Property-Driven Design
A new approach for hardware design

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

Doktor der Ingenieurswissenschaften (Dr.-Ing.)

genehmigte Dissertation von

Tobias Ludwig
geboren in Merzig, Deutschland

D 386

Datum der mündlichen Prüfung:	16.07.21
Dekan des Fachbereichs:	Prof. Dr.-Ing. Ralph Urbansky
Vorsitzender der Prüfungskommission:	Prof. Dipl.-Ing. Dr. Gehard Fohler
Gutachter:	Prof. Dr.-Ing. Wolfgang Kunz
	Prof. Dr.-Ing. Ulrich Heinkel

Acknowledgments

This thesis is the result of several years of research carried out while I have been part of the “electronic design automation group” at the University of Kaiserslautern. This work benefits from the close collaboration of our group during this period. Furthermore I’d like to thank all my student and research assistants that contributed to my work within the recent years.

Most important, I would like to thank my supervisors, Prof. Wolfgang Kunz and Prof. Dominik Stoffel, who have made my research possible. Thank you for all the good advice, many hours in the seminar room and the possibilities I had during my time as a PhD. I am really glad that I had the chance to work with you.

Specifically, I would like to mention: Joakim Urdahl for supporting me as a colleague, mentor and friend; Nawras Altaleb, Ibrahim Alkoudsi, Lucas Deutschmann, Alex Wezel and Paulius Morkunas for their major code contributions; Michael Schwarz for the reviews, as colleague and friend.
Many thanks also to Prof. Ulrich Heinkel for his in-depth analysis in the review of this thesis and to Prof. Gerhard Fohler for chairing the examination procedures.

To all my friends that where always ready to support me and listen to my ideas and dreams. A special thanks goes to Frank and Joachim for following along the same path with me since school and to Lea for always supporting me.
To my parents who made this entire journey possible with their endless support during all these years.

To everyone that made my time in Kaiserslautern cheerful.

Kaiserslautern, 30.12. 2020
Tobias Ludwig

Contents

Chapter 1

Introduction

Today, the Register Transfer Level (RTL) is still the central anchor point in most methodologies for designing the hardware of a System-on-Chip (SoC) or Embedded System. After decades of evolutionary progress, industrial design environments for hardware have reached a high level of sophistication. Yet — despite all the progress related to design at the Electronic System Level (ESL) with new system-level languages and design environments integrating virtual prototypes and advanced verification techniques — the actual conceptual process of RTL design based on languages like VHDL and Verilog has not much changed over the last decades.

Although High-Level Synthesis (HLS) has evolved to be applicable in certain domains, such as implementations of signal processing algorithms, the bulk of RTL designs in industry is still created manually, starting from informal specifications such as natural language descriptions, conceptual diagrams of finite state machines (FSMs), timing diagrams, flow charts et cetera. In almost all practical settings, RTL descriptions are the reference point for verification. Even in HLS-based flows, usually, the RTL description remains the "golden model" for sign-off. System-level models, on the other hand, are usually considered "prototypes" employed for early assessments of functional and non-functional design goals. The extent to which these prototypes really reflect the characteristics of the final RTL implementation, however, is often difficult to determine. In fact, this disconnection between System Level and RTL implementation is one of the main risk factors in today's industrial design flows.

In the scope of this thesis, we work towards a new approach for RTL design, starting from a system-level description. The proposed methodology is based on a manual design process and does not impose any restrictions on designers w.r.t. the developed implementations. However, new tool support is provided, based on formal techniques for verification and abstraction, ensuring that both the hardware IP and the verification IP are developed in a systematic and compositional manner. The most important benefit of the proposed method is that a formally well-defined relationship is established between the RTL description and the system-level model used as the starting point for our method. This aspect is key. It allows us to safely deviate from today's design practices and to move towards employing abstract system models as *golden* design models.

The proposed approach called "Property First Hardware Design" or "Property-Driven Hardware Design (PDD)" takes inspiration from software engineering and widely used practices in software development like "Test First Development" or "Test-Driven Development (TDD)"" [1]. The TDD paradigm deviates from the classical V-model for software development and is based on the conviction that the actual software development process is positively affected by creating software tests *prior* to writing the actual program code. Such gray-box tests have shown to lead to higher fault coverage when compared to the classical white-box tests of the V-model.

Transferring this idea to the hardware domain suggests a methodology that integrates verification steps early and systematically into the design process. The role of software tests in TDD can be assumed in hardware design by *properties* or *assertions* formulated in property languages like the Property Specification Language (PSL) or System Verilog Assertions (SVA). A distinct feature of our methodology – and under this aspect the analogy with the software domain is no longer valid – is the systematic creation of properties for which abstract descriptions are automatically generated from the design's system-level model. Adapting this to the special needs of RTL hardware design and running modern formal verification techniques in the background of our design environment results in a systematic and intuitive design procedure (without restricting the designer's freedom). Finally, this allows for formal statements about the functional correctness of the obtained RTL implementation.

There is a key attribute to the methodology that is formally guaranteed by the specific way the properties are initially generated and later refined: when all created properties are proven on the developed design it can be concluded with mathematical rigor that the RTL design represents a correct refinement of the system-level description from which the properties were generated. This is ensured by a well-defined formal relationship established between the abstract model and its concrete implementation, called Path Predicate Abstraction (PPA) [2]. Conversely, this means that after successfully completing the design process the system-level model represents a formally *sound* abstraction of the implementation.

Establishing system-level models as PPAs of RTL designs can change the role of system-level models fundamentally: rather than being "prototypes" and having only a loosely defined relationship with the implementation they may now be trusted as "design models", just like RTL design models are trusted to be sound abstractions of the underlying gate level (by merit of formal equivalence checking). Based on Path Predicate Abstraction (PPA), the theoretical framework we provide establishes a formal link between the abstract system model and the concrete RTL design. In our methodology, the semantics of the system model is defined by compositional PPA.

We show how this can be used in a realistic design methodology based on standard languages. This is subject of Chap. 3. We introduce a subset of SystemC called SystemC-PPA for describing system-level models and present the PDD flow in which abstract property descriptions are automatically generated from SystemC-PPA and subsequently refined during the design process. A major contribution of this work is the open-source tool *DeSCAM* that reads the SystemC-PPA description and automatically generates the properties. In Chap. 5 we introduce the tool and describe how the properties are generated correct-by-construction. A challenging task is to implement a *pipelined* design from an abstract model. We develop a special methodology for this problem in Chap. 6. It helps the designer to ensure a sound implementation and to solve possible hazards.

In Chap. 7 we report on case studies conducted for several industrial and open-source designs. Our experimental results show that fully verified RTL designs can be created by

PDD as formally well-defined refinements of system-level descriptions. All design steps are based on standard languages and only employ state-of-the-art formal property checking, as it is commercially available. The manual effort for design and verification is reduced substantially.

Chapter 2

Background

This section presents general background and notations that are relevant for the understanding of later chapters. Sec 2.1 elaborates on the Electronic System Level. We introduce formal techniques in Sec 2.2 and Sec 2.3, that are used to bridge the semantic gap and enable a new hardware design methodology, as presented in Chap. 3. Finally, Sec. 2.4 briefly reviews SMT solving (SMT = *Satisfiability Modulo Theories*) which is a class of decision procedures that are the basis for code optimizations in our hardware design flow.

2.1 Electronic System Level

The complexity of today's circuits poses great challenges for design and verification productivity, in particular at the *Register-Transfer Level* (RTL). Technology enhancements allow us to implement more complex designs. However, the current design and verification methodologies do not scale with the growing design complexity. This is mainly due to the lack of new abstraction levels above the RTL, preventing an increased productivity. The *Electronic System Level* (ESL) aims to establish a new abstraction level above the RTL. In contrast to RTL and lower abstraction levels (gate level, switch level, transistor level etc.) the ESL, as of today, is not clearly defined. To date, there is no common, agreed definition of the ESL.

Modeling systems at the ESL before actually implementing them pursues the goal of reducing time to market by arriving at a feasible system architecture faster. [3] describe ESL design goals as follows: "*The utilization of appropriate abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner, while meeting necessary constraints.*"

An ESL model captures the desired functionality abstractly, more or less independent of the implementation. The designer chooses the required amount of detail depending on the system-level behavior and functionality that is modeled. We call any model that is more abstract than an (RTL) implementation an ESL model. An ESL model may describe a simple model of a bus system or an abstract model of a *System-on-Chip* (SoC). The model of the bus aims to model a correct arbitration and routing of messages by ensuring a correct implementation of the protocol. The model of the SoC is designed to test firmware and inter-module communication. The main difference between these models is not only the scope of the model, it's more in the modeled amount of detail. The model of the SoC abstracts anything that is not required to develop the firmware. For example, the bus connecting the components of the SoC does not model the bit- and timing-accurate protocols and implements only the routing of the messages. The bus and the SoC model focus on different aspects and describe the functionality on different abstraction levels. It is the task of the designer to choose the appropriate abstraction level for his purpose.

We describe in Sec 2.1.1 how systems are modeled at the ESL and introduce the core concepts of system-level modeling, such as transaction-level modeling. In practice, the C++ library *SystemC* is is the most common way of modeling the ESL. We cover the core ideas of SystemC briefly in Sec. 2.1.2. For design exploration and virtual prototyping, ESL modeling is already widely adopted in the industry. Sec. 2.1.3 provides an overview of ESL design methodologies currently used in industrial practice. The ESL is, however, barely used as the golden design model for the hardware design process. This is due to a semantic gap, as explained in Sec. 2.1.4.

2.1.1 Modeling Systems

In this section, we explain how systems are modeled at the ESL. As opposed to the RTL, ESL models may exist at various levels of abstraction, even for the same design. Depending on the application domain, the understanding of what is specified by the system level, may vary. In order to have a common understanding, in the context of this work, we provide a definition of our understanding of the system level.

Definition 1. *(System-level model). A system-level model is composed of modules that communicate with each other. The model may be* executable*, i.e., it may be simulated in order to analyze its behavior. Communication is modeled on the transaction level as blocking message passing. The behavior is described as a time-abstract, word-level description and the behavior of each module is described as an FSM. The finite state machines of the modules send each other messages based on synchronization events.*
The outputs are defined as a function of the current state and inputs. The behavior of the entire system level is defined by an FSM that results from an asynchronous composition *of the individual FSMs (cf. Def. 23).* □

As mentioned in Def. 1, the behavior of each module is described in terms of a *Finite State Machine* (FSM). The asynchronous composition of the FSM allows for a modeling of all interleavings of messages being passed between the modules, and ensures capturing the full behavior of the system-level model. Due to the untimed behavior of the system-level model, each module is allowed to run at its own speed, in the context of system-level modeling this is also called temporal decoupling. In order to exchange a message between two modules they need to synchronize through a handshake. In the executable model this handshake is implemented with events.

The system-level model, as defined in Def 1, is also referred as *transaction-level* model and constructing executable models at the transaction level is denoted as *Transaction-level Modeling* (TLM). This term was coined in the early 2000s by the *Electronic Design Automation* (EDA) companies. TLM aims to model the communication components of the system, especially the bus systems. The core idea of TLM is to separate communication from computation [4].

For example, consider modeling a synchronous bus system at the RTL. This is done by modeling the different signals (e.g., clock, control, address or data) and the respective time behavior of the signals, as defined by the bus protocol. As a consequence, the RTL model is cycle-accurate and bit-accurate. The transaction-level model abstracts from implementation details such as how control and data are represented by logic signals and how these signals are timed according to the specified bus protocol. It instead models the bus communication in terms of abstract data transfers called *transactions*.

As noted, the core idea of TLM is to separate communication and computation. The bus is abstracted to a so called *Channel*, i.e., a single module specifying the routing and arbitration between communicating components. In contrast to the RTL, the time and pin-accurate protocol is abstracted and sending and receiving is modeled by a distinct event-based handshaking. This results in a vast increase of simulation speed compared to the RTL model.

In TLM, the computational part responsible for data processing can be described without the micro-architecture of an RTL model. Instead, the behavior of the component is described by an algorithm, analogous to a software function. This data processing model is called an *algorithmic model*.

The term TLM is misleading, because, as opposed to the RTL, the transaction level does not describe a distinct level of abstraction. The abstraction of the RTL has a clear semantics. It describes the desired behavior by means of register-transfers. The transaction level lacks this clear semantics. For example, the communication can be modeled time- and pin abstract and the computation pin- and cycle-accurate, and vice versa. This results in different possible model abstractions with different semantics. In practice, every model that is more abstract than the RTL is considered to be a transaction-level model and time behavior is modeled at various accuracy levels. Hence, the notion of a "level" does not fully apply and TLM is considered more a modeling technique. In TLM, the level of abstraction is dependent on the use case.

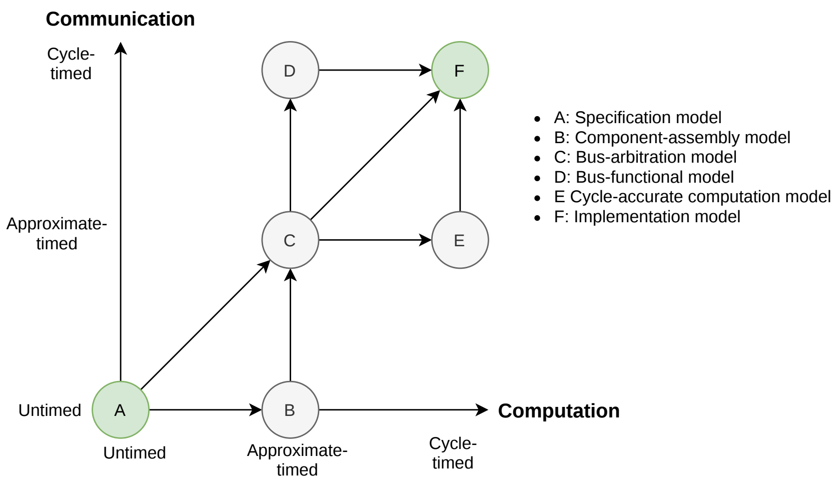

Communication
Cycle-timed
Approximate-timed
Untimed
A
B
C
D
E
F
Computation
Untimed
Approximate-timed
Cycle-timed
• A: Specification model
• B: Component-assembly model
• C: Bus-arbitration model
• D: Bus-functional model
• E Cycle-accurate computation model
• F: Implementation model

Figure 2.1: Abstracting time in TLM [4]

Fig. 2.1 shows different levels of abstraction when modeling time in TLM: Untimed, Approximately-timed, and Cycle-timed. An untimed model describes the behavior without any notion of time, whereas approximate-timed adds an estimated timing behavior without the need to completely match with the real time behavior. The cycle-timed model exhibits the exact timing behavior. We may describe computation and communication with different notions of time. The untimed model is denoted as the *Specification Model* and the cycle-timed is denoted as the implementation. The other models result from combining different abstraction levels. For example, "E"in Fig. 2.1 is a model with a cycle-accurate timing behavior for computation, i.e., all the clock events are simulated, combined with approximately timed communication behavior.

According to Def. 1, we only consider the specification model when we talk about system-level models. The definition also states that a system-level model has to be executable. The industry standard for executable system-level designs is SystemC [5]. The next section gives an overview on SystemC.

