of a concrete FSM can be described in terms of the state colors, input colors and output colors. In the PPA derived from this operational coloring the state colors, input colors and output colors are directly used as the states, input symbols and output symbols, respectively. The soundness of the abstraction guarantees that in both, the original FSM and the PPA, a sequence of input colors applied from an initial state causes paths in both, the original FSM and the PPA, with the same sequence of state colors and produces output sequences with the same sequence of output colors.

To aid the formalization of this soundness in Theorem 2 we first define state-color sequences, input-color sequences and output-color sequences.

Definition 32 [State Color Sequence]:
Let M be an operationally colored FSM according to Def. 28 with a set of colored states $W \subseteq S$. Then, the state color sequence of a path $(s_0, s_1, s_2, \ldots)$ in M is the sequence, $\hat{\pi} = (\mathtt{f}_{\mathtt{SC}}(s_i), \mathtt{f}_{\mathtt{SC}}(s_{i+j}), \ldots)$, of state colors of all colored states, $s_i, s_{i+j}, \ldots \in W$, on the path. □

Definition 33 [Input Color Sequence and Output Color Sequence]:
Let M be an operationally colored FSM according to Def. 28, and let $\rho = (x_0, x_1, x_2, \ldots)$ be an input sequence to M causing the state sequence $\pi = (s_0, s_1, s_2, \ldots)$ and producing the output sequence $\gamma = (y_0, y_1, y_2, \ldots)$ when applied to s_0. Further, let $(s_i, s_{i+j}, \ldots)$ be the sequence of colored states on π, let ρ_k denote the infinite suffix, $(x_k, x_{k+1}, x_{k+2}, \ldots)$, of ρ starting from x_k, and let γ_k denote the infinite suffix, $(y_k, y_{k+1}, y_{k+2}, \ldots)$, of γ starting from y_k. Then, the input color sequence is $\hat{\rho} = (\mathtt{f}_{\mathtt{XC}}(s_i, \rho_i), \mathtt{f}_{\mathtt{XC}}(s_{i+j}, \rho_{i+j}), \ldots)$, and the output color sequence is $\hat{\gamma} = (\mathtt{f}_{\mathtt{YC}}(s_i, \gamma_i), \mathtt{f}_{\mathtt{YC}}(s_{i+j}, \gamma_{i+j}), \ldots)$. □

Theorem 2 [Soundness of FSM PPA w.r.t. colors]:
Let M be an operationally colored FSM and let $\hat{M}$ be its path predicate abstraction according to Def. 31. Let ρ be a sequence of input symbols. When ρ is applied to an initial state s_0 in M it causes the path π and produces the output sequence γ. Further, let $\hat{\rho}$ be the input color sequence of ρ, let $\hat{\pi}$ be the state color sequence of π and let $\hat{\gamma}$ be the output color sequence of γ observed in M when ρ is applied in s_0 of M.

1. For any input sequence ρ there exists an abstract input sequence $\hat{\rho}$ which causes the abstract path $\hat{\pi}$ and produces the abstract output sequence $\hat{\gamma}$ when applied to the initial state $\hat{s}_{reset}$ of $\hat{M}$.
2. For any abstract input sequence $\hat{\rho}$ such that $\hat{\rho}$ causes the abstract path $\hat{\pi}$ and produces the abstract output sequence $\hat{\gamma}$ when applied to the initial state $\hat{s}_{reset}$ of $\hat{M}$, there exists a concrete input sequence ρ causing π and γ in M.

□

Proof. The proof follows from the four requirements in Def. 28 for operationally colored FSMs and from the construction of the related PPA in Def. 31 which is defined directly from the color transition function $\hat{\delta}$ (cf. Def. 29) and the output color function $\hat{\lambda}$ (cf. Def. 30) of the original FSM.

Requirement 2 of Def. 28 for an operational coloring ensures that all initialized paths begin with a colored state of the same color. This state color is the initial state $\hat{s}_{reset}$ of the PPA.

Requirement 1 ensures that if an operational path exists from a state of one color, say "blue", to a state of the same or another color, say "green", then an operational path exists from all states colored "blue" to some state colored "green". Further, the first part of requirement 3 ensures that an input sequence has one unique input color for all states of equal state color. (A single concrete input sequence may be given several input colors depending on the state it is applied from, but only if the states have different state colors.) Finally, the second part of requirement 3 ensures that if an operational path exists from "blue" to "green" then the color of the input sequence (when applied to a "blue" state) causing this operational path to be taken is different from the color of any input sequence causing an operational path from "blue" to a color different to "green". Since all "blue" states can reach "green" states for some input sequence (guaranteed by requirement 1) and since an input sequence is given a unique input color for all "blue" states (guaranteed by first part of requirement 3) it is ensured that for all paths starting in states given an equal state color, the color of the next colored state is equal if the input sequence applied from the starting state are of an equal input color.

In other words, the state color of the next colored state on a path is uniquely defined only in terms of the color of the current state and the color of the applied input sequence. The color transition function $\hat{\delta}$ (cf. Def. 29) defines exactly this relationship and this function is used directly as the transition function $\hat{\delta}$ of the PPA.

Note that not all abstract input sequences $\hat{\rho}$ are applicable, because the transition function $\hat{\delta}$ is defined only partially. However, no concrete sequences with the corresponding sequence of colors exist. Only the applicable abstract input sequences, where $\hat{\rho}$ causes a defined abstract path $\hat{\pi}$, are therefore considered in (2).

Requirement 4 for an operational coloring states that, when applied to states of the same state color, all input sequences of the same input color produce output sequences of the same output color. In other words, the output color produced from a colored state is uniquely defined by the color of the state and the color of the input sequence applied to it. The output color function $\hat{\lambda}$ (cf. Def. 30) defines exactly this relationship and this function is used directly as the

output function $\hat{\lambda}$ of the PPA.

It therefore follows, from the requirements of an operational coloring and the construction of the related PPA, that Theorem 2 holds. □

Before ending this section, it is in order to better clarify the reasons why the transition function and output function of a PPA are partial functions. The partialness of these functions is directly caused by the state-dependent mapping of input sequences, which, as mentioned in the discussion for requirement 3 of an operationally colored FSM (cf. Def. 28), allows for an input sequence to be mapped to several input colors, namely, one for each state color (but not more than one). As a result, we may in fact understand the PPA to be an FSM with a separate input alphabet for each state. (In this view each alphabet creates a partitioning of all input sequences.) The following discussion clarifies the necessity of this complication of our formalism.

Consider that we did not allow each abstract state to have its own input alphabet. Then, the coloring of the input sequences creates a partitioning of input sequences which is independent of the state the sequence is applied from. Also in this construction, it must be ensured that if any two input sequences given the same color are applied from any two states of the same color, then the color of the next colored state is the same on both paths. Such a partitioning clearly exists and, in the worst case, can be determined by enumerating all operational paths. However, its characterization may require a rather long prefix to be evaluated, in the worst case the length of the longest operational path segment.

As an example, consider the situation that in some state of a specific color, say "blue", the input sequence (0, 1, ...) guarantees a transition to some other colored state. In the same FSM, there is another colored state, say "purple", where we need to distinguish the input sequences (0, 1, 0, ...) and (0, 1, 1, ...) from each other, since they cause transitions to states of different color. We may choose to characterize the input sequence (0, 1, 0, ...) by the input color "red" and the input sequence (0, 1, 1, ...) by a second input color "yellow". Now, if we also gave the input sequence (0, 1, ...) at state "blue" its own input color, say "orange" this would lead to ambiguities. For example, the same input sequence could be described by "orange" and by "red". In order to avoid this, we only include the input colors "red" and "yellow" in our input alphabet. The input sequence (0, 1, ...) at the blue state would then be modeled by "red" $\vee$ "yellow". This approach, in principle, is always applicable for an isolated FSM and its PPA.

The situation, however changes when looking at communicating FSMs and the corresponding composition of PPAs. To enable system verification based on a composition of PPAs (cf. Chap. 5), it will be required that abstract symbols communicated between PPAs have a consistent definition in terms of concrete sequences, i.e., communicated I/O symbols must characterize the same concrete sequences both for the machine producing the symbol as an output and for the machines receiving the symbol as input. Consider again the above example of an FSM with an "over-specification" of the available abstract input symbols for a state "blue". Consider now a second FSM producing the required inputs as outputs. The output alphabet of the second FSM must match the input alphabet of the first FSM, thus, no unique color is available to describe the output sequence (0, 1, ...) produced by the second FSM. Obviously, it would be required to model the output sequence (0, 1, ...) by "red" $\vee$ "yellow". This, however, would lead to an abstract model for the second FSM with a non-deterministic output.

The discussion shows that it is desirable to evaluate input and output sequences of operational path segments for different lengths depending on which colored state is considered. Therefore, we allow each abstract state to define its own partitioning of the input and output se-

quences. As a consequence, as developed above, this results in partially defined color transition and color output functions as well as a partially defined abstract FSM.

4.3 Operational Coloring by C-IPC

The requirements for an operationally colored FSM, detailed in the previous section, are fulfilled by a complete set of interval properties (cf. Chapter 3) when the properties are written according to the convention presented in this section.

In this convention, properties are written according to a "template" such that references to the concrete RTL implementation are never made directly, but always encapsulated in macros evaluated either for FSM states, input sequences or output sequences. The assumption of the property consists of only one macro describing the FSM state at the start of the operation and one macro describing input sequences triggering the operation. The commitment of the property consists of a macro describing the FSM state at the end of the operation and a macro describing the output sequences produced.

This construction will be exploited to create complete sets of interval properties, such that each property can be viewed as a transition in an abstract state machine where the macros are abstract states, abstract inputs and abstract outputs. Hence, the FSM-state macro in the commitment of an operation will be re-used as FSM-state macro for the assumption of all of its successors.

Before we formalize this convention for writing properties a few notions must be defined to describe FSM states, input sequences and output sequences within our interval properties. Since interval properties have been defined for Kripke models (based on LTL) we will in the following consider a Kripke model K_{FSM} derived from an FSM such that all elements of the FSM are distinguishable. We consider the set of atomic formulas of K_{FSM} to be the union $A = A_S \cup A_X \cup A_Y$ where the subset A_S identifies exactly the FSM states, A_X exactly the input symbols, and A_Y exactly the output symbols.

Definition 34 [FSM-State Predicate]:
A set of FSM states can be characterized by an *FSM-state predicate*. An FSM-state predicate, $\eta(s)$, is a Boolean function only over atomic formulas of A_S in the derived Kripke model evaluating to *true* for an FSM-state *s* iff *s* is in the characterized set. □

Definition 35 [Input Sequence Predicate]:
A set of infinite input sequences can be characterized by an *input sequence predicates*. An input sequence predicate, $\iota(\rho)$, is a *sequence predicate* (see Def. 17) only over atomic formulas of A_X in the derived Kripke model. □

Definition 36 [Output Sequence Predicate]:
A set of infinite output sequences can be characterized by an *output sequence predicates*. An output sequence predicate, $\mu(\gamma)$, is a *sequence predicate* (see Def. 17) only over atomic formulas of A_Y in the derived Kripke model. □

Using the above definitions, a template for writing operation specifications (cf. Def. 19) can be formalized such that a complete set $\{(P_0, l_0), (P_1, l_1), \ldots\}$ of such specifications creates an operational coloring. In the context of the complete set, we let the set of all employed FSM-state predicates be denoted by $\{\eta_i\}$, the set of all input sequence predicates be denoted by $\{\iota_i\}$ and the set of all output sequence predicates be denoted by $\{\mu_i\}$. To emphasize that a predicate

belongs to one of these sets, the predicates will in the following be referred to as *operational predicates*.

To create a PPA we create a complete set of operation specifications $\{(P_0,l_0),(P_1,l_1),\ldots\}$ where all properties $\{P_0,P_1,\ldots\}$ are written according to the convention of an *operation property template*.

Definition 37 [Operation Property Template]:
Let η^{start}, η^{end}, ι and μ be arbitrary state predicates, input sequence predicates and output sequence predicates, respectively, for an FSM M, and let Ψ be the disjunction of all FSM-state predicates in $\{\eta_i\}$ (which includes η^{start} and η^{end}). Then, an operation specification $((A,C),l)$ for M has the form of an operation property template iff

- $A := \eta^{start} \wedge \iota$
- $C := \mu \wedge \mathsf{X}^l(\eta^{end}) \wedge (\mathsf{X}^j(\neg\Psi)$ for $0 < j < l)$ □

A property following the form of the above template ensures that, from all states characterized by η^{start} in combination with any input sequence characterized by ι some output sequence is generated that satisfies the output sequence predicate μ and after l transitions we are in some state characterized by η^{end} and no state satisfying any predicate in $\{\eta_i\}$ was visited in between. (The last part is needed to ensure that only operational path segments are described.)

In order to refer to the operational predicates of a specific property $P \in \{P_i\}$ the following notation is used: $P.\eta^{start}$ is the *starting state predicate*, i.e., the state predicate used in the assumption A, $P.\eta^{end}$ the *ending state predicate*, i.e., the predicate used in the commitment C at time point l, $P.\iota$ is the input sequence predicate and $P.\mu$ is the output sequence predicate.

We let the complete set of properties create a coloring for the FSM by mapping each operational predicate to a color. This simple one-to-one mapping is defined formally by the functions below.

Definition 38 [Correspondence of Operational Predicates and Colors]:
The correspondence between operational predicates and colors is defined by the following bijective mappings:

- $\mathtt{B}_{\eta\mathtt{C}} : \{\eta_i\} \mapsto \hat{S}$ maps FSM-state predicates to state colors,
- $\mathtt{B}_{\iota\mathtt{C}} : \{\iota_i\} \mapsto \hat{X}$ maps input sequence predicates to input colors,
- $\mathtt{B}_{\mu\mathtt{C}} : \{\mu_i\} \mapsto \hat{Y}$ maps output sequence predicates to output colors, and
- $\mathtt{B}_{\mathtt{C}\eta}$, $\mathtt{B}_{\mathtt{C}\iota}$ and $\mathtt{B}_{\mathtt{C}\mu}$ denote the respective inverse mappings. □

With the above defined mappings between predicates and colors, a set of properties, written in terms of such predicates, can be used to define a coloring of the considered FSM. It will be shown that this coloring is an operational coloring if the set of properties fulfills certain conditions.

Definition 39 [Property-Induced Coloring]:
A set of operation specifications, $\{(P_0,l_0),(P_1,l_1),\ldots\}$, all written according to the operation property template of Def. 37, induces a coloring on an FSM M in the following way:

1. a state s has the state color $\hat{s} = \mathtt{B}_{\eta\mathtt{C}}(\eta)$ iff $\eta(s) = \mathit{true}$,
2. a pair of an input sequence ρ and a colored state s has the input color $\hat{x} = \mathtt{B}_{\iota\mathtt{C}}(P.\iota)$, for a property P such that $P.\eta^{start}(s) = \mathit{true}$ and $P.\iota(\rho) = \mathit{true}$, and
3. a pair of an output sequence γ and a colored state s has the output color $\hat{y} = \mathtt{B}_{\mu\mathtt{C}}(P.\mu)$, for a property P such that $P.\eta^{start}(s) = \mathit{true}$ and $P.\mu(\gamma) = \mathit{true}$.

□

Theorem 3 [Operational Coloring by C-IPC]:
The coloring of an FSM M induced by the set of operation specifications V for M (cf. Def. 39) is an operational coloring according to Def. 28 if:

1. the FSM, M, satisfies all operation specifications in V,
2. the set of operation specifications, V, is complete,
3. all operation specifications in V are formulated according to the operation property template (cf. Def. 37),
4. the set $\{\eta_i\}$ of FSM state predicates disjointly partitions the state set, i.e., no single state satisfies more than one state predicate,
5. operational input or output predicates used with the same starting state predicate disjointly partition the set of input sequences or output sequences, respectively, and
6. no state predicate and no input sequence predicate is a contradiction. □

Proof. The FSM states are colored by the state predicates they fulfill (cf. Def. 39-1). We let this coloring be defined by the relation $R \subseteq S \times \hat{S}$ where $(s,\hat{s}) \in R \iff \mathtt{B}_{\eta\mathtt{C}}(\eta) = \hat{s} \wedge \eta(s) = \mathit{true}$.

To meet requirement 1 of Def. 28 for an operationally colored FSM, the color of a state must be uniquely defined for some subset of S, i.e., R must be a right-unique relation. This is clearly the case since the FSM-state predicates of $\{\eta_i\}$ are constructed such that they characterize disjoint sets of states (as stated in point 4).

The state coloring function must also fulfill the two rules of Def. 23: Rule 1 states that every cyclic path must contain at least one colored state. Rule 2 states that, if an operational path segment exists from some state with the color $\hat{s}_i$ to a state colored $\hat{s}_j$, then also all other states given color $\hat{s}_i$ must have an operational path to some state colored $\hat{s}_j$.

A property in the form of an operation property template checks that for all states of some color $\mathtt{B}_{\eta\mathtt{C}}(\eta_i)$ a (finite) path segment exists which ends in some state of color $\mathtt{B}_{\eta\mathtt{C}}(\eta_j)$ without visiting any other colored state along the way. Hence, a complete property suite, where all properties are written according to the template and where all properties are proven on the FSM (point 1, 2 and 3 of the theorem), ensures a coloring of states such that rule 1 and rule 2 are both fulfilled. Requirement 1 is therefore fulfilled.

Requirement 2 of Def. 28 simply ensures an unambiguous behavior from reset, by mapping all initial states to the same state color. This is ensured by the special reset property (cf. Sec. 3.3.4) which is required for a complete property set. The requirement is fulfilled when the reset property checks that all initial states are characterized by the same state predicate $\eta \in \{\eta_i\}$.

Requirement 3 of Def. 28 specifies how input sequences are characterized in an operationally colored FSM. It is required that a function $\mathtt{f}_{\mathtt{XC}}$ exists which ensures that any two operational path segments starting from states of the same color but ending in states of different colors are distinguished by their input color. It follows that the color of the next colored state must be a function of this input color and the color for the current state.

The coloring induced by the complete property set colors pairs of an input sequence and a colored state with the color of the input sequence predicate which is applicable at the given state color, and which is satisfied by the input sequence (cf. Def. 39-2).

Consider the relation $R \subseteq (W \times \{\rho_i\}) \times \hat{X}$ where $((s,\rho),\mathtt{B}_{\iota\mathtt{C}}(\iota)) \in R$ iff $\exists P \in \{P_i\}$ such that $P.\eta^{start}(s) = \mathit{true}$ and $P.\iota(\rho) = \mathit{true}$. We will show how R becomes a function possessing the attributes specified for $\mathtt{f}_{\mathtt{XC}}$ in requirement 3.

The case split test (cf. Sec. 3.3.1) ensures that all input scenarios for any ending state of a property are characterized by the assumption of a successor property. For a complete set of properties in the form of operation property templates this test ensures that R is left-total.

For right-uniqueness we must additionally ensure that no pair of a state and an input sequence can be related to more than one input color (included as point 5 in the theorem). For the complete set of properties $\{P_i\}$ this means that no two properties exist which characterize the same operational path segment.

Consider that requirement 3 was not fulfilled for R. Then, two input sequences exist with the same input color which cause for two states of the same state color operational path segments ending in states of different colors. This would imply that two properties with the same assumption A but with different ending state predicates would exist. Since the operational state predicates characterize disjoint sets of states, and the input sequence predicates characterize disjoint sets of input sequences for this state predicate, this is clearly not possible, unless A itself is a contradiction. Point 6 in the theorem states that A is not a contradiction. Requirement 3 is therefore fulfilled.

Note that the successor test does not have to be considered in this proof. Since the properties have the form of the operation property template, the assumption of each property comprises only a starting state predicate and some input sequence predicate. Since the state predicates describe pairwise disjoint sets of states the starting state predicate of each property is the same as the ending state predicate of each of its predecessor properties. The successor test (cf. 3.3.2) is hence fulfilled by construction.

Requirement 4 of Def. 28 specifies how output sequences must be characterized for an operationally colored FSM. The requirement is specified as the existence of a function $\mathtt{f}_{\mathtt{YC}}$ which ensures that any two operational path segments starting from a state of the same color and caused by input sequences of the same input color will produce output sequences of the same color, i.e., the output color is a function of the state color and the input color.

The coloring induced by the complete property set, colors pairs of an output sequence and a colored state with the color of the output sequence predicate which is applicable at the given state color, and which is satisfied by the output sequence (cf. Def. 39-3).

Consider the relation $R \subseteq (W \times \{\gamma_i\}) \times \hat{Y}$ where $((s,\gamma),\mathtt{B}_{\mu\mathtt{C}}(\mu)) \in R$ iff $\exists P \in \{P_i\}$ such that $P.\eta^{start}(s) = \mathit{true}$ and $P.\mu(\gamma) = \mathit{true}$. We show how R becomes a function possessing the attributes specified for $\mathtt{f}_{\mathtt{YC}}$ in requirement 4.

The determination test (cf. Sec. 3.3.3) ensures that all operation properties describe the value of each concrete signal for the length of the property. If the operation properties have the form of an operation property template, this test ensures that any prefix of an output sequence produced between colored states of the FSM is characterized by an operational output predicate $\mu \in \{\mu_i\}$. It follows that R is left-total.

To also ensure right-uniqueness we must ensure that no pair of a state and an output sequence can be related to more than one output color (included as point 5 in the theorem). For the set of properties $\{P_i\}$ this means that no two properties exist with the same starting state predicate and input predicate but with differing operational output predicates.

Analog to the requirement for the operational input predicates, right-uniqueness is ensured by the fact that output sequence predicates used in properties starting from the same state predicate characterize disjoint sets of output sequences (included as point 5 in the theorem).

Consider that requirement 4 was not fulfilled for R, then two operational path segments starting from a state of the same color and caused by input sequences of the same input color exist whose produced output sequences are given a differing color. This would imply that two properties with the same assumption A but with different operational output predicates exist. Since we require for any such pair of properties that their operational output predicates

characterize disjoint sets of output sequences these properties cannot both hold. Requirement 4 is therefore fulfilled. □

In practice, all the requirements listed in Theorem 3 (points 1 through 6) for creating an operational coloring can be handled even for large industrial designs.

The properties written according to the operation property template (point 3) can be proven as properties on the actual design (point 1) using any available property checking tool. The form of these properties makes them, however, well suitable for interval property checking (cf. 2.4.4) which scales well even for large industrial implementations. The completeness of a property set (point 2) can be proven as elaborated in Sec. 3.3, which is tractable even for large industrial designs and commercially available.

The additional requirements of point 4, point 5 and point 6, are simple to ensure in practice. In fact, we believe that a well structured verification suite is likely to possess these attributes anyway.

Point 4 states that no two states can fulfill the same state predicate. It can easily be checked using an additional property. In most cases, however, it is simply ensured through the construction of the state predicates. Note that this requirement also ensures that the starting state predicates are the ending state predicates of the predecessor properties.

Point 5 states that the same operation cannot be described by different properties. It can be easily checked by formulating additional properties checking that different input (output) sequence predicates used with the same starting state predicate are never simultaneously fulfilled. This can, however, also simply be ensured by the construction of the input (output) sequence predicates.

Point 6 states that no assumption is a contradiction, hence, no property is a tautology. An explicit SAT check can be used to ensure this. However, most property checking tools will automatically check for such cases and mark such properties as "vacuous" instead of *true* .

4.4 Model Checking with Path Predicate Abstraction

The soundness of path predicate abstraction with regards to color sequences stated in Theorem 2, will be extended to a soundness between LTL properties of an abstract and a concrete model in this section. This will make it possible to do model checking on a highly abstract PPA to obtain formally valid proofs for corresponding properties of the actual RTL implementation.

Since LTL is defined for Kripke models, Theorem 2, which was stated for FSMs, needs to be adapted to Kripke models. As already discussed in Sec. 4.3, a Kripke model can be derived from an FSM such that all inputs, outputs and states are distinguishable. Theorem 2 is then translated simply by substituting states, inputs and outputs of FSMs with their corresponding subsets of atomic formulas. The proof of this adapted soundness theorem follows trivially.

The proof of Theorem 2 guarantees that all reachable color sequences of the concrete FSM are modeled by the partially defined PPA. In other words, the combinations of abstract input and abstract state for which the PPA is undefined cannot occur in its concrete FSM.

In practice, the PPA may have to be extended to a completely defined FSM model before submitting it as input to available model checking tools. (Deriving a Kripke model from this FSM model will, most likely, be an internal pre-processing step carried out in the model checking tool.) The extension can be done by adding a state $\hat{s}_{never}$ and an output $\hat{y}_{never}$ and letting all undefined transitions end in $\hat{s}_{never}$ while producing $\hat{y}_{never}$. (Note that this implies that also the

	Abstract formula	Concrete formula
(1)	$\hat{s}_i$	η_i
	$\hat{x}_i$	$\Psi_{\hat{x}_i} \wedge \iota_i$
	$\hat{y}_i$	$\Psi_{\hat{y}_i} \wedge \mu_i$
(2)	$\mathsf{X}\,\hat{f}$	$\mathsf{X}(\neg\Psi\,\mathsf{U}\,(\Psi \wedge f))$
	$\mathsf{F}\,\hat{f}$	$\mathsf{F}(\Psi \wedge f)$
	$\mathsf{G}\,\hat{f}$	$\mathsf{G}(\Psi \Rightarrow f)$
	$\hat{g}\,\mathsf{U}\,\hat{f}$	$(\Psi \Rightarrow g)\,\mathsf{U}\,(\Psi \wedge f)$
(3)	$\neg\hat{f}$	$\neg f$
	$\hat{f} \wedge \hat{g}$	$f \wedge g$
	$\hat{f} \vee \hat{g}$	$f \vee g$

Table 4.1: Abstract formulas vs. concrete formulas

transitions from $\hat{s}_{never}$ all end in $\hat{s}_{never}$.)

In our formalism such an extension is, however, not necessary. A Kripke model does not have a transition *function* dependent on inputs but a transition *relation*. In the Kripke model derived from our partially defined FSM, we therefore simply do not add Kripke states representing undefined combinations of abstract input and abstract FSM state.