2.1.2 SystemC

In this section we briefly introduce the core concepts of SystemC. As mentioned in Sec. 2.1.1, we consider system models described at different levels of abstraction. SystemC is a software library implemented in C++ and allows to model any desired level of detail as mentioned in Sec. 2.1. In the scope of this work, we are focusing on untimed system-level models. We discuss only the relevant components for modeling on this abstraction level. For a more detailed introduction on SystemC we would like to refer to [6] or [7].

The development of SystemC started in the late 90s, with preliminary efforts from both, academia and EDA companies (e.g., SpeC [8] and Handel-C[9]). In 1999 the *Open SystemC Initiative* (OSCI) formed and started coordinating the development of SystemC. Due to the participation of major EDA and semiconductor companies in the initiative, SystemC quickly became the dominant language for describing system-level models. Version 1.0 of SystemC was released in 2000 and it primarily rebuilt the functionalities of RTL languages, to allow for fast RTL simulation by *native execution* rather than by a simulation engine that interprets VHDL or Verilog like, e.g., *ModelSim*™ [10]. A SystemC model is compiled together with a process scheduler and a simulation engine into an executable simulation program for the host computer. Executing native code delivers higher simulation performance compared to an interpreting simulator, and it also allows employing standard software debugging technology. Over time, SystemC has been continuously extended with transaction-level features [4, 11] and version 2.0 was standardized by the IEEE [12] in 2005. The new version provides features that allow to abstract from certain implementation details (e.g., timing) resulting in faster simulation models.

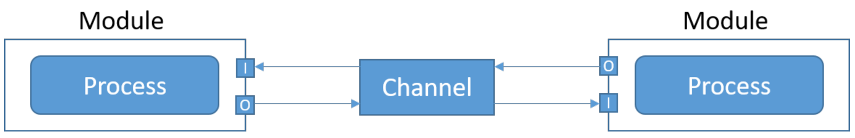


Figure 2.2: Structure of a SystemC TLM model

In SystemC, a C++ class hierarchy models the structural composition of modules and their behavior into a system of concurrent processes. The SystemC library provides a scheduler for simulating the concurrent execution of the processes. Fig. 2.2 provides an overview of the structure of a SystemC model. The structure of the model is defined by instantiating objects of the module class. The designer may add input / output ports by instantiating a port object within the class. The modules can exchange messages between each other by connecting the ports through channels. Channels are an object structure specifically designed to connect ports and to specify a communication protocol.

For example, let us consider a channel implementing communication through a FIFO buffer (FIFO = *First In – First Out*). A port connected to this channel can write (read) a value to the FIFO, if and only if the FIFO is not full (empty). Otherwise, the execution of the module is blocked (resp. has to wait) until the FIFO is free.

The behavior (i.e., the FSM) of each module is specified by providing methods to the module class. By using SystemC-specific macros within the constructor the method is linked to the scheduler. When the binary is executed, the scheduler calls the methods upon events and thereby models the behavior of the modules and the overall system. Methods may alter state variables of the class or assign values to output ports. A method that is registered to the scheduler is called a process. There are different types of processes in SystemC, which differ semantically in the concepts they intend to model. In this scope we are only going to use a *thread* process, that specifies the desired behavior within an infinite `while(true)` loop.

The provided scheduler implements an event-driven simulation algorithm, concurrently executing the processes, in order to model asynchronous behavior. The correct asynchronous execution of the processes is ensured by the simulation algorithm. The core idea of the scheduler is the following: At the start of the program all threads are marked as "pending" for execution. The scheduler randomly chooses a thread that is marked as pending and executes it until it is blocked by a communication request. In this case, the thread is marked as "suspended" until the desired event occurs. Returning to the example of the FIFO, a thread that writes to the FIFO is blocked, if the FIFO is full. The thread is suspended until a notification from the FIFO is received. Upon notification, the suspended thread is marked as "ready" for execution. After the scheduler executed every thread marked as pending, all threads marked as ready are then again marked as pending. The simulation continues until the simulation is stopped by the user or there is no more thread marked as pending.

In practice, suspending threads is used to enforce a specific execution order of threads. However, we assume an untimed asynchronous behavior and each thread may run at it's own speed. The threads only need to synchronize, if they want to exchange a message. From a simulation point of view, the thread keeps the priority, i.e., it keeps executing, until it communicates. Every communication results in an event based handshaking with the communication partner and gives other threads the possibility to execute. For untimed processes the execution order results from the handshaking of the processes. Hence, every process may run at its own speed. In case of an approximate or cycle-accurate timing model the execution order depends additionally on some notion of time. Processes may execute after a specific number of clock ticks (as in hardware). Here, the progress of time is modeled by events and a process is suspended until the desired number of cycles has passed. However, because we are only interested in untimed models, we always consider blocking behavior as related to communication.

In Chap. 3 we provide a basic example of a SystemC module. Later on, Sec. 4.4 introduces SystemC-PPA, a subset of SystemC, used to efficiently model a system according to Def. 1. In the following section we are going to explain how SystemC is used in industrial practice.

2.1.3 Design Flows

In this section, we explain how the ESL and the language SystemC is currently used in industrial design flows. There are two main use cases for ESL models: *High Level Synthesis* (HLS) and *Virtual Prototyping* (VP). Furthermore, we will talk about the current limitations of these flows, which are mostly related to the *semantic gap*.

Virtual Prototyping

In virtual prototyping the designer models the desired behavior of the system, e.g., an SoC. There are two dominating use cases for virtual prototyping. By providing an early prototype of the hardware, the respective software development can start before the actual hardware is implemented. However, because the prototype is not used as a golden reference in the following hardware design process, the hardware model is rarely functionally sound w.r.t. the resulting RTL implementation.

A second use case of a virtual prototype is *design exploration*. Here, different architecture types, bus protocols and implementations of algorithms are explored and evaluated. The goal is to estimate the resulting performance, throughput and power consumption of the hardware design. Furthermore, within design exploration the task of partitioning is performed. The designer decides which part of the functionality is implemented in hardware and which part is implemented as software. The benefit of virtual prototyping, compared to RTL, is that it allows for quick architectural changes.

High Level Synthesis (HLS)

Synthesis, in the domain of digital hardware, is, in the understanding of most people, the process of translating an RTL model into a functionally equivalent gate-level model or, respectively a gate-level model into a corresponding transistor-level model. With the upcoming of the ESL and SystemC, noticeable effort has been put towards synthesizing RTL from ESL descriptions. As of today, HLS is mostly used in the context of digital signal processing in complex ASIC and FPGA designs. As described above, a core concept of TLM is the separation of computation and communication. HLS synthesis tools are strong in synthesizing the algorithmic descriptions into a digital design.

The algorithms are described and tested at the ESL. Most synthesis tools accept a description as a function in a C style language. The function has to have a deterministic and stateless behavior. The result must be a function of the provided parameters only, not of the internal state. Such a function cannot describe an FSM. Using HLS in a top-down design flow has two benefits: First, it allows for a very abstract, algorithmic description without the need to worry about micro-architectural decisions (e.g., pipelining) or timing. Second, the immensely increased simulation speed at the ESL allows for a more thorough verification of the implemented algorithms compared to RTL simulation.

Besides functional design goals HLS tools are able to consider also non-functional design goals such as timing, area and performance. In order to meet these goals, various optimization strategies (e.g., loop unrolling, pipelining, rescheduling) can be applied. The resulting RTL description behaves equivalently with respect to the I/O behavior of the algorithm function. It is possible to use equivalence checking techniques to prove equivalence between the algorithm and the resulting RTL implementation.

However, these tools are less suited for synthesizing state machines. When synthesizing algorithms it is always possible to match up the I/O signals of the ESL and the RTL models and thereby establish a sound relationship. This is, however, not possible for FSMs and communication protocols. In this case, the RTL timing as well as the logic signals are abstracted away at the ESL. A refinement step is required, specifying how the abstract objects of the ESL are related to the generated RTL. This problem is called *semantic gap*, which is discussed in more detail in Sec. 2.1.4.

To date, there is no commercially available technique to establish a sound relationship between a TLM SystemC model and an RTL model. Within the scope of this work we will use a technique, as presented in [13, 2] that allows to build such a sound relationship for communication protocols and FSMs. In Chap. 4 we will discuss the theory of Path Predicate Abstraction, as introduced in [13], that provides the means for building such a relationship. In Chap. 3 we will show how this relationship can be established in practice.

2.1.4 Semantic gap

This section discusses the problem of the semantic gap between ESL and RTL in more detail. In Fig. 2.3 the relationships between the different abstraction levels are illustrated. A new abstraction level is established only, if it has a well defined (i.e., *sound*) relationship with lower abstraction levels. If this is the case, the new abstraction level can be used for design and verification. Otherwise, the abstract model lacks clear semantics w.r.t. the physical circuit. The core idea of sound abstraction levels is that verification results of a sub-circuit, obtained at a lower abstraction level, should not need to be verified again at a higher abstraction levels. Let's assume that there exists a verified transistor implementation of an AND gate. At the gate level it is now possible to use the more abstract AND gate for further design verification. There is no need to re-verify the transistor implementation of the AND gate at the gate level. If there exists confidence in the relationship between two abstraction levels then it is not necessary to use the lower level for design sign-off.

As shown in Fig. 2.3, the lowest considered level is the transistor level. The transistor level is simulated (e.g., with SPICE [14]), in order to build confidence that a transistor netlist realizes the intended functionality correctly. Such netlists may describe a Boolean operator (e.g., *and* or *not*) or a binary storage element (e.g., D-FlipFlop). At the gate level an abstract representation of the operators and storage elements is used (e.g., AND-gate or NOT-gate). There is no more need to model the behavior of the individual transistors. Here, the sound relationship results from an exhaustive simulation at the transistor level.

The *gate level* is a netlist describing the actual Boolean representation of the circuit or device. It consists of combinational blocks and storage elements in between. The gate level can be seen as a structural implementation of Boolean formulae. The level above the gate level is called the *Register Transfer Level* (RTL).

The RTL models the functional behavior of digital circuits. The main difference to the gate level is that it allows to describe behavioral models. It relates well to the theoretical

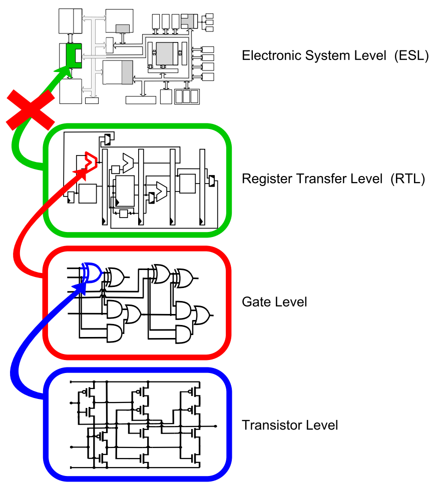


Figure 2.3: Abstraction levels for circuit descriptions

model of a *Finite State Machine* (FSM). An FSM models the behavior only in terms of output and register changes as functions of the input and state. The gate-level details, i.e., how the change is realized by a combinational circuit, is abstracted away, to a large extent.

In order to show that the RT level is sound w.r.t. the gate level a formal *Equivalence Check* (EC) is used. The EC ensures that a gate-level netlist correctly implements the desired functionality. This allows to abstract from gate-level details and use the more abstract RTL description. Due to these methods, it is possible to show that RT-, gate- and transistor-level behave equivalently for identical input sequences. We call this "the chain of trust", because designers trust the higher abstraction level to be suitable for design and verification.

However, EC techniques cannot be employed to establish a sound ESL model, because there is no notion of equivalence that can be easily applied in this context. This problem is referred to as the "semantic gap". As implied by the name, functional descriptions at high levels of abstraction do not have a well defined semantics w.r.t. the RTL and, as a result, to the physical circuits. However, the problem does not result from a lack of standardization of ESL descriptions. It is a theoretical problem of describing a sound relationship between the bit- and cycle-abstract ESL descriptions and the RTL.

A system engineer models large functional blocks as modules with event-driven communication. This allows modeling the overall behavior of the system without the need to worry about synchronization and details of any specific communication protocol. However, implementing such functionalities at the RT level requires an important refinement step. This refinement step forces the engineer to make major RT design decisions such as making area/performance trade-offs, selecting communication protocols, employing pipelining, etc. An automated synthesis, as described in Sec. 2.1.3, is in most cases not possible, or at least unwanted, because this takes away many of the RT design decisions.

A notion of equivalence, as defined for the RTL and gate level, does simply not apply in case of the ESL and the RTL. Nevertheless, a formal relationship between these levels is necessary to establish trust in the ESL and use it as the new abstraction level for design sign-off. In [13],[2] and [15, 16] a new formalism has been developed, specifically tailored for the purpose of describing such a relationship and, thereby, of closing the semantic gap. This formalism is described in Chap. 4 and adapted in the scope of this work in order to fit a new top-down design flow called *Property-Driven Development* (PDD).

2.2 Model Checking

Model checking, also referred as property checking, aims to formally prove or disprove, with fully automatic methods, whether a digital design fulfills a specific property. The behavior of the design is formalized by a sequential model and the desired property is formalized with a temporal logic expression. The task of a model checker is to prove that the specified property holds on the design. As opposed to simulation, the proof mechanism is based on mathematical methods and the model checker does not apply stimuli and assert responses. Instead, the design model and the property together form a computational verification model and the result is computed by an algorithm.

Model checking allows to verify hardware (or software) with tools providing front-ends for common hardware description languages (e.g., VHDL and Verilog) or common software programming languages. The properties are formulated in specific property languages such as *Property Specification Language* (PSL) [17] and *System Verilog Assertions* (SVA) [18].

First, we explain how digital circuits are formalized as sequential models by introducing two different automata formalisms: Finite State Machines and Kripke Models in Sec. 2.2.1. The desired properties are formulated by a specific form of temporal logic expression, as explained in Sec. 2.2.2. There are different model checking techniques, of which, within the scope of this work, interval property checking plays an important role. We explain the idea of this technique in Sec. 2.2.3.

2.2.1 Sequential model

In the following, we explain how the behavior of digital circuits can be formalized by formal, sequential models. Digital circuits are composed of storage elements and combinational blocks. The combinational behavior of the circuit can be described by Boolean formulas. However, storage elements cannot be represented by Boolean logic. Instead, various formalisms for automata are used, also known as state machines or transition systems.

These automata, represent the value of the storage elements by a state and are often visualized as a *state transition graph (STG)*, a directed graph with a node for each state, with an edge for each transition, and with a labeling dependent on the specifics of the automaton formalism under consideration. The terminology used in graph theory also partly applies in the context of automata. The *reachable states* are states that are reachable by a path starting at the initial state. The *sequential depth* is the length of the longest cycle-free path from the initial state to any reachable state.

Finite State Machine

The digital behavior of an electronic circuit can be described by a discrete and deterministic *Finite State Machine* (FSM). In the scope of this work we are using *Mealy-type*

FSMs. *Moore-type* FSMs only differ in the definition of 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 2 (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 3 (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.

Kripke Model

In this section we introduce Kripke models, an automaton formalism widely used in computer science. It is used, e.g., for defining temporal logic languages, as introduced in Sec. 2.2.2. In the scope of this work, it is also used in Chap. 4 to establish a sound relationship between ESL and RTL.

Definition 4 (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 transition 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.

2.2.2 Linear Temporal Logic

Temporal logic languages allow us to reason over logical properties for the sequential behavior of automata. In practice, standardized languages such *Property Specification Language* (PSL) [17] and *SystemVerilog Assertions* (SVA) [18] are used to formalize temporal logic. They provide an extended syntax for describing temporal logic expressions and they can be mapped to formal temporal logic languages, as presented here.

In this section, we introduce the language *Linear Temporal Logic* (LTL), as proposed in [19]. There are other forms of temporal logic such as Computation Tree Logic (CTL) [20] and CTL* [13]. These languages and the according model checking techniques, however, are not required in the scope of this work.

LTL extends the set of Boolean logic operators by a set of temporal operators: X ("next"), G ("globally"), F ("finally"), U ("until"), W ("weak until"), R ("release"), to express logical properties quantified over time. LTL formulas describe a set of paths in the considered Kripke model. If an LTL formula is satisfied in all paths starting from a specific state, we may say that an LTL formula holds for this state.

Definition 5 (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 using $\neg$ and $\vee$.* $\square$

Definition 6 (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, $\pi \models \phi$ 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 $i \leq k < j$, $\pi_k \models \phi_1$*
- $\mathsf{F}\phi_1 \equiv \mathsf{true}\,\mathsf{U}\, \phi_1$
- $\mathsf{G}\phi_1 \equiv \neg(\mathsf{F}\neg\phi_1)$
- $(\phi_1 \mathsf{W}\, \phi_2) \equiv (\phi_1 \mathsf{U}\, \phi_2) \vee \mathsf{G}\phi_1$
- $(\phi_1 \mathsf{R}\, \phi_2) \equiv \phi_2\, \mathsf{W}\, (\phi_2 \wedge \phi_1)$ $\square$

2.2.3 IPC

In this section we introduce Interval Property Checking (IPC), as proposed in [21]. It is a SAT-based model checking technique, rooted in the industrial developments of the 1990s. With this model checking technique, it is possible to produce globally valid, unbounded proofs. It uses Kripke models as the sequential model and the desired property is formulated by a restricted form of LTL formulas (cf. Sec. 2.2.2) known as *interval properties*. An interval property describes behavior over a finite time interval and the temporal logic expression is formulated in form of an implication.

Definition 7 (Interval Property)**.** *An interval property ϕ is an LTL formula of the form* $\mathsf{G}\ (A \to C)$ *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. □

We further refer to interval properties as *operation properties*, because in practice they are often used to specify "operations" of the model under consideration. We provide a formal introduction of operations in Sec. 4.2.

The computational model for IPC is related to *Bounded Model Checking* (BMC) [22]. In BMC properties are formulated from a specific starting state and are restricted to describe behavior over a bounded time interval. Here, the sequential model, can be mapped to Boolean logic by "unrolling" it over a finite time window. The inputs of the circuit are left as free input to the SAT problem.

However, in order to have an unbounded proof that is not restricted to a certain period of time, the unrolled model has to cover the entire reachable state space. This can be done by unrolling the model from the initial state with a period greater than or equal to the sequential depth of the circuit. This is, in most cases, not feasible due to the computational complexity of the SAT problem as well as the complexity of calculating the sequential depth in the first place. In a nutshell, BMC gives bounded proves for properties describing a finite time interval.

The computational model for IPC is, with one important exception, the same as for BMC. The difference is that in IPC no assumption is made about the starting state, i.e., the starting state is left as a free input in the SAT problem. In other words, IPC assumes a *symbolic initial state*. This is why the properties proven on this model are valid for an arbitrary concrete starting state. This is why IPC provides *unbounded proofs* for properties describing a finite time interval.

The proof computation for a property, ϕ, is illustrated in Fig. 2.4. Property ϕ expresses an interval property over the finite interval of three clock periods. It uses a nested *next* operator X at most three times.

The combinational logic is unrolled three times and the starting state, s_t and the input in each clock period, x_t, x_{t+1}, and x_{t+2} are left as free input with respect to the assumption A. Every value that is not a contradiction to the assumption is considered to be a valid value for the input and the initial state. The property is disproved if any set of values fulfills the negation of the commitment C. This set is, then, a counterexample of the property. If no such counterexample exists, the property is valid for the model under consideration.

Due to the fact that the starting states are modeled as free inputs, all states are considered as possible starting states. This results in an over-approximation of the state

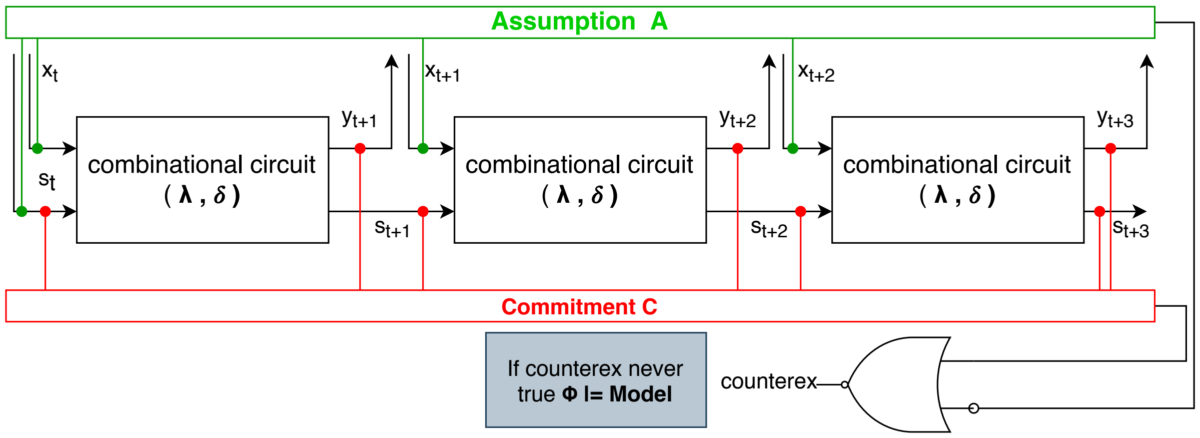

Assumption A
x_t
s_t
combinational circuit (λ , δ)
y_{t+1}
x_{t+1}
s_{t+1}
combinational circuit (λ , δ)
y_{t+2}
x_{t+2}
s_{t+2}
combinational circuit (λ , δ)
y_{t+3}
s_{t+3}
Commitment C
If counterex never true Φ |= Model
counterex

Figure 2.4: Proof computation for interval properties

space, because not all considered states are necessarily reachable in the design. This might result in "false negatives" or "spurious counterexamples". These are counterexamples starting from an unreachable state of the design. However, "false positives" are not possible, because the property is proven in all reachable and unreachable states. The term "positive" refers to a successful proof of a property and "negative" to an unsuccessful one that returned a counterexample. Therefore, the computational model of IPC is *conservative*.

The problem of false negatives is solved in IPC by proving strengthened invariants for the design. Either the counterexample is manually inspected and assertions are formulated for the design or automatic methods, e.g., as proposed in [23], are used.

If the properties are formulated such that they describe important "operations" of the design, false negatives can be avoided, to a large extend. An operation, in the context of digital hardware, describes sequences of register transfers with a common purpose. An operation could be, e.g., an instruction of a simple processor, or, the functionality triggering the protocol for the transmission of a message over a bus.

These operations tend to have sparse, manageable inter-dependencies. Note that the correct execution of a processor instruction is, in most cases, independent of previous behavior. Interdependencies between instructions can, for example, occur in pipelined designs if forwarding of datapath variables is required to resolve hazards. Here, the reachable values, at the time of the operation, are determined completely by previous operations.

A designer, usually, thinks about the behavior of design in terms of operations (e.g., an ADD instruction as one operation). The process of creating and proving the operation properties is, from an abstract point of view, closely related to the design process. Also, the verification engineer needs to identify the operations to create the properties. In that sense, the operation properties formalize and document the operations of the design. We consider IPC to be a *white-box* approach, because the process of creating the properties requires to reason on internal signals. This is in contrast to *black-box* approaches like simulation-based verification. Here, the verification engineer is not required to understand the internals of the design.

During the process of creating the properties an abstract engineering view of the design is formalized. Even for complex circuits, from our experience, the overall functionality can be well understood in terms of operations. Formulating properties for important operations of the design is common practice in industry and well supported by prover technology tailored for this purpose such as in [21, 24, 25]. The high-quality of the verification results from the established validity of the individual operations.

2.3 Complete Interval Property Checking

In the context of simulation-based verification methods certain coverage metrics, such as code coverage and functional coverage, are used to measure which portion of the design is verified. These metrics, however, do not apply in the context of formal methods. Proving a formal property on a design shows that the property is valid for all possible input stimuli. A measurement, reflecting which portion of the design behavior is covered by the proven property, is still needed.

This chapter presents an absolute cover measure, that is a formal criterion for the *completeness* of a set of interval properties. It was first developed in [26, 27]; [28] later independently obtained a similar result. It is ensured that a set of properties fulfilling this criterion completely describes the output behavior of the design in terms of the design's input.

This chapter briefly summarizes *Complete Interval Property Checking* (C-IPC) in the terminology and notations of this thesis. For a more comprehensive and illustrated elaboration of this technique the reader may refer to [27]. The chapter is structured as follows: First we will introduce some important notions related to completeness. This formalization is required to introduce the path predicate abstraction in Chap. 4 — the main theoretical foundation for Property-Driven Development. In Sec. 2.3.2 the completeness criterion will be formally defined. Sec. 2.3.3 presents an algorithm to check the fulfillment of this criterion. In practice, this completeness check is computationally tractable even for large designs and it is commercially available [25].

2.3.1 Terminology

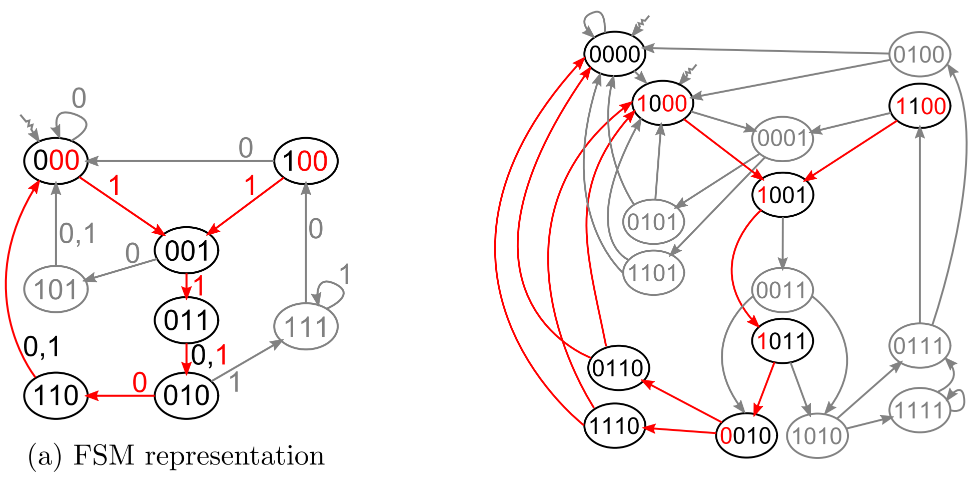
The behavior over a finite time interval is characterized by the notion of a *sequence predicate.*

Definition 8 (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 9 (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* next_k *is defined by:* $\text{next}_0(\phi) := \phi$*, and* $\text{next}_i(\phi) := \mathsf{X}(\text{next}_{i-1}(\phi))$ *when* $i > 0$*.* □

The completeness check is tightly coupled with the concept of describing the design behavior with *operations*, as described in Sec. 2.2.3. The basic idea is the following: A set of properties is considered complete if and only if every input sequence and every output



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

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

Figure 2.5: Example model to illustrate formalisms

sequence of the FSM under verification is fully described by a sequence of operations. For the following discussion, operations are formalized by means of interval properties augmented with a length.

Definition 10 (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, $P := \mathsf{G}(A \to C)$, 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 $\pi \models A$.* $\square$

In other words, the pair (P, l) characterizes path segments with l transitions (between $l + 1$ states). These segments 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. From now on, when it is clear from the context whether the specification or the actual design behavior is meant, we may refer to both, (P, l) and O, as "operation".

Practically, l specifies the length of the finite behavior of an operation. This means that it describes the number of transitions needed to check an assumption A and to produce an output sequence fulfilling the commitment C of the property. There are special cases where a shorter value for l is chosen. In pipelined designs, for example, the computed results are only visible at the outputs a number of cycles after the operation was issued. A new operation may already be in progress while the current operation is still computing a result. In this case, the length of l is chosen such that the start of the current operation, O, and the start of the new operations align. This is important when considering chains of operations, as explained below.

Example (1): Fig. 2.5 shows an FSM and the corresponding Kripke model. We created an example operation $(\mathtt{op}, 5)$, such that inputs, outputs and FSM can be distinguished by the labeling of the Kripke model.

The operation is characterized by the property $\mathtt{op}$ and the length $l = 5$ for the model of Fig. 2.5. Let $\mathtt{op} := G(A \to C)$ where $A := \neg b \wedge \neg c \wedge i \wedge \mathrm{next}_1(i) \wedge \mathrm{next}_2(i) \wedge \mathrm{next}_3(\neg i)$ and $C := \mathrm{next}_1(\neg a \wedge \neg b \wedge c) \wedge \mathrm{next}_2(\neg a \wedge b \wedge c) \wedge \mathrm{next}_3(\neg a \wedge b \wedge \neg c) \wedge \mathrm{next}_4(a \wedge b \wedge \neg c)$.

The path satisfying A are all paths with a prefix matching the states codes (`1x00`, `1xxx`, `1xxx`, `0xxx`). The reader may verify by inspection of the Kripke model that the paths with prefix (`1000`, `1001`, `1011`, `0010`) and the paths with prefix (`1100`, `1001`, `1011`, `0010`) fulfill this condition. The reader may further verify by inspection of Fig. 2.5b that for these paths the commitment C, given by (`xxxx`, `x001`, `x011`, `x010`, `x110`), is fulfilled and, hence, the interval property `op` holds. The evaluation of the operation results in the following path segments. It shows the paths fulfilling A and having a prefix of length $l = 5$:

```
{(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)}
```

$\square$

The completeness criterion relies on sequencing of a set of operations. Each operation has a set of states reachable at the beginning of the operation and a set of states reachable at the end of the operation. In order to fulfill the criterion, it is required that the end of an operation is the start of another. The set of reachable states at the end of an operation are therefore of special interest and are referred to as *important ending states.*

Definition 11 (Important Ending State)**.** *The set of important ending states of an operation O of length l is the set of states*
$\{s \mid \exists\,(s_0, s_1, \ldots, s_l) \in O : s = s_l\}$. $\square$

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

Definition 12 (Important States)**.** *The* important states *of a set of operations are the union of all* important ending states *of all operations in the set.* $\square$

In order to create a complete set of properties the verification engineer has to identify important modes of the system. The design behavior is partitioned into operations transitioning between these modes. Hence, the modes describe the important states of the design. In our methodology, the important states are specified in terms of an expression over the RTL state variables. Practically, macros / functions are created for each important state of the design. They are then refined by formulating logic expressions over the RTL state variables. The operation properties are then 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.