Let $\Psi_{\hat{x}}$ be the disjunction of all state predicates mapped to an abstract state where $\hat{x}$ is a defined input:

$\Psi_{\hat{x}} := \bigvee_{\hat{s} \in \hat{S}_{\hat{x}}} \mathtt{B}_{\mathtt{C}\eta}(\hat{s})$, where $\hat{s} \in \hat{S}_{\hat{x}} \iff \hat{\delta}(\hat{s},\hat{x})$ is defined.

In order to understand why we consider a disjunction over several states, $\hat{s} \in \hat{S}_{\hat{x}}$, remember that we allow a separate input alphabet for each abstract state. However, this is not required. It is also possible and quite common that an abstract input is applicable in several abstract states.

Similarly, let $\Psi_{\hat{y}}$ be the disjunction of all state predicates mapped to an abstract state where $\hat{y}$ can be produced:

$\Psi_{\hat{y}} := \bigvee_{\hat{s} \in \hat{S}_{\hat{y}}} \mathtt{B}_{\mathtt{C}\eta}(\hat{s})$, where $\hat{s} \in \hat{S}_{\hat{y}} \iff \exists \hat{x} \in \hat{X} : \hat{\lambda}(\hat{s},\hat{x}) = \hat{y}$.

Table 4.1 shows the mapping between abstract LTL formulas and the corresponding LTL formulas for the concrete model. Section (1) of the table is concerned with the atomic formulas of the abstract model.

An atomic FSM-state formula $\hat{s}_i$ in the abstract Kripke model corresponds to the concrete LTL formula $\eta_i = \mathtt{B}_{\mathtt{C}\eta}(\hat{s}_i)$. It characterizes the set of all paths beginning in a state satisfying the FSM-state predicate η_i.

An atomic input formula $\hat{x}_i$ corresponds to the LTL formula $\Psi_{\hat{x}_i} \wedge \iota_i$, where $\iota_i = \mathtt{B}_{\mathtt{C}\iota}(\hat{x}_i)$. It characterizes the set of paths starting in some colored state followed for any input sequences satisfying the input sequence predicate ι_i.

In the same way, an atomic output formula $\hat{y}_i$ corresponds to the LTL formula $\Psi_{\hat{y}_i} \wedge \mu_i$, where $\mu_i = \mathtt{B}_{\mathtt{C}\mu}(\hat{y}_i)$. It characterizes the set of paths starting in some colored state producing an output sequence satisfying the output sequence predicate μ_i.

Section (2) of Tab. 4.1 shows the translations for the temporal operators. The formulas

$\hat{f}$ and $\hat{g}$ are arbitrary LTL formulas based on the atomic formulas of the abstract model. An abstract formula of the form $\mathsf{X}\hat{f}$ corresponds to the concrete LTL formula $\mathsf{X}(\neg\Psi\,\mathsf{U}\,(\Psi \wedge f))$. Intuitively, $\mathsf{X}\hat{f}$ means "in the operation immediately following the current operation, $\hat{f}$ holds". In the concrete model, this characterizes paths that begin in a colored state for which the (infinite) suffix starting from the second colored state on these paths is characterized by f. (Note: The concrete formula using the Until operator represents paths of arbitrary length. In fact, the length of the prefix between the first and second colored state is bounded by the length of the longest operational path segment in the Kripke model. For a given Kripke model, the concrete formula can be strengthened by a constraint on this maximum length for this prefix.)

The abstract Until operator U expresses that on the abstract paths considered, $\hat{g}$ holds until eventually a state is reached where $\hat{f}$ holds. (For convenience, it is said that an LTL formula holds for a state when it is meant that it holds for all paths starting in this state, see Sec. 2.2.2.) These paths correspond to concrete paths where g holds on the colored states along the path until, finally, a colored state is reached where f holds.

The "eventually $\hat{f}$" LTL formula, $\mathsf{F}\hat{f}$, is a shorthand notation for *true* $\mathsf{U}\,\hat{f}$, leading to the translation shown in the table. The formula $\mathsf{G}\hat{f}$ states that in all abstract states on the path, $\hat{f}$ holds. In the concrete Kripke model, this is expressed by $\mathsf{G}(\Psi \Rightarrow f)$: on all colored states on the path the formula f holds.

The Boolean operators $\neg$, $\wedge$ and $\vee$ are simply transferred from the abstract to the concrete formula as shown in section (3) of Tab. 4.1.

Theorem 4 [LTL Soundness]:
Let $\hat{p}$ be an abstract LTL formula of one of the forms (1), (2) or (3) of the left-hand side of Tab. 4.1, where $\hat{f}$ and $\hat{g}$ are sub-formulas also of these forms. The concrete LTL formula p is obtained by recursively applying the correspondences of Tab. 4.1 to replace the sub-formulas of $\hat{p}$ by their concrete formulas.

Path predicate abstraction is sound for LTL in the sense that if $\hat{p}$ holds for $\hat{M}$ then p holds for M, and vice versa, i.e., $\hat{p} \models \hat{M} \Longleftrightarrow p \models M$. □

Proof. The proof is based on the soundness of FSM path predicate abstraction with respect to color sequences (cf. Theorem 2). We consider a Kripke model K_{FSM} derived from an FSM such that all elements of the FSM are distinguishable. This is achieved by defining the set of atomic formulas of K_{FSM} to be the union $A = A_S \cup A_X \cup A_Y$ where the subset A_S identifies exactly the FSM-states, A_X exactly the input symbols, and A_Y exactly the output symbols.

If an FSM is operationally colored in terms of the elementary formulas (colors) $\{\eta_i\}$, $\{\iota_i\}$ and $\{\mu_i\}$, as given by Def. 28 and Def. 38, then, also the Kripke model K_{FSM} is operationally colored. This operational coloring for K_{FSM} constitutes a path predicate abstracted Kripke model which is equal to the Kripke model that can be derived from the path predicate abstracted FSM. The rest of the proof is dedicated to show how the correspondences of Table 4.1 result from correspondences between color sequences in the concrete and abstract models.

We show the correspondence of each pair of LTL formulas of Tab. 4.1 by considering the color sequences produced on the paths defined by the LTL formulas. For each pair of LTL formulas, the correspondence between abstract and concrete formula is shown by the following reasoning:

- The concrete formula characterizes a set of concrete paths.
- The subset of these paths starting at colored states can be described by a set of color sequences.

- The abstract formula characterizes a set of abstract paths representing the *same* set of color sequences.

We now show, for each pair of LTL formulas in Tab. 4.1, the equivalence of the formulas with certain sets of color sequences, based on the semantics of LTL formulas.

Elementary LTL formulas (section (1) in Tab. 4.1) characterize paths that begin at a colored state. An elementary formula η_i characterizes all paths whose first state satisfies η_i. By construction, the same set of paths is described by the set of state color sequences whose first color is $\hat{s}_i = \mathtt{B}_{\eta\mathtt{C}}(\eta_i)$. The atomic abstract input formula $\hat{x}_i$ is an input color and corresponds by construction to the set of paths satisfying $\iota_i = \mathtt{B}_{\mathtt{C}\iota}(\hat{x}_i)$ and starting in a state satisfying $\Psi_{\hat{x}_i}$, i.e., starting in a concrete state mapped to one of the abstract states where $\hat{x}_i$ is applicable. Analogously, the atomic abstract output formula $\hat{y}_i$ corresponds by construction to the set of paths which satisfies $\mu_i = \mathtt{B}_{\mathtt{C}\mu}(\hat{y}_i)$ and which starts in a state satisfying $\Psi_{\hat{y}_i}$, i.e., starting in a concrete state mapped to one of the abstract states from where $\hat{y}_i$ can be produced.

Consider the Boolean LTL formulas of section (3) in Tab. 4.1. We assume that the theorem holds for the pairs of sub-formulas $\hat{f}$—f and $\hat{g}$—g. (The validity of this assumption is proven below.) Then, the LTL sub-formula f characterizes the same set of paths that can, equivalently, be described by the set of color sequences $(\hat{w}_0, \hat{w}_1, \ldots)$ characterized by the abstract LTL sub-formula $\hat{f}$. The formula $\neg f$ holds in a colored state s_0 if all paths $(s_0, s_1, \ldots)$ do not fulfill f. The set of all such paths can be described by a set of color sequences, none of which fulfill $\hat{f}$. (If a color sequence in this set would fulfill $\hat{f}$ then, according to our assumption and to Theorem 2, a concrete path starting in s_0 would exist that satisfies f, contradicting the validity of the LTL property $\neg f$ in this state.) The Boolean operators $\wedge$ and $\vee$ correspond to intersection and union, respectively, on sets of paths or sets of color sequences.

At last, we consider the temporal operators, one at a time, and prove that the correspondence shown in section (2) of Tab. 4.1 holds.

The concrete formula $\mathsf{X}(\neg\Psi\,\mathsf{U}\,(\Psi \wedge f))$ describes paths $(s_0, \ldots, s_f, \ldots)$ that start at some state s_0 and continue with a finite sequence of zero or more non-colored states until they visit a colored state s_f. The suffix $(s_f, \ldots)$ fulfills the LTL formula f. We assume that Theorem 4 holds for the pair of sub-formulas $\hat{f}$ and f, i.e., sub-formula f characterizes path suffixes that can, equivalently, be described as the set of color sequences described by $\hat{f}$. (Validity of assumption is proven below.) Consider now all paths fulfilling the concrete LTL formula that start at *colored* states s_0 colored with $c(s_0) = \hat{w}_0$. In every such path, no state is colored between s_0 and s_f, hence, s_f is colored with the *next* color, $\hat{w}_1$, in the color sequence $(\hat{w}_0, \hat{w}_1, \ldots)$ produced by the path. This means that the set of paths beginning at colored states and characterized by the concrete LTL formula can, equivalently, be characterized by color sequences $(\hat{w}_0, \hat{w}_1, \hat{w}_2, \ldots)$ where the color sequence suffix $(\hat{w}_1, \hat{w}_2, \ldots)$ is the set of path suffixes described by the LTL sub-formula f. The set of color sequences $(\hat{w}_0, \hat{w}_1, \hat{w}_2, \ldots)$ describes exactly the set of paths in the abstract model where the suffix $(\hat{w}_1, \hat{w}_2, \ldots)$ fulfills sub-formula $\hat{f}$. This set of paths $(\hat{w}_0, \hat{w}_1, \hat{w}_2, \ldots)$ is described by the abstract LTL formula $\mathsf{X}\hat{f}$.

The concrete formula $\mathsf{F}(\Psi \wedge f)$ describes paths $(s_0, s_1, \ldots, s_f, \ldots)$ with a finite prefix ending at a colored state s_f. The suffix $(s_f, \ldots)$ fulfills the LTL formula f. We assume that Theorem 4 holds for the pair of sub-formulas $\hat{f}$ and f, i.e., sub-formula f characterizes path suffixes that can, equivalently, be described as the set of color sequences described by $\hat{f}$. (Validity of assumption is proven below.) Every concrete path fulfilling $\mathsf{F}(\Psi \wedge f)$ produces a color sequence with a finite prefix and a suffix fulfilling $\hat{f}$. Hence, the color sequences produced by the path

fulfill $\mathsf{F}\hat{f}$.

The validity of the theorem for the formula pair $\mathsf{G}\hat{f}$ — $\mathsf{G}(\Psi \Rightarrow f)$ follows from the duality law $\neg\mathsf{F}p = \mathsf{G}\neg p$.

— In the sub-proofs, we made the assumption that the theorem holds for sub-formula pairs $\hat{f}$—f and $\hat{g}$—g. This assumption is proven by recursively decomposing the formulas into sub-formulas, constructing a recursion tree, until, at the leaves of the tree, pairs of an abstract atomic formula and a concrete elementary formula are reached. The validity of the theorem for these formulas does not rely on sub-formulas. The validity of the assumption follows by induction from the leaves of the recursion tree backwards to the formula at the root. □

4.5 Comparison with other Abstraction Techniques

It is interesting to compare PPA with the well-known notion of a *bisimulation* and with the two weakened forms, *stuttering bisimulation* and *bisimulation modulo silent actions* as introduced in Sec. 2.5 For this comparison we consider a Kripke model's state transition graph and we let the labeling of states correspond to the coloring of nodes, as defined for directed graphs in Sec. 4.1.

PPA can be viewed as a two-step process where the first step is to find an operational coloring and the second step is to create an abstraction based on this colored model. If we consider both steps, then it becomes clear that the relationship between a concrete model and its PPA is a relationship between models labeled over different alphabets.

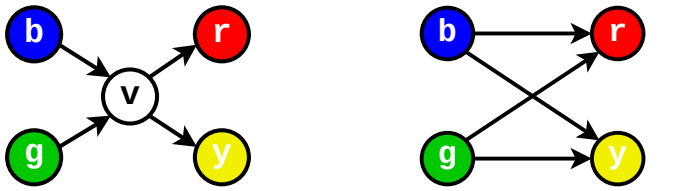
Bisimulation and its two weakened forms, on the other hand, describe relationships between a concrete and an abstract model labeled over the same alphabet. This marks an important difference between previous work based on these classical notions and our work which aims at closing the semantic gap between descriptions of RTL designs, as practiced in industry based on standard hardware description languages, and their system-level models.

Besides this different nature of PPA resulting from the first step, also the differences related to the second step are worthwhile to be illuminated. In the following, we therefore compare bisimulation and its two weakened forms with our approach for creating an abstraction based on operational coloring.

A bisimulation and a stuttering bisimulation require that a color is defined for every node in G, i.e., that $V = W$. The relation between G and $\hat{G}$ is a *bisimulation* iff every path in G can be mapped to a path in $\hat{G}$ such that the sequence of colors is the same. In other words, if G does not contain any uncolored nodes, the notion of path predicate abstraction and bisimulation become identical. In general, path predicate abstraction describes a *weaker* relationship than bisimulation since every bisimulation is also a PPA but not vice versa.

The example in Figure 4.3 shows that a path predicate abstraction cannot be mapped to the concept of a stuttering bisimulation. In a stuttering bisimulation the bisimulation is weakened by additionally allowing that paths along nodes in G with identical color (stuttering) can be mapped to a single node in $\hat{G}$ with that color.

For obtaining the abstract graph $\hat{G}$ from the concrete graph G in Figure 4.3 by a stuttering bisimulation instead of path predicate abstraction it would be required to make $V = W$ by assigning one of the colors "green", "blue", "red" or "yellow" to the uncolored node v. However, no matter what color we choose for v, an additional edge is created in $\hat{G}$. For example, if v becomes a green node, this wrongly creates an edge "blue" $\rightarrow$ "green" in $\hat{G}$.



(a) Operational coloring (b) Resulting abstraction

Figure 4.3: Abstraction not obtainable with stuttering bisimulation

As the above example illustrates, an important difference is that PPA only maps a subset of the nodes. This mapping is similar to *bisimulation modulo silent actions*, which was presented in a version adapted to Kripke-models in Sec. 2.5.3. For comparison, we let the notion of the *silent action* correspond to the concept of "uncolored nodes". Then, for a corresponding labeling/coloring, it could indeed be shown that $\hat{G}$ and G are *silent-action-bisimilar*.

However, we cannot assume that a labeling exists which fulfills our rules of an operational coloring (cf. Def. 23). Bisimulation, and its weakened forms, are defined for an arbitrary labeling, which is required since the labeling is the inherent labeling of the derived model. The coloring in PPA, on the other hand, complies with the rules of an operational coloring. This is possible because the "pre-processing" step creates a new labeling based on the labeling inherent to an automatically derived model rather than using this inherent labeling itself.

In fact, this difference accounts for a different focus of previous work using bisimulation and its derivatives. Previous work in this area mostly has the objective to identify an efficient abstraction based on an arbitrary labeling of the concrete model. In our work, the objective is to find a special labeling that fulfills the conditions of operational coloring. Then, the abstraction is immediately given.

The classical work [3] of Abadi and Lamport assumes that a concrete and an abstract model are given. The concrete model *implements* the abstract model if the externally visible behavior of the concrete is a subset of the externally visible behavior of the abstract. Then, the existence of a refinement mapping proving this relationship is examined. In this work, we are only given the concrete model and we are searching for the abstract model. We find this abstract model by constructing a mapping function, which is formulated based on operational coloring and which *defines* the abstract model. The refinement mapping for an operationally colored model is immediately obvious and maps each label (color) of the concrete model to one abstract state. In other words, PPA describes a special case of a bisimulation modulo silent actions, where the refinement mapping is obvious but, as a result of the operational view, can still be linked to practical RTL designs by step 1 of our approach.

Chapter 5

Compositional Path Predicate Abstraction

In this chapter the compositionality of PPA [58] is discussed. It will be shown how a system model composed from several communicating path predicate abstractions is constructed to become a sound model with respect to corresponding LTL properties for the concrete system. Note that this soundness does not follow trivially from the soundness of a single PPA since the modules are *time-abstract*. A transition in an abstract module corresponds to an operation of an arbitrary, but finite, length. Hence, in a system model composed of such path predicate abstractions all interleavings between operations in neighboring modules must be considered. In Sec. 5.1 this abstract system model is defined as an *asynchronous composition* communicating over *messages*.

The abstract model to be introduced comprises an over-approximation of interleavings and is, as discussed in Sec. 5.5, a conservative model with regards to LTL properties of the corresponding concrete system. The false negatives introduced in this conservative model may be avoided by ensuring that synchronization mechanisms of the concrete model are made explicit through the property suite and translated to corresponding mechanisms in the asynchronous abstract model.

In Sec. 5.2 the communication and synchronization approaches used in digital hardware are discussed in general terms and categorized. Methods for modeling these categories of communication schemes, without breaking the formal relationship between the abstract and the concrete model, are introduced in Sec. 5.3. To enable a more flexible abstraction of communication, a minor extension is made to our abstraction technique in Sec. 5.4 that allows for a weakened relationship between *wait messages* and abstract symbols. Finally, in Sec. 5.5, it is shown that the composed abstract system model is indeed sound with respect to a certain set of LTL-properties.

5.1 Abstract System Model

We model a concrete system of n finite state machines, $M_i = (S_i, I_i, X, Y_i, \delta_i, \lambda_i)$. The machines are interconnected through a common, global, input alphabet $X = Y_0 \times Y_1 \times Y_2 \times \ldots \times Y_n$, i.e., every machine receives the primary input, Y_0, and the outputs of every other machine, Y_i for $i > 0$, as a possible input. (To simplify later notations, the primary input, i.e., the input to the system from external sources, is denoted Y_0, as opposed to the more appropriate X_0.)

In order to create consistent abstractions, in our compositional approach, input sequence predicates and output sequence predicates are replaced by *message specifications*. Message

specifications are used to define output sequence segments of some FSM of the system that serve as input sequence segments in other FSMs triggering operations there. Each message specification is syntactically defined in terms of the global input space X, however, semantically it relates only to the outputs of a specific sending machine. A receiving machine can access the output of several sending machines simultaneously and may therefore evaluate its input at any point in time.

In order to ensure that the behavior of the abstract system corresponds to the behavior of the concrete system we must ensure that the input/output sequences considered are consistent. In Chapter 4 we let sequence predicates define a correspondence between abstract symbols and concrete sequences. However, for the soundness of the composed abstract system this correspondence is insufficient.

In the composed system, where the output of a module can be the input of another, it is required, for a sound abstraction, that the abstraction of these concrete I/O sequences is done consistently for all modules. That is, any initialized I/O sequence must be abstracted to the same sequence of abstract symbols for all modules in the system. In the current definition, where it is possible that modules exchange an abstract symbol while executing operations of different lengths, this is not ensured. The abstract symbols may then characterize different concrete scenarios, because, when no length is specified then the evaluation of the next abstract symbol may be done based on differing suffixes of the initialized concrete I/O sequence. For compositional PPA we therefore let abstract symbols characterize sequence segments of a specified length. For this characterization sequence predicates are paired with a length, a pair referred to as a *message specification*.

Definition 40 [Message Specification]:
A *message specification* is a pair, (μ, l), of a sequence predicate, μ, (cf. Sec. 17) only over atomic formulas A_X encoding the global input alphabet and a length l. It characterizes the finite prefix of length l of the I/O sequences characterized by μ. The length l is the length of the operation specification (P, l) it is used in. (A sequence predicate used in operations of different lengths describes different messages and is therefore abstracted to different abstract symbols.) □

Note that, for simpler notation, we write $\mu_j((y_0, \ldots, y_{l-1}))$ when characterizing an *output* sequence of length l sent by a specific machine M_j in a system. The same specification written as $\mu_j((x_0, \ldots, x_{l-1}))$ characterizes all global system *input* sequences of length l where M_j produces the specified output sequences. In the former notation the message predicate describes outgoing messages, in the latter the same messages are considered as ingoing. In practice, such message specifications may be defined as *macros* in a property language, relating to logic values on the interconnect lines between communicating machines. The same macro can be used without modification for ingoing and outgoing message specifications.

When an individual FSM in the system reads its inputs these values may be outputs of more than one other machine. In the context of a composed system the input sequence predicate, ι, of an operation property (cf. Def. 37) is therefore expressed as such a conjunction of message predicates, one from each machine M_j in a system of n machines:

$$\iota = \bigwedge_{j=0}^{n} \mu_j, \text{ where } \mu_j \in Q_j$$

Note that, formally, we need to define "null" message specifications for outputs from ma-

chines in the system that are ignored by a specific FSM.

The message specifications are mapped to abstract symbols through a mapping function $\beta_j(\mu)$. There exists a mapping function β_j for every FSM M_j in the system.

Definition 41 [Message Abstraction]:
For every message specification (μ, l) there is a distinct, unique, *abstract message* symbol $\hat{y} \in \hat{Y}_j$. There is a one-to-one mapping between the message specifications and the abstract message symbols by the mapping function: $\beta_j : \{(\mu_j, l)\} \mapsto \hat{Y}_j$. □

Using the β_j we can map every operational input predicate ι to a corresponding tuple of abstract messages, $(\hat{x}_0, \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_n,)$, where $\hat{x}_j = \beta_j((\mu_j, l))$, (and vice versa).

Definition 42 [Abstract System Alphabet]:
The abstract system alphabet is a set of $(n+1)$-tuples, $\hat{X} = \hat{Y}_0 \times \hat{Y}_1 \times \hat{Y}_2 \times \ldots \times \hat{Y}_n$ where $\hat{Y}_0$ is the set of abstract primary input messages and where $\hat{Y}_j$ for $j > 0$ is the set of abstract message symbols of the j-th sending machine according to Def. 41. □

The abstract state set is the product of the individual FSM state sets. But how should the transition behavior be modeled? In the concrete system, every finite state machine, M_i, corresponds to a path predicate abstraction $\hat{M}_i$. An operation in a machine M_i may comprise a sequence of state transitions but it corresponds to a single transition in the abstract model $\hat{M}_i$. The temporal relationship between the operations in different machines, e.g., based on a common clock, is lost in the abstraction. Hence, in our abstract system model the abstract FSMs communicate *asynchronously* with each other by exchanging messages. The unknown temporal relationship between the modules is modeled using non-determinism: While each abstract FSM $\hat{M}_i = (\hat{S}_i, \hat{I}_i, \hat{X}, \hat{Y}_i, \hat{\delta}_i, \hat{\lambda}_i)$ is still a deterministic FSM the composed model $\hat{M}$ has a non-deterministic transition behavior modeled by a relation $\hat{T}$ rather than a function $\hat{\delta}$.

Definition 43 [Asynchronous composition]:
The *asynchronous composition* $\hat{M}$ of n path-predicate-abstracted FSMs $\hat{M}_i$ is given by $\hat{M} = (\hat{S}, \hat{I}, \hat{X}, \hat{Y}, \hat{T}, \hat{\lambda})$, where

- $\hat{S} = \hat{S}_1 \times \hat{S}_2 \times \hat{S}_3 \times \ldots \times \hat{S}_n$, is the set of states,
- $\hat{I} = \hat{I}_1 \times \hat{I}_2 \times \hat{I}_3 \times \ldots \times \hat{I}_n$, is the set of initial states,
- $\hat{X} = \hat{Y}_0 \times \hat{Y}_1 \times \hat{Y}_2 \times \ldots \times \hat{Y}_n$ is the input alphabet where $\hat{Y}_0$ is the set of abstract primary input messages and, for all i, $1 \leq i \leq n$, $\hat{Y}_i$ is the set of messages produced by the abstract FSM $\hat{M}_i$,
- $\hat{Y} = \hat{Y}_1 \times \hat{Y}_2 \times \hat{Y}_3 \times \ldots \times \hat{Y}_n$ is the output alphabet where $\hat{Y}_i$ is the set of messages produced by the abstract FSM $\hat{M}_i$,
- $\hat{T}$ is the transition relation:
  $((\hat{s}_1, \ldots, \hat{s}_n), \hat{x}, (\hat{s}'_1, \ldots, \hat{s}'_n)) \in \hat{T}$ iff $\exists i, 1 \leq i \leq n$ such that $\hat{s}'_i = \hat{\delta}_i(\hat{s}_i, \hat{x})$ and $\forall j, 1 \leq j \leq n, j \neq i : \hat{s}'_i = \hat{s}_i$.
- $\hat{\lambda} : \hat{S} \mapsto \hat{Y}$ is the output function, labeling every state with the output messages produced by all sub-modules:
  $\hat{\lambda}((\hat{s}_1, \ldots, \hat{s}_n)) = (\hat{\lambda}_1(\hat{s}_1), \hat{\lambda}_2(\hat{s}_2), \ldots \hat{\lambda}_n(\hat{s}_n))$. □

This notion of an asynchronous composition is illustrated in an example below (Fig. 5.5). All possible interleavings of operations between machines are represented. This model is closely related to the asynchronous product of ω-automata in Spin [31]. Note that the transitions in the asynchronous composition represent single transitions of sub-modules, i.e., modules never transition simultaneously but always "one after the other".

When checking liveness properties based on this asynchronous composition fairness constraints need to be added to make sure that every sub-FSM $\hat{M}_i$ infinitely often makes a transition (cf. *finite progress assumption* in [31]). This is implemented as *weak fairness* in Spin.

5.2 Communication schemes in digital hardware

Communication in digital hardware relies on a few basic principles. The communicating parties need to synchronize with each other before messages can be transmitted. Also, there needs to be an agreement on how the message is actually transferred from the sender to the receiver(s).