2.3.2 Completeness Criterion

A property set is considered to be complete if the properties describe the output sequences according to certain *determination requirements.* The determination requirements specify which output signals in the design need to be *determined* under which circumstances. An output 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/or earlier time points.

In addition to the outputs, state variables can also be declared as determined. If state variables are declared to be determined at the time of reference, these state variables may be used to express output signals. The expression for determining an output signal's value must be in terms of inputs and/or other determined values.

As an example, let us consider the determination requirement for a data bus with a valid flag "datavalid". If the flag is not set, the data of the bus is irrelevant. The determination requirement for the data signal, "data", is specified as "if (datavalid = true) then determined(data)", assuming that the valid flag has no determination requirements. The determination requirements thereby specify a set of time points for each property considered in the set for which the value of the data bus should be determined. It is defined as a pair (s, σ_s) for a signal s (i.e.,: data bus) and the condition σ_s defines 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 13 (Complete Property Set). *A property set $V = \{P_1, P_2, \ldots, P_n\}$ is complete if any two 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 the scope of this work the properties we consider reason about the labels of a Kripke model. The labeling is chosen such that they discriminate the signals of the RTL design. One way for choosing the labeling is that each RTL signal is represented by its own Boolean atomic formula, e.g., the aforementioned bus is represented by a set of atomic formulas, one for each of the state and data bits.

2.3.3 Completeness Check

The completeness of a property set is checked formally, by a proof by induction starting from the initial states. In practice, the initial states are defined by the possible states of the design when the reset signal is asserted. This algorithm has shown to be computationally feasible. The algorithm proves that starting from an initial state that any input sequence drives the design through a sequence of operations, each described by an operation property. The starting point of each operation "mode" is determined by the history of input sequences (i.e., the sequence from reset). The mode is described in terms of expressions over state variables of the system.

The base case of the inductive proof is the reset property which determines the state of the system a finite number of clock cycles after reset. The inductive step is constituted by the other properties of the property suite. 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. In order to ensure that an operation ends in a determined state it is checked whether the operation is a function of only inputs and its starting states.

The basic of idea of *Complete Interval Property Checking (C-IPC)* is implemented by a collection of four tests. These tests can be performed automatically on the set of operation properties. Note that the design is not taken into consideration and the tests are solely performed on the property suite. The user needs to specify:

- the input signals,

- what signals should be determined according to determination requirements,
- a sequencing of operations.

The sequencing is specified by a *property graph* $G = (V, E)$ where the nodes $V = \{P_i\}$ are the operation properties and the edges describe their potential 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.

In order to prove the completeness criterion of a set of properties, four checks are performed on the property graph G: a *case split test*, a *successor test*, a *determination test* and a *reset test*, all described below.

Example (2): In order to practically demonstrate the idea of completeness, a complete set of properties is created for the model shown in Fig. 2.5. First, we need to specify the determination requirements. Here, we only list one requirement for the atomic formula c as $(c, \textit{true})$, i.e., it is an unconditional requirement. Atomic formulas a and b are considered internal states and do not need to be determined.
We consider the set of operations:

$\{(\texttt{long}, 5), (\texttt{short}, 3), (\texttt{idle}, 1), (\texttt{wrong}, 4), (\texttt{readErr}, 1), (\texttt{keepErr}, 1), (\texttt{reset}, 0)\}$
The interval properties are defined by:

$$
\begin{aligned}
&A_{\texttt{long}} := \neg b \wedge \neg c \wedge i \wedge \text{next}_1(i) \wedge \text{next}_3(\neg i)\\
&C_{\texttt{long}} := \text{next}_1(c) \wedge \text{next}_2(c) \wedge \text{next}_3(\neg c) \wedge \text{next}_4(\neg c) \wedge \text{next}_5(\neg b \wedge \neg c)\\
&A_{\texttt{short}} := \neg b \wedge \neg c \wedge i \wedge \text{next}_1(\neg i)\\
&C_{\texttt{short}} := \text{next}_1(c) \wedge \text{next}_2(c) \wedge \text{next}_3(\neg b \wedge \neg c)\\
&A_{\texttt{idle}} := \neg b \wedge \neg c \wedge \neg i\\
&C_{\texttt{idle}} := \text{next}_1(\neg b \wedge \neg c)\\
&A_{\texttt{wrong}} := \neg b \wedge \neg c \wedge i \wedge \text{next}_1(i) \wedge \text{next}_3(i)\\
&C_{\texttt{wrong}} := \text{next}_1(c) \wedge \text{next}_2(c) \wedge \text{next}_3(\neg c) \wedge \text{next}_4(a \wedge b \wedge c)\\
&A_{\texttt{readErr}} := a \wedge b \wedge c \wedge \neg i\\
&C_{\texttt{readErr}} := \text{next}_1(\neg b \wedge \neg c)\\
&A_{\texttt{keepErr}} := a \wedge b \wedge c \wedge i\\
&C_{\texttt{keepErr}} := \text{next}_1(a \wedge b \wedge c)\\
&A_{\texttt{reset}} := \texttt{reset}\\
&C_{\texttt{reset}} := \neg b \wedge \neg c
\end{aligned}
$$

□

The example will be used to explain how the four individual tests for establishing completeness are applied and how they ensure the overall completeness of the property set. It will be continued for each test in each test's respective subsection.

2.3.4 Case Split Test

The case split test checks, for an arbitrary input sequence, that 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 deterministic sequence of operations.

This is ensured by checking that for every operation (P, l_P) the commitment C_P of P is covered by the disjunction of the assumptions $\{A_{Q_j}\}$ of all successor properties Q_j after l_P transitions. This implies that, 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 that path.

Let $\{A_{Q_1}, A_{Q_2}, \ldots\}$ be the set of assumptions of the successor operations, then the case split test checks whether the implication $C_P \rightarrow \text{next}_{l_P}(A_{Q_1} \vee A_{Q_2} \vee \ldots)$ holds or fails. The assumptions A_{Q_i} of the property 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): Fig. 2.6 shows the property graph as defined previously by the interval properties of our running example. Lets consider the case split test applied to operation $(\texttt{wrong}, 4)$. By inspecting the property graph, the successors operations are identified as `readErr` and `keepErr`. The case split tests checks for `wrong` 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, i.e., $\text{next}_1(c) \wedge \text{next}_2(c) \wedge \text{next}_3(\neg c) \wedge \text{next}_4(a \wedge b \wedge c) \Rightarrow \text{next}_4((a \wedge b \wedge c \wedge \neg i) \vee (a \wedge b \wedge c \wedge i))$. Clearly, the case split test holds for $(\texttt{wrong}, 4)$.

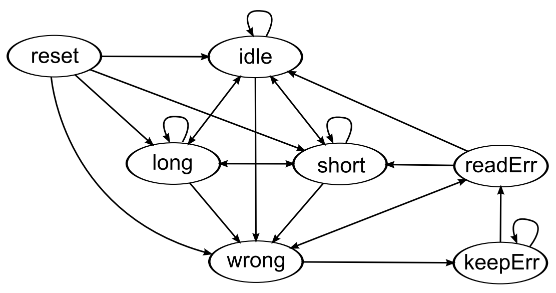


Figure 2.6: Property Graph

□

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.

2.3.5 Successor Test

The successor test ensures that the execution of an operation (Q, l_Q) is completely determined by its predecessor operations (P, l_P).

Considering a pair $(P, Q) \in G$ with a predecessor operation P and a successor operation Q. The test checks whether the assumption A_Q of Q depends only on inputs and signals determined by P. The problem is translated to a SAT instance by the following steps:

1. The set of signals mentioned in the properties P and Q is duplicated. If a symbol z' is marked with a tick then it belongs to the copy, otherwise to the original set. 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 a different operation.

2. The same input sequence and the same values for determined state variables are applied to both executions.

3. The set of determination requirements $D = \bigwedge d_i$ is expressed as a conjunction of the determination requirements 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.

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

   $A_P \wedge C_P \wedge A'_P \wedge C'_P \wedge D \wedge \mathrm{next}_{l_P}(A_Q) \Rightarrow \mathrm{next}_{l_P}(A'_Q)$

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"). In the case that this implication does not hold, there exists an input sequence such that operation property P is executed and the assumption of operation property Q may 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.

Example (4): For the pair (`wrong`, `keepErr`) of the example, the successor test is:

$$A_{\texttt{wrong}} \wedge C_{\texttt{wrong}} \wedge A'_{\texttt{wrong}} \wedge C'_{\texttt{wrong}} \wedge (c \Leftrightarrow c') \wedge \mathrm{next}_4(A_{\texttt{keepErr}}) \rightarrow \mathrm{next}_4(A'_{\texttt{keepErr}})$$

The commitment of `wrong` fully specifies the FSM at time point next_4. Clearly, the formula holds for this example.

In order to demonstrate the successor test we modify `wrong`, such that next_4 specifies only states b and c. Now, we add a new successor property to the property graph with the an assumption including $a = 0$. In this case, the case split test for `wrong` still holds. Due to the fact that a is not determined at the end of `wrong` it is not possible to know whether the new property or one of the two original successor properties would trigger at next_4. Hence, the successor test fails and correctly flags such a set of properties as incomplete. □

With the successor test, we ensure that a unique chain of operations for every input trace exists. In the following, we introduce the *Determination Test* that shows that these operations determine the output signals as stated in the determination requirements.

2.3.6 Determination Test

The determination test performs a check whether an operation (Q, l_Q) and its predecessor operation (P, l_P), in turn, fulfills its respective determination requirements. The test creates a SAT instance that evaluates to *true* if a determination requirement is violated.

This means that a variable required to be determined by Q is actually not a function of the variables P and/or of inputs during the operation (Q, l_Q).

As in the successor test, the set of signals is duplicated for both operations Q and P. This technique allows to describe two executions where Q is the successor of P and identical input sequences are applied. The state variables are assumed to be equal and are given the same values, in the time points specified by the guards. 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):

$$A_P \wedge C_P \wedge A'_P \wedge C'_P \wedge D_P \rightarrow \text{next}_{l_P}(D_Q)$$

Disproving this implication means that there exists a sequence of input signals and/or signals mentioned in the operation properties such that signals that are supposed to be determined in Q may have different values in different executions of P.

Example (5): In the provided example we only have a single, unconditional determination requirement — c should be determined at all times, 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 set example.

In reality, 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. □

2.3.7 Reset Test

The inductive proof is rooted at the reset state. The case split, successor and determination test form the inductive step of the proof with a hypothesis that states: Assuming that an operation $(P, l_P) \in V$ uniquely determines 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).

For the reset test, a special *reset property* is required to describe the state after reset. Here, the assumption contains only the reset condition and the test whether reset behaves deterministically and that it fulfills all determination requirements. The construction is similar to the other test. However, the reset property does not have any predecessor properties.

Example (6): The reset operation $(\texttt{reset}, 0)$ specifies the state after reset. It does not refer to the state before the reset condition. Hence, it guarantees that reset can be applied deterministically. Furthermore, 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).

2.4 Satisfiability Modulo Theories

In the context of digital design Boolean formulas and their satisfiability are of special importance. For example, they are used to specify the logic of the circuit or to prove important design properties with IPC. The problem of determining whether there exists an interpretation that satisfies a given Boolean formula is called *SATISFIABILITY* or *SAT*. The solvers for these problems are efficient in solving Boolean decision problems, which is why they are frequently used as engines behind verification applications. In practice, decision problems are sometimes described with "richer" languages features (e.g., arithmetic operators) and the resulting formula (for example, $a + b \leq 20$) models a higher abstraction level. In order to use the existing efficient SAT solvers, the abstract formula has to be translated to a Boolean formula. However, translating such formulas to Boolean decision problems quite often results in computationally very expensive problems. Let us consider the example $a + b \leq 20$. The addition of two 32-bit integer values has to be transformed to the Boolean logic of a 32-bit adder to decide if the problem is satisfiable. The arithmetic operation is mapped to the theory of Boolean logic. In order to capture the meaning of these formulas without the need of translation to Boolean logic, a different technique named *Satisfiability Modulo Theories* (SMT) [29] is used. The key idea of SMT is to combine a SAT-based approach with specific problem specific theories (e.g., theory of integer arithmetic) allowing capturing the high-level semantics of the formula, without the need of translating it to Boolean logic.

The primary goal of research in SMT is to create verification engines that can reason natively at a higher level of abstraction, while still retaining the speed and degree of automation of today's Boolean engines. SMT formulas provide a much more expressive modeling language than what is possible with Boolean SAT formulas. For example, an SMT formula allows us to model datapath operations of a microprocessor at the word rather than the bit level. A problem formulated as $(x - y \leq c)$, where x and y are variables and c is a constant, can be solved by a theory solver for *difference arithmetic*. Considering a set of these formulas, instead of translating the problem to propositional logic, a computationally far less expensive method is used that represents that equation with directed weighted graphs. The theory solver searches for negative cycles within the graph and thereby decides on the satisfiability of the problem.

An SMT solver works as follows: First, the solver creates an abstraction that maps the atoms (i.e., non-Boolean expressions used within the formula) of an SMT formula into fresh Boolean variables $p1, ..., pn$. For example, the formula $\neg(a \geq 3) \wedge (a \geq 3 \vee a \geq 5)$ is translated into $\neg p1 \wedge (p1 \vee p2)$, where the atoms $a \geq 3$ and $a \geq 5$ are, respectively, replaced by the Boolean variables $p1$ and $p2$.

If a SAT solver finds the abstract formula to be unsatisfiable, then the SMT formula is, too. On the other hand, if the SAT solver finds a model (i.e., the formula is SAT), then a specific theory solver is used to check the model produced by the SAT engines. A model for the abstract formula is $\{p1 \rightarrow false, p2 \rightarrow true\}$ that translates to $\{\neg(a \geq 3), (a \geq 5)\}$ for the concrete formula. Here, a theory solver for the theory of *arithmetic* is used and finds this formula to be unsatisfiable.

The abstract formula is extended by the theory lemma $\neg(\neg(a \geq 3) \wedge (a \geq 5)) \Rightarrow \neg(\neg p1 \wedge p2)$ and is checked for satisfiability again. The extended formula $\neg p1 \wedge (p1 \vee p2) \wedge \neg(\neg p1 \wedge p2)$ is not satisfiable and thus the entire formula is not satisfiable. The engine continues adding the theory lemmas until the formula is disproven or the concrete model holds.

In Chap. 5.5 we use SMT solving for optimizing the generated set of operation properties. The operation properties are described at the word level and may contain complex datapath operations. This makes a Boolean SAT-based pruning approach computationally expensive. We identify and detect unreachable operations and use SMT-based techniques for further simplification of expressions (e.g., by computing that $(x+1+1+1+1+1)$ is equal to $(x+5)$).