Let us discuss the different communication schemes in digital hardware that we address in this work. A first fundamental distinction is between *asynchronous* communication, relying on dedicated event signaling, and *synchronous* communication, relying on a common hardware clock. (Note that the terms "asynchronous" and "synchronous" are here used in the context of data transmission at the hardware level. They have a different meaning at the software level, where they usually imply non-blocking or blocking communication, respectively.) Another distinction is to be made between implementations of communication systems that rely on timing constraints/guarantees (*implicit timing*) and those that do not.

In asynchronous communication the synchronization of sender and receiver is carried out through event signaling: one or more communication partners signal their being ready for communication by asserting a synchronization signal. If only one partner sends a synchronization signal then local timing constraints must guarantee that the other is ready to communicate when the synchronization signal comes. The message is then transferred either through implicit timing, meaning that that the communication partners comply to timing constraints such as latency periods or setup/hold times. Or, if no implicit timing information is used, proper transmission is signaled through a handshake.

In synchronous communication the situation is similar, however, all communication partners rely on a common clock. Before a message is sent the communication partners synchronize by asserting synchronization signals (*explicit synchronization*). If only one communication partner sends such a signal (*unilateral synchronization*) then it must be guaranteed (through local timing constraints) that the other partner(s) are ready to receive the signal and the message. Because of the common clock, in synchronous communication the actual data exchange may comprise several steps (such as the beats in a burst operation on a bus). Proper reception of the data may be signaled explicitly (e.g., to accommodate for access latencies) or it may not need to be signaled because of the implicit synchronization through the clock.

5.3 Modeling Communication

Based on the categorization of communication schemes described in Sec. 5.2 we show how communication can be modeled within our methodology using explicit synchronization at the abstract level.

Since our computational model is based on an asynchronous product of the individual FSMs, it is interesting to note that a system composed at the abstract level simply by connecting abstract inputs and outputs and without any abstract synchronization mechanism is actually sound with respect to LTL. However, such a simple model will usually introduce many false negatives due to an over-approximation of interleavings between operations of different modules.

Therefore, the synchronization mechanisms of the concrete system should be reflected in an abstract synchronization model that is created to avoid these false negatives. In the following, it is explained how to model an abstract synchronization that is sound by construction for the standard communication schemes considered here.

Fig. 5.1 shows an example of a four-phase handshake between two Moore machines M_1 and M_2. The handshake is carried out using signals s and r. The signal s, produced by M_1, is asserted only in state S and de-asserted in all other states. Likewise, signal r, produced by M_2, is asserted in no state other than R.

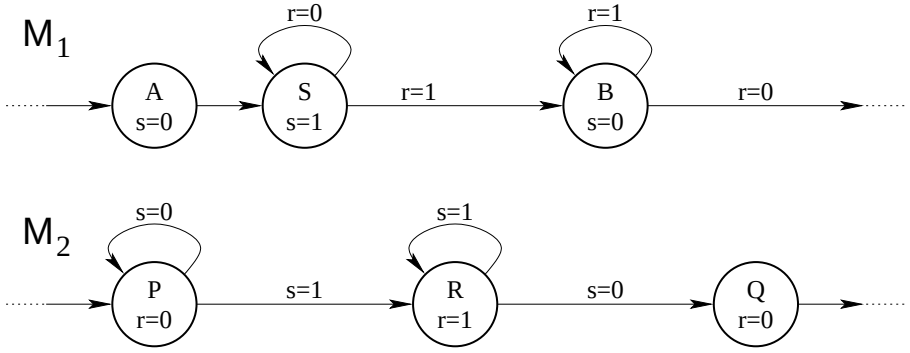


Figure 5.1: Asynchronous communication based on four-phase handshake

Before data can actually be transferred both machines need to synchronize. Assume that M_2 is waiting in P. When M_1 moves from A to S it sends a synchronization signal $s = 1$, possibly together with some data. M_2 is triggered by this signal and moves into R. Because there are no timing guarantees machine M_1 needs to wait in its sending state S until M_2 has actually received the message, moved into state R and acknowledged back to M_1 by sending the signal $r = 1$, again possibly together with some data. Machine M_1 then de-asserts s and waits for M_2 to de-assert r as well. Note that M_1 needs to wait for M_2 in state B, otherwise a new message sent during some state sequence $(B,\ldots,A,S,B)$ could go unrecognized if machine M_2 remains in state R during that time. The four-phase handshake built with the signals s and r ensures certain reachability constraints according to the four phases: state P is not left unless state S is taken, S is not left while in P, R is not left while in S and B is not left while in R.

In order to obtain a composed model of path-predicate-abstracted state machines, each of the communication schemes discussed in Sec. 5.2 needs to be mapped to this four-phase handshake. This is trivial in the case of asynchronous communication without local timing guarantees. In this case a four-phase handshake communication is present in the concrete implementation as well as in its path predicate abstraction. In the other schemes discussed above only parts of the four-phase handshake are present in the path predicate abstraction. We then need to extend the abstract models of sender and receiver(s) with the missing elements.

As an example, consider the unilateral synchronous scheme: two machines M_1 and M_2 communicating in a synchronous system with M_1 sending the synchronization signal and M_2 receiving it. Fig. 5.2 shows parts of their state transition graphs. The system relies on an implicit timing guarantee stating that machine M_2 is always in state P whenever machine M_1 enters state S. (Such implicit timing guarantees result from the communication protocol and can usually be identified easily.) Machine M_1 sends the synchronization signal, $z = 1$, in state S to indicate that a communication operation begins. Machine M_2 waits in P until the synchronization signal triggers a communication operation. The operation lasts for several clock cycles in

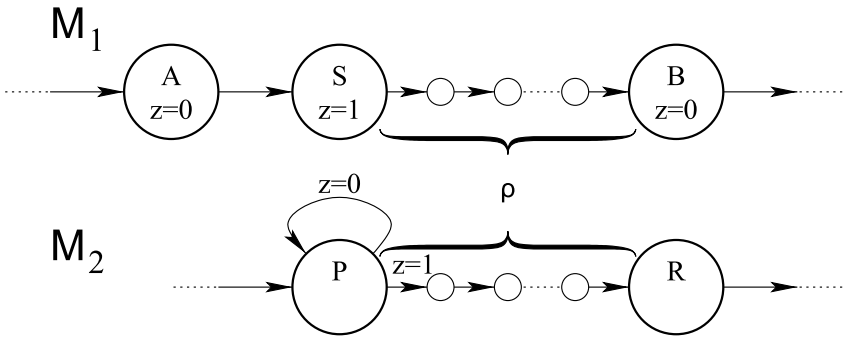


Figure 5.2: Synchronous communication with unilateral synchronization

which data can be exchanged between the machines. During this operation, both machines remain in synchrony due to the common clock while they each traverse the non-important states of the operation (indicated by the smaller circles). A message specification (μ, l) is used to characterize the I/O sequences, ρ, exchanged in the operation, i.e., $\mu(\rho) = \mathit{true}$.

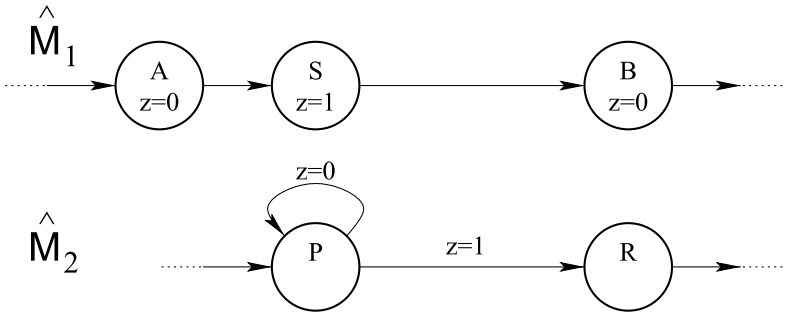


Figure 5.3: Path predicate abstractions of M_1 and M_2

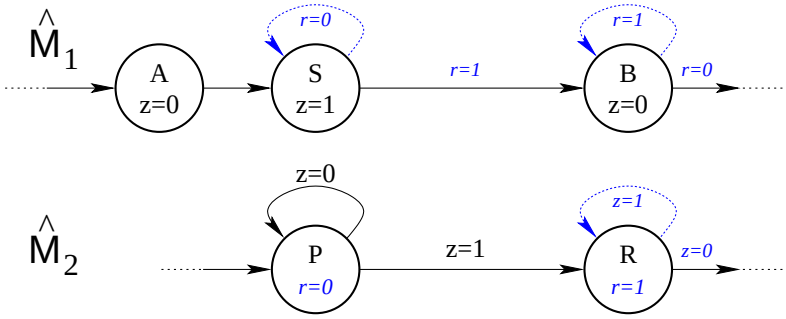


Figure 5.4: Path predicate abstractions of M_1 and M_2 with extensions

The path predicate abstractions of M_1 and M_2 (cf. Sec. 4.2) are given by the state transition graphs in Fig. 5.3. The abstract states S and P correspond to the important starting states of the communication operation between M_1 and M_2, the abstract states B and R mark its end. Comparing this with the four-phase handshake of Fig. 5.1 we see that states with the same names correspond to each other. The synchronization signal z serves as one of the handshake signals, namely s in Fig. 5.1. However, for a full four-phase handshake, the dotted/italic elements of Fig. 5.4 need to be added so that we soundly model the communication in an abstract system

that does not rely on timing guarantees. In this example, an abstract handshake signal r needs to be introduced that is asserted only in state R. Self loops and guard conditions are added to the state transition graphs of the abstract machines $\hat{M}_1$ and $\hat{M}_2$ as shown in Fig. 5.4.

When are we allowed to add these elements to the state transition graphs of the path-predicate-abstracted models? As stated above, the elements (states, transitions, guard conditions) of a four-phase handshake produce a behavior with certain reachability constraints on the product states of the composed abstract system. We may extend path predicate abstractions to full four-phase handshake communication if and only if the corresponding concrete system has the same reachability constraints on the involved important states as the extended abstract system. This must be shown for all communication schemes considered in our methodology.

Let us begin with the case of unilateral synchronization as shown in the above example. Referring to Fig. 5.1, the first reachability constraint requires that state S must not be entered before P is entered, (i.e., M_2 is always ready for M_1 in P). This reachability constraint must be guaranteed by the implicit timing constraints used in the implementation. (Note that the soundness of our model relies on the validity of this constraint; see discussion below.) The other three reachability constraints (S not left while in P, R not left while in S and B not left while in R) are fulfilled through the synchronous communication operation following state S. They are verified by the fact that the machines transition synchronously throughout the communication operation and that this operation is unambiguously described by the message specification (μ, l). This same predicate is used in the formal property proofs of the communication operations in both, the sending and the receiving machine.

The discussion of the remaining communication schemes is analogous. For the case of synchronous communication with bilateral synchronization both signals, s and r, exist in the concrete implementation and therefore also in the path predicate abstractions, as do the self-loops in state S and P. The extension towards a full four-phase handshake requires only the self-loops in the communication ending states, B and R. This is justified in the same way as for the unilateral synchronous case.

For the case of asynchronous communication we identify two cases: bilateral synchronization and unilateral synchronization with an implicit timing guarantee. The first case yields a four-phase handshake on the concrete as well as on the abstract level, as mentioned before, and needs no extension. The second case is similar to the synchronous unilateral scheme in the following respect: Instead of having a feedback signal r from machine M_2 to M_1 we have implicit timing constraints (enforced, e.g., through timer circuits or counters) that enable state transitions in machine M_1 only if M_2 is guaranteed to have moved into the corresponding communication states. In other words, the timing constraints enforce the reachability constraints of the four-phase handshake abstraction.

The soundness of the abstract model can also be established, for the considered communication schemes, by the following argument. If the reachability constraints are fulfilled by the implementation then the following construction yields a concrete system which is functionally equivalent with the original design and has a path predicate abstraction with a four-phase handshake: Assume we extend the original concrete implementation of M_2 with an additional output signal r that is evaluated by M_1 as in Fig. 5.4. Obviously, the operations corresponding to the dotted arcs are never triggered if the implementation fulfills the set of reachability constraints. Therefore, the extended implementation which has a full four-phase handshake communication abstraction is functionally equivalent to the original implementation.

In all standard communication schemes, as they are considered here, the extended path

predicate abstractions composed in an asynchronous system with communication through four-phase handshakes, by construction, soundly model the concrete system. It only needs to be ensured that the concrete system indeed matches with one of the described communication schemes. This can be done by a simple manual inspection or can also be automated by going through a formalized check list that examines whether the path predicate abstractions created for the individual modules match with the characteristics of the considered communication schemes. In cases where implicit timing constraints are used in the implementation, the soundness of our model relies on the validity of the timing constraints. In practice, however, these timing constraints are often obvious by inspection because the respective state machines have only very few states and may be always in a state ready for communication.

Fig. 5.5 shows the asynchronous composition of the machines in Fig. 5.1. As can bee seen

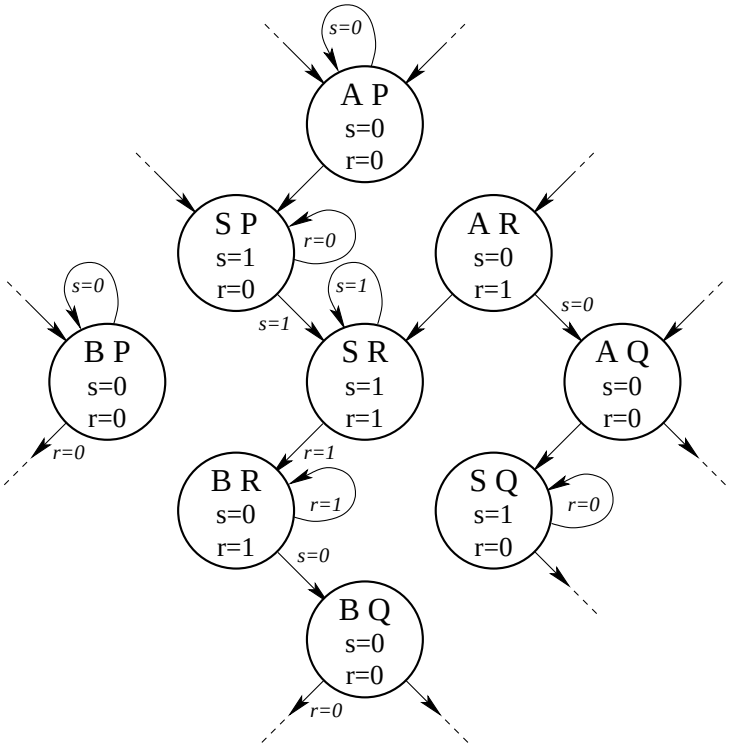


Figure 5.5: Asynchronous composition of machines in Fig. 5.1

from the state transition graph the four-phase handshake between the two machines is capable of modeling a synchronous communication as, for example, in Fig. 5.4: The starting states of the communication operation are *S* and *P*, the ending states are *B* and *Q*. In a synchronous communication the product state *BQ* is reached always some time after product state *SP*. This is reflected in the asynchronous composition of Fig. 5.5: All fair paths leaving *SP* always reach *BR*. (Fairness forbids infinite cycling in *SP*, *SR* or *BR*.)

5.4 Synchronization and wait-stuttering

The composed abstract model contains no concrete timing information. The soundness of the abstract model relies, instead, solely on the corresponding reachability established for each individual PPA with its original concrete module. Without violating this soundness we may

therefore safely ignore any output that does not cause a state change, neither directly nor indirectly, in the composed system.

In this section a categorization of messages is introduced such that for messages of specific *types* the soundness of the model is ensured also when the correspondence required for message abstraction is weaker than what was originally required for message abstraction in Def. 41. Thus, the extensions made in this section enable a stronger abstraction and allows a more flexible abstraction technique.

Consider the effect that an operation starting and ending in states characterized by the same FSM-state predicate, i.e., an operation mapped to a self-transition in the PPA, has on the composed abstract system. As a result of composing the abstract system as an asynchronous product, all possible interleavings of transitioning among the individual modules are modeled. It follows that a self-transition in an individual abstract module will always result in a self-transition in the composed abstract system model. It implies that if an operation is reachable in the system which starts and ends in the same state predicate, i.e., an operation abstracted to self-transition in an abstract module, then in the composed abstract system any number of consecutive repetitions of the corresponding self-transition is reachable. (That number is always finite since we will only consider fair paths.)

Definition 44 [Wait Message]:
A *wait message* is an I/O sequence which only satisfies input sequence predicates of operations starting and ending in states satisfying the same FSM state predicate. □

Fig. 5.6 illustrates the definition of a wait message for a system composed of modules M_1 and M_2. In the system, the message $z = 0$ can only trigger the self-transition of length 1 in P of M_2. The I/O sequences specified by $((\mu_w := z = 0), 1)$ are examples of waiting messages.

From the above discussion it follows that, in the composed abstract system, the effect of a single wait message cannot be distinguished from the effect of any number of consecutive wait messages. We may therefore safely model any number of consecutive wait messages using the same abstract symbol — the set of unique paths in the (over-approximated) abstract system remains unchanged.

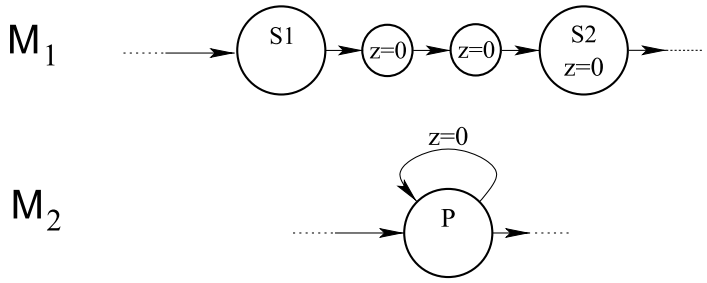


Figure 5.6: Consecutive wait messages

As an example, consider the situation shown in Fig. 5.6 where module M_1 produces $z = 0$ during an operation of length 3 from S_1 to S_2. According to Def. 41, the mapping of this "long message", $((\mu_{tot} := z = 0 \wedge \mathsf{X}(z = 0) \wedge \mathsf{X}^2(z = 0)), 3)$, cannot be mapped to the same symbol as used for the "short message", $((\mu_w := z = 0), 1)$ which triggers the self-transition in M_2. We also cannot use a different symbol because the long message "contains" the short message, i.e., μ_w and μ_{tot} would not characterize disjoint sets of I/O sequences. In order to create a legal

abstraction the long operation would therefore have to be split up into several shorter operations such that each produces the short message.

The extension to be introduced in this section allows for a more flexible mapping where such an abstraction is possible. The short message is a wait message, and the long message can be viewed as several consecutive short messages. In the abstract system, the effect of several consecutive wait messages cannot be distinguished from the effect of a single occurrence. In such a case, we will therefore allow the long message to be mapped to the same abstract symbol as used for the short message.

Notice, that the wait message is not completely disregarded. We may only simplify our model by representing any fixed number of repeated wait messages by a single occurrence. A complete disregard of a wait message could make the self transition it triggers unreachable and thereby hide the output of the self transition which in turn could possibly trigger an actual state change in its neighboring modules. Such a disregard could therefore change the reachable product space of the composed system and would make the abstraction unsound.

The above situation does, however, not occur for many typical wait messages. That is, no chain effect occurs in the sense that, the wait message cause only self-transitions whose produced output does not causes state changes in neighboring modules. To define such a *strict* wait message we first define a *waiting operation*.

Definition 45 [Waiting Operation]:
A waiting operation is an operation where the start state predicate is the same as the end state predicate and whose output is either not used as a trigger for any operation in another module or it only triggers waiting operations in other modules. □

Definition 46 [Strict Wait Message]:
A *strict* wait message is an I/O sequence which only satisfies input sequence predicates of waiting operations. □

Based on the above classification of messages the abstraction requirements will now be weakened. We will allow any finite number of consecutive wait messages to be represented by the same abstract symbol, and for strict wait messages, we allow an abstraction where the occurrence of such a message is fully ignored.

The abstraction of messages was described in Def. 41 by a one-to-one mapping between message specifications and abstract message symbols. The message specifications are I/O sequence predicates paired with the length of the operation (cf. Def. 40). According to Def. 41, operations of different lengths can therefore not be described using the same message symbols. In the following we replace Def. 41 by the weakened message abstraction of Def. 47, which, under specific circumstances in the presence of wait messages, may allow I/O sequences in operations of different lengths to be abstracted using the same abstract symbols.

To simplify later notation we let $stutter((\mu,l),k)$ denote the I/O sequence predicate, $\mu \wedge \mathsf{X}^{l}(\mu) \wedge \mathsf{X}^{2\cdot l}(\mu) \wedge \ldots \wedge \mathsf{X}^{(k-1)\cdot l}(\mu)$, characterizing k consecutive repetitions of μ for length l.

Definition 47 [Message Abstraction weakened for Wait Stuttering]:
Let β_j be the bijective function of Def. 41 mapping message specifications to abstract message symbols, let μ_{tot} and μ_{msg} be arbitrary I/O sequence predicates, and let μ_w and μ_{sw} be I/O sequence predicates characterizing only wait messages and strict wait messages, respectively, then a surjective message abstraction function F^{msg} exists such that $F^{msg}((\mu_{tot},l)) = \hat{y}$ implies one of the following:

1. $\beta_j(\mu_{tot},l) = \hat{y}$

2. $\beta_j(\mu_w, q) = \hat{y}$ and
   $(\mu_{tot}, l) = stutter((\mu_w, q), i)$ and
   $l = i \cdot q$
3. $\beta_j(\mu_{msg}, a) = \hat{y}$ and
   $(\mu_{tot}, l) = stutter((\mu_{sw}, q), i) \wedge \mathsf{X}^{q \cdot i}(\mu_{msg}) \wedge \mathsf{X}^{a+q \cdot i}(stutter((\mu_{sw}, r), j)$
   and $l = i \cdot q + a + j \cdot r$

□

The first form of the message abstraction is the original form of Def. 41. It is actually contained in the third form as a special case (when $i = 0$ and $j = 0$) but is kept here for clarity. The second form describes a finite number of repetitions of a wait message μ_w, mapped to an abstract "wait" symbol. The third form describes a message with three "phases": the first phase is a (possibly empty) stuttering of strict wait messages, followed by a single instance of a non-wait message, followed by a (possibly empty) stuttering of wait messages. These three phases together are abstracted into the single abstract message symbol $\hat{y}$ corresponding to the non-wait message.

The message abstraction requirements can be weakened further by taking predecessor and successor operations into consideration. If all predecessor operations have an output mapped to the abstract symbol for any stuttering of a wait message, and this same wait message is further stuttered in a prefix of the current operation, then in the current operation this message can be ignored, i.e., $\mu_{tot} = stutter(\mu_w, a) \wedge \mathsf{X}^a(\mu_{msg})$ could be mapped to $B((\mu_{msg}, k))$. Analogously, if the current operation has a suffix where it stutters over a wait message and all successor operations have an output mapped to the abstract symbol representing any stuttering of this wait message then this suffix can be ignored.

In practice the form of message abstraction relates to elements of the standard communication schemes (cf. Sec. 5.2). Signals for synchronization are abstracted using stuttering of message specifications as introduced in this section (case two and case three of Def. 47) while the output of data exchange operations are abstracted using the unmodified message abstraction function (case one of Def. 47).

5.5 Model checking on abstract system

The abstract system model is an asynchronous product of path predicate abstractions. The concrete system model we consider here can either be a standard synchronous product of finite state machines if all design modules share a common clock or an asynchronous composition in the form of Def. 43 if the design modules communicate asynchronously with each other. We here consider LTL model checking. A Kripke model $\hat{K} = (\hat{S}_K, \hat{I}_K, \hat{R}_K, \hat{A}_K, \hat{L}_K)$ is derived from an asynchronous composition $\hat{M}$ in the following way. Let $\hat{M} = (\hat{S}, \hat{I}, \hat{X}, \hat{Y}, \hat{T}, \hat{\lambda})$ be the asynchronous composition of n path-predicate-abstracted FSMs $\hat{M}_i$, $1 \leq i \leq n$. For the Kripke model $\hat{K}$ the set of states is $\hat{S}_K = \hat{S} \times \hat{X}$, the set of initial states is $\hat{I}_K = \hat{I} \times \hat{X}$ and the transition relation is $\hat{R}_K \subseteq \hat{S}_K \times \hat{S}_K = \{((\hat{s}, \hat{x}), (\hat{s}', \hat{x}')) \,|\, (\hat{s}, \hat{x}, \hat{s}') \in \hat{T}\}$. The set of atomic formulas, $\hat{A}_K$, is composed from subsets of atomic formulas, one for each component of the system. The subset for component $\hat{M}_i$ comprises the state set $\hat{S}_i$, the input alphabets $\hat{X}_i$ and the output alphabet $\hat{Y}_i$ of that component: $\hat{A}_i = \hat{S}_i \cup \hat{X}_i \cup \hat{Y}_i$. The overall set of atomic formulas for the system's Kripke model is $\hat{A}_K = \bigcup_{i=0}^{n} \hat{A}_i$.

For compositional path predicate abstraction, soundness can be established only in the one

direction — a proof of a property based on the abstract model ensures that the corresponding property is valid for the concrete model, but not vice versa. Due to the asynchronous composition the abstract composed system model over-approximates the possible interleavings of operations. In other words, the paths reachable in the abstract composed system represent a superset of the sequences of operations that are actually executable in the concrete composed system.

The soundness of the asynchronous composition with respect to LTL relies on the fact that every path in the concrete system is represented by a path in the abstract system. This means that if there is a counterexample for a property in the concrete model then there also exists a counterexample on the abstract model. (An existential operator does not exist in LTL, the over-approximation by asynchronous composition is therefore conservative for all properties.)

Note that a property in the abstract model may fail if the corresponding property on the concrete level only holds for a specific processing speed in between synchronization events. In the asynchronously composed abstract model all possible component speeds and the resulting interleavings of operations are represented, including the specific set of speeds of the concrete system. Therefore, not all initialized (i.e., reachable) abstract paths must have a corresponding concrete path. As a result, only system behavior that is *speed independent*, i.e., it does not depend on the processing speed of the individual modules, can be proven on the abstract level.

We may also consider the degenerate case that *none* of the abstract paths described by an LTL formula have a correspondence with initialized concrete paths. Note that such a formula cannot hold on the abstract model. Since every initialized concrete path corresponds to some initialized abstract path, it follows in the considered situation that at least one initialized abstract path exists which is not included by the formula, thus, representing a counterexample. (Remember, that an LTL formula holds iff it includes all initialized paths of the model.)

While such a formula by itself cannot be proven as a property of the abstract model it can be included as an assumption sub-formula of a global formula such that the global formula holds on the abstract model. Since the abstract assumption has no corresponding initialized concrete paths, the corresponding concrete global formula has a “vacuous” assumption sub-formula. Note that, although the abstract formula has a non-vacuous assumption, the corresponding concrete assumption is vacuous. This reflects the fact that the abstract property has no meaning in the concrete system. Formally, however, the concrete property holds trivially (“ex falso quod libet”). Therefore, also in this corner-case the soundness of compositional path predicate abstraction is fulfilled.

In practice, model checking tools would flag such a trivially fulfilled property as vacuous. Therefore, in [58] a specific set of “allowed LTL formulas” was defined to exclude such properties beforehand.

Theorem 5 [LTL Soundness of Composed Model]:
Consider an LTL formula $\hat{\varphi}$ for the abstract model and the corresponding LTL formula φ for the concrete model as obtained by applying the translation rules of Tab. 4.1. If the formula $\hat{\varphi}$ holds on the abstract model then the formula φ holds also for the concrete model. □

Proof. We state the proof for the communication scheme of synchronous communication on the concrete level. For the other communication schemes the proof is similar.

Consider an arbitrary (possibly infinite) path π from the initial state in the concrete system. In every product state on the path an arbitrary number of machines may be in a respective important state. In a synchronous communication operation, the sender and the receiver begin

and end the communication in synchronized start and end states, i.e., at product states that are important in both, sender and receiver. We now split up the path π into fragments between such synchronized states. A communication fragment is one that begins at a communication start state and that ends at a communication end state. Obviously, this fragment has a corresponding abstract path fragment, because the start and end states, by construction (cf. Sec. 4.2 and Sec. 5.3), have corresponding sets of product states in the asynchronous composition.

A non-communication fragment is a path fragment that begins at the end of some communication and that ends at the beginning of the next communication. In between these states the machines do not communicate and the specific product states occurring on the path fragment are a result of the specific speed at which each module actually runs in the implementation. The asynchronous model abstracts from concrete timing and represents all possible interleavings of operations in the two modules. In every individual module, an operation on the concrete level always corresponds to an abstract transition. Hence, for such a concrete path fragment at least one abstract path fragment exists.

Each path in the concrete and in the abstract machine consists of one or more path fragments and represents a sequence of communications, in the concrete and in the abstract model, respectively. As a result of the asynchronous composition, the sequence of communications contained in the abstract model is a superset of the sequence of communications in the concrete model. Therefore, every path in the concrete system beginning at the initial state has a representation in the abstract system. If a concrete LTL formula does not hold on a specific path leaving the initial state in the concrete model then there is an abstract path leaving the initial state where the abstract LTL formula is violated, also. □

As a last observation in this section, we note that our framework has to be based on LTL formulas rather than CTL. In LTL, all expressible properties can be understood as expressions within a universal operator, i.e., properties expressing a meaning in the form “for all paths ...”. In contrast, CTL formulas can also existentially quantify over paths. Since not *every* abstract path must have a concrete counterpart, the asynchronous composition, in general, is not a sound model with respect to CTL.

Chapter 6

Practical Methodology

This chapter demonstrates how path predicate abstraction (PPA) can be used in practice to create a sound, yet highly abstract, system-level model from RTL implementations. First, Sec. 6.1 summarizes the main steps taken in our approach for creating a PPA. Then, in Sec. 6.2, an example design is introduced for which an abstraction is created in Sec. 6.3 and Sec. 6.4. Finally, Sec. 6.5 shows the results from two case studies where the presented approach for PPA has been applied to industrial implementations.

The approach described in this section is bottom-up and is based on extensions to the verification methodology by C-IPC, as described in Sec. 3. This bottom-up approach is applicable, if only an RTL design is given and no system-level model is yet available. A top-down approach integrating PPA into a design flow starting from a system level model will be subject of Sec. 7.

6.1 Abstraction Flow

The basic steps of the proposed bottom-up abstraction flow are as follows:

1. Identify all operations of the design. (Operations are finite sequences of behavior between important control states of the design, cf. Def. 19 in Sec. 3.1.) Write macro definitions in the used property checking language for the following objects:
   - State predicates describing the important control states between the operations of the design
   - Input sequence predicates (trigger conditions)
   - Output sequence predicates (output messages)
2. Write an operation property in the form of an operation property template as described in Def. 37. Assumption and commitment must both be expressed solely in the macros defined above. Formally verify the property.
3. Formally check the completeness of the property set as described in Sec. 3.3.
4. The abstract model is obtained by regarding the *macro names* as abstract state, input and output symbols as in Sec. 4.3.

Step 1, finding the operations of a design, is a creative step with a certain degree of freedom. In practice, the intended operations are usually obvious from the specification or from the design itself. However, the choice of operations determines the level of detail of the abstract model obtained in the end. The flow above normally involves several iterations to refine the macros and properties until a complete set of properties holding on the design has been found. Every

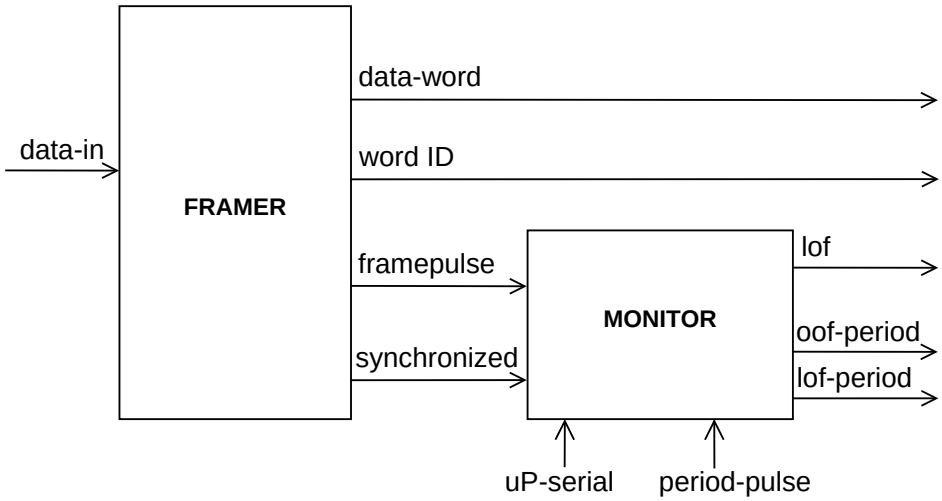


Figure 6.1: Block diagram of example system

property is individually proven. Since the complete set of properties fully specifies the expected behavior of the design, any deviation from this will be detected and can be fixed. We illustrate the abstraction flow by means of an example and show how the macros and properties relate to the theoretical notions introduced earlier.

The example design is inspired by a SONET/SDH framer circuit from Alcatel Lucent that has been subject to a comprehensive case study as will be reported in Sec. 6.5. For the illustrations of this section, we created a simplified version of the industrial design. For this simplified version, the VHDL source code as well as the SVA properties and the abstract model in Promela are available at *http://www.eit.uni-kl.de/eis/forschung/ppa.*

6.2 Example Design

SONET and SDH are two very similar transport protocols developed for transferring large amounts of data with minimal latency. Data is structured and sent in *frames* where each frame can have a complex hierarchical structure of payload and header information. For our example we only look at one such header field, the *frame marker*. The frame marker is a constant data string used to mark the start of a frame.

Data frames are sent at a fixed frequency, however, no explicit timing information is transferred alongside the data. Small differences in the clock speeds of sender and receiver can cause loss of synchronization. Common signal distortion may also lead to data incoherencies. In this non-ideal, physical environment it is the task of a framer to synchronize with the incoming data stream.

For our example we consider a simplified protocol where a frame consists of 64 data words with a width of 8 bits each. The frame marker is the 16-bit string 'A738' (hex) and constitutes the first and second word of every frame.

Fig. 6.1 is a block diagram of our system consisting of a framer and a monitor. The framer synchronizes with the incoming bit stream, signal *data-in*, by comparing actual position of the frame marker with its expected position. The outgoing signal *data-word* is simply a copy of *data-in*, re-aligned to match the actual occurrence of the frame marker. Each word is assigned

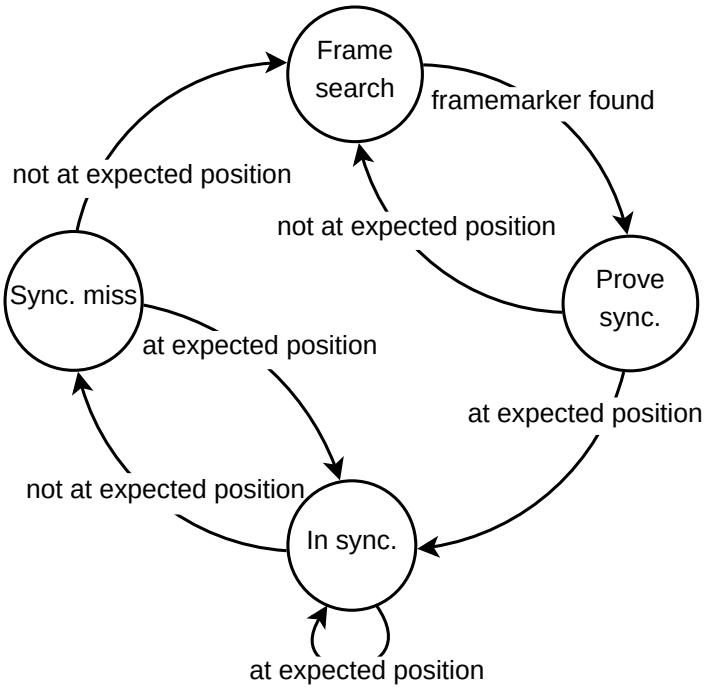


Figure 6.2: Conceptual State Machine of the Framer

a unique *word ID* that identifies it within its frame. The current state of the synchronization process is output by the *synchronized* flag indicating whether or not outgoing data words are considered to be correctly synchronized.

On a positive clock edge the framer receives 8 bits of the bit stream, latched in from the bit stream by a shift register external to this design and operating at 8 times the clock speed. Note that these 8 bits are possibly unaligned to the words of the frame, i.e., they may contain 8 bits which are part of two different words.

Fig. 6.2 explains the main control flow of the framer as we could expect to find it documented in the specification sheet of such a module. We will refer to it as the Conceptual State Machine (CSM) of the design.

The framer is implemented in VHDL. Subtasks were identified and then realized as dedicated VHDL processes. Implementing a module in this way, as the product machine of several processes, is common practice for hardware descriptions. The following short discussion of the implementation should suffice as background when we later consider the abstraction mechanisms.

The incoming data, *data-in*, is buffered by a shift register. The width of this register is chosen such that any bit string containing the frame marker can be identified. One VHDL process implemented is concerned with identifying any occurrence of the frame marker within this shift register and reporting at what offset the marker was found. Note that the frame marker can be found at 8 possible offsets since it receives 8 bits of the bit-stream every cycle. The found offset is used to align data words from the incoming bit stream.

We also implemented a process that associates a *word-ID* with the outgoing data words. The frame marker itself is two words wide. These words have the word IDs 0 and 1. For each subsequent word in the frame the word ID is incremented by 1.

At last we have a process checking for synchronization hits and setting the mode of op-

eration according to Fig. 6.2. A synchronization hit is a frame marker found at the expected alignment and at the expected cycle according to the protocol, i.e., 64 cycles later and at unchanged alignment.

For illustrating compositional PPA we introduce this framer as a module in the simple system shown in Fig. 6.1. An error reporting unit, the monitor, communicates with the framer. The monitor calculates various performance metrics such as error rates based on this communication. (Through a serial interface, uP_serial, towards a microprocessor some of the constants of this calculation can be changed.)

Loss of frame, *lof*, is set for consecutive frames out of synchrony and reset by consecutive frames in synchrony. Configuration registers, *lof-set* and *lof-reset* are used, respectively, to define the number of consecutive frames. A period is defined as the interval from one *period-pulse* to the next. Out of frame period, *oof-period* is set if any frame within the last period was out of frame. Similarly, loss of frame period, *lof-period*, is set if lof was set within the last period.

6.3 Obtaining a complete property set

From a practical point of view, finding a PPA means formalizing a Conceptual State Machine (CSM) as an abstract system model. A first candidate for such a CSM is obtained by conceptualizing over the design as it is described in the specification sheet.

In order to establish a PPA the edges of the CSM are formalized as operations. Proving the existence of an operation is done by writing and proving operation properties for the concrete design. In the following we show how operation properties are created to establish a PPA as defined in previous sections.

We used the CSM in Fig. 6.2 to explain the behavior of the design. The same CSM will now be used as a starting point to create a PPA. We consider the edge from “prove synchronization” mode to “in synchrony” mode. The edge is formalized as an operation property and can be checked against the concrete implementation.

```
1  property proven_sync is
2  assume:
3    at t: PROVE_SYNC;
4    at t: MarkerIn;
5  prove:
6    at t+1: OutOfFrame64;
7    at t+1: FramePulse64;
8    during[t+1,t+63]: not ImportantState;
9    at t+64: SYNC;
10 end property;
```

We have used a pseudo property language for illustration purposes. Of course, the same property could also be expressed using standard property languages.

Note that although the operation is a property evaluated on the concrete design it is described in terms of abstract objects. The above operation describes an abstract transition from the abstract state *PROVE_SYNC* to the abstract state *SYNC* under the trigger condition *MarkerIn*, producing as abstract output symbol the pair (*OutOfFrame64*, *FramePulse64*). This can be done since the abstract objects are actually macros of the property language encapsulating its relation to the actual concrete design. The state predicates, η, input sequence predicates, ι, and

output sequence predicates, μ, of our formalisms in Sec. 4 and Sec. 5 are realized through these macros. They define the semantics of the abstract objects in terms of concrete RTL signals. The name of the macro is used as abstract symbol while the content defines the corresponding predicate.

The above property holds if the concrete module satisfies the following. If we start at any state characterized by the macro *PROVE_SYNC* and receive any concrete input sequence characterized by the macro MarkerIn, then, 64 clock cycles later we are in some state characterized by the macro *SYNC*. The macros *OutOfFrame64* and *FramePulse64* characterize all possible output sequences in this interval. In the property these macros are "anchored" at t+1, (i.e., the first state of the respective sequence predicate is assumed at $t+1$), but since they describe sequence predicates of length 64 they do characterize the entire interval of the operation. (If this were not the case the completeness check of the property set would fail.) According to our formalism we must also ensure that no other important state is visited during this interval. An explicit check, as seen above, is used to ensure this. The macro *ImportantState* describes all important states, Ψ. It is created as a disjunction over all abstract state macros.

```
1  macro PROVE_SYNC : boolean :=
2    not next(prev_miss,2) and not next(synchronized) and
3    next(frame_cnt,2)=3 and
4    not next(synchronized) and
5    not next(prev_miss,2) and
6    (not next(sync_hit) or not next(synchronized,2));
7  end macro;
```

The macro above defines the abstract state symbol *PROVE_SYNC* and its corresponding important-state predicate. The predicate characterizes a set of concrete important states. The identifiers used inside the macro are the RTL identifiers from the VHDL implementation. Note that we allow previous and future values of state variables to be used in the predicate. This is a useful technical extension of our formalism that allows to succinctly capture a current-state set by implicitly taking into account the unrolled transition relation of the circuit model for resolving past and future state variable references. The same state set as characterized by *PROVE_SYNC* could also be described using only present-state variables; however, this would result in a much more complex macro definition.

```
1  macro MARKER_C : unsigned := 'A738'hex; end macro;
2
3  macro InputHist : unsigned :=
4   (prev(data_in,2)&prev(data_in)&data_in);
5  end macro;
6
7  macro MarkerIn : boolean :=
8    not prev(reset_n)=0 and (
9      if prev(reset_n,2)=0 then
10       prev(data_in)&data_in = MARKER_C;
11     else
12       InputHist(15 downto 0)=MARKER_C or
13       InputHist(16 downto 1)=MARKER_C or
14       ...
15       InputHist(22 downto 7)=MARKER_C;
16     end if;
17   );
18 end macro;
```

The abstract input *MarkerIn* characterizes the set of concrete input sequences for which a frame marker is found. Macros *MARKER_C* and *InputHist* are used only to enhance readability.

Similarly, abstract output symbols *FramePulse64* and *OutOfFrame64* are created using macros. They encapsulate sequence predicates of length 64.

```
1  macro FramePulse64 : boolean :=
2    next(framepulse,1)=1 and
3    next(framepulse,2)=0 and
4    next(framepulse,3)=0 and
5    ...
6    next(framepulse,64)=0;
7  end macro;
8
9  macro OutOfFrame64 : boolean :=
10   next(synchronized,1)=0 and
11   next(synchronized,2)=0 and
12   next(synchronized,3)=0 and
13   ...
14   next(synchronized,64)=0;
15 end macro;
```

So far, all state information has been encoded into macros corresponding to our important state predicates. This makes the relation to our formalism straightforward. Sometimes it may, however, be beneficial to allow additional state information to be stored as state variables. By doing so we can separate what we consider control state and what we consider data path state.

In our example, the operation property “proven_sync” as shown earlier does actually not hold on our RTL implementation as is. The property checker returns a counterexample where a frame marker comes at the correct cycle, but with an alignment differing from previous alignment. Correct synchronization depends on both, correct timing and correct alignment of the frame marker. Therefore we need to refine our CSM such that also the alignment is represented. We could do this by creating new abstract state macros, e.g., *PROVE_SYNC_Align0*, *PROVE_SYNC_Align1*, etc. However, a better approach is to separate this information out by adding an abstract state variable called *Alignment*:

```
1  macro Alignment : unsigned :=
2    next(prev_alignment,2);
3  end macro;
```

We also create new abstract input symbols reflecting the position of an eventual frame marker hit. A single macro encapsulates eight abstract symbols, one for each possible alignment.

```
1  macro MarkerPosition : unsigned :=
2    if    InputHist(22 downto 7)=MARKER_C then 7;
3    elsif InputHist(21 downto 6)=MARKER_C then 6;
4    ...
5    elsif InputHist(16 downto 1)=MARKER_C then 1;
6    else  0;
7    end if;
8  end macro;
```

The refined operation property, “proven_sync”, now holds for the concrete implementation.

```
1  property proven_sync is
2  assume:
```

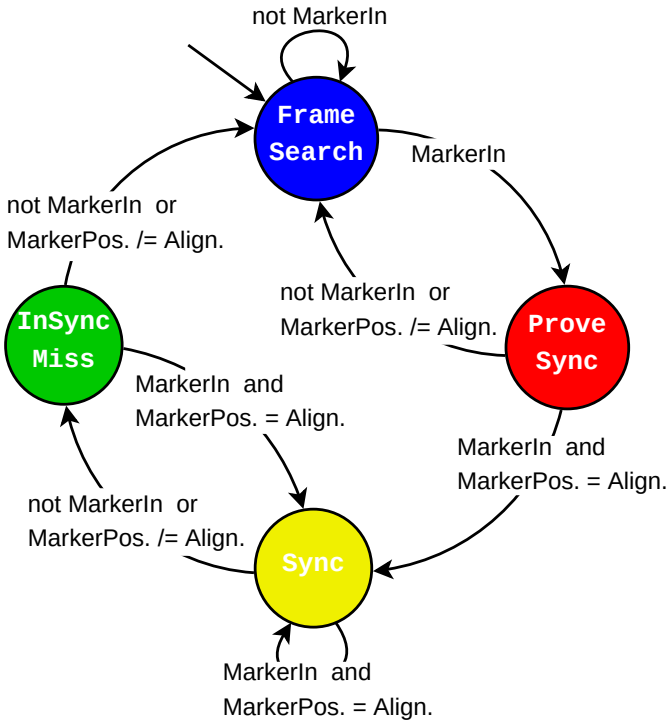


Figure 6.3: Formalized Conceptual State Machine of the Framer

```
3    at t: PROVE_SYNC;
4    at t: MarkerIn;
5    at t: MarkerPosition = Alignment;
6  prove:
7    at t+1: OutOfFrame64;
8    at t+1: FramePulse64;
9
10   during[t+1,t+63]: not ImportantState;
11   at t+64: Alignment = Alignment @ t;
12   at t+64: SYNC;
13 end property;
```

Note that the abstract state is now defined both by our abstract state macros, referred to as conceptual states, and the value of *Alignment*. We must therefore also determine *Alignment* together with the conceptual ending state of the operation (at t+64). (The completeness check of Sec. 3.3 automatically ensures that all such state information is determined.)

Note that describing the abstract state by two different macros, *Alignment* and *SYNC*, is merely a technical extension that allows us to concisely describe a set of operations using a single property. For example, the above property corresponds to 8 operations in our formalism: One from *PROVE_SYNC_Align0* to *SYNC_Align0*, another from *PROVE_SYNC_Align1* to *SYNC_Align1*, etc.

In the same way as just described for property “proven_sync”, a complete set of operation properties has been formulated, proved and checked for completeness. The abstraction shown in Fig. 6.3 is obtained as a result of this verification.

The concrete and the abstract model could be viewed as sequentially equivalent if the abstraction functions for inputs and outputs given through the macros, e.g., MarkerIn, MarkerPos

and OutOfFrame, are included in the definition of equivalence. Even if an equivalence checker was available to handle this, finding the appropriate mappings is not an easy task and would require a similar operational design view as in the approach presented here. Instead, we propose to create the abstraction step by step by formulating and proving operation properties. This also has the added advantage that the global correctness proof is split into a number of local proofs.

We have now described the abstraction of the framer in some detail. Before we discuss the composition of our abstract system we quickly summmarize how the second module in our composed system, the monitor, was abstracted.

The writing of configuration registers through the microprocessor interface was verified but will not be considered in our system. We rather assume the configuration registers to remain unchanged during system operation. The generation of *lof*, (loss of frame) flag is independent of the algorithms for setting *OOFperiod* and *LOFperiod*. We therefore create one abstract FSM for the *LOF* generation and one for the *OOFperiod* and *LOFperiod* generation. Since these concrete functionalities are almost trivial, their abstractions do not differ much from their corresponding concrete FSMs. Their abstraction will be understood from the following discussion on compositionality.

6.4 Creating a compositional abstraction

In this section we consider the abstraction of the communication. For global input and output (i.e., input and output of the system), obviously, no such additional considerations are necessary. The bit stream is a concrete global input of the considered system. The abstraction of this global input was created to reflect how it is interpreted in the framer. From the many possible scenarios of the incoming bit stream we have created abstract input symbols that enumerate the triggers of the control path. By characterizing the bit stream, *data-in*, we created two abstract input signals, *MarkerIn* and *MarkerPosition*. The values of the abstract input signals correspond to the abstract symbols of our formalism. *MarkerIn* is a Boolean signal that is set when a frame marker is found. *MarkerPosition* identifies the alignment where the frame marker is found (value range from 0 to 7).

The communication within the system requires special consideration. The definitions for compositional PPA from Sec. 5 lead to a set of additional criteria that need to be ensured for the soundness theorem to hold for systems of PPAs. In essence these are restrictions on the input/output abstraction ensuring a one-to-one relation between concrete communication and abstract messages.

Within our system the framer and the monitor communicate using the concrete signals *synchronized* and *framepulse*. We showed the operation “proven_sync” of the framer earlier. Output message symbols *FramePulse64* and *OutOfFrame64* were used to abstract these signals. These symbols correspond to sequence predicates of length 64, the entire length of the operation. We follow this procedure for all operations and obtain a total of eight output symbols, e.g., *FramePulse1*, *not_FramePulse1*, *FramePulse64*, etc. For every operation we now have a single symbol that uniquely represents an entire output sequence.

Let us now consider the composition of our abstract system, i.e., how we can use the same message symbols as input symbols of the monitor. The concrete monitor has a simple behavior: at a *framepulse* it reads in the value of *synchronized* and accordingly updates a counter. For compositionality the PPA of the monitor should use the eight output symbols of the framer

as input symbols (obviously, the symbols also need to represent the same concrete interpretation/predicate). Creating such an abstraction may be possible but is not very convenient. Considering predicates of length 64 to describe this simple behavior would be cumbersome and result in a questionable abstraction not representing a natural view on this module. Instead, let us rather change our message symbols so that they describe the concrete situations in the way the monitor interprets them. Our new symbols describe only three situations. No frame pulse is present:

```
1 macro NoFramePulse : boolean :=
2   not next(framepulse);
3 end macro;
```

Frame pulse and inactive *synchronized*:

```
1 macro OutOfFrame : boolean :=
2   next(framepulse) and next(synchronized)=0;
3 end macro;
```

Frame pulse and active *synchronized*:

```
1 macro NotOutOfFrame : boolean :=
2   next(framepulse) and next(synchronized)=1;
3 end macro;
```

In the monitor, *NoFramePulse* will trigger waiting operations only. In Sec. 5.4 we redefined the abstract output function to allow such stuttering. We will now make use of this stuttering extension and prove the output in terms of the macros above. For “proven_sync” the changed property becomes:

```
1  property proven_sync is
2  assume:
3    at t: PROVE_SYNC;
4    at t: MarkerIn;
5    at t: MarkerPosition = Alignment;
6  prove:
7    at t+1: NotOutOfFrame;
8    during[t+2, t+64]: NoFramePulse;
9
10   during[t+1,t+63]: not ImportantState;
11   at t+64: Alignment = Alignment @ t;
12   at t+64: SYNC;
13 end property;
```

We use form (3) of the definition of an abstract symbol as described in Sec. 5.4, i.e., the one-cycle message *NotOutOfFrame* followed by a 63 cycle stuttering of *NoFramePulse*. The abstract output value is the message symbol *NotOutOfFrame*.

For the soundness theorem to hold the abstract communication must also adhere to one of the synchronization schemes of Sec. 5.2. Our concrete system modules share a common clock. The communication between the framer and the monitor is abstracted into a unilateral synchronization followed by a message transfer. The monitor waits as long as it receives the *NoFramePulse* symbol. The two other abstract symbols, *OutOfFrame* and *NotOutOfFrame*, represent all non-waiting messages appearing in the concrete system. All communications happening in the concrete system are described based on only these messages and match the communication scheme of unilateral synchronization as described in Sec. 5.2. Therefore, the abstract synchronization by a four-cycle handshake is sound by construction.