2.5 S^{2}QED

In this section, we review S^{2}QED, a formal processor verification approach targeting complex instruction pipelines presented by [30]. S^{2}QED proves that every instruction executes independently of the previous pending instructions in the pipeline, i.e., independently of its program context.

The computational model of S^{2}QED consists of two identical and independent instances of the processor under verification which are constrained to execute the same instruction, at an arbitrary time point. It is important to note that S^{2}QED does not necessarily result in a *complete* formal verification, as specified in Sec. 2.3.2. For example, the existence of a single-instruction bug is not detected. With the S^{2}QED-property it is only possible to prove that the instruction produces equivalent results independent of the context. There exists an extension to S^{2}QED, presented in [31], that ensures a complete coverage along the S^{2}QED proof. In the remainder of this section we review the fundamentals of S^{2}QED.

Fig. 2.7 shows the computational model in which the two CPU instances of the same processor are unrolled for a time window as large as the upper bound of the execution time of an instruction in the pipeline.

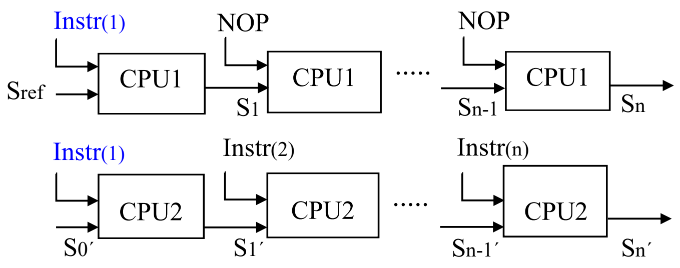


Figure 2.7: S^{2}QED Computational model

Definition 14. *(QED consistency): In the S^{2}QED computational model, the two CPU instances are* QED-consistent *at a time point t, if the corresponding architectural state elements of both instances at time point t hold the same values.* □

The architectural state of a processor is defined by its register file and general purpose registers. For a processor with N registers a QED-consistent register state is characterized by the named logic expression:

$$qed\_consistent\_registers := \bigwedge_{i=0}^{N-1} \left(R^i_{cpu1} = R^i_{cpu2}\right) \tag{2.1}$$

This expression is a Boolean predicate that is implemented as a *macro* in the property language of the verification tool. It represents architectural states in which the register files R of both CPU instances have identical contents.

```
assume:
    at $t_{\mathrm{IF}}$:                    $cpu2\_fetched\_instr() = cpu1\_fetched\_instr()$;
    during [$t_{\mathrm{IF}}$+1, $t_{\mathrm{WB}}$]: $cpu1\_fetched\_instr() = \mathrm{NOP}$;
    at $t_{\mathrm{IF}}$:                    $cpu1\_state() = S_{\mathrm{ref}}$;
    at $t_{\mathrm{WB}}$:                    $qed\_consistent\_registers()$;
prove:
    at $t_{\mathrm{WB}}$+1:                  $qed\_consistent\_registers()$;
```

Figure 2.8: S^{2}QED property

Consider an S^{2}QED computational model, in which R_{cpu1} and R_{cpu2} represent the CPU 1 and CPU 2 register files, respectively. Fig. 2.8 shows the S^{2}QED property that is to be proven on this model. The property specifies that if two independent CPU instances fetch the same instruction and the register files are consistent with each other before the write-back (the macro *qed_consistent_registers*() specifies this consistency), then the two CPU instances must be QED-consistent also after the write-back, independently of the pipeline context.

The CPU 1 instance is constrained to start from a flushed-pipeline state S_{ref} and fetches only NOPs in the time frames for $t > 1$. A flushed-pipeline state S_{ref} is forced on the CPU 1 instance by letting it execute only NOPs for as many time frames before time point t as there are pipeline stages. This results in a significant reduction of proof complexity and excludes any false counterexample to the property that can result from an inconsistent pipeline register.

The CPU 2 instance is left unconstrained to start from a symbolic initial state and is allowed to execute an arbitrary sequence of instructions for the time frames $t > 1$. In this computational model, the SAT solver compares the scenario 1, where the *Instruction Under Verification* (IUV) is executed in a flushed-pipeline context, with all scenarios 2 where the IUV is executed in an arbitrary context including the ones where bugs are activated and propagated.

In Sec. 6 we extend this idea for general hardware models other than only processors. The main challenge is to define the "architectural state" of the two design instances and to generalize the idea of an "instruction" to a general hardware design.

2.6 Publication List

The main contributions of thesis have been published at conferences and in journals as listed chronologically below:

1. T. Ludwig, M. Schwarz, J. Urdahl, L. Deutschmann, S. Hetalani, D. Stoffel and W. Kunz Property-Driven Development of a RISC-V CPU. In *Proc. Design and Verification Conference United States (DVCON-US '19).*

2. T. Ludwig, J. Urdahl, D. Stoffel and W. Kunz Properties First – Correct-By-Construction RTL Design in System-Level Design Flows In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

3. J. Urdahl and S. Udupi and T. Ludwig and D. Stoffel and W. Kunz Properties first? A new design methodology for hardware, and its perspectives in safety analysis In ACM International Conference on Computer-Aided Design (ICCAD), 2016.

4. A publication for the results of Sec. 6 is in preparation.

The theoretical foundation of the PPA theory (see Chap. 4) has been developed by our chair and has been published in:

1. 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.

2. 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.

2.7 Related work

The theory presented in Chap. 4, introduced in [13], is inspired by the advances in theorem proving and the use of *refinement maps* [32, 33] to model complex relationships between abstract and concrete models in hardware design such as [34, 35, 36, 37, 38, 39]. Notable success was obtained in the hardware domain by the notion of "Bisimulation modulo stuttering", as demonstrated in [40] for hardware verification. The difference between bisimulation modulo stuttering and PPA was already discussed in [2].

Generally, there is an important difference between these classical approaches and our work. Bisimulation and its weakened forms are defined for an *arbitrary* labeling. The objective in most of this work is to identify an efficient abstraction based on a labeling as given by a concrete model. Both the abstract and the concrete model are described in terms of the same alphabet.

In our work, we exclusively consider a special labeling called "operational coloring", as defined in Def. 15, which is derived directly from the given abstract model. As will be developed in later sections, the whole point of our methodology is to enforce this operational coloring along the design flow. 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 fact, PPA may be understood as a special case of a "bisimulation modulo silent actions" [33] where the refinement mapping is obvious, but as a result of the operational view, it can be linked to RTL designs. Thanks to the special labeling, our approach can formally link models described in terms of completely different alphabets. This aspect is very important for the practicality of our approach.

Our PPA-based methodology is formulated in such a way that no knowledge of higher order logic or related languages is needed to implement the proposed PDD paradigm. Based on PPA only standard design and verification languages are needed. All proofs can be based on bounded circuit models using SAT-based property checking [41, 21], as it is commercially available.

A completely different approach to closing the semantic gap between a system model and concrete RTL is *High-Level Equivalence Checking (HLEC)* [42, 43]. This approach has the advantage that a high degree of automation is possible. It is promising in applications where the notion of equivalence or a weakened form of it, such as equivalence modulo latency, is adequate to describe the relationship between the abstract and the concrete model. This can be the case in data-centric designs, for example, when an abstract specification of a signal processing algorithm is matched against its RTL implementation.

In [44, 45, 38] equivalence checking is combined with theorem proving to obtain a more general framework for linking concrete RTL implementations with high-level models. Note, however, in contrast to our approach, equivalence checking always assumes that both the concrete design and the abstract model are available. HLEC is usually meant to be used in scenarios where also *High-Level Synthesis (HLS)* is available. While HLS and HLEC are most appropriate for data-centric designs, the proposed PDD methodology is equally suitable for the control and communication structures of a system. It may therefore complement design flows in those cases where HLS and HLEC are not applicable. The approach described in [46] enables a hardware generation from operation properties. This approach requires a complete set of properties to be available prior to generation. This is actually the case in a PDD based flow and a combination of the methods in [46] and PDD may lead to a top-down hardware generation flow.

Specification based on Instruction-Level-Abstraction (ILA), as proposed in [47], pursues a similar goal as our work. It is centered around a generalized notion of processor instructions. In contrast, our approach is based on structuring an arbitrary design in terms of the operations of its control flow and establishes compositionality based on the notions of complete property sets and PPA.

Finally, it should be noted that the proposed methodology is loosely related to practices where high-level simulation patterns or test cases as in [48, 49] are used to derive properties for implementation verification. In contrast to our work, these approaches do not aim at establishing a formal relationship between the system-level model and the RTL implementation, but instead to maintain coherence between the test cases at the different abstraction levels.

Chapter 3

Property Driven Development

In this chapter we will introduce the *Property-Driven Design* (PDD) flow, as depicted in Fig. 3.1. Starting at the Electronic System Level (ESL) and following common practices we can formalize the abstract design by an executable model description. In our current implementation of the flow we use the language SystemC for this purpose.

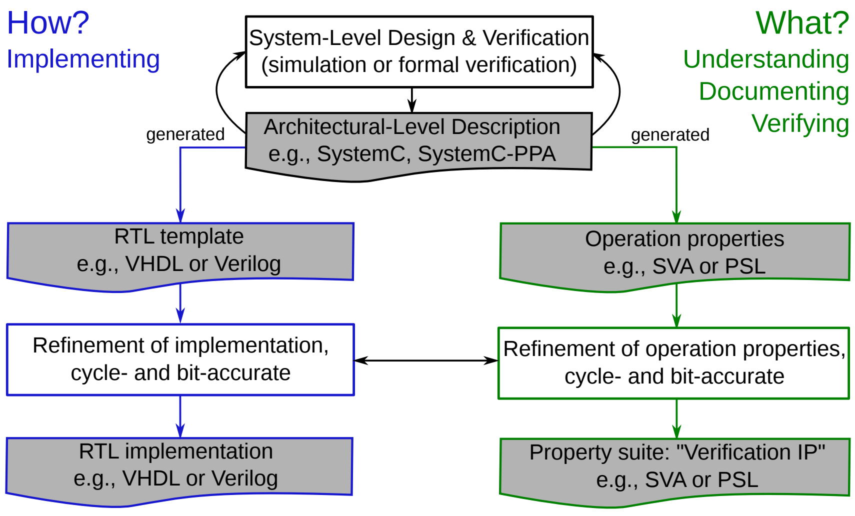


Figure 3.1: Workflow of the proposed design methodology

Already in the system-level model first refinements are made and described at an *architectural level*. The model and the refinements can be based on SystemC. In PDD, the architectural level comprises any system-level description for which clear semantics with respect to PPA can be defined. For this purpose, the language SystemC-PPA has been created (cf. Sec.4.4).

At the architectural level, the system is modeled in a time-abstract way by communicating abstract automata and their interaction with data paths. It is described what data manipulations are performed between communication points of different components. This does not dictate, however, that a particular algorithm chosen at the system level must be implemented at the RTL. For example, a multiplication at the system level may be implemented using Booth's multiplication algorithm, a Wallace Tree or any other suitable method at the RTL. The property checker allows abstracting from such implementation details. SystemC descriptions that have been refined to the architectural level using SystemC-PPA can then be processed by our tools that support the new flow.

Example (7): Fig. 3.2 shows an example of a module modeled in SystemC-PPA. We will continue to use this example in the sequel in order to illustrate the PDD flow, tool-generated properties and manual refinements. It is a slightly simplified version of a SystemC-PPA model that is publicly available online and that can be evaluated using our PDD tool DeSCAM [50]. □

```
enum status_t {in_frame, oof_frame};
struct msg_t {status_t status; int data; };
SC_MODULE(Example) {
    SC_CTOR(Example):
        nextsection(idle) { SC_THREAD(fsm)};
    enum Sections {idle, frame_start, frame_data};
    Sections section, nextsection;
    blocking_in<msg_t> b_in;
    master_out<int>m_out;
    shared_out<bool> s_out;
    int cnt; bool ready; msg_t msg;
    void fsm() {
        while (true) {
            section = nextsection;
            if (section == idle) {
                s_out→set(false);
                b_in→read(msg);
                if (msg.status == in_frame) {
                    s_out→set(true);
                    nextsection = frame_start;
                    cnt = 3;
                }
            } else if (section == frame_start) {
                m_out→write(cnt);
                cnt = cnt - 1;
                if (cnt == 0) {
                    cnt = 15;
                    nextsection = frame_data;
                }
            } else if (section == frame_data) {
                ready = b_in→nb_read(msg);
                if (!ready) {
                    m_out→write(msg.data);
                    if (cnt == 0) {nextsection = idle; }
                    cnt = cnt - 1;
                }
            }
        }}};
```

Figure 3.2: Example of a SystemC-PPA module

The SystemC-PPA language subset is discussed in more detail in Chap. 5. At this point, the reader may examine the source code of the example in order to gain a first intuition into the nature of PPA models. The basic structure of a module is a finite state machine (FSM), defined in a method called *fsm*(). General FSM control is divided into *sections*. Each section can define arbitrary computations and determine the next section to be executed, and it can invoke communication transactions with other modules. At the current state of our implementation, communication needs to be specified based on predefined transaction-level interfaces (SystemC-PPA header files). The interfaces implement *message passing* in various forms (blocking, non-blocking, synchronous/asynchronous).

Our software tool DeSCAM reads a SystemC-PPA description and extracts from it the abstract FSM that represents a PPA of the RTL model to be designed.

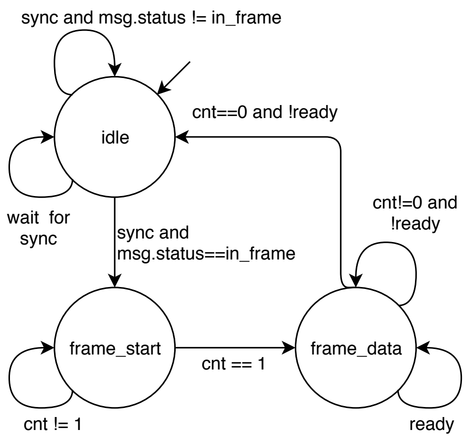


Figure 3.3: Abstract FSM of SystemC module in Fig. 3.2

Example (8): Fig. 3.3 shows the state transition graph of the abstract FSM defined by the SystemC-PPA module in Fig. 3.2. The edges between the nodes are labeled with communication events and control conditions. Each edge represents the computations performed on the respective control paths in the SystemC code, i.e., an edge represents an *operation*, a fundamental concept of PPA (cf. Sec. 4). □

The subsequent design process can be split into two separate threads. The first one is shown on the left side of Fig. 3.1 and corresponds to the conventional design process for RTL descriptions from abstract specifications. Here, it is determined *how* the abstract specification is *implemented* in a cycle-accurate RTL model. All common design practices can be applied and arbitrary solutions can be chosen for the micro-architectures of the system. Optionally, a PDD-based design environment can automatically generate templates for the main control structures of the system as they result from the abstract model. This, however, only serves convenience purposes. The soundness of the final design does not rely on this optional automation step.

The second thread on the right side of Fig. 3.1 is used to formally describe *what* is implemented by the design on the left side. A "formal data sheet" of the design is created. It provides a well readable documentation of the design in standardized property languages. We can use System Verilog Assertions (SVA) or any other suitable property language for this purpose. Importantly, this description also forms the basis for the formal proofs by which the soundness of the system model and, accordingly, the correctness of the design refinements are established.