We have created a system of compositional PPAs from an RTL implementation. For this system model Theorem 5 on LTL soundness for compositional path predicate abstractions holds. System-level properties proven on the abstraction are valid proofs for system-level properties of the actual implementation. A PPA is an FSM that can be implemented in any standard language and can used for verification purposes based on formal methods as well as based on simulation.

6.5 Experimental Results

Besides exercising the proposed methodology on our own circuit example as described above, we also evaluated the approach on two industrial RTL designs.

The first design is an implementation of the on-chip bus protocol *Flexible Peripheral Interconnect (FPI) bus*, owned by Infineon Technologies. We chose the FPI bus implementation since it is a highly optimized design realizing a complex communication protocol whose correctness of implementation can only be evaluated when considering the system as a whole.

A second case study was conducted on a SONET/SDH framer implementation from Alcatel-Lucent. The source code spans 27,000 lines of RTL code. It is highly configurable so that the module can be used in a wide range of SONET/SDH applications differing in various parameters, e.g., the data rate of the analyzed input stream. Different configurations are synthesized into circuits that differ substantially with respect to their logic functionality. High configurability is implemented using a comprehensive set of library functions and modules. With this case study we intend to demonstrate that our method copes efficiently with different configurations and that it allows to create abstract modules possessing the same re-usability as the implementation.

6.5.1 Case Study: Flexible Peripheral Interconnect (FPI) bus

The FPI bus is a modular system — every master and slave interfaces the bus using a dedicated master agent or slave agent, respectively. For our experiments, we constructed a system where two peripherals act as masters and two simple memories act as slaves. As an implementation of such a system we instantiated the FPI bus with two master agents and two slave agents.

Module	#inputs	#outputs	#state bits	#LoC
MasterAgent	199 / 17	202 / 13	292 / 17	3568 / 91
SlaveAgent	199 / 10	202 / 5	292 / 7	3568 / 49
BCU	258 / 9	215 / 4	941 / 14	8966 / 41

Table 6.1: Sizes of concrete/abstract system components

The modules of the FPI bus are listed in Table 6.1. For each module we created path predicate abstractions, as described in the previous sections, based on C-IPC. A total of 32 properties, 51 macros and 3 environment constraints were developed in 1850 lines of code (LoC). The property checker OneSpin 360 MV [44], running on an Intel Core i7 CPU 860 at 2.8 GHz, proved all properties in 1 minute and 30 seconds. The property describing block transfer operations had the longest run time with 25 seconds. The size of each module in its

concrete and path predicate-abstracted version is shown in Table 6.1. The last column shows LoC in VHDL for the concrete model and LoC in Promela for the abstract model. Promela was chosen since it is the input language for Spin [31] — a model checker using the same asynchronous composition for modeling systems as specified for compositional PPA in Def. 43.

The effort to create a complete set of properties for SoC module verification is estimated to be around 2000 LoC per person month [44]. This matches also with our experiences in this project. Since the abstract model results from these properties based on macro coding conventions, as was illustrated in this chapter, the additional effort for creating the abstract model is small. Coding conventions similar to the ones described are, anyways, considered "good practice" among verification engineers.

In the concrete implementation, MasterAgent and SlaveAgent are designed as functions of a single agent module which can be configured at system start-up to act as an agent for either the master or the slave. Our abstractions of the MasterAgent and SlaveAgent, however, cover only the relevant behavior of either role. We estimate that the module configured as SlaveAgent covers about 2/3 of the inputs, outputs and state bits listed in Tab. 6.1, while the module configured as MasterAgent covers nearly all of them. Similarly, for the Bus Control Unit (BCU) we ignored certain features like debugging mode and starvation protection, and abstracted only the behavior relevant for address decoding and bus arbitration. We estimate that about 1/3 of the BCU inputs, outputs and state bits contribute to these functions. But even taking this into account, our composed abstract system soundly models concrete system functions involving approximately 750 inputs, 750 outputs and 1300 state bits. Proving global safety and liveness properties on a concrete system of this size is clearly beyond the capacities of today's model checking technology.

Note that separating out the considered system functionality is practically impossible in the highly optimized concrete system. It is an advantage of our methodology that it can selectively extract functionality which is in the focus of the intended verification plan.

In our experiments, it was our goal to verify that the FPI bus conforms to its protocol in any environment, independently of the specific system into which it is integrated. Therefore, we implemented a peripheral which behaves in as general a way as possible. In order to achieve this, we exploited the non-determinism native to Spin and implemented a device that randomly makes requests from a set containing all types of FPI bus transactions.

Spin features a number of pre-defined checks that can be run on a Promela model. In particular, since we model the communication between modules using the Promela notion of a channel, Spin globally proves freedom of deadlock in the composed system. In our experiments this global proof took 3 minutes of CPU time on a state-of-the-art PC using 3 GB of main memory.

Moreover, we formulated a representative LTL property of type $\mathsf{G}(a \Rightarrow \mathsf{F}c)$ to verify that a requested transaction is executed correctly in the system. This and similar properties to verify the correctness of transactions can be proven in about 1 minute using 0.5 GB of main memory.

6.5.2 Case Study: SONET/SDH Framer

For an introduction to the functionality of a framer, please refer to Sec. 6.2 where a simplified framer implementation was created to demonstrate our practical methodology. For the industrial design, the top-level VHDL entity was verified by two complete property sets, thereby creating two abstract modules. The first module, called *main*, comprises all functionality belonging

to the actual framer. The second module, called *monitor*, computes performance data of the framer.

The behavior described by the *main* module is distributed over all VHDL components of the design spanning 27k LoC. The size of the module depends on the configuration parameters: The number of inputs varies between 132 and 549, the number of outputs between 88 and 280, and the number of state bits between 4054 and 47213. The large state space is due to input stream buffering. The *monitor* functionality is contained within a subset of the components spanning 850 LoC. The module has 20 inputs, 6 outputs, and 30 state bits.

The property suites for both modules were set up to be configurable with the configuration parameters of the design. Parametrization was confined to the macro definitions so that the top-level properties defining the abstraction were generic and parameter-independent. We were able to formally verify all configurations of the design using the same property suite. As a consequence, the constructed PPA is the same for every configuration of the actual design.

Table 6.2 compares the size of the largest design configuration with that of the abstract model.

Module	#inputs	#outputs	#state bits	#LoC
Framer	549 / 9	280 / 7	47213 / 11	27k / 122
Monitor	20 / 13	6 / 3	30 / 12	850 / 80

Table 6.2: Sizes of concrete/abstract system components

Despite the fact that the abstract models are very compact, they contain all information for describing the behavior relevant at the system-level. Complex functionality such as re-alignment of the input stream is fully verified by the property suite creating the abstract model, but it is implementation-level behavior that is not relevant for the interaction of the module with its system environment and is therefore not included in the abstract model.

The property suites comprise a total of 29 properties based on 21 macros in a total of 954 LoC. For the intended abstract model the operations where chosen to cover a full data frame with a length between 2430 and 19440 cycles, depending on the configuration.

The total manual effort including formal verification of 27k LoC in VHDL was about six person months. Compared with our estimate of 2k LoC per person month the productivity figure here is more than doubled. The main reason for this is the sparse description of functionality within the highly generic and re-usable implementation.

For the configuration with the largest state space, the entire set of properties was proven on an Intel Core i7 CPU 860 at 2.8 GHz with 8 GB of memory in less than 2 minutes. No individual property took longer than 20 seconds to prove. Maximum memory usage was 1067 MB.

As can be seen from the design examples of this case study, different implementations can have the same abstract model. This ensures the re-usability of the abstract model for all configurations of the implementation. The possibility to choose an appropriate abstraction makes it possible to extract exactly the information needed in a system-level model. The abstract model obtained here for the SONET/SDH framer fully describes the relevant interaction of the model with the system environment. Yet, it is so small that the model by itself will not impose any complexity challenges for LTL property checking at the system level, for example, when conducting the automatic checks provided by the Spin model checker.

Chapter 7

A Novel Design Flow

In this chapter, a novel design methodology is proposed [57] where the bottom-up approach of Sec. 6 for creating sound system-level models for given RTL implementations is turned around into a top-down procedure. A design flow is proposed where descriptions at high abstraction levels are systematically refined into RTL implementations ensuring that the original high-level description represents a sound abstraction of the final implementation. Verification results obtained at the system level can therefore be trusted for the actual implementation.

Sound models at the system level require abstract and yet clearly defined descriptions of the system's modules and their communication. Existing system description languages such as SystemC, due to their universal applicability and their origin in software programming languages, are not immediately adequate as a basis for establishing a formal relationship between the system level and the RTL. This chapter therefore proposes a language for architectural models that can be used as an intermediate format in system-level design flows.

In our envisioned methodology, the refinement from system-level models into RTL will be a largely manual process although we expect that the method can also be exploited for various automatic refinement steps. In general, however, we argue that a fully automatic synthesis from high-level models is, in many cases, not wanted since engineering decisions are still to be made in the refinement to the cycle-accurate and bit-accurate RTL descriptions.

7.1 Design Flow

The envisioned design flow is depicted in Fig. 7.1. After initial phases of concept building it is proposed that ideas and concepts be formalized as executable model descriptions, e.g., written in SystemC. The refinement of these descriptions will then result in a model at an abstraction level that will here be referred to as the *architectural level*. The same high-level languages used for the conceptual models that may be built in early design phases can be used for describing this architectural model.

The architectural-level description plays a key role in our methodology. If the proposed design flow is followed it will be guaranteed that this description is in a well-defined relationship with the concrete design implementation. Specifically, the architectural model represents a *path predicate abstraction* of the underlying RTL implementation. This has important consequences: verification results obtained at the architectural level will be valid also for the implementation without any further verifications steps. In other words, the architectural model is a *sound abstraction* of the RTL design.

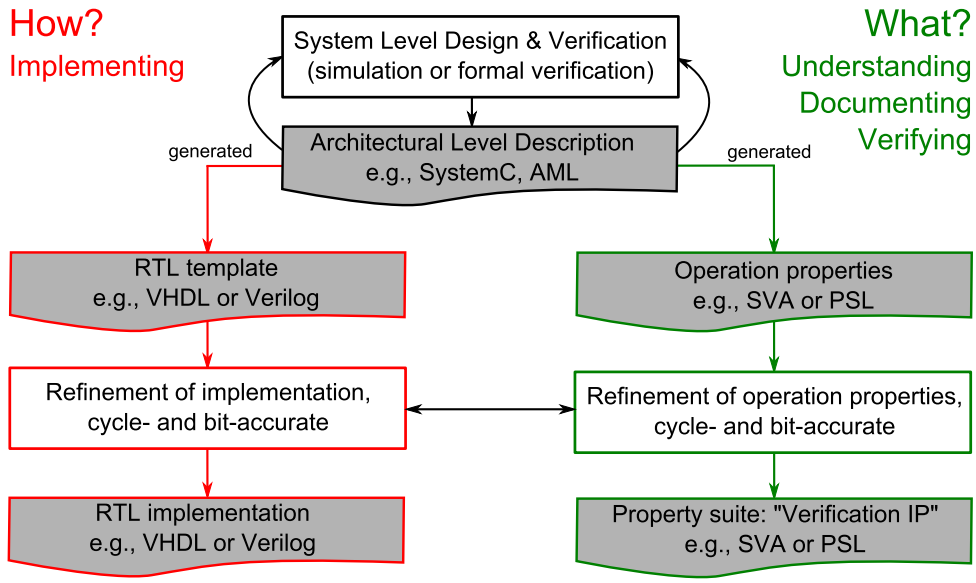


Figure 7.1: Novel design flow based on path predicate abstraction

It was shown in previous chapters how a path predicate abstraction can be created bottom-up for an existing RTL description. The technique requires the design behavior to be structured by *operations* which are formulated as properties according to Def. 37. These operation properties are expressed in terms of the abstract objects given by the *operational predicates*. Features of the chosen property language for supporting a hierarchical code structure, such as macros or functions, are used to encapsulate all direct references to elements of the RTL design and allows for describing concrete behavior in terms of these abstract objects. Note that this also ensures a well readable design documentation.

Instead of using a bottom-up verification process the design flow in Fig. 7.1 creates a path predicate abstraction (PPA) top-down by starting from an architectural design description. In the work of this thesis the modeling language AML has been contributed to give this level of abstraction, which precisely matches that of PPA, a clear semantics. The full syntax of the new language is described in Backus-Naur form in Appendix A.

Note that the proposed language is intended merely as an intermediate language to which standardized high-level languages preferred in industry, or subsets thereof, can be automatically mapped. Compositional PPA describes a relationship between highly abstract systems and RTL which is sound with respect to LTL.

The soundness relies solely on a correspondence for the ordering and for the content of communication events, ensured for the individual PPA module, and a modeling of the composed system as an asynchronous composition of such modules. For our suggested design flow, the semantics of the architectural level must describe this, and only this, semantics. Any additional information which may be specified for an abstract system such as delay ranges, or implementation details for specific data manipulations, are not captured in the semantics of a PPA. A PPA describes what data manipulations are performed between communication points but it does not dictate that a particular algorithm chosen at the system level be implemented at the RTL. In our top-down design flow, such additional details will therefore not be reflected by any generated properties and no correspondence with the resulting implementation can be

guaranteed.

Many language constructs presented for AML have resemblance with those of Esterel [6]. However, synchronous languages, such as Esterel, are not appropriate for describing the asynchronous composition used for PPAs.

Promela, the input language of the Spin model checker [31], has a semantics that comes close to the needs of the intended design flow but has quite a different syntax. We expect that a mapping to AML from a subset of Promela will follow quite easily if communication constructs depending on global variables are avoided and by collecting all statements between communication calls in the Promela construct for defining atomic sequences.

Our main objective, however, is to support design flows based on SystemC. We believe that the described AML semantics matches well with that of un-timed SystemC TLM models. Directly using the SystemC language as a specification for PPAs, however, is not possible. SystemC, as well as many other high-level modeling languages employed in industry are primarily software programming languages. For example, SystemC is used with a framework of class hierarchies and macro definitions to describe the structure of hardware systems, with the associated behavior being modeled in C++. While C++ has a clearly defined semantics as a programming language, the high-level objects defined in the SystemC class framework lack a precise semantics with respect to the abstract hardware designs they are intended for. We envision that this can be solved by defining restrictions to a certain subset of constructs in SystemC for which a mapping to AML can be performed. A formal semantics then applies for this subset through the mapping.

The proposed *architectural modeling language* (AML) is intended for a bit- and time-abstract design description. It precisely describes the decomposition of the design into its sub-components. The interaction between these components is modeled by event-driven communication. AML models are intended as the starting point for the RTL design phase. They specify the functionality that each module fulfills in the system without regards to timing and bit-widths.

In the proposed flow, from the final architectural description a complete set of *abstract operation properties*, written in SVA [2] or PSL [1], is generated automatically. This will be detailed in Sec. 7.3. These properties describe the behavior of the architectural model in terms of *abstract operations*. Using macros, functions or related language features for introducing a code hierarchy the abstract design behavior is described in terms of high-level objects in a hierarchical property code. The detailed encoding of these high-level objects in terms of RTL design elements, for example by formulating the actual body of an SVA function, is left to later stages of the design flow. Any design implementation that later fulfills these properties, for any encoding of the abstract property objects, is ensured to preserve the relationship of a path predicate abstraction with the original architectural description. In this sense, the resulting implementation is "correct by construction" and represents a *correct refinement* of the architectural model.

We also intend to reduce the efforts of implementing the design by generating, from the architectural description, an RTL template containing an entity definition and a state machine for the main control, as depicted on the implementation side of Fig. 7.1. This is, however, only a feature for improved convenience. All implementations fulfilling the properties, including those not using the template, are *correct refinements* by construction of our methodology.

Both the implementation and the macros of the property suite must be refined into bit- and cycle-accurate descriptions. Design refinements must match the concretizations of properties to

be proven. The horizontal arrow in Fig. 7.1, pointing in both directions between the refinement on the "implementation side" and the refinement on the "understanding, documenting, verifying side", is intended to indicate this correspondence. Note that the refinement can be "driven" from both sides. In fact, it may be an attractive option to start with the refinement of the properties. Especially the property specifications for input and output sequences are good candidates as starting points since they represent explicit protocol descriptions and should be subject of early and conscious design decisions.

The proposed approach for the RTL development closely resembles modern software programming techniques using *test-driven development*, in that the verification suite and the implementation are developed in parallel. Thus, in addition to ensuring the soundness of the system model, we also expect the RTL implementation to greatly benefit from such an approach. Rather than being a hurdle for productivity, the property suite serves as a reference while implementing, as an efficient test to the correctness of optimizations (analog to benefits for re-factoring of software), and as a documentation.

At the current state of development, a tool has been implemented which successfully generates the set of operation properties from an architectural description in the defined language. Note that by virtue of this generation step the set of properties resulting in the right part of the flow in Fig. 7.1 is automatically complete. The completeness check of [9] is only necessary when creating an abstraction bottom-up from an existing implementation [56]. It is however not required in the proposed design flow. The generation of the suggested RTL template, as mentioned above, only serves convenience purposes and has not yet been implemented. Also an automatic mapping from SystemC or other standardized ESL languages to the proposed AML language is not yet developed.

7.2 Architectural Modeling Language

The language can be used to describe system architecture in a hierarchical manner. Systems can be composed of sub-systems and/or behavioral descriptions (modules). The full description of the language is available on our website at *http://www.eit.uni-kl.de/eis/forschung/ppa*. In the following, we will sketch important aspects and features, partly using an example.

7.2.1 Global Scope

An AML description is composed of definitions of enumeration types, constants, functions, modules and systems. These definitions reside in the global definition scope.

System definitions are primarily intended for system modeling and verification purposes. Future development could also exploit system analysis to simplify the operational structure of the generated property suite. In a system definition behavior is described as a collection of connected module instances and/or instances of (sub-)systems. **Connections** can be made between pairs of input and output from different instances, as well as "forwarding" connections from input of the system to input of an instance and from output of an instance to output of the system.

A **constant** is a symbolic representation of an already defined data type value and may be used anywhere in the description in place of that value. Constants allow creating configurable descriptions and improve readability of both the AML as well as the operation property suite.

Enumeration types define custom data types that can be used in later declarations. The symbolic values of the data types will be present in the operation property suite as macros and are later given an encoding on the RTL. (This is in contrast to most programming languages where values are assigned to the enumeration symbols already in the definition.)

Also, **functions** can be defined in the global scope. Functions are defined in the mathematical sense: they compute a result from a set of values given as arguments. Unlike in many programming languages, functions are not subroutines and a function "call" neither affects the state of the model nor does it have any side effects. The purpose of function definitions is to encapsulate computation for code reuse and readability.

Module definitions contain the behavioral description of the modules which will later be implemented in RTL. For each module a complete operation property suite will be generated.

7.2.2 Example

Listing 7.1 will serve as an example in the following discussion. It describes a performance monitoring circuit for a telecom protocol framer with the same functionality as the monitor in the example design in Sec. 6.2. The monitor counts the number of times the framer is synchronized with the data stream ("in frame") or not ("out of frame"). The functional behavior is inspired from a similar device implemented in the industrial framer reported in Sec. 8.3. An intuitive understanding of the functionality should be easily gained from the code.

The framer reports de-synchronization through its output `oof` ("out of frame"), which is an input to the monitor. The monitor updates `lof` ("loss of frame") as a performance metric collected over a longer period of time. The exact setting of this metric can be configured through a microprocessor interface, which is abstracted here by the input `setup`.

```
1  MODULE monitor {
2    in<bool> oof;
3    in<bool intmod, int set, int reset> setup;
4    out<bool> lof;
5
6    FSM behavior {
7      states = {REGULAR, INTEGRATING};
8      var<int> OOFcnt; var<int> IFcnt;
9      var<int> set; var<int> reset;
10   @init:
11     nextstate = REGULAR;
12     OOFcnt = 0;
13     IFcnt = 0;
14     lof = false;
15     set = 7;
16     reset = 7;
17   @REGULAR:
18     read(oof);
19     if (oof) {
20       IFcnt = 0;
21       if (OOFcnt >= set) {
22         lof = true;
23       } else {
24         OOFcnt++; }
25     } else {
26       OOFcnt = 0;
27       if (IFcnt >= reset) {
28         lof = false;
29       } else {
30         IFcnt++; }
31     }
32     write(lof);
33     if (pending(setup)){
```

```
34        read(setup);
35        set = setup.set;
36        reset = setup.reset;
37        if (setup.intmod) {
38          nextstate = INTEGRATING; }
39      }
40    @INTEGRATING:
41      read(oof);
42      if (oof) {
43        IFcnt = 0;
44        if (OOFcnt >= set) {
45          lof = true;
46        } else {
47          OOFcnt++; }
48      } else {
49        if (IFcnt >= reset) {
50             OOFcnt = 0;
51          lof = false;
52        } else {
53          IFcnt++; }
54      }
55      write(lof);
56      if (pending(setup)){
57        read(setup);
58        set = setup.set;
59        reset = setup.reset;
60        if (not setup.intmod) {
61          nextstate = REGULAR;}
62      }
63    }
64  };
```

Listing 7.1: AML description of monitor

7.2.3 Module Definitions

We will explain module definitions at the example shown in Listing 7.1. The syntax is quite intuitive but there are some fundamental differences when compared to RTL descriptions. The most important difference originates in the modeling of communication. A communication interface is called a *port* and it is declared using the `in` and `out` keywords (see lines 2–4). Communication is realized through these ports as `read` and `write` calls. The syntax and the semantics is that of a function call, similar to communication modeling in SystemC. The data type carried on the port is defined within angle brackets. A port may also be of a composite data type, as shown for `setup`. It "carries" data at the *event* of a write call. Both, `read` and `write` functions, block until the communication is completed.

Models at the architectural level are event-driven. No clock or similar construct is present that drives the behavior through the sequential description. Between communication points (read/write calls) behavior is only ensured to be executed within some finite time. For the system, all behavior described between communication points is unobservable and can be treated as a single atomic expression. Note that details of how data is actually modified between communication points is irrelevant to the semantics, i.e., two algorithms modifying data in the same way can be used interchangeably without affecting the model's semantics.

The behavior of a module is described as a finite state machine within an `FSM` block (see lines 6–63 in the example). The first part of the definition holds declarations used to describe the state of the module. The control states are enumerated in `states` while data variables are declared using the keyword `var` followed by the data type, again within angle brackets.

The remainder of the definition holds the behavior of the module for each of the control states. The initial state of the module is always called `init`. This keyword is implicitly added to the set of `states`. The *init section* (lines 10–16) is mandatory. No read/write calls to the ports are allowed within the init section.

The behavior of the module is specified for each of the states defined in the `states` set. A standard set of operators and syntax elements exists for defining behavior. Most of it is borrowed from the C programming language. Assignments are defined using the = operator. Conditional execution is declared using if/else.

The special keyword `nextstate` can be understood as a variable of the same enumeration type as defined by `states`. Assigning a control state to `nextstate` defines the state the FSM assumes after the execution of the current state section has finished.

A `for` loop construct is also present. It serves, however, only for conveniently defining a constant number of repetitions. This is useful when iterating over arrays or in order to create generic designs. Actual behavioral loops cannot be modeled using `for` but can instead be described as repeated executions of a section.

Read and **write** operations model, per default, *blocking* communication. A `read` is applied to an `in` port, a `write` is applied to an `out` port. After a read, the `in` port contains the received data. It can be viewed as a variable that cannot be assigned to, (it never appears on the left side of an assignment). It holds its value until the next read on this port. Analogously, an `out` port can only be written and never read, i.e., it never appears on the right hand side of an assignment. It must be assigned a value before the corresponding `write` can be called. The AML parser enforces that `in` ports are read before used and that `out` ports are assigned a value before written.

Non-blocking communication can also be modeled. The predicate `pending` applied to a port name returns *true* if data is available at the port. Unlike `read` or `write`, it does not block. If `pending` returns *true* , a subsequent `read` or `write` will return immediately (see lines 33–39).

7.3 Correct Refinement

The sound relationship between the architectural level and the implementation is that of a *path predicate abstraction*. This relationship is formally proven by describing the complete behavior of the implementation as a set of operation properties where each property describes a transition between abstract control states in a module. A transition is triggered by abstract input, and it is accompanied by the abstract output produced by the module. In practice, the abstract objects are defined using language constructs for defining macros or functions. Such constructs are available in all standard property languages. The macro name is considered an atomic object in the abstraction, while the macro body defines its bit-accurate and cycle-accurate implementation on the RTL. The macro body can thus be viewed as an *encoding* of the abstract object on the RTL.

Based on the example of Listing 7.1, the following discussion clarifies how it is ensured in the proposed design flow that the RTL is a *correct refinement* of the original architectural-level description.

An automatic tool called *refinement synthesizer* takes as input the AML description of the system and produces a set of operation properties together with macro skeletons for the abstract objects. For the above example the generated operation property property suite can be found in

Appendix B.2. Fig. 7.2 shows a graph representation of the operational structure of the example. Nodes in the graph represent control states, edges represent operation properties. The numbers attached as edge labels refer to the corresponding operation property in the example suite.

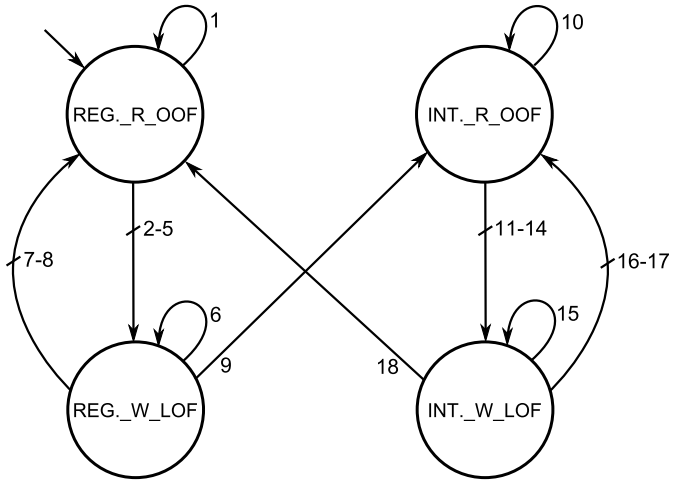


Figure 7.2: Control Mode Graph of the Monitor

7.3.1 Objects of the Abstraction

The macro skeletons consist of only a name with a return type. The macro body is empty and must be filled by the designer to specify how the abstract object is encoded in the RTL implementation. For example, an abstract state macro needs to be encoded by a set of Boolean constraints that defines the set of implementation states corresponding to the abstract control state.

In our example the generated property suite has four control states, one for each call to a blocking communication, i.e., a read or a write not immediately following a pending condition. In general, also a control state may be required to define the beginning of a section. The tool, however, recognizes the cases where this state can be merged with the first read/write state to keep the property suite as simple as possible. (For our example, this check is trivial because both abstract state sections immediately begin with a read.) Also, the abstract state information represented by the module's variable set are directly mapped to macro skeletons.

Macros for the input and output ports of the module are also created, for receiving and sending the actual data, for synchronization, and, if needed, also for storing data. Synchronization signal macros are generated for each port. The macro names ending in `_notify` represent incoming synchronization signals, the ones ending in `_sync` represent outgoing synchronization signals.

The set of generated communication macros also includes datapath macros which are given the ending `_sig`. The encoding of these macros may be spread over a finite number of clock cycles. These macros describe input and output sequence predicates, and they must refer only to input or output signals of the module, respectively. When modeling a system, this requirement is met automatically because connected ports are forced to share the same encoding, by construction.

Ports may be referenced throughout the entire architectural description, not only in the read-/write calls. State datapath macros may therefore also have to be created for the ports. Note that this does not imply that such ports will actually have additional RTL state variables for every port. The encoding may even be the same as the for the corresponding `_sig` macro.

In the example only the `lof` port causes the creation of a state macro (the macro with name `lof`). For the two other ports, the `oof` port and the `setup` port, it can be observed that any reading reference to a value is preceded by a `read` call to this port, and any writing reference to a value is followed by a `write` call to this port. Therefore, no data must be stored across operations and no data storage macro must be created for these cases.

7.3.2 Operational Structure

The refinement synthesizer creates operation properties by, effectively, enumerating all "execution paths" that can be taken between abstract control states. Consider, for example, the `@REGULAR` section: after the `read` four such execution paths exist which all end in a `write`. These paths correspond to the operations 2 through 5 in the generated suite. They differ in the evaluation of the conditions of the if-else blocks. Each execution path is characterized by the conjunction of all condition expressions along the path. This conjunction forms the assumption of the operation property corresponding to the execution path.

The translation from ports with blocking read/write calls is reflected in the operational structure. (In its current state, the tool only supports clock-synchronous communication.) The read-/write calls to `oof` and `lof` (lines 18, 32, 41, and 55) are blocking calls. The operation property suite is structured in a way such that the modeled blocking behavior is imposed on the RTL implementation. In particular, a *waiting state* is generated, modeling a mode where the module waits for an incoming synchronization signal (`_sync`). The wait is modeled by a waiting operation (e.g., operations 1, 6, 10, and 15 in the example). This operation is "triggered" by the absence of the `_notify` flag on the corresponding port.

The read calls to `setup` (line 34 and 57), are both immediately preceded by a `pending` if-condition and are therefore examples of non-blocking communication. This is exploited to simplify the operation property suite. No additional wait state is required. Note that the blocking communication scheme does not force any wait operation to actually be executed. Communication which is non-blocking in the concrete implementation could therefore be modeled at this abstract level as blocking communication (and the later realized implementation would ensure that the wait operation is never triggered). When the non-blocking read/write is executed the `_notify` flag is active for one clock cycle to inform the communication partner of the communication event. In a synchronous system the event is always safely captured by the communication partner. The incoming `_sync` (being the outgoing `_notify` of the paired port) ensures that the communication partner is in a state where it can react to the call. Due to the common clock any signal value kept stable for one clock cycle will be captured.

7.3.3 Modeling Communication at the RTL

The transfer of data must be soundly modeled by the property suite. In a read call the input data, encoded in `_sig`, is captured at the time when `_sync` is set active. In the following operation the read value is referenced as the input at this time point. If a reference is made to this port value also in other operations the value must be stored within the abstract state. This is realized as an

additional datapath state macro which, at the ending control state of the operation, is proven to encode the read value.

A write call works just in the opposite way. The outgoing data transported in the `_sig` macro must have the value of the state datapath macro at the time point of the `_sync` event.

By a proper encoding of the communication data macro, `_sig`, the actual data transfer may be specified to occur also at any (fixed) time later in the actual RTL description. The structure of the properties simply ensures that the data macros explicitly specify the data encoding with time reference "anchored" at the synchronization event.

The generated macros and operations together form a communication framework that forces the RTL to implement a communication infrastructure with proper synchronization and correct blocking behavior. In RTL design, however, there exist many different signaling, synchronization and data transfer mechanisms. The generated communication framework must allow the designer to implement these by simply encoding them properly into the macro bodies. It is therefore important that the generated property set is "generic" enough and does not lead to redundant structures on the RTL.

The various common communication schemes and their modeling through synchronization and data transfer sequence predicates were discussed in Sec. 5.2. All these schemes can be modeled using blocking read or write in AML and translated into corresponding mechanisms on the RTL, e.g. by event-signaling or handshaking. This is encoded through two synchronization macros for each port, an outgoing, `_notify`, and an incoming, `_sync`.

7.4 Experimental Results

In order to evaluate the novel design methodology we conducted two case studies. The first study was a student group design project and is reported in the following. The second study used the new methodology for a power-aware redesign of an industrial telecommunications IP component and is reported in Sec. 8.3.

7.4.1 Student Project

Four students were given the task to use AML and the new methodology for designing, implementing and verifying a musical game to be realized on an FPGA evaluation board. Prior to the actual design phase, the students were trained in formal property checking and completeness checking with a commercial tool suite (OneSpin 360 DV). All system components were to be designed purely as hardware descriptions; no processor was used. The task was focused on the design of the central game controller. A set of hardware IP blocks for input and output (keyboard, LCD, audio) were given.

The students were asked to describe the game controller as an AML module communicating with peripherals through freely chosen ports. As a next step, the AML description was used as an informal specification while implementing the RT level using VHDL. The RTL implementation consists of the given IPs, a "main game controller", a pseudo-random generator, and several communication modules providing interfaces between the game controller and the given IPs.

In contrast to the suggested design flow, the operation property suite was not given to the students until after the implementation was done. In this case study we did not want the implementation to be affected by the details of the operational structure. The purpose was to

investigate the possible discrepancies and their cause, and to check if the working communication structures of the implementation could be encoded within the boundaries of the generated operation property skeletons.

The case study showed that the students could identify a suitable encoding of the abstract objects in the RTL implementation quite easily. The property suite that was created and refined from the architectural description initially failed the formal verification on the implementation, pointing to an actual design discrepancy between the manual RTL design and the architectural specification. After these issues were resolved, the property suite could be proven on the RTL design. The students were exchange students from a different university and had no previous experience with formal verification and the design methods developed in this research at the host university. They did have some prior knowledge in VHDL-based RTL design from courses in their home university, however no extensive design experience. It was encouraging to see that the students, who, in contrast to industrial design engineers, were not biased by extensive experience in "classic" RTL design, quickly picked up the new concepts, learned to use AML and adopted the new methodology. The team of four students took about two weeks for acquiring property checking skills before they actually began design. They then used about four weeks of team effort to complete the design. The students communicated their impression that the existence of AML models greatly eased the integration of the developed modules and that the pursued design flow significantly reduced their work effort.

Chapter 8

Perspectives for Low-Power Design

The last technical chapter of this thesis is dedicated to a preliminary demonstration of how PPA and the new design flow can open new opportunities to achieve non-functional design goals. The formalization of an "operational view", either obtained *bottom-up* for an existing implementation using C-IPC as described in Chapter 6, or *top-down* in the novel design flow suggested in Chapter 7, can be exploited for purposes of low-power design. This has been researched as a part of the EnerSave project funded by the German Ministry of Science and Education (BMBF) under Grant Agreement 16BE1103. Some of the presented results have been published in this context [59].

The three sections of this chapter each present a distinct use case. In Sec. 8.1 a method for improved energy estimation at the system level is presented. This method is based on annotating operations with average energy consumption. We only expect minor deviations in the energy consumptions for different runs of the same operation, hence, we expect that total energy consumption of longer behavioral scenarios can be estimated fairly precisely based on the average energy consumptions of the operations executed in the scenarios.

Further, in Sec. 8.2, perspectives for power optimization of the implementations are discussed. An analysis method is presented that finds signal toggling in the implementation that do not influence the input/output behavior. The analysis uses the complete set of operation properties as a description of the input/output behavior and applies a backtrace procedure to find a cone of influence for output at a specific "time point" relative to the operations.

Finally, in Sec. 8.3, our suggested novel design flow (cf. Chapter 7), is discussed in the context of power optimization. To evaluate the potential for power optimization a full re-design of the SDH/Sonet framer from Alcatel Lucent has been executed. The results are presented at the end of this section.

8.1 Energy Estimation at System Level

Theoretically, every possible sequence of state transitions that may occur in an FSM model for a considered SoC module can be associated with a specific amount of energy that is consumed along this path. It is, of course, intractable to individually identify this amount of energy for every single FSM path when considering SoC modules of realistic size. In a path predicate abstraction the design behavior is decomposed into a relatively small number of operations that each describe a set of FSM paths that are functionally closely related. Intuitively, paths being functionally related, in many cases, can also be expected to consume a similar amount

of energy. Therefore, it seems promising to associate energy consumption with operations. In less frequent cases, a notable difference may be observed in the energy consumption for different runs of the same operation. Note however, that in a power-aware design flow, this may be reflected in the choice of important states and operations. If an operation exhibits a heterogeneous power behavior, it may be wise to split the operation into sub-operations with similar energy consumptions, or to choose an entirely different operational structure of the design. As explained in earlier chapters, it is left to the ingenuity of the designer or verification engineer to formulate operations according to his/her needs.

Using path predicate abstraction, as explained in Chapter 6, opens the opportunity to create system models that predict energy consumption in a reliable way. Abstract models created by path predication are in a strict formal relationship with the underlying concrete RTL design. From the soundness of a path predicate abstraction, stated in Theorem 2, if follows that every state sequence in an abstract model corresponds to a well-defined chain of concrete operations in the RTL. Therefore, the energy values associated with operations can be used to annotate the abstract model. Due to the formal soundness of the abstract model the same accuracy that is achieved for the energy models of the individual operations is also available for the system model.

Obviously, energy calculations can first be conducted when an implementation is at hand. In any top-down design flow, e.g., in the suggested design flow of Chapter 7, accurate energy estimates are therefore not at hand for modules which are not yet given a realization.

System developments does often rely on a re-use of modules, intellectual property (IP) developed internally or bought from external developers. Such modules are, already today, sometimes paired with *virtual prototypes*, i.e., a system-level module that explains the interface and the functionality of the implementation to the degree of detail necessary for its integration in a system. For the gained benefit of accurate energy estimation, we imagine that these virtual prototypes could be described as architectural descriptions with such annotation (possibly a set of annotations for various production technologies).

Note that we consider power annotations to be useful in a top-down design flow also when they are first available in a later development stage. The implementation of operations which are frequently executed in system-level use-cases or have a specifically high energy consumption may be revised without changing the obligations of the operation, i.e., the functionality actually fulfilled as specified by the operation property template.

We have conducted a case study based on an hardware implementation of Euclid's greatest common divisor (GCD) algorithm. Its exact verified behavior will be presented in terms of operations described in the pseudo property language also used in Chapter. 6. This provides an example on how operations can be identified and described for such datapath oriented RTL designs and how the obtained control oriented abstraction compares to a purely algorithmic description. It is then examined how energy estimates can be associated with the chosen operations and to what extent the energy consumption varies within a single operation.

8.1.1 Energy Estimation for GCD circuit

The following C function describes Euclid's algorithm to find the greatest common divisor of two integers.

```
1 int euclid(int a, int b){
2   if (a == 0)
```

```
3     return b;
4   while (b != 0) {
5     if(a > b)
6       a = a - b;
7     else
8       b = b - a;
9   }
10  return a;
11 }
```

A usable hardware realization of this algorithm needs basic control mechanisms. These mechanisms are best described in terms of the interface of the module.

```
1  entity euclid is
2  port (
3    clock  : IN  STD_LOGIC;
4    reset  : IN  STD_LOGIC;
5    start  : IN  STD_LOGIC;
6    done   : OUT STD_LOGIC;
7    idle   : OUT STD_LOGIC;
8    enable : IN  STD_LOGIC;
9    a      : IN  STD_LOGIC_VECTOR(31 downto 0);
10   b      : IN  STD_LOGIC_VECTOR(31 downto 0);
11   return : OUT STD_LOGIC_VECTOR(31 downto 0) );
12 end;
```

Note that even a data-centric module like this one needs a well-defined protocol for the communication with the environment. Although this protocol may not be explicitly documented, understanding it is required in order be able to successfully integrate the module into a system environment.

Input signal `start` indicates the start of a new calculation taking the values of `a` and `b` in the following clock cycle where the chip is enabled as argument, i.e. where the signal `enable` is high. Output signal `done` flags the end of the calculation and that the current value of signal `return` is valid. Output signal `idle` is set when the module is ready to start a new calculation. Input signal `enable` is used to enable/disable the module. When disabled, i.e., `enable` is low, the internal states of the module freeze and any input is ignored.

In our operational methodology the calculation of the greatest common divisor may be viewed as a single operation taking `a` and `b` as operands. A correct implementation of Euclid's algorithm can indeed be described by a single operation as a behavioral sequence of bounded length. In practice, however, the sequence length grows linearly with the number of loop iterations of the algorithm and may contain millions of cycles. The complexity of such a long operation cannot be handled by any state-of-the-art model checker.

Therefore, the behavior of this module is decomposed into smaller operations that result from considering the above C function as a specification. By inspection of the RTL code and by viewing counterexamples of preliminary properties we gain knowledge about implementation-specific details required to formulate the operations.

The verified operation properties and the resulting abstract state machine for the control of the module are shown in Fig. 8.1. The abstract FSM consists of two abstract control states shown as nodes in the graph and seven operations shown as directed edges between the control states. After reset we are in a state READY where the module can accept a request for a new calculation. From READY, if now a request is made, we wait in READY (`wait` operation). If

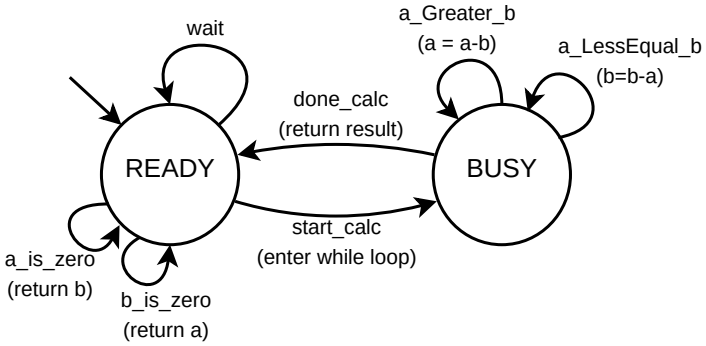


Figure 8.1: Abstract model

a request is accepted and one of the operands is zero the other operand is returned. No further calculation is necessary and we may directly return to control state READY (`a_is_zero` and `b_is_zero` operation). A request where neither operand is zero causes the control state BUSY to be entered and the operands to be stored in registers (`start_calc` operation). In control state BUSY the register values are iteratively computed as follows:

```
if (b_reg != 0 and a_reg > b_reg)
  a_reg = a_reg - b_reg; //a_Greater_b operation
if (b_reg != 0 and a_reg ≤ b_reg)
  b_reg = b_reg - a_reg; //a_LessEqual_b operation
if (b_reg == 0)
  exit to control state READY; //done_calc operation
```

The operations can easily be verified against the algorithmic specification. Additionally, they explicitly describe the communication protocol.

The abstract symbols are created using macros, as explained in Chapter. 6. The name of the macro is our abstract symbol while the body of the macro describes its semantics in terms of RTL objects. As an example, the abstract control state READY is defined as:

```
1  macro READY : boolean :=
2     (ap_CS_fsm = ap_ST_st0_fsm_0) and ap_ce or
3     (ap_CS_fsm = ap_ST_st2_fsm_2) and ap_done and ap_ce;
4  end macro;
```

We can then refer to these abstract symbols as demonstrated by the formulation of operation `start_calc` in Listing 8.1.

```
1  property start_calc is
2    assume:
3      at t: READY;
4      at t: start;
5      at t+1: a /= 0 and b /= 0;
6    prove:
7      at t+1: not IMPORTANT_S;
8      during[t+1,t+2]: not done;
9      during[t+1,t+2]: not idle;
10     at t+2: a_reg = a @ t+1;
11     at t+2: b_reg = b @ t+1;
12     at t+2: BUSY;
13 end property;
```

Listing 8.1: Operation start_calc

Operation	Avg. energy (fJ)	Std. dev.
a_is_zero	401.9	19.9 %
b_is_zero	1155.4	18.2 %
done_calc	283.4	23.7 %
reset	13.6	0 %
start_calc	4506.0	13.4 %
a_Greater_b	1386.4	7.2 %
a_LessEqual_b	1315.1	6.6 %
wait	187.36	78.2 %
after splitting:		
wait_tmpReg0	374.9	0.1 %
wait_tmpReg1	78.7	0.3 %

Table 8.1: Energy consumption of GCD operations

This property is proved on the RTL implementation. It ensures that if we are in any state satisfying READY, `start` is asserted, and both operands are different to zero, then (among other commitments) we enter some state satisfying BUSY and the operand values are stored to registers at the correct time points.

For a power-aware design flow each operation can be annotated with an energy value. In order to compute this energy value we use the formal property checker to compute a number of *witnesses* for the property. A witness is a set of timing diagram waveforms corresponding to sequences of value assignments to the signals in the circuit such that the operation is executed by the design (and the correctness of the property for the given set of waveforms can be observed by the user). The witnesses are generated under input data constraints taken from a representative use case of the design and are evaluated by Synopsys PowerCompiler (R) with respect to their energy consumption. In the experiments presented here, we ran witness computation for about 30 sets of input constraints for each operation.

Tab. 8.1 summarizes energy consumption (average and standard deviation) for the operations of Fig. 8.1. Our experiments confirm that the energy consumption of an operation does not depend heavily on the different input scenarios covered by the operation. In fact, in this experiment the energy values did not vary much within the samples for an operation. An exception to this, however, is operation `wait` which therefore deserves a closer consideration. When measuring the energy consumption for runs of `wait` it was observed that the results always fell into one of two narrow bands at around 375 fJ and 79 fJ, accounting for a std. dev. of 78 % (cf. Tab. 8.1). The problem could be traced back to a specific signal visible in the RTL. The operation was then split for the values of this internal signal, now reducing the std. dev. to marginal values.

Interestingly, this complication is a consequence of poorly optimized circuitry. It is proven, as a result of the complete verification, that the `wait` operation is not affected by the value of the internal signal. Therefore, the RTL could be modified to always use the less energy-consuming value without changing the input/output behavior of the module. This shows that such an analysis could serve as a strong heuristic for finding “fine-grained” optimization potential.

In general, as can be noted in Tab. 8.1, even in this data-centric design, the differences between energy values for different operations are significantly larger than the differences between

energy values for different data inside the same operation. This confirms that useful estimates of energy consumption can also be obtained in a system-level description derived from these operations by path predicate abstraction. In the given example, this system model is obtained by annotating the transitions in Fig. 8.1 with the energy values from Tab. 8.1.

8.2 Dependency Analysis for Power Optimization

A second observation opens up new opportunities for power optimization. Operation properties may provide important functional information that may help us to identify parts of the design that are irrelevant in certain situations and which therefore can be switched off. One possible approach is to specifically formulate properties to probe possible behaviors in the design with the objective to identify design components that can be switched off [32]. Here, we pursue another approach which exploits the completeness of the property set. If an RTL design is completely described in terms of operation properties, these properties provide a formal specification of the expected behavior and contain the exact information about what parts of the design are relevant at every clock tick to produce the specified behavior. We may therefore directly derive from the property set what parts of the design are needed in specific time intervals and what parts are not needed so that they can be temporarily switched off.

We have used the SONET/SDH Framer from Alcatel-Lucent, as already introduced in Sec. 6.5.2, as a case study and we use this industrial design to demonstrate the potential of a formal specification obtained by C-IPC to identify power optimizations at a micro-architectural level.

A complete property suite has been developed for this design. The properties and their completeness have been formally proven and a sound path predicate abstraction of the design behavior was obtained (see Sec. 6.5.2).

We are experimenting with this design to explore ideas for operation-based power optimization. As explained above, the intuition is that not all circuitry needs to be active during an operation. The property describes precisely what registers and inputs are evaluated to trigger the operation and also what output sequences and state values are produced during the operation. Anything that does not contribute to these computations can be turned off during an operation.

We give here a simplified explanation of the approach taken in this experiment. For each operation:

1. Analyze the operation to identify any circuitry that is not involved in implementing its behavior.
2. Based on the *assumption A* of the operation property, construct a monitor circuit that identifies situations and time points when the operation is triggered.
3. Insert disabling logic for switching elements that can be turned off and connect it with the monitor circuit.

The analysis of 1) takes into account the actual timing of the operation. The circuit under consideration is unrolled for the number of clock cycles the operation takes. The circuitry implementing the behavior specified in the property is identified using a back-tracing procedure. We start at the registers and output signals in the design that hold the end states and output values specified in the *commitment c* of the operation property. We trace back in the unrolled circuit and mark all gates until we reach the inputs or start states mentioned in the assumption *a*. The unrolled circuitry that has been marked in the tracing procedure takes part in the computation

of next states and output values. Because we have unrolled the circuit, each marked gate in the unrolling is associated with a specific time point in which its output contributes to the operation's computation. Therefore, the disabling logic can be fine-tuned to the actual clock cycles within an operation in which a particular switching element can be turned off.

For the SONET/SDH framer, the operation `sync_iterate` can be identified as making dominant contributions to the overall power consumption of the module. Therefore, as a proof of concept, we applied the above approach to this operation. Using power estimations based on Synopsys PowerCompiler (R) it was obtained that power consumption was reduced from 187 μW to 103 μW amounting to power savings of 45 % for this operation. This clearly demonstrates the potential of exploiting detailed functional information from the design's operational structure for reducing switching activity.

8.3 Power Aware Design Flow

In this section, we apply the novel design flow presented in Chapter 7 in the context of a power aware RTL development.

Resulting from the architectural description a complete set of operation properties is obtained in the form of operation property templates. The operation properties, together with the details of the external communication protocols, represent a formal specification that can be automatically checked against the implementation. Any two implementations satisfying all of these properties can be used interchangeably for the same task in a system.

This is a major aid to the development of the RT level implementation. Clearly, in the comfort of an automated check to the functional correctness, aggressive optimizations and changes to legacy code can be done without the risk of introducing bugs. Any changes that violate the specified functionality will fail in one or more properties.

A complete re-design of the SONET/SDH framer from Alcatel Lucent was done to gain experience with what optimization potential we may expect and to get comparable numbers for such an optimized design. From the case study reported in Sec. 6.5.2 a complete set of operation properties had already been developed that proves the existence of a path predicate abstraction. For the re-design we have deleted the encoding of the internal abstract objects but otherwise used this set of operation properties directly. In other words, for the re-design the same (bit- and cycle-accurate) protocol is ensured for the interface, only the internal, non-observable, behavior is changed.

The new design was created from scratch, but may be compared with respect to how some of the commonly required functional tasks are realized.

- The new design relies, as opposed to the original, on only a single counter that keeps track of the data words irrespective of mode operation. Additional circuitry was added to reflect the respective different use, but this circuitry only causes toggling while in the respective modes of operation.
- The buffering of the incoming bit-stream could successfully be reduced.
- The sensitivity of the originating RTL processes were all synchronous. In the new design, functionality that rarely occurs is instead implemented in combinational processes. Specifically, a very large comparator circuitry realizing a search for the frame marker at all possible alignments could thereby be restrained from toggling at any stage where the search result is not used.

In total (dynamic and static) the energy consumption was reduced by about 50% compared to the original design. For this measurement we have assumed that the framer remains in a synchronized mode during a predominant part of the time.

We argue that path predicate abstraction is a considerable aid for a power aware implementation. It enables the automatic analysis for optimization potential. This analysis, in future work, could possibly also be used in automated methods for optimizing the RTL or the gate-level models. In a top-down design flow, as suggested in Chapter 7, aggressive optimizations can be implemented directly. This is owed to the formalization of an operational design view supporting the design process with an intuitive understanding of "what is needed when", as well as with an automated formal check ensuring the correctness of the design refinement.

Chapter 9

Conclusion

A formalism called *path predicate abstraction (PPA)* was contributed in this thesis. PPA is a formalism suitable to describe the relationship between cycle-accurate circuit descriptions at the register transfer level (RTL) and time-abstracted system-level models whose functionality is specified in terms of sequences of transactions.

For a pair of an abstract and a concrete model in which this relationship holds, properties of the one model are reflected by corresponding properties of the other model. Important properties, including both liveness and safety properties, can in this way be proven for the concrete implementation by reasoning on the much simpler abstract model. The computational complexity for model checking is vastly reduced when considering the abstract model. Through path predicate abstraction it is therefore possible to prove large global properties of the actual realization – properties far out of scope for model checking approaches on the RT-level model.

Sound abstraction based on a stack of sound functional models above the RTL have been investigated extensively in the context of formal verification by theorem proving, for example in [48, 39, 7]. An important difference between these previous works and sound abstractions based on path predicate abstraction, as proposed in this thesis, lies in the fact that the methodology developed in our work is entirely based on standardized property languages such as System Verilog Assertions (SVA) and relies exclusively on fully automatic property checking using a bounded circuit model [16, 43]. This does not only support an intuitive formulation of design properties but also facilitates proof procedures that can handle industrial RTL designs of realistic complexity.