To this end, abstract properties are automatically generated from the architectural-level description. In analogy to TDD-based software development this is done before the start of RTL code development ("Properties First"). In the actual RTL design process, the designer maintains a formal relationship between the abstract properties and the RTL code by refining auto-generated macros and/or function definitions. This consists of filling in pre-defined template fields and function bodies. This automation step contributes to substantially reducing verification efforts. Moreover, in order to ensure soundness by PPA it is essential that all of the generated properties are refined and proven on the design. By this methodology it is ensured that the resulting verification IP constitutes a *complete set of operation properties* in the sense of a well-defined *completeness criterion* [26, 28]. This results in the powerful theoretical properties of the new methodology which establishes a formal relationship between the architectural level and the RTL, thus, closing the "semantic gap" between the two levels. In the following chapter, we will explain in detail how this formal relationship is built.

Chapter 4

Path Predicate Abstraction

In this chapter we will introduce the theory of *Path Predicate Abstraction* (PPA) and how it is used to establish a well-defined formal relationship between the system level and the RTL. The chapter is structured as follows: In Sec. 4.1 we introduce the main idea of PPA, as well as the meaning of the terms *soundness* and *operation properties.* By example of directed graphs we aim at providing an intuitive understanding of PPA. A more general definition of the PPA for FSMs is introduced in Sec.4.2.

In order to model entire systems with PPA, a compositional approach is elaborated in Sec. 4.3. The theory of PPA has been published in [13, 2, 16, 15]. In the scope of this work, we will define a subset of SystemC, called SystemC-PPA, that provides the semantics of a PPA. As explained in Sec.2.1.4, there is an established chain of trust from the transistor level to the gate level, and from the gate level to the RTL. The correctness of a model at one abstraction level can be verified with respect to the next lower level, e.g., an RTL model can be verified against its gate-level implementation by *formal equivalence checking* techniques such as combinational or sequential equivalence checking. Two sequential circuits are equivalent if and only if they produce identical output sequences for all possible input sequences. Such a notion of equivalence does not exist between design models at the ESL and the RTL. At the ESL, a system is described by modules that communicate abstractly based on untimed message passing, whereas at the RTL, communication is specified with bit and clock cycle accuracy.

For example, an ESL designer specifies reading a message from a bus as $bus \rightarrow read()$. At the ESL this is modeled event-based, e.g., by using handshaking. The RTL design implements the same operation with an arbitrarily complex, cycle-accurate bus protocol. The RTL design is considered sound w.r.t. the ESL if the protocol ensures, under all circumstances, that if $bus \rightarrow read()$ is triggered correctly and a valid message is transmitted from the writer to the reader. In this case, the RTL is a correct *refinement* of the ESL, and the ESL is a *sound* abstraction of the RTL. The term *soundness* describes a well defined formal relationship between an ESL and an RTL model. In the following we describe how this relationship is established. First, we give an intuition into the underlying theory of *path predicate abstraction* [2], and then we show how it is applied in practice in the context of RTL design.

4.1 PPA for graphs

Instead of revisiting the formal development of PPA from [2], we motivate its basic idea by considering a special graph labeling or coloring called "operational coloring". Later, we show how this type of coloring is created using the concept of "operations" in digital circuits.

Definition 15 (Operational Graph Coloring)**.** *Consider a directed graph $G = (V, E)$, a subset $W \subseteq V$ of the graph nodes called* colored *nodes, a set of colors $\hat{W} = \{\hat{w}_1, \hat{w}_2, \ldots\}$ and a surjective coloring function $c : W \mapsto \hat{W}$. A path $(v_0, v_1, \ldots, v_n)$ such that $v_0, v_n \in W$ and $v_1, \ldots, v_{n-1} \in V \backslash W$ is called* operational path *in G. The set W must be chosen and colored such that:*

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

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

In other words, this definition considers a graph with two types of nodes, colored and uncolored. We call this coloring an operational coloring if the following two conditions are fulfilled: Every cycle in the graph must contain at least one colored node. A path starting in a colored node moving through uncolored nodes and ending in a colored node is called an "operational" path. The coloring must fulfill a second condition: If there exists an operational path that starts in some node with color 1 and ends in some node with color 2 then for any other node of color 1 there also must exist an operational path to color 2.

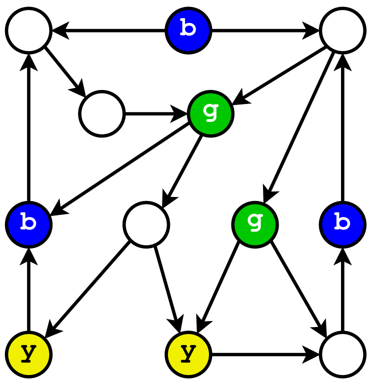


Figure 4.1: Example of an operational coloring

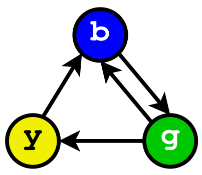


Figure 4.2: Path predicate abstraction of Fig. 4.1

Fig. 4.1 shows a directed graph which may be interpreted as a finite state transition structure such as a Kripke model or an FSM. There are two types of nodes: colored (blue, green, yellow), and uncolored (white nodes). The coloring in Fig. 4.1 is special in that it has some interesting properties w.r.t. paths between colored nodes. An operational path, as defined by Def. 15, is a path that visits only uncolored (white) nodes along the way.

We call this path an *operational path*, or *operation*, for short. Such a coloring is called "operational coloring".

Definition 16 (Path Predicate Abstraction)**.** *We consider a graph $G = (V, E)$ with a set of colored nodes $W \subseteq V$, a set of colors $\hat{W}$ and an operational coloring function $c : W \mapsto \hat{W}$.*

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

This definition states that the abstracted graph contains exactly one node for each color in the original graph, and path segments through uncolored nodes in the original graph are replaced by single edges in the abstract graph. It naturally allows for a powerful abstraction. Since all colored nodes of identical color support the same operations we can create an abstract graph that has one node for each color and one edge representing each operation. All white nodes are removed and all operational paths of the same kind are collapsed into a single edge. Fig. 4.1 shows the abstract graph of the example. Note that, by following this procedure, both graphs, the concrete one in (a) and the abstract one in (b) contain the same set of operations: $b \rightarrow g$, $g \rightarrow b$, $g \rightarrow y$, $y \rightarrow b$. Each operation in the abstract graph is represented by a single edge, whereas each operation in the concrete graph may consist of more than one edge. This is key to modeling multi-cycle RTL operations/transactions. In our PPA formalism, we need this kind of abstraction for the input sequences and the output sequences of operations spanning multiple clock cycles.

What is interesting about this abstraction? There is a well-defined formal relationship between the abstract graph and the original graph: Any path in the abstract graph can be described by a sequence of colors. The underlying theory is briefly introduced in Sec. 4.2 and is published in [13, 2, 16, 15]. The goal here is to provide an intuitive understanding of PPA, as well as the meaning of the terms *soundness* and *operation properties*.

4.2 PPA for FSM

PPA defines a relationship between two automata where one is referred to as the implementation M and the other the abstraction $\hat{M}$. In [51, 2], PPA was defined based on Kripke models and it was shown how can they be translated to Mealy-type FSMs, as commonly used for designing and reasoning about hardware models. PPA describes a relationship between two automata operating on a different set of labels. (For FSMs, this translates to different input and output alphabets.) This is essential when describing the relationship between an ESL model where communication is realized by sending abstract messages, and the RTL implementation where a message may be transmitted in different encodings, and possibly as a sequence of packets. A PPA is defined by a special labeling called *operational coloring* of the implementation M.

For a comprehensive introduction of the PPA, the reader should refer to [2, 13], where an operational coloring is formally defined for Kripke models. Here, we will discuss PPA informally based on FSMs, aiming to provide an intuitive understanding of the abstraction mechanism.

When translated to FSMs, the set of Kripke states becomes a set of three separate coloring functions applied to objects of the RTL implementation: one maps a *subset of* the FSM states to *state colors* $\hat{S}$, another one maps input sequences to *input colors* $\hat{X}$ and the third one maps output sequences to *output colors* $\hat{Y}$. These colors serve as elements of an abstract FSM $\hat{M} = (\hat{S}, \hat{s}_{reset}, \hat{X}, \hat{Y}, \hat{\delta}, \hat{\lambda})$.

In practice, the abstract state set is created by the designer or, in case of PDD, generated from a system-level specification. Abstract states correspond to "conceptual states" or "operation modes" of the design. Every abstract state corresponds to a unique set of implementation states. However, not every implementation state is represented by an abstract state. Some states are "uncolored", i.e., they are "unimportant" intermediate states visited during the execution of an operation. An abstract transition according to the transition function $\hat{\delta} : \hat{S} \times \hat{X} \mapsto \hat{S}$ corresponds to an abstract operation. Operations are "triggered" by concrete input sequences that are represented by abstract input symbols from the set $\hat{X}$ and, according to the output function $\hat{\lambda}$, "produce" concrete output sequences that are represented by abstract output symbols from the set $\hat{Y}$.

Example (9): Consider a bus protocol and a master executing a *write*() transaction. The master node can begin the transaction when the bus is in an *IDLE* state, ready to receive a transaction request. In the abstract finite state machine of the PPA, *IDLE* is an abstract state $\hat{s} \in S$, and *write* is an abstract output symbol $\hat{y} \in Y$, produced by the FSM during the abstract operation from *IDLE* to the same abstract state again, *IDLE*. In the properties describing the cycle-accurate RTL implementation, *IDLE* is a state predicate (to be defined below) in terms of RTL objects modeling all design states in which the bus is idle, and *write* is described as a sequence predicate (to be defined below) representing the set of value assignments to the bus signals such that a write transaction is carried out. For example, *write* could specify a burst transaction beginning with an address phase and followed by a sequence of data items to be written. □

Example (10): As another example, consider the abstract FSM of Fig. 3.3. The transition from state "idle" to state "frame_start" represents an operation that performs a *read* in line 17 of Fig. 3.2, and that receives data such that the *if* condition in line 18 becomes true. The further actions taken by the operation are specified in lines 19 to 21. While *read* and *in_frame* are abstract symbols (alphabet of the abstract level), they may define multi-cycle sequences of value assignments to multiple RTL signals (alphabet of the concrete level) in the implementation. □

Previous work [26, 2] has shown how a PPA can be extracted bottom-up from an already existing RTL implementation. Bottom-up, the coloring, and hence, the abstraction, can be chosen quite freely; however, certain restrictions apply: The coloring functions must obey the requirements for an *operational coloring* as detailed in [2]. These requirements ensure that the abstraction $\hat{M}$ and the implementation M have the same "color behavior". Fig. 4.1 and Fig. 4.2 for graphs, this means that for any sequence of input colors the abstract FSM and the implementation FSM produce the same sequence of output colors. This can be viewed as "sequential equivalence in terms of colors". The colors have a defined semantics for the implementation (based on the coloring functions) and for the abstraction, as they directly represent FSM objects.

Any property that can be expressed in terms of colors has the same meaning and validity in the abstraction and in the implementation. Tab. 4.1 (cf. [2]) shows how any LTL property $\hat{p}$ formulated for the abstraction can be translated to a property p formulated for the implementation, and vice versa. For such properties, PPA ensures behavioral

	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

equivalence between any single abstract model $\hat{M}$ and its implementation M, i.e., $\hat{p} \models \hat{M} \Longleftrightarrow p \models M$.

This establishes the *soundness* of path predicate abstraction. In Tab. 4.1, η_i refers to an abstract-state predicate, and ι_i and μ_i refer to input and output sequence predicates, respectively (to be defined below). The Ψ-predicates are used to characterize implementation states that are colored ("important states"). These predicates evaluate to false for intermediate, non-important states of the design.

A path predicate abstraction is uniquely defined by an operational coloring. In practice, the operational coloring is established through formal property checking. We need to formalize a property for every operation of a design, i.e., we need to create a *complete set of operation properties*. (Note that the properties we talk about at this point *establish* the coloring and the PPA, whereas the properties of Tab. 4.1 *exploit* the soundness and are used to specify verification targets for the system under verification.) Previous work [26, 2] has shown how a PPA can be extracted bottom-up from an already existing RTL implementation. Also the bottom-up approach relies on the completeness of a set of properties. As explained in Sec. 2.3, a set of operation properties is complete if every design behavior is described by one of the properties in the set. Completeness of a property set can be proven formally [26, 28].

In the top-down PDD methodology described in this thesis, the properties are first generated automatically from the SystemC-PPA code in an abstract version, and are then refined together and concurrently with the RTL design implementation. The properties are generated in a standard property checking language like SVA. Their structure follows a certain coding style in which abstract objects like states, input and output sequences are encapsulated in macros/functions of the property checking language. The verification engineer does not touch the abstract properties themselves but merely refines by filling in the bodies of the macros/functions. A main benefit of PDD is that the properties are generated correct-by-construction, they automatically fulfill the completeness criterion (see Sec. 2.3.2) and, thus, cover the entire state space of the design. Formally, such macros are *state predicates* or *sequence predicates*:

Definition 17 (State Predicate)**.** *A state predicate is an LTL formula without temporal operators (Boolean subset of LTL).* □

Definition 18 (Sequence Predicate)**.** *A sequence predicate is an LTL formula where the only temporal operator used is the next operator,* X. □

By allowing only X for expressing temporal relationships, sequence predicates describe finite time intervals (more specifically, finite prefixes of infinite paths).

Definition 19 (Operation Property)**.** *An operation property is a pair* (P, l) *of an LTL formula* P *and the finite length* l *of the operation. The property is of the form of an implication* $A \implies C$ *with* $A = \eta_{start} \wedge \iota$ *and* $C = \mathsf{X}^l \eta_{end} \wedge \mu$.

The state predicate η_{start} *specifies the starting state set; the state predicate* η_{end} *specifies the state set reached at the end of the operation after* l *clock cycles. The sequence predicate* ι *specifies the input sequences triggering the operation. The sequence predicate* μ *specifies the expected output sequences during the operation.*

States fulfilling a starting state predicate η_{start} *or an ending state predicate* η_{end} *are called* important states. □

The antecedent, A, and the consequent, C, are also called "assumption" and "commitment", respectively, of the property.

```
 1: property idle_3_read_5_p(length);
 2:     // freeze variables
 3:         int b_in_data_0;
 4:     // freeze values @t
 5:         t ##0 hold(b_in_data_0, `b_in_data) and
 6:     // triggers
 7:         t ##0 idle_3()  and
 8:         t ##0 b_in_sync()  and
 9:         t ##0 b_in_status() == in_frame
10:     implies
11:         t_end(length) ##0 frame_start_2() and
12:         t_end(length) ##0 cnt() == 3 and
13:         t_end(length) ##0 m_out_sig() == 3 and
14:         t_end(length) ##0 msg_data() == b_in_data_0 and
15:         [...];
16: endproperty;
```

Figure 4.3: Operation property example

Example (11): Fig. 4.3 shows an example of an operation property in SVA, as it is generated in our PDD framework. The property describes the transition from state *idle* to state *frame_start* in Fig. 3.3 (see Example 8). It describes the operation of the module when the *read* in line 17 of Fig. 3.2 has completed and the condition in the *if* statement in line 18 evaluates to *true* .