Methods to apply PPA in practice have been researched in the context of this work. In Chapter 6 it is shown how a path predicate abstraction can be created for an already existing RTL implementation based on *complete* interval property checking (cf. Chapter 3). Compared with traditional simulation-based verification, complete interval property checking has, in several industrial case studies [7], been shown to outperform traditional simulation-based verification approaches, not only in terms of design bugs found, but also in terms of effort spent.

Path predicate abstraction is established based on complete interval property checking structured by *operation property templates* (cf. Sec. 4.3). We argue that this structure also improves the quality of the verification itself by ensuring that the resulting properties serve as an intuitive and understandable documentation where cycle-accurate implementation details are placed in context with a high-level functional model.

A top-down design flow integrating PPA is proposed in Chapter 7. Instead of creating an abstraction by describing and proving properties of an existing design, a set of operation

properties is “synthesized” from a system-level description which serves as formal requirements for the concrete implementation. In other words, from a methodical refinement of a system-level description a “correct-by-construction” RTL implementation is created.

In the context of the top-down design flow, an “architectural modeling language (AML)” has been developed (cf. Appendix A). The language is intended to be used only as an intermediate description automatically derived from standardized ESL languages such as SystemC when these descriptions are restricted to a mappable subset. Such an intermediate representation is needed to overcome the limitations of SystemC in precisely defining the semantics of the design model and its interfaces as well as to cope with the overwhelming expressive power of SystemC and the large syntactical diversity it allows.

In sum, the work presented in this thesis shows how the semantic gap between the system level and the RT level can be closed through the contributed formalism of compositional path predicate abstraction. The proposed work thereby enables a verification approach ensuring correctness of the overall system architecture based on system-level models, as opposed to current verification methods which still explore system behavior at the RTL.

Having a known and formalized relationship between the system level and the RTL is of great advantage not only in reaching functional design and verification goals. It also paves the way for new approaches to reach non-functional design targets. Motivated by the current need for new design flows with better power closure, in Chapter 8 of this thesis, initial research has been conducted demonstrating the use of PPA for energy estimation and power optimization.

In fact, if the proposed correct-by-construction design flow is followed, all functional dependencies between the relevant design components are made visible to the designer with formal precision. This provides comprehensive and valuable insights into the implementation and sets the frame for highly optimized designs including even those that, in today’s design flows, are lying outsides the “comfort zone” of designers .

Thus, the new design flow based on PPA, as proposed in this thesis, combines an increased productivity of functional design and verification with the promise of improved design closure for non-functional targets such as power consumption. Sec. 8.3 is only a preliminary study in this context. Future work will be dedicated to better elaborate and demonstrate these aspects, as well as to include low-level software and firmware into PPA-based design.

Bibliography

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

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

[3] ABADI, M., AND LAMPORT, L. The existence of refinement mappings. *Theoretical Computer Science 82*, 2 (1991), 253–284.

[4] BALL, T., COOK, B., LEVIN, V., AND RAJAMANI, S. K. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In *IFM* (2004), Springer, pp. 1–20.

[5] BARBACCI, M. R. A comparison of register transfer languages for describing computers and digital systems. *IEEE Trans. Comput. 24*, 2 (Feb. 1975), 137–150.

[6] BERRY, G., AND GONTHIER, G. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program. 19*, 2 (Nov. 1992), 87–152.

[7] BMBF. Verisoft XT. http://www.verisoftxt.de.

[8] BORMANN, J. *Vollständige Verifikation*. Dissertation, Technische Universität Kaiserslautern, 2009.

[9] BORMANN, J., AND BUSCH, H. 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.

[10] BRAND, D. Verification of large synthesized design. In *Proc. International Conference on Computer-Aided Design (ICCAD)* (1993), pp. 534 –537.

[11] BROWNE, M. C., CLARKE, E. M., AND GRÜMBERG, O. Characterizing finite kripke structures in propositional temporal logic. *Theor. Comput. Sci. 59*, 1-2 (July 1988), 115–131.

[12] CHAUHAN, P., GOYAL, D., HASTEER, G., MATHUR, A., AND SHARMA, N. Non-cycle-accurate sequential equivalence checking. In *Proc. Design Automation Conference (DAC)* (2009), pp. 460–465.

[13] CIMATTI, A., GRIGGIO, A., MICHELI, A., NARASAMDYA, I., AND ROVERI, M. KRATOS: a software model checker for SystemC. In *Proceedings of the 23rd international conference on Computer aided verification* (Berlin, Heidelberg, 2011), CAV'11, Springer-Verlag, pp. 310–316.

[14] CIMATTI, A., MICHELI, A., NARASAMDYA, I., AND ROVERI, M. Verifying SystemC: a software model checking approach. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design* (Austin, TX, 2010), FMCAD '10, FMCAD Inc, pp. 51–60.

[15] CLAESSEN, K. A coverage analysis for safety property lists. In *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)* (2007), pp. 139–145.

[16] CLARKE, E., BIERE, A., RAIMI, R., AND ZHU, Y. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des. 19*, 1 (July 2001), 7–34.

[17] CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst. 8*, 2 (Apr. 1986), 244–263.

[18] CLARKE, E. M., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. Counterexample-guided abstraction refinement. In *Proc. International Conference on Computer Aided Verification (CAV)* (2000), pp. 154–169.

[19] CLARKE, E. M., GRUMBERG, O., AND LONG, D. E. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems 16*, 5 (September 1994), 1512–1542.

[20] CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. *Model Checking*. MIT Press, London, England, 1999.

[21] CORTES, C. F. C. High level synthesis of control flow units: Synthesizable model of a pipelined data cache for the cadence c-to-silicon compiler. Master's thesis, Technische Universität Kaiserslautern, 2013.

[22] DE NICOLA, R., AND VAANDRAGER, F. Action versus state based logics for transition systems. In *Proceedings of the LITP Spring School on Theoretical Computer Science on Semantics of Systems of Concurrent Processes* (New York, NY, USA, 1990), Springer-Verlag New York, Inc., pp. 407–419.

[23] EMERSON, E. A., AND HALPERN, J. Y. "sometimes" and "not never" revisited: On branching versus linear time temporal logic. *J. ACM 33*, 1 (Jan. 1986), 151–178.

[24] EVEKING, H., DORNES, T., AND SCHWEIKERT, M. Using SystemVerilog Assertions to relate non-cycle-accurate to cycle-accurate designs. In *Proc. of 16th IEEE International High Level Design Validation and Test Workshop (HLDVT)* (2011), pp. 17–24.

[25] GENTILINI, R., PIAZZA, C., AND POLICRITI, A. From bisimulation to simulation: Coarsest partition problems. *J. Autom. Reasoning 31*, 1 (2003), 73–103.

[26] GONTHIER, G. The Four Colour Theorem: Engineering of a Formal Proof. In *Computer Mathematics, 8th Asian Symposium, ASCM, Singapore, December 15-17, 2007. Revised and Invited Papers* (2007).

[27] GRAF, S., AND SAÏDI, H. Construction of abstract state graphs with PVS. In *Proc. International Conference Computer Aided Verification (CAV)* (London, UK, 1997), vol. 1254 of *LNCS*, Springer-Verlag, pp. 72–83.

[28] GROSSE, D., LE, H. M., AND DRECHSLER, R. Proving transaction and system-level properties of untimed SystemC TLM designs. In *Proc. ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)* (2010), pp. 113–122.

[29] HAO, K., RAY, S., AND XIE, F. Equivalence checking for behaviorally synthesized pipelines. In *Proceedings of the 49th Annual Design Automation Conference* (New York, NY, USA, 2012), DAC '12, ACM, pp. 344–349.

[30] HAO, K., XIE, F., RAY, S., AND YANG, J. Optimizing equivalence checking for behavioral synthesis. In *Proceedings of the Conference on Design, Automation and Test in Europe* (3001 Leuven, Belgium, Belgium, 2010), DATE '10, European Design and Automation Association, pp. 1500–1505.

[31] HOLZMANN, G. J. The SPIN model checker. *IEEE Transactions on Software Engineering 23* (1997), 279–295.

[32] HORN, T., SAUPPE, M., MARKERT, E., HEINKEL, U., RÖSSEL, W., AND SAHM, H.-W. Einsatz formaler Methoden zur Energieeinsparung. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV* (2013), pp. 141–146.

[33] KOELBL, A., JACOBY, R., JAIN, H., AND PIXLEY, C. Solver technology for system-level to RTL equivalence checking. In *Design, Automation Test in Europe Conference (DATE)* (april 2009), pp. 196 –201.

[34] KÖLBL, A., BURCH, J. R., AND PIXLEY, C. Memory modeling in ESL-RTL equivalence checking. In *Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007* (2007), pp. 205–209.

[35] KROENING, D., AND SHARYGINA, N. Formal verification of SystemC by automatic hardware/software partitioning. *Formal Methods and Models for Co-Design* (2005), 101 – 110.

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

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

[38] LE, H. M., GROSSE, D., HERDT, V., AND DRECHSLER, R. Verifying systemc using an intermediate verification language and symbolic simulation. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013* (2013), pp. 116:1–116:6.

[39] MANOLIOS, P., AND SRINIVASAN, S. K. A refinement-based compositional reasoning framework for pipelined machine verification. *IEEE Transactions on VLSI Systems 16* (2008), 353–364.

[40] MCMILLAN, K. L. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

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

[42] NAGEL, L. W. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. PhD thesis, EECS Department, University of California, Berkeley, 1975.

[43] NGUYEN, M. D., THALMAIER, M., WEDLER, M., BORMANN, J., STOFFEL, D., AND KUNZ, W. Unbounded protocol compliance verification using interval property checking with invariants. *IEEE Transactions on Computer-Aided Design (TCAD) 27*, 11 (November 2008), 2068–2082.

[44] ONESPIN SOLUTIONS GMBH. Germany. OneSpin 360MV. http://www.onespin-solutions.com.

[45] PNUELI, A. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on Foundations of Computer Science* (31 1977-nov. 2 1977), pp. 46 –57.

[46] PRICE, D. Pentium fdiv flaw-lessons learned. *IEEE Micro 15*, 2 (Apr. 1995), 86–88.

[47] RAY, S., HAO, K., CHEN, Y., XIE, F., AND YANG, J. Formal verification for high-assurance behavioral synthesis. In *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis* (Berlin, Heidelberg, 2009), ATVA '09, Springer-Verlag, pp. 337–351.

[48] RAY, S., AND HUNT, JR., W. A. Deductive verification of pipelined machines using first-order quantification. In *Proceedings of the* 16*th International Conference on Computer-Aided Verification (CAV 2004)* (Boston, MA, 2004), Springer, pp. 31–43.

[49] SEGER, C.-J. H., AND BRYANT, R. E. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design 6*, 2 (1995), 147–189.

[50] SHANNON, C. A symbolic analysis of relay and switching circuits. *American Institute of Electrical Engineers, Transactions of the 57*, 12 (dec. 1938), 713 –723.

[51] SUTTER, H. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal 30*, 3 (2005).

[52] TABAKOV, D., AND VARDI, M. Y. Monitoring temporal SystemC properties. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on* (July 2010), pp. 123 –132.

[53] TABAKOV, D., VARDI, M. Y., KAMHI, G., AND SINGERMAN, E. A temporal language for SystemC. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design* (Piscataway, NJ, USA, 2008), FMCAD '08, IEEE Press, pp. 22:1–22:9.

[54] THALMAIER, M., NGUYEN, M., WEDLER, M., STOFFEL, D., BORMANN, J., AND KUNZ, W. Analyzing k-step induction to compute invariants for SAT-based property checking. In *Proc. International Design Automation Conference (DAC)* (2010), pp. 176 –181.

[55] URDAHL, J., STOFFEL, D., AND KUNZ, W. System- versus RT-level verification of systems-on-chip by compositional path predicate abstraction. Tech. Rep. EDA-2013-01-29, University of Kaiserslautern, Dept. of Electrical and Computer Engineering, EDA group, Jan 2013.

[56] URDAHL, J., STOFFEL, D., AND KUNZ, W. Path predicate abstraction for sound system-level models of RT-level circuit designs. *IEEE Transactions on Computer-Aided Design (TCAD) 33*, 2 (Feb. 2014), 291–304.

[57] URDAHL, J., STOFFEL, D., AND KUNZ, W. Architectural system modeling for correct-by-construction RTL design. In *Forum on Specification and Design Languages (FDL)* (Barcelona, Spain, September 2015).

[58] URDAHL, J., STOFFEL, D., WEDLER, M., AND KUNZ, W. System verification of concurrent RTL modules by compositional path predicate abstraction. In *Proceedings of the 49th Annual Design Automation Conference* (New York, NY, USA, 2012), DAC '12, ACM, pp. 334–343.

[59] URDAHL, J., UDUPI, S., STOFFEL, D., AND KUNZ, W. Formal system-on-chip verfication: An operation-based methodology and its perspectives in low power design. In *Proc. 23th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)* (2013).

Kurzfassung

Die Verifikation funktionaler Eigenschaften stellt die moderne Hardwareentwicklung vor immer größere Probleme. Tatsächlich ist sie sogar ein limitierender Faktor bei der Weiterentwicklung digitaler Architekturen. Doch während moderne Entwurfsmethoden von kompositionalen Verfahren und der Wiederverwendung älterer Entwürfe profitieren, sind Versuche ähnliche Ansätze auf die Verifikation anzuwenden in der Praxis des Entwurfs auf Registertransferebene (RTL) bislang gescheitert. Die Verfikation des Systemverhaltens kann sich deshalb, nach wie vor, nicht ausschließlich auf Teilverifikationen verlassen, bei denen lediglich die einzelnen Systemkomponenten verifiziert werden. Als Konsequenz daraus benötigt eine ausreichend gründliche Verifikation Wochen oder sogar Monate an Rechenzeit in großen Rechenzentren, da das Verhalten des gesamten Chips auf der Registertransferebene simuliert werden muss.

Ein kurzer historischer Rückblick auf die Modellierung von Schaltungen soll helfen die Ursache des Problems identifizieren zu können. In frühen Modellen wurden alle elektrischen Bauelemente explizit spezifiziert. Obwohl es gründsätzlich möglich ist, auch moderne Schaltungsentwurfe so zu modellieren, sind derartige Modelle viel zu komplex, als dass man aus ihnen, selbst mit computerautomatisierten Methoden, Schlussfolgerungen ziehen könnte. Aus diesem Grunde wird heute die Funktionalität einer digitalen Schaltung anhand abstrakter Modelle beschrieben und verstanden.

Es lassen sich auf der anderen Seite jedoch auch nicht beliebige oder beliebig abstrakte Modelle für die Schaltungsmodellierung verwenden, denn nur wenn sich bestimmte Eigenschaften des abstrakten Modells auf seine Entsprechung auf der physikalischen Ebene übertragen lassen, ist es möglich durch Analyse des abstrakten Modells Rückschlüsse auf sein physikalisches Gegenstück ziehen zu können.

In der Arbeit von Claude Shannon [50] wurde gezeigt wie jeder boolsche Ausdruck in eine entsprechende elektrische Schaltung überführt werden kann. Auf Grundlage dieser Arbeit können boolsche Ausdrücke auch als abstrakte Schaltungsbeschreibung verstanden werden, welche bezüglich des funktionalen Verhaltens die im letzten Abschnitt genannte Eigenschaft besitzt. Aufbauend auf dieser, als boolsche Logik bezeichneten, Modellbildung ist es möglich selbst aus großen und komplexen Schaltungen Schlüsse zu ziehen und dabei gefahrlos physikalische Effekte ignorieren zu können.

Die steigende Komplexität digitaler Schaltungen machte weitere Modellabstraktionen notwendig. Als solche setzte sich schließlich die Registertransferebene durch, auf welcher sich Funktionalität ähnlich beschreiben lässt wie in einem Softwareprogramm. Dank der Entwicklung formaler Äquivalenzbeweise [36] wurde es zudem möglich die Korrektheit von RTL-Modellen in Bezug zu ihren entsprechenden Modellen auf der boolschen Ebene (Gatterebene) zu zeigen, und damit auch zu ihrer physikalischen Verwirklichung. Die nach wie vor immer komplexer werdenden digitalen Schaltungen machen die Einführung noch höherer Abstraktionsebenen für die Schaltungsmodellierung erforderlich. Für diesen Zweck existiert eine große Anzahl an

Beschreibungssprachen mit deren Hilfe sich Modelle auf der sogenannten Systemebene (engl. electronic system level, kurz ESL) beschreiben lassen. In der Regel gibt es jedoch keine eindeutige Beziehung zwischen Modellen auf der Systemebene und entsprechenden Modellen auf der RT-Ebene. Es ist also nicht möglich, anhand eines Systemmodells, Rückschlüsse auf seine Entsprechung auf der RT-Ebene oder gar der physikalischen Ebene zu ziehen. Dieses Problem ist schon lange bekannt und wird auch als semantische Lücke zwischen der Systemebene und der RT-Ebene bezeichnet.

Der in dieser Arbeit vorgeschlagene Formalismus der Pfadprädikatenabstraktion soll dazu beitragen dieses Problem zu lösen, indem er erlaubt eine Beziehung zwischen transaktionsbasierten Systemmodellen und zyklengenauen RTL-Modellen formal zu beschreiben. Darüber hinaus enthält die Arbeit Methoden für die Anwendung des Formalismus, die auf Basis standardisierter Eigenschaftssprachen eingesetzt werden können. Dazu wird in dieser Arbeit ein SAT-basierter Ansatz zur Eigenschaftsprüfung, genannt "Interval Property Checking", kurz IPC, benutzt. Dieser ist besonders gut geeignet, um die nötigen Eigenschaften zu beweisen und kann außerdem selbst bei großen industriellen Entwürfen eingesetzt werden [7].

Pfadprädikatenabstraktion (PPA)

Modellbeschreibungen auf der Systemebene sind zeitabstrakt und unterscheiden sich grundsätzlich von den zyklengenauen Schaltungsbeschreibungen auf der Registertransferebene. Im Unterschied zum Verhältnis zwischen Registertransferebene und Gatterebene, kann das Verhältnis zwischen Systemebene und Registertransferebene, nach unserer Erkenntnis, nicht im allgemeinen als Äquivalenz beschrieben werden. Aus diesem Grunde führen wir stattdessen die Pfradprädikatenabstraktion ein, bei der anhand von Ein- und Ausgabesequenzen eine Beziehung zwischen der transaktionsbasierten Kommunikation von zeitabstrakten Systemmodellen und der zyklengenauen Kommunikation von RT-Modellen hergestellt werden kann.

Zunächst wird die Pfadprädikatenabstraktion durch eine operationelle Färbung endlicher Automaten (FSM) definiert (siehe Abschnitt 4.2). Im Anschluss daran wird gezeigt, wie mit Hilfe von Eigenschaftsprüfungen auch die FSM einer RTL-Beschreibung operationell eingefärbt werden kann, wenn eine vollständige Menge von Eigenschaften vorliegt. Eine Menge von Eigenschaften ist in diesem Zusammenhang dann vollständig, wenn sie das gesamte Eingabe- und Ausgabe-Verhalten der FSM überdeckt. Jede dieser Eigenschaften ist dabei in der Form, dass selbst bei großen industriellen Beispielen, das Beweisverfahren der Eigenschaftsprüfung (IPC) vollständig automatisiert durchgeführt werden kann.

Die operationelle Färbung übernehmen drei Färbefunktionen, welche jedem Zustand, jeder Eingabesequenz und jeder Ausgabesequenz eine individuelle Farbe aus je einer von drei Farbmengen zuordnet, wobei die Farbmengen untereinander disjunkt sind. Durch Anforderungen an die Erstellung dieser Färbefunktionen wird sichergestellt, dass die Berechnung der Farbe einer Ausgabesequenz sowie der Farbe des nachfolgenden Zustandes ausschließlich von der Farbe des aktuellen Zustandes und der Farbe der Eingabesequenz abhängt. Hiervon lässt sich die Pfadprädikatenabstraktion in Form einer FSM, deren Elemente den Farben entsprechen, direkt ableiten.

Es sei an dieser Stelle darauf hingewiesen, dass die ursprüngliche FSM und die FSM der Pfadprädikatenabstraktion sich in ihrem Ein- und Ausgabealphabet unterscheiden. In der Praxis wird dieser Unterschied ausgenutzt, um Transaktionen mit zyklengenauen Sequenzen in

Beziehung zu setzen.

Zwischen einem abstrakten und einem konkreten Modell, welche in der formalen Beziehung einer PPA zueinander stehen, lassen sich Eigenschaften des einen Modells auf das andere übertragen. Dadurch können Beweise von wichtigen, globalen Eigenschaften, die im Falle von Modellen niedrigerer Abstraktionsebenen eine derzeitig noch utopische Rechenleistung benötigen, stattdessen auch auf Basis des viel einfacheren abstrakten Modells durchgeführt werden. Die in dieser Arbeit vorgeschlagene Abstraktion durch PPA stellt damit ein Werkzeug dar, mit dessen Hilfe man nicht nur zuverlässige Modelle erhält, sondern auch Beweise, die auch für die Eigenschaften des entsprechenden konkreten Modells gültig sind.

Natürlich ist eine zuverlässige Übertragung von Eigenschaften für Einzelkomponenten noch nicht ausreichend, um die semantische Lücke zu schließen. Aus diesem Grund muss die Zuverlässigkeit auch auf ein System, dass aus einzelnen zeitabstrakten Modellen zusammengesetzt ist, erweiterbar sein. Eine sorgälltige Modellierung der Kommunikation zwischen den einzelnen Modellen ist daher unerlässlich. In Kapitel 5 wird diese Erweiterung eingeführt und dabei aus einer kleinen Menge an Synchronisationsmechanismen einer ausgewählt, welcher dazu genutzt wird die intermodulare Kommunikation abzubilden.

Anwendungsverfahren

Anhand mehrerer Fallstudien konnte gezeigt werden, wie durch vollständige IPC die Pfadprädikatenabstraktion aus existierenden RTL-Beschreibungen erzeugt werden kann. Von unserem Standpunkt aus ist dies das Produkt einer systematisch durchgeführten formalen Verifikation des zugrundeliegenden RTL-Modells.

Der damit verbundene Arbeitsaufwand ist zwar erheblich, unsere Erfahrungen mit einer Vielzahl von IPC-Verifkationen - darunter auch mehrere Beispiele aus der Industrie - zeigen allerdings, dass der Aufwand tendenziell kleiner ist als der einer traditionellen, simulationsbasierten Verifikation [6]. Zieht man nun auch die Zeit für die Systemverifikation mit in die Betrachtung ein, so lässt sich ein Wechsel zu formalen Methoden gut begründen.

In Verbindung mit neuentwickelten Modulbeschreibungen ist es aber umständlich und unpassend zuerst die konkreten RTL-Beschreibungen zu erstellen, um davon dann ein abstraktes Modell für die Systemverifikation zu erzeugen. Zur Lösung dieses Problems wird deshalb in Kapitel 7 ein neues Entwurfsverfahren vorgestellt. In diesem wird PPA nicht als eine Abstraktion, sondern stattdessen als eine Verfeinerung verwendet.

Um die Semantik einer, für PPA geeigneten, Systemebenenbeschreibung erfassen zu können, ist in diesem Zusammenhang die Sprache AML (siehe Appendix A) entwickelt worden. Sie wurde speziell für den Zweck entwickelt, dass Sprachen, welche für die Beschreibung von Modellen auf der Systemebene verwendet werden, wie beispielsweise SystemC, durch eine Übersetzung nach AML eine klar definierte Semantik erhalten, die es ermöglicht das jeweilige Systemmodell als Schaltungsbeschreibung zu verstehen.

Unterstützt wird das oben erwähnte Entwurfsverfahren durch ein von uns entwickeltes Softwareprogramm namens Refinement Synthesizer. Dieses erzeugt aus Modulbeschreibungen, die in AML vorliegen, eine Menge von Eigenschaften, die automatisch vollständig ist. Die hieraus erhaltenen Eigenschaften können als Anforderungen an die bevorstehenden Implementierung des Moduls auf der RT-Ebene verstanden werden. Folglich bezeichnen wir jede Implementierung die alle diese Eigenschaften besitzt als “correct by construction”. Das ursprüngliche

Modul auf der Systemebene stellt dann für jede solche Implementierung die Pfadprädikatenabstraktion dar.

Es sei an dieser Stelle ausdrücklich darauf hingewiesen, dass sich das in dieser Arbeit vorgestellte Entwurfsverfahren und eine High-Level-Synthese im beachtlichen Maße unterscheiden. Die aus AML-Beschreibungen generierten Eigenschaften enthalten nämlich keinerlei zyklengenauen Informationen. In unserem Verfahren ergeben sich derartige Informationen erst aus einer, im Anschluss getätigten, manuellen Verfeinerung. Wir können uns zwar vorstellen, dass sich unser Formalismus gut mit datenpfadbasierten Methoden der High-Level-Synthese verbinden lässt, im Allgemeinen vertreten wir jedoch den Standpunkt, dass solche Methoden nicht immer erwünscht sind, da einige grundlegende Entscheidungen noch nicht auf der Systemebene getroffen wurden.

Perspektiven für Energiesparende Entwürfe

Die vorgestellten Methoden liefern nicht nur ein zuverlässiges Systemmodell, sondern ermöglichen auch eine formale Verfikation einer zyklengenaue Implementierung dieses Systems. Die Verifikation stellt hierbei auch eine Dokumentation dar, die beschreibt wie das Systemmodell mit der zyklengenauen Implementierung zusammenhängt. Diese Ergebnisse können auch bei anderen Aufgaben, wie zum Beispiel der Optimierung nichtfunktionaler Eigenschaften, nützlich sein. Ein Anwendungsfall dieser Art stellt beispielsweise der Entwurf von digitalen Systemen mit anspruchsvollen Anforderungen an die Energieeffizienz dar.

Energieabschätzung

Energieverbrauch ist im Allgemeinen sehr abhängig von der Wahl der Systemarchitektur. Es wäre daher von Vorteil möglichst frühzeitig ein Modell zur Verfügung zu haben, mit dem sich der Energieverbrauch abschätzen lässt, so dass das Wissen darüber in die Wahl der geeigneten Systemarchitektur einfließen kann.

Einen Ansatzpunkt dafür liefert die Feststellung, dass bei der Erstellung der Pfadprädikatenabstraktion der Kontrollfluss in Operationen und Operationsmodi zerlegt wird. Kombiniert man dies mit der Annahme, dass der Energieverbrauch hauptsächlich von dem Kontrollflussverhalten abhängig ist, während abweichende Datenwerte nur geringe Unterschiede verursachen, kann man zu dem Schluss kommen, dass die Pfadprädikatenabstraktion geeignet sein müsste, um auch als Energiemodel dienen zu können. Genau diese Eignung haben wir anhand einer Fallstudie geprüft. Für diesen Zweck verwendeten wir eine Schaltung, die den größten gemeinsamen Teiler berechnet. Um unseren Schluss zu prüfen, haben wir eine große Anzahl an Energiemessungen mit wechselnden Datenwerten und derselben Operation durchgeführt. Die Ergebnisse dieser Studie werden in Abschnitt 8.1.1 vorgestellt.

Abhängigkeitsanalyse für Energieersparnisse

Wie bereits erwähnt stellt die Pfadprädikatenabstraktion eine Menge von formalen Eigenschaften dar, die ein RTL-Modul besitzen muss, um seine Aufgaben im System zu erfüllen. Zusammen mit zyklengenauen Beschreibungen der externen Kommunikationsprotokolle wird dies zu einer vollständigen Anforderung für die Korrektheit einer RTL-Beschreibung. Daraus lässt sich

folgern, dass jenes Verhalten, welches zu diesem Eingabe-/Ausgabe-Verhalten nichts beiträgt, irrelevant oder redundant ist und deshalb wegoptimiert werden kann. Ein automatisiertes Verfahren zu so einer Abhängigkeitsanalyse wird in Abschnitt 8.2 vorgestellt. Im selben Abschnitt werden auch die Ergebnisse präsentiert, die auf Basis einer Schaltung von Alcatel Lucent - einem Demonstrator aus dem BMBF Projekt EnerSave - gewonnen wurden.

Verfahren für energiesparende Entwürfe

Das neue Entwurfsverfahren, welches in Kapitel 7 vorgestellt wird, hat beachtliche Vorteile beim Entwurf energiesparender Systeme. Anhand der Systembeschreibung kann eine Menge an Eigenschaften abgeleitet werden, die automatisch gegen eine RTL-Implementierung geprüft werden kann. Zwei beliebige Implementierungen können für die gleiche Aufgabe eingesetzt werden, wenn beide diese Eigenschaften erfüllen. In der Gewissheit, dass eine Optimierung die Funktionalität der ursprünglichen Beschreibung nicht ändert, werden Optimierungen, die sonst vielleicht zu einem viel zu hohen Verifikationsaufwand geführt hätten, dank dieser Methodik erst möglich. Die dabei entstandenen Energieersparnisse werden in Abschnitt 8.3 vorgestellt.

Appendix A

AML Syntax

```
<architectural_description> ::= <top_element>*

<top_element> ::= <enum_def> | <const_def> | <func_def> | <system_def> | <module_def>

<enum_def> ::= "enum" NAME "=" "{" NAME ["," NAME]* "}" ";"
   /*left side becomes a valid data type (ENUM_TYPE), right side this data types values (ENUM_VALUE)*/

<const_def> ::= <const_num_def> | <const_bool_def> | <const_enum_def>
<const_num_def> ::= "const" "int" NAME "=" NUM ";"
<const_bool_def> ::= "const" "bool" NAME "=" BOOL ";"
<const_enum_def> ::= "const" ENUM_TYPE NAME "=" ENUM_VALUE ";"
   /*must be set to an enum value of the used type*/

<func_def> ::= <func_header> <stmt_list> "}" ";"
<func_header> ::= "func" "<" <datatype> ">" NAME "(" <arg_list>  ")" "{" <func_var_list>
<arg_list> ::= [<var_decl>] ["," <var_decl>]*
<func_var_list> ::= [<var_decl> ";"]*

<arrayindex> ::= "[" <num_expr> "]"    /*<num_expr> must be static*/

<sync_key> ::= "sync" | "async"

<datatype> ::= "int" | "bool" | ENUM_TYPE

<structdef>  ::= <datatype> NAME [<arrayindex>]

<in_decl> ::= [<sync_key>] "in" "<" <datatype> ">" NAME [<arrayindex>] ";"
          |   [<sync_key>] "in" "<" <structdef>* ">" NAME [<arrayindex>] ";"

<out_decl> ::= [<sync_key>] "out" "<" <datatype> ">" NAME [<arrayindex>] ";"
           |   [<sync_key>] "out" "<" <structdef>* ">" NAME [<arrayindex>] ";"

<shared_decl> ::= "shared" "<" <datatype> ">" NAME [<arrayindex>] ";"
              |   "shared" "<" <structdef>* ">" NAME [<arrayindex>] ";"

<submod_decl> ::= "module" "<" NAME ">" NAME [<arrayindex>] ";"
              |   "system" "<" NAME ">" NAME [<arrayindex>] ";"

<var_decl> ::= "var" "<" <datatype> ">" NAME [<arrayindex>]
           |   "var" "<" [<structlist>] ">" NAME [<arrayindex>]

<var_ref> ::= <var_ref_simple> | <var_ref_struct>
<var_ref_simple> ::= NAME [<arrayindex>]
<var_ref_struct> ::= NAME [<arrayindex>] "." NAME [<arrayindex>]


<system_def> ::= "system" NAME "{" <system_port>* <connect_decl> "}" ";"

<system_port> ::= <in_decl> | <out_decl> | <shared_decl> | <submod_decl>

```

```
<connect_decl> ::= "connect" "{" <connection>* "}"
               |   <for_header> "{" <connect_decl> "}"

<connection>
    ::= NAME [<arrayindex>] "." NAME [<arrayindex>] "->" NAME [<arrayindex>] "." NAME [<arrayindex>] "
     ;"
    |   NAME [<arrayindex>] "->" NAME [<arrayindex>] ";"
    |   NAME [<arrayindex>] "." NAME [<arrayindex>] "->" NAME [<arrayindex>] ";"
    |   NAME [<arrayindex>] "->" NAME [<arrayindex>] "." NAME [<arrayindex>] ";"


<module_def> ::= "module" NAME "{" <module_port>* <fsm_decl> "}" ";"

<module_port> ::= <in_decl> | <out_decl>

<fsm_decl> ::= "fsm" NAME "{" "states" "=" "{" NAME ["," NAME]* "}" ";" [<var_decl> ";"]* <
    behave_init> <behave_part>* "}"
<behave_init> ::= "@" "init" ":" <stmt_list>
<behave_part> ::= "@" NAME ":" <stmt_list>


<stmt_list> ::= <a_stmt>+
<a_stmt> ::= <assign_stmt> | <if_stmt> | <case_stmt> | <read_stmt> | <write_stmt> | <for_stmt>

<assign_stmt> ::= <var_ref> = <expr>";" | "++" <var_ref>";" | "--" <var_ref>";"
  /*variable and expression must have matching type, increment and decrement operators only legal for numeric variables*/

<if_stmt> ::= <if_part> <elsif_part>* [<else_part>]
<if_part> ::= "if" "(" <bool_expr> ")" "{" <stmt_list> "}"
<elsif_part> ::= "else" "if" "(" <bool_expr> ")" "{" <stmt_list> "}"
<else_part> ::= "else" "{" <stmt_list> "}"

<case_stmt> ::= "switch" "(" <expr> ")" "{" <case_element>+ [<case_default>] "}"
<case_element> ::= "case" <expr> ":" <stmt_list>
<case_default> ::= "default" ":" <stmt_list>

<read_stmt> ::= "read" "(" NAME [<arrayindex>] ")" ";"    /*must be a valid in port*/
<write_stmt> ::= "write" "(" NAME [<arrayindex>] ")" ";"  /*must be a valid out port*/

<for_stmt> ::= <for_header> "{" <stmt_list> "}"
<for_header> ::= "for" "(" <num_expr> ".." <num_expr> ")"     /*<num_expr> must be static*/
<forcnt> ::= @ | <forcnt>+ /*used within a for loop to refer to the current loop count*/

<func_ref> ::= NAME "(" <arg_ref_list> ")"
<arg_ref_list> ::= <expr>*




<expr> ::= <num_expr> | <bool_expr> | <enum_expr>

<bool_expr> ::= BOOL
            | "(" <bool_expr> ")"
            | "pending" "(" NAME [<arrayindex>] ")" /*Must be a valid port*/
            | <func_ref> /*to a Boolean function*/
                       | <var_ref> /*to a Boolean variable*/
            | <bool_expr> "and" <bool_expr>
            | <bool_expr> "or" <bool_expr>
            | <bool_expr> "nor"  <bool_expr>
            | <bool_expr> "nand" <bool_expr>
            | <bool_expr> "xor" <bool_expr>
            | <bool_expr> "xnor" <bool_expr>
            | "not" <bool_expr>
            | <expr> "==" <expr> /*expressions must be of same type*/
            | <expr> "!=" <expr> /*expressions must be of same type*/
            | <num_expr> ">"  <num_expr>
            | <num_expr> ">=" <num_expr>
            | <num_expr> "<"  <num_expr>
            | <num_expr> "<=" <num_expr>
```

```
119
120
121 <num_expr> ::= NUM
122             | "(" <num_expr> ")"
123             | <func_ref> /*to a numeric function*/
124             | <forcnt> /*the loop counter*/
125             | <var_ref> /*to a numeric variable*/
126             | <num_expr> "+" <num_expr>
127             | <num_expr> "-" <num_expr>
128             | <num_expr> "*" <num_expr>
129             | <num_expr> "/" <num_expr> /*integer division*/
130             | <num_expr> "%" <num_expr> /*remainder of integer division*/
131             | "-" <num_expr>
132             | <num_expr> "&" <num_expr> /*bitwise and*/
133             | <num_expr> "|" <num_expr> /*bitwise or*/
134             | <num_expr> "^" <num_expr> /*bitwise xor*/
135             | <num_expr> "<<" <num_expr> /*shift left*/
136             | <num_expr> ">>" <num_expr> /*shift right*/
137             | "~" <num_expr> /*bit inversion*/
138
139 <enum_expr> ::= ENUM_VALUE
140              | "(" <enum_expr> ")"
141              | <func_ref> /*to a function returning an enum constant*/
142              | <var_ref> /*to an enum variable*/
```

Appendix B

Design Flow Demonstrator: Monitor

B.1 Architectural Description

```
1  MODULE monitor {
2    in<bool> oof;
3    in<bool intmod, int set, int reset> setup;
4    out<bool> lof;
5
6    FSM behavior {
7      states = {REGULAR, INTEGRATING};
8      var<int> OOFcnt; var<int> IFcnt;
9      var<int> set; var<int> reset;
10   @init:
11     nextstate = REGULAR;
12     OOFcnt = 0;
13     IFcnt = 0;
14     lof = false;
15     set = 7;
16     reset = 7;
17   @REGULAR:
18     read(oof);
19     if (oof) {
20       IFcnt = 0;
21       if (OOFcnt >= set) {
22         lof = true;
23       } else {
24         OOFcnt++; }
25     } else {
26       OOFcnt = 0;
27       if (IFcnt >= reset) {
28         lof = false;
29       } else {
30         IFcnt++; }
31     }
32     write(lof);
33     if (pending(setup)){
34       read(setup);
35       set = setup.set;
36       reset = setup.reset;
37       if (setup.intmod) {
```

```
38          nextstate = INTEGRATING; }
39      }
40    @INTEGRATING:
41      read(oof);
42      if (oof) {
43        IFcnt = 0;
44        if (OOFcnt >= set) {
45          lof = true;
46        } else {
47          OOFcnt++; }
48      } else {
49        if (IFcnt >= reset) {
50              OOFcnt = 0;
51          lof = false;
52        } else {
53          IFcnt++; }
54      }
55      write(lof);
56      if (pending(setup)){
57        read(setup);
58        set = setup.set;
59        reset = setup.reset;
60        if (not setup.intmod) {
61          nextstate = REGULAR;}
62      }
63    }
64  };
```

B.2 Operation Property Suite

```
1  -- INCOMING SYNC SIGNALS (1-cycle macros) --
2  macro oof_sync : boolean := end macro;
3  macro setup_sync : boolean := end macro;
4  macro lof_sync : boolean := end macro;
5
6  -- OUTGOING SYNC SIGNALS (1-cycle macros) --
7  macro oof_notify : boolean := end macro;
8  macro setup_notify : boolean := end macro;
9  macro lof_notify : boolean := end macro;
10
11 -- INCOMING DP SIGNALS --
12 macro oof_sig : boolean := end macro;
13 macro setup_sig_intmod : boolean := end macro;
14 macro setup_sig_set : numeric := end macro;
15 macro setup_sig_reset : numeric := end macro;
16
17 -- OUTGOING DP SIGNALS --
18 macro lof_sig : boolean := end macro;
19
20 -- VISIBLE REGISTERS --
21 macro lof : boolean := end macro;
22 macro OOFcnt : numeric := end macro;
23 macro IFcnt : numeric := end macro;
24 macro set : numeric := end macro;
25 macro reset : numeric := end macro;
```

```
26
27
28 -- MODES --
29 macro REGULAR_R_OOF : boolean := end macro;
30 macro REGULAR_W_LOF : boolean := end macro;
31 macro INTEGRATING_R_OOF : boolean := end macro;
32 macro INTEGRATING_W_LOF : boolean := end macro;
33
34
35
36 -- OPERATION PROPERTIES --
37
38 property reset is
39   assume: reset_sequence;
40   prove:
41     during[t+1, t_end-1]: oof_notify = false;
42     at t_end: oof_notify = true;
43     during[t+1, t_end]: setup_notify = false;
44     during[t+1, t_end]: lof_notify = false;
45
46     at t_end: lof = false;
47     at t_end: OOFcnt = 0;
48     at t_end: IFcnt = 0;
49     at t_end: set = 5;
50     at t_end: reset = 5;
51     at t_end: REGULAR_R_OOF;
52 end property;
53
54 property wait_1 is
55   assume:
56     at t: REGULAR_R_OOF;
57     at t: not oof_sync;
58   prove:
59     at t+1: oof_notify = true;
60     at t+1: setup_notify = false;
61     at t+1: lof_notify = false;
62
63     at t+1: lof = lof @ t;
64     at t+1: OOFcnt = OOFcnt @ t;
65     at t+1: IFcnt = IFcnt @ t;
66     at t+1: set = set @ t;
67     at t+1: reset = reset @ t;
68     at t+1: REGULAR_R_OOF;
69 end property;
70
71 property op_2 is
72   assume:
73     at t: REGULAR_R_OOF;
74     at t: oof_sync;
75     at t: (OOFcnt >= set) and oof_sig;
76   prove:
77     during[t+1, t_end]: oof_notify = false;
78     during[t+1, t_end]: setup_notify = false;
79     during[t+1, t_end-1]: lof_notify = false;
80     at t_end: lof_notify = true;
```

```
81
82      at t_end: lof = true;
83      at t_end: OOFcnt = OOFcnt @ t;
84      at t_end: IFcnt = 0;
85      at t_end: set = set @ t;
86      at t_end: reset = reset @ t;
87      at t_end: REGULAR_W_LOF;
88  end property;
89
90  property op_3 is
91    assume:
92      at t: REGULAR_R_OOF;
93      at t: oof_sync;
94      at t: (not (OOFcnt >= set)) and oof_sig;
95    prove:
96      during[t+1, t_end]: oof_notify = false;
97      during[t+1, t_end]: setup_notify = false;
98      during[t+1, t_end-1]: lof_notify = false;
99      at t_end: lof_notify = true;
100
101     at t_end: lof = lof @ t;
102     at t_end: OOFcnt = (OOFcnt @ t) + 1;
103     at t_end: IFcnt = 0;
104     at t_end: set = set @ t;
105     at t_end: reset = reset @ t;
106     at t_end: REGULAR_W_LOF;
107 end property;
108
109 property op_4 is
110   assume:
111     at t: REGULAR_R_OOF;
112     at t: oof_sync;
113     at t: (IFcnt >= reset) and (not oof_sig);
114   prove:
115     during[t+1, t_end]: oof_notify = false;
116     during[t+1, t_end]: setup_notify = false;
117     during[t+1, t_end-1]: lof_notify = false;
118     at t_end: lof_notify = true;
119
120     at t_end: lof = false;
121     at t_end: OOFcnt = 0;
122     at t_end: IFcnt = IFcnt @ t;
123     at t_end: set = set @ t;
124     at t_end: reset = reset @ t;
125     at t_end: REGULAR_W_LOF;
126 end property;
127
128 property op_5 is
129   assume:
130     at t: REGULAR_R_OOF;
131     at t: oof_sync;
132     at t: (not (IFcnt >= reset)) and (not oof_sig);
133   prove:
134     during[t+1, t_end]: oof_notify = false;
135     during[t+1, t_end]: setup_notify = false;
```

```
136     during[t+1, t_end-1]: lof_notify = false;
137     at t_end: lof_notify = true;
138
139     at t_end: lof = lof @ t;
140     at t_end: OOFcnt = 0;
141     at t_end: IFcnt = (IFcnt @ t) + 1;
142     at t_end: set = set @ t;
143     at t_end: reset = reset @ t;
144     at t_end: REGULAR_W_LOF;
145 end property;
146
147 property wait_6 is
148   assume:
149     at t: REGULAR_W_LOF;
150     at t: not lof_sync;
151   prove:
152     at t+1: oof_notify = false;
153     at t+1: setup_notify = false;
154     at t+1: lof_notify = true;
155
156     at t+1: lof = lof @ t;
157     at t+1: OOFcnt = OOFcnt @ t;
158     at t+1: IFcnt = IFcnt @ t;
159     at t+1: set = set @ t;
160     at t+1: reset = reset @ t;
161     at t+1: REGULAR_W_LOF;
162 end property;
163
164 property op_7 is
165   assume:
166     at t: REGULAR_W_LOF;
167     at t: lof_sync;
168     at t: not setup_sync;
169   prove:
170     at t: lof_sig = lof;
171     during[t+1, t_end-1]: oof_notify = false;
172     at t_end: oof_notify = true;
173     during[t+1, t_end]: setup_notify = false;
174     during[t+1, t_end]: lof_notify = false;
175
176     at t_end: lof = lof @ t;
177     at t_end: OOFcnt = OOFcnt @ t;
178     at t_end: IFcnt = IFcnt @ t;
179     at t_end: set = set @ t;
180     at t_end: reset = reset @ t;
181     at t_end: REGULAR_R_OOF
182 end property;
183
184 property op_8 is
185   assume:
186     at t: REGULAR_W_LOF;
187     at t: lof_sync;
188     at t: setup_sync;
189     at t: not setup_sig_intmod;
190   prove:
```

```
191     at t: lof_sig = lof;
192     during[t+1, t_end-1]: oof_notify = false;
193     at t_end: oof_notify = true;
194     during[t+1, t_end]: setup_notify = false;
195     during[t+1, t_end]: lof_notify = false;
196
197     at t_end: lof = lof @ t;
198     at t_end: OOFcnt = OOFcnt @ t;
199     at t_end: IFcnt = IFcnt @ t;
200     at t_end: set = setup_sig_set @ t;
201     at t_end: reset = setup_sig_reset @ t;
202     at t_end: REGULAR_R_OOF;
203 end property;
204
205 property op_9 is
206   assume:
207     at t: REGULAR_W_LOF;
208     at t: lof_sync;
209     at t: setup_sync;
210     at t: setup_sig_intmod;
211   prove:
212     at t: lof_sig = lof;
213     during[t+1, t_end-1]: oof_notify = false;
214     at t_end: oof_notify = true;
215     during[t+1, t_end]: setup_notify = false;
216     during[t+1, t_end]: lof_notify = false;
217
218     at t_end: lof = lof @ t;
219     at t_end: OOFcnt = OOFcnt @ t;
220     at t_end: IFcnt = IFcnt @ t;
221     at t_end: set = setup_sig_set @ t;
222     at t_end: reset = setup_sig_reset @ t;
223     at t_end: INTEGRATING_R_OOF;
224 end property;
225
226 property wait_10 is
227   assume:
228     at t: INTEGRATING_R_OOF;
229     at t: not oof_sync;
230   prove:
231     at t+1: oof_notify = true;
232     at t+1: setup_notify = false;
233     at t+1: lof_notify = false;
234
235     at t+1: lof = lof @ t;
236     at t+1: OOFcnt = OOFcnt @ t;
237     at t+1: IFcnt = IFcnt @ t;
238     at t+1: set = set @ t;
239     at t+1: reset = reset @ t;
240     at t+1: INTEGRATING_R_OOF;
241 end property;
242
243 property op_11 is
244   assume:
245     at t: INTEGRATING_R_OOF;
```

```
246     at t: oof_sync;
247     at t: (OOFcnt >= set) and oof_sig;
248   prove:
249     during[t+1, t_end]: oof_notify = false;
250     during[t+1, t_end]: setup_notify = false;
251     during[t+1, t_end-1]: lof_notify = false;
252     at t_end: lof_notify = true;
253
254     at t_end: lof = true;
255     at t_end: OOFcnt = OOFcnt @ t;
256     at t_end: IFcnt = 0;
257     at t_end: set = set @ t;
258     at t_end: reset = reset @ t;
259     at t_end: INTEGRATING_W_LOF;
260 end property;
261
262 property op_12 is
263   assume:
264     at t: INTEGRATING_R_OOF;
265     at t: oof_sync;
266     at t: (not (OOFcnt >= set)) and oof_sig;
267   prove:
268     during[t+1, t_end]: oof_notify = false;
269     during[t+1, t_end]: setup_notify = false;
270     during[t+1, t_end-1]: lof_notify = false;
271     at t_end: lof_notify = true;
272
273     at t_end: lof = lof @ t;
274     at t_end: OOFcnt = (OOFcnt @ t) + 1;
275     at t_end: IFcnt = 0;
276     at t_end: set = set @ t;
277     at t_end: reset = reset @ t;
278     at t_end: INTEGRATING_W_LOF;
279 end property;
280
281 property op_13 is
282   assume:
283     at t: INTEGRATING_R_OOF;
284     at t: oof_sync;
285     at t: (IFcnt >= reset) and (not oof_sig);
286   prove:
287     during[t+1, t_end]: oof_notify = false;
288     during[t+1, t_end]: setup_notify = false;
289     during[t+1, t_end-1]: lof_notify = false;
290     at t_end: lof_notify = true;
291
292     at t_end: lof = false;
293     at t_end: OOFcnt = 0;
294     at t_end: IFcnt = IFcnt @ t;
295     at t_end: set = set @ t;
296     at t_end: reset = reset @ t;
297     at t_end: INTEGRATING_W_LOF;
298 end property;
299
300 property op_14 is
```

```
  assume:
    at t: INTEGRATING_R_OOF;
    at t: oof_sync;
    at t: (not (IFcnt >= reset)) and (not oof_sig);
  prove:
    during[t+1, t_end]: oof_notify = false;
    during[t+1, t_end]: setup_notify = false;
    during[t+1, t_end-1]: lof_notify = false;
    at t_end: lof_notify = true;

    at t_end: lof = lof @ t;
    at t_end: OOFcnt = OOFcnt @ t;
    at t_end: IFcnt = (IFcnt @ t) + 1;
    at t_end: set = set @ t;
    at t_end: reset = reset @ t;
    at t_end: INTEGRATING_W_LOF;
end property;

property wait_15 is
  assume:
    at t: INTEGRATING_W_LOF
    at t: not lof_sync;
  prove:
    at t+1: oof_notify = false;
    at t+1: setup_notify = false;
    at t+1: lof_notify = true;

    at t+1: lof = lof @ t;
    at t+1: OOFcnt = OOFcnt @ t;
    at t+1: IFcnt = IFcnt @ t;
    at t+1: set = set @ t;
    at t+1: reset = reset @ t;
    at t+1: INTEGRATING_W_LOF;
end property;

property op_16 is
  assume:
    at t: INTEGRATING_W_LOF;
    at t: lof_sync;
    at t: not setup_sync;
  prove:
    at t: lof_sig = lof;
    during[t+1, t_end-1]: oof_notify = false;
    at t_end: oof_notify = true;
    during[t+1, t_end]: setup_notify = false;
    during[t+1, t_end]: lof_notify = false;

    at t_end: lof = lof @ t;
    at t_end: OOFcnt = OOFcnt @ t;
    at t_end: IFcnt = IFcnt @ t;
    at t_end: set = set @ t;
    at t_end: reset = reset @ t;
    at t_end: INTEGRATING_R_OOF;
end property;

```

```
property op_17 is
  assume:
    at t: INTEGRATING_W_LOF;
    at t: lof_sync;
    at t: setup_sync;
    at t: setup_sig_intmod;
  prove:
    at t: lof_sig = lof;
    during[t+1, t_end-1]: oof_notify = false;
    at t_end: oof_notify = true;
    during[t+1, t_end]: setup_notify = false;
    during[t+1, t_end]: lof_notify = false;

    at t_end: lof = lof @ t;
    at t_end: OOFcnt = OOFcnt @ t;
    at t_end: IFcnt = IFcnt @ t;
    at t_end: set = setup_sig_set @ t;
    at t_end: reset = setup_sig_reset @ t;
    at t_end: INTEGRATING_R_OOF;
end property;

property op_18 is
  assume:
    at t: INTEGRATING_W_LOF;
    at t: lof_sync;
    at t: setup_sync;
    at t: not setup_sig_intmod;
  prove:
    at t: lof_sig = lof;
    during[t+1, t_end-1]: oof_notify = false;
    at t_end: oof_notify = true;
    during[t+1, t_end]: setup_notify = false;
    during[t+1, t_end]: lof_notify = false;

    at t_end: lof = lof @ t;
    at t_end: OOFcnt = OOFcnt @ t;
    at t_end: IFcnt = IFcnt @ t;
    at t_end: set = setup_sig_set @ t;
    at t_end: reset = setup_sig_reset @ t;
    at t_end: REGULAR_R_OOF;
end property;
```

Appendix C

Design Flow Student Project

C.1 Architectural Description

Available @ http://www.eit.uni-kl.de/en/eis/research/ppa

C.2 Operation Property Suite

Available @ http://www.eit.uni-kl.de/en/eis/research/ppa

C.3 RTL implementation

Available @ http://www.eit.uni-kl.de/en/eis/research/ppa