The property has the structure of an implication: The *assumption* part is the set of sequences before the *implies* keyword in line 10 of Fig. 4.3; the *commitment* part is the set of sequences thereafter. The sequences are conjoined by the "and" keyword. Some custom keywords have been defined to enhance readability: The *t* keyword has a void definition; *t ##0* refers to "time point $t = 0$" in the respective sequence, i.e., the start of the operation. The keyword *t_end(length)* marks the time point the operation ends. (Note that *## 0* is needed for syntactic correctness.) It serves as an abstract time reference, and is concretized (refined) by specifying the actual length that the operation has in the RTL. All objects referred to in Fig. 4.3 are abstract objects. For example, *idle_3()* and *frame_start_2()* are state predicates, realized in SVA by functions. □

Once the RTL design process has started, the abstract properties are refined by filling in the macro/function bodies.

Example (12): The expression in line 9 of Fig. 4.3 is a sequence predicate according to Def. 8. It relates two abstract objects to each other: *in_frame* is of the *enum* type defined in line 1 of Fig. 3.2, and *b_in_status()* refers to the *status* element of the message object received in the the *read()* call in line 17 of Fig. 3.2. Once the RTL implementation of the module exists, the abstract object *b_in_status()* must be refined. In SVA, predicates are realized as functions, and refinement means providing bodies to the functions. Fig. 4.4 shows an example of such a refinement.

```
function status_t b_in_status();
    return to_status_t({ \
    $past(in,4), $past(in,3), $past(in,2), $past(in,1) });
endfunction
```

Figure 4.4: Refinement example

As can be seen, the refinement makes use of the SVA special operator *$past()* in order to collect and concatenate several input values received sequentially over multiple clock cycles. (For simplicity, a function *to_status_t()* is used to perform a conversion from RTL values to a symbolic *enum* value.) □

When the design implementation is completed and all properties have been refined and proven to hold on the design, we have obtained an implementation of a module that is a correct refinement of its SystemC-PPA description.

4.3 Compositional PPA

Up to this point, we have reviewed PPA for a single module as introduced in [2]. Systems, however, are usually composed of several interacting modules. We now review the compositional approach as presented in [15]. Here, the sound relationship between a system of communicating SystemC-PPA modules and its corresponding system of modules at the RTL is introduced.

At the RTL, we model a system of n finite state machines, $M_i = (S_i, I_i, X, Y_i, \delta_i, \lambda_i)$, where $1 \leq i \leq n$. 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 can access the output of every other machine, Y_i for $i > 0$, as a possible input. Y_0 represents the input from the environment. Of course, in practical systems, not every machine reads every other machine's output all the time. By formulating appropriate input sequence predicates (Def. 8) we can select what machine outputs are evaluated at what time points.

In order to create a sound abstract system model, the coloring of the I/O must be consistent among the communicating modules in the system. This means that any initialized I/O sequence must be abstracted to the same sequence of abstract symbols for all modules in the system. We therefore extend the PPA definitions in [2] to *compositional PPA* by defining a *message specification* below.

A message specification characterizes a set of I/O sequences of a specified length for the global input alphabet X. 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 may relate only to the output of one or more specific machines. A receiving machine can access the output of several sending machines simultaneously.

Definition 20 (Message Specification). *A* message specification *is a pair, (μ, l), of a sequence predicate, μ (see Def. 8), and a length l. The predicate μ is specified using only atomic formulas encoding the global input alphabet, X.* □

In this definition, l denotes the length of the operation that produces an output sequence characterized by μ. In practice, such message specifications are defined as *macros* in a property language, relating to logic values on the interconnect lines between communicating machines like in the example of Fig. 4.4. 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 (see Def. 19) is therefore expressed as such a conjunction of message predicates, one from each machine M_j in a system of n machines: $\iota = \bigwedge_{j=0}^{n} \mu_j$, where each message predicate $\mu_j \in Q_j$, with Q_j being the set of all message predicates describing outputs of machine M_j.

Example (13): In the example SVA property of Fig. 4.3, the module is triggered by an input reading only a single port *b_in*. However, if it were to make decisions based on more than one input this would be specified by additional sequence definitions placed in the assumption part. The conjunction of all sequence predicates is formed explicitly by the SVA *and* operator. □

In our PPA theory, the message specifications are mapped to abstract symbols through a mapping function β_j. There is a mapping function β_j for every FSM M_j in the system.

Definition 21 (Message Abstraction). *For every message specification (μ, l) there is a 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 22 (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 message symbols and where $\hat{Y}_j$ for $j > 0$ is the set of abstract message symbols of the j-th sending machine according to Def. 21.* □

Example (14): In practice, the abstract system alphabet consists of a set of abstract message symbols that are defined globally for all modules. In the SystemC-PPA example of Fig. 3.2, the definitions in lines 1 and 2 define a message data type as an *enum* type and message objects as a *struct msg_t*. All modules exchanging messages of this type use the same global definition. □

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 $\hat{M}_i$, i.e., $\hat{M}_i$ is *time-abstract*. 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 23 (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, $1 \le i \le n$, $\hat{Y}_i$ *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* $\exists i, 1 \le i \le n$ *such that* $\hat{s}'_i = \hat{\delta}_i(\hat{s}_i, \hat{x})$ *and* $\forall j, 1 \le j \le n, j \ne i: \ \hat{s}'_i = \hat{s}_i$.
- $\hat{\lambda} : \hat{S} \mapsto \hat{Y}$ *is the output function, labeling every state with the output messages produced by all sub-modules:*
  $\hat{\lambda}\left((\hat{s}_1, \ldots, \hat{s}_n)\right) = (\hat{\lambda}_1(\hat{s}_1), \hat{\lambda}_2(\hat{s}_2), \ldots \hat{\lambda}_n(\hat{s}_n))$. □

This notion of an asynchronous composition is illustrated in an example below (Fig. 4.9). It is important to observe that all possible interleavings of operations between different machines are represented. This model is closely related to the asynchronous product of ω-automata in Spin [52]. 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".

Note that when using a model checker on this asynchronous composition, a liveness property may return counterexamples that are spurious in the implementation because abstract paths exist where sub-FSMs $\hat{M}_i$ remain in waiting states infinitely long. This is usually solved by adding fairness constraints to make sure that every sub-FSM $\hat{M}_i$ infinitely often makes a transition (cf. *finite progress assumption* in [52]). (This is implemented as *weak fairness* in Spin.)

4.3.1 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 a 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.

Asynchronous communication

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 arrives. The message is then transferred either through implicit timing, meaning that the communication partners comply to timing constraints such as setup/hold times of the synchronization signals or guaranteed latency periods. Or, if no implicit timing information is used, proper transmission is signaled through a handshake.

Synchronous communication

In synchronous communication the situation is similar. While the transfer of the individual information bits of a message is controlled by a common clock, the overall orchestration of message exchange is synchronized through certain *explicit synchronization* signals. If only one communication partner sends such a signal (*unilateral synchronization*), then it must be guaranteed (through local timing constraints) that the other partners 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 is either guaranteed through the implicit synchronization by the clock, or may be be signaled explicitly, e.g., to accommodate for varying access latencies such as “wait states” in bus protocols.

4.3.2 Modeling Communication

When using compositional PPA for designing systems in practice, one of the key issues is to model communication between the different modules. Therefore, based on the categorization of communication schemes in Sec. 4.3.1 we show, in the following, how communication can be modeled within our methodology using explicit synchronization at the abstract level, as realized by the SystemC-PPA communication primitives.

Our computational model is based on an asynchronous product of the individual FSMs. An important fact is that a system composed at the abstract level by connecting abstract inputs and outputs, without any abstract synchronization mechanism, is actually sound with respect to LTL. However, such a simple model will usually introduce many spurious transitions due to an over-approximation of the possible 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 spurious transitions. In the following, we explain how to model an abstract synchronization that is sound by construction for the standard communication schemes considered here.

Modeling asynchronous communication

Fig. 4.5 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.

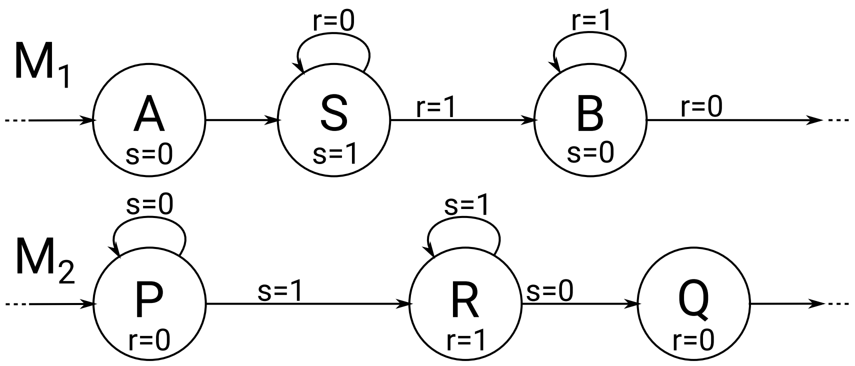


Figure 4.5: Asynchronous communication based on four-phase handshake

(This example corresponds to a “blocking port” communication in SystemC-PPA. For example, the module described in Fig. 3.2 has one such interface, *b_in* (cf. line 8). With respect to the abstract FSMs of Fig. 4.5, the SystemC module of Fig. 3.2 resembles machine M_2, signal s corresponds to the *sync* component of the port *b_in* and the *read*() communication call in line 17 corresponds to the shown state sequence.)

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 $(B, \dots, A, S, B)$ 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.

For creating the communication primitives provided by SystemC-PPA we need to map each of the communication schemes of Sec. 4.3.1 to the four-phase handshake above so that it can be interpreted as event-based message passing on the abstract level. This is trivial for the scheme of asynchronous communication without local timing guarantees at the RTL, because in this case the RTL implementation communicates exactly like the abstract model, using a four-phase handshake. The abstract modeling of the other communication schemes needs more consideration.

Modeling synchronous communication

In case of synchronous communication the four-phase handshake results in an unnecessary overhead regarding the required synchronization signals. In this case, we need auxiliary constructs to establish soundness, because 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 at the RTL with M_1 sending the synchronization signal and M_2 receiving it. Fig. 4.6 shows parts of their state transition graphs. The

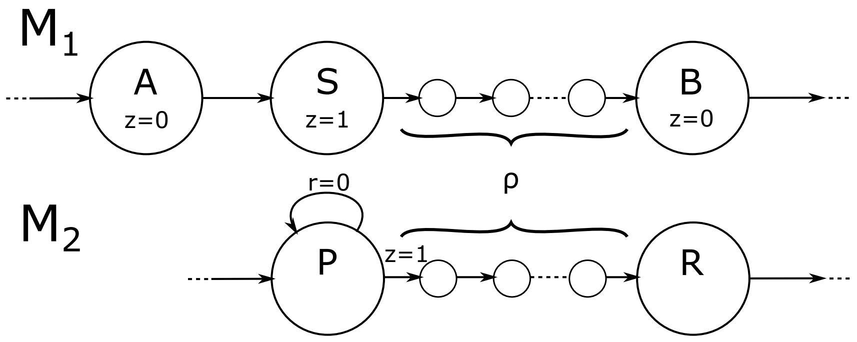


Figure 4.6: Synchronous communication with unilateral synchronization

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 an 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 which data can be exchanged between the machines. During this operation, both machines remain in synchron due to the common clock while they each traverse the operation. A message specification (μ, l) is used to characterize the I/O sequences exchanged in the operation.

The path predicate abstractions of M_1 and M_2 are given by the state transition graphs in Fig. 4.7. The abstract states S and P correspond to the 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. 4.5 we see that states with the same names correspond to each other. The synchronization signal z serves as one of the

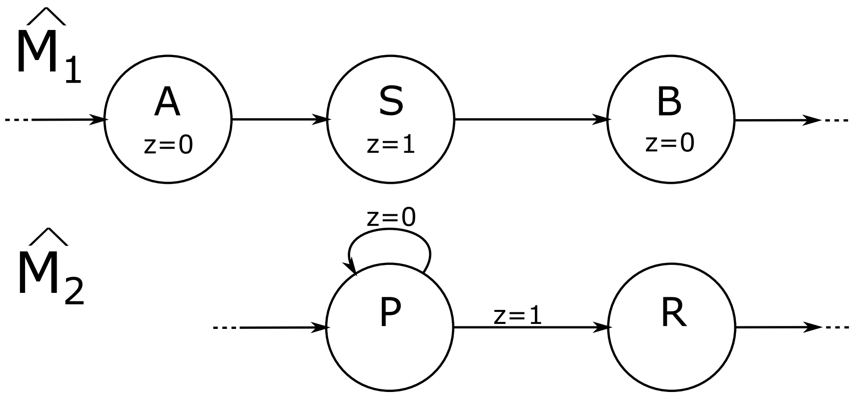


Figure 4.7: Path predicate abstractions of M_1 and M_2 from Fig. 4.6

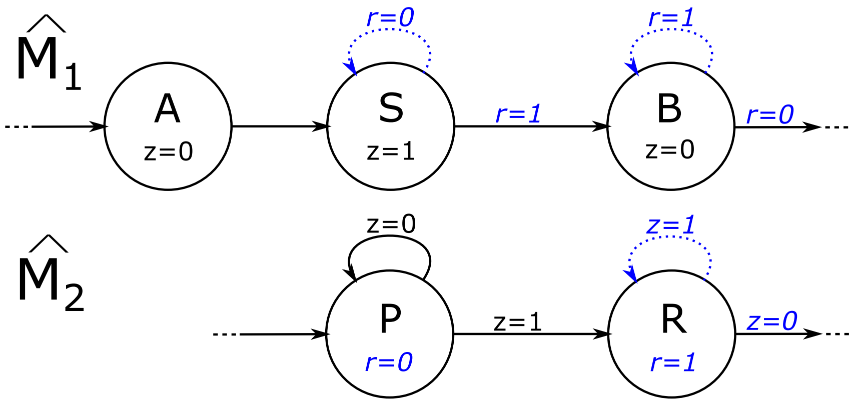


Figure 4.8: Path predicate abstractions of M_1 and M_2 with extensions

handshake signals, namely s in Fig. 4.5. However, for a full four-phase handshake, the dotted elements of Fig. 4.8 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. 4.8.

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 starting and ending states of the operation as the extended abstract system. This must be shown for all communication schemes considered in our methodology.

In the top-down methodology, we account for the different synchronization schemes by providing corresponding communication primitives in SystemC-PPA. For example, the unilateral synchronization scheme is implemented by a "master/slave port" in SystemC-PPA, as, for example, used in Fig. 3.2 in line 9. For this interface, our PDD tool DeSCAM automatically generates properties checking for unilateral synchronization, as well as checking that the timing constraints are indeed met by the RTL.

Soundness for specialized communication schemes

Let us begin with the case of unilateral synchronization as shown in the above example. Referring to Fig. 4.5, 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. 4.8. 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 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. 4.9 shows the asynchronous composition of the machines in Fig. 4.5. As can be seen from the state transition graph, the four-phase handshake between the two machines is capable of modeling a synchronous communication, e.g., Fig. 4.8: 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. 4.9: All fair paths leaving SP always reach BR. (Fairness forbids infinite cycling in SP, SR or BR.)

4.3.3 Synchronization and wait-stuttering

The asynchronous composition of Def. 23 pays tribute to the fact that the concrete timing of an operation in a PPA module is lost in the abstraction. The abstract system model represents *all interleavings* of the transitions in the abstract modules, by considering every possible operation timing instead of just the one in the concrete implementation. Two different implementations have the same abstract system model if the concrete systems differ only in the timing but not in in the sequencing of input/output messages. We may exploit this fact for simpler and more flexible abstraction functions while preserving the soundness of the abstract model.

In this section we introduce a new message category for specific *types* that still ensures soundness of the model even when the correspondence for the message abstraction is

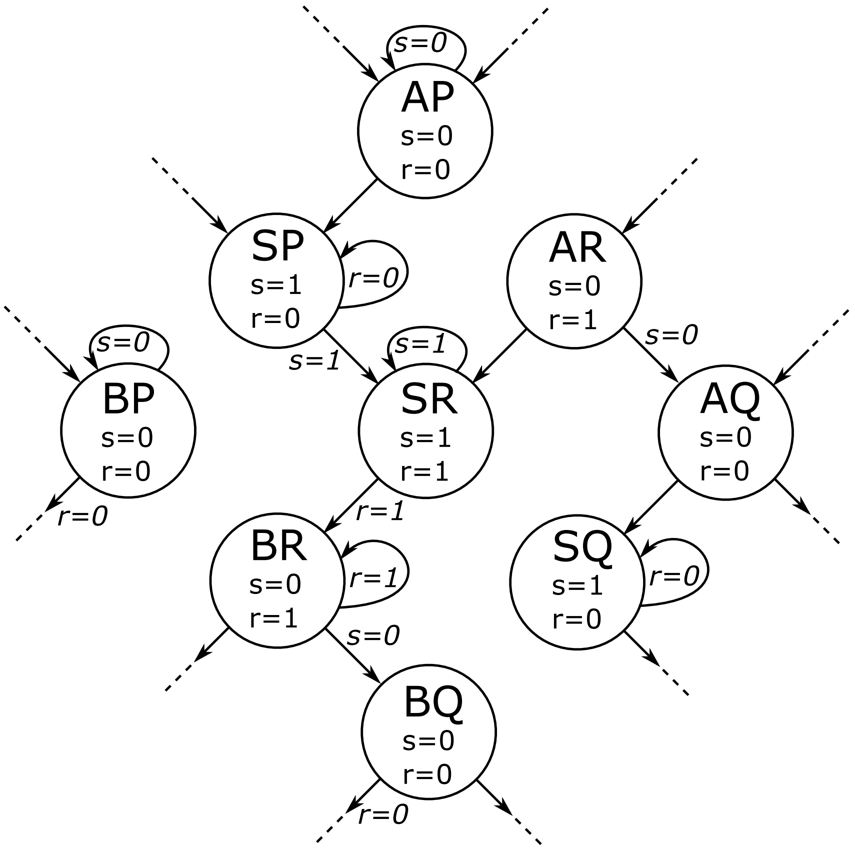


Figure 4.9: Asynchronous composition of machines in Fig. 4.5

weaker than what was originally required by Def. 21. Thus, the extensions made in this section enable a stronger abstraction and allow for a more flexible abstraction technique.

Consider an operation starting and ending in states that are characterized by the same FSM state predicate. Such an operation is mapped to a state with a self-transition in the PPA. If an abstract FSM containing such a state $\hat{w}$ with a self-transition is composed with other abstract FSMs in an asynchronous product then every product state containing $\hat{w}$ has a self-transition, too. If such a product state is reachable then an arbitrary number of consecutive repetitions of the self-transition is reachable. In order for the product machine to be meaningful, the actual number of repetitions cannot be relevant for functional correctness, because the modeled behavior includes any number of repetitions.

Definition 24 (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. 4.10 illustrates the definition of a wait message for a system composed of modules M_1 and M_2. M_1 produces output sequences $\mu_w := (z = 0)$ of length 1 in three states. M_2 receives these messages which only trigger the self-transition of length 1 in state P. The I/O sequences specified by $(\mu_w, 1)$ are examples of wait 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.

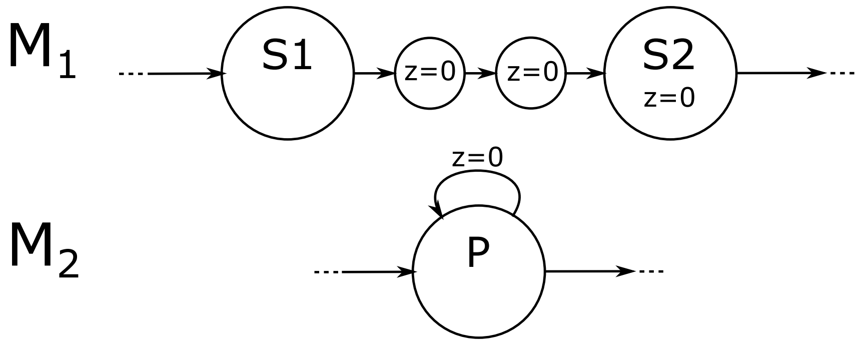


Figure 4.10: Consecutive wait messages

In the example of Fig. 4.10, according to Def. 21, 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", $(\mu_w, 1)$ which triggers the self-transition in M_2. (We write $\mathsf{X}^{i+1}p = \mathsf{X}\mathsf{X}^{i}p$ to represent a generalized LTL "next" operator.) 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 preventing a possible triggering of an actual state change in one of its neighboring modules. Such a disregard could therefore change the reachable product space of the composed system and would make the abstraction unsound.

In many cases, however, the output during an operation triggered by a wait message is completely irrelevant because it does not trigger operations in neighboring modules. We call such wait messages “strict”, as defined below.

Definition 25 (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 26 (Strict Wait Message)**.** *A* strict *wait message is a wait message satisfying only input sequence predicates of waiting operations.* □

Based on the above classification of messages the abstraction requirements can now be weakened without affecting soundness. We allow any finite number of consecutive wait messages to be represented by the same abstract symbol, and we allow abstractions where strict wait messages are not represented at all.

The abstraction of messages was described in Def. 21 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 (see Def. 20). According to Def. 21, operations of different lengths can, therefore, not be described using the same message symbols. In the following we replace Def. 21 by the weakened message abstraction of Def. 27, 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 notation we let $\text{stutter}((\mu, l), k)$ denote the I/O sequence predicate, $\mu \wedge \text{next}_l\mu \wedge \text{next}_{2l}\mu \wedge \ldots \wedge \text{next}_{(k-1)l}\mu$, characterizing k consecutive repetitions of μ of length l.

Definition 27 (Message Abstraction with Wait Stuttering)**.** *Let β_j be the bijective function of Def. 21 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 function $F^{msg} : \{(\mu_{tot}, l)\} \mapsto \hat{Y}$ is a message abstraction function if $F^{msg}((\mu_{tot}, l)) = \hat{y}$ implies one of the following:*

1. $\beta_j((\mu_{tot}, l)) = \hat{y}$
2. $\beta_j((\mu_w, q)) = \hat{y}$ *and* $(\mu_{tot}, l) = \text{stutter}((\mu_w, q), i)$ *and* $l = i \cdot q$

3. $\beta_j((\mu_{msg}, a)) = \hat{y}$ *and*
$(\mu_{tot}, l) = \text{stutter}((\mu_{sw}, q), i) \ \wedge\ \text{next}_{q \cdot i}(\mu_{msg}) \ \wedge\ \text{next}_{a+q\cdot i}(\text{stutter}((\mu_{sw}, r), j)$
and $l = i \cdot q + a + j \cdot r$ $\square$

The first form of the message abstraction is the original form of Def. 21. 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 $\hat{y}$. The third form describes a message with three phases. The first phase and the third phase are (possibly empty) stutterings of strict wait messages. The second phase is a single instance of a non-wait message. These three phases together are abstracted into the single abstract message symbol $\hat{y}$ corresponding to the non-wait message.

With Def. 27 we can now concisely model any communication scheme in practical systems including the schemes presented in Sec. 4.3.1 by abstracting synchronization signals using stuttering of wait messages.

4.3.4 Model checking on the 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. 23 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 $\hat{R}_K \subseteq \hat{S}_K \times \hat{S}_K = \{((\hat{s}, \hat{x}), (\hat{s}', \hat{x}')) \mid (\hat{s}, \hat{x}, \hat{s}') \in \hat{T}\}$. 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 $\hat{A}_K = \bigcup_{i=0}^{n} \hat{A}_i$.

In the following we consider an abstract model and a concrete model with an already established sound relationship. Due to its 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. Hence, successfully proofing a property on the abstract model guarantees that the corresponding property is also valid for the concrete model, but not vice versa.

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 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 existing in the implemented, 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., does not depend on the processing speed of the individual modules, can be verified on the abstract level.

Theorem 1 (LTL Soundness of Composed Model)**.** *Consider an LTL formula $\hat{\varphi}$ for the composed abstract model and the corresponding LTL formula φ for the concrete system 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* (see Def. 19). 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 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. This fragment always has a corresponding abstract path fragment, because the start and end states, by construction (see Def. 23), 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.

4.3.5 Data Path Abstraction

The formalism of PPA is well suited to create an abstraction for the control of a hardware implementation. Including the data path is straightforward. We simply view all state and input variables to be part of the control. In the PPA formalism, this would lead to separate operations and operation properties for every possible data path value. Although formally correct, this is, clearly, not practical.

However, there is a technical solution to this problem. We allow each property to distinguish between a set of control states, a set of data path variables and several input and output message types. The properties that are created describe operations as transitions between control states in terms of a set of input messages and a set of data path variables. This distinction is a purely syntactical one, and an operation described in this way has a direct translation into sets of operations matching our formal model. Soundness of the abstraction is not compromised. A translation always exists when the abstract state space is the product of the control states and the data path states, and the "input message space" is the product of the set of input messages.

The operation properties written in this way describe transitions between concrete data path states *implicitly* using functions. The functions describe the data path functionality and map between the data path variables representing operands and those representing the results of a computation. This allows for efficient abstractions of data path objects of various kinds, including predicate abstraction and word-level abstractions. Note that due to the time-abstract nature of PPA, cycle accuracy of concrete data path computations is always abstracted away.

4.4 SystemC-PPA

In the following, we sketch the subset of SystemC, called "SystemC-PPA", that can be used for sound hardware modeling. We need to restrict ourselves to a subset because SystemC, *per se*, lacks a clear semantics with respect to modeling digital hardware.

The subset requires the user to write the SystemC model in a certain format such that the notions of Compositional PPA (cf. Sec. 4.3) apply and a set of communicating FSMs can be extracted. A SystemC-PPA model of a system consists of modules communicating with each other using event-based message passing. To enable extracting a module's behavior as a PPA, the module must:

1. model an FSM in a time-abstract fashion,
2. have a single thread executing infinitely (infinite outer loop),
3. communicate only through an allowed set of communication ports,
4. not block unless a blocking communication interface is called,
5. have no cyclic execution path without a blocking communication,
6. use only objects whose size are known at compile time (e.g., no dynamic memory allocation).

We provide a set of predefined communication interfaces to simplify the mapping of messages between modules to sequence predicates. These interfaces are necessary, because DeSCAM needs to know the underlying communication protocol of a communication interface to correctly generate the properties. These interfaces, however, implement the SystemC TLM 2.0 standard in order to enable all performance optimizations available by SystemC. They do not impose any restrictions on the system-level designer and SystemC-PPA modules are able to interact with other SystemC modules.

Despite these restrictions, SystemC-PPA supports a large subset of the C language including arithmetic operators, built-in C++ datatypes, *enum* and compound types (*structs*), and arrays. *If-then-else* is used for flow control. Note that – as of the time of this writing – loops must be implemented in FSM style by jumping to and iterating over *sections*. Future work will extend the subset to bounded *for* and *while* loops as well as *switch-case* statements. Since any hardware behavior can generally be described using the concept of finite state machines, SystemC-PPA places no restrictions on *what* can be modeled (only on *how* it is described). Intuitively, we may view SystemC-PPA as the "hardware-designable subset" of SystemC at a transactional level.

We illustrate some key features of SystemC-PPA using Fig. 4.11 and in the following sections, we explain some key language features in more detail. A full documentation of the features can be found in the online documentation to DeSCAM [50].

The module provides communication interfaces to other modules in the system. Lines 8 to 10 define three such interfaces. There is a *blocking* interface, corresponding to a full four-phase handshake between the communicating modules (cf. Sec. 4.3.2). During simulation, the *read*() in line 17 blocks and suspends the module's SC_THREAD by calling a SystemC *wait*() until the counterpart of the communication has called the corresponding *write*(). When a blocking interface is refined to the RTL, the generated set of properties contains special synchronization functions/macros that need to be implemented by the RTL developer in order to guarantee a full four-phase handshake.

When RTL designs are synchronized through a common clock, four-phase handshakes bear unnecessary overhead. Line 9 shows an example of a so-called *master/slave* interface, avoiding such overhead in the RTL refinement. A master/slave interface refines to a *unilateral synchronization* scheme (cf. Secs. 4.3.1, 4.3.2). It relies on the timing constraint that the *slave* module is guaranteed to be ready for communication whenever the *master* module sends a synchronization signal. The generated properties ensure that the timing constraint is met by the RTL design.

As a third communication primitive, line 10 shows an example of an unsynchronized port. It can be used to model volatile data (like sensor I/O) or to provide additional information together with some other communication port that is of the blocking or master/slave kind.

The control and computation behavior of the module is described in the form of an FSM description (cf. line 12 in the example). It is divided into *sections* that loosely correspond to operations of the design. However, the actual abstract states of the PPA defined by the module are given *implicitly* through the communication primitives *read*() and *write*(), as shown, e.g., in lines 17, 24 and 31.

Fig. 4.12 shows the state transition graph of the abstract FSM defined by the SystemC-PPA module in Fig. 3.2. The nodes represent communication states of *read* or *write* calls in the different sections in which the FSM waits for synchronization signals from the mod-

```
enum status_t {in_frame, oof_frame};
struct msg_t {status_t status; int data; };
SC_MODULE(Example) {
  SC_CTOR(Example):
      nextsection(idle) { SC_THREAD(fsm)};
  enum Sections {idle, frame_start, frame_data};
  Sections section, nextsection;
  blocking_in<msg_t> b_in;
  master_out<int>m_out;
  shared_out<bool> s_out;
  int cnt; bool ready; msg_t msg;
  void fsm() {
      while (true) {
          section = nextsection;
          if (section == idle) {
              s_out→set(false);
              b_in→read(msg);
              if (msg.status == in_frame) {
                  s_out→set(true);
                  nextsection = frame_start;
                  cnt = 3;
              }
          } else if (section == frame_start) {
              m_out→write(cnt);
              cnt = cnt - 1;
              if (cnt == 0) {
                  cnt = 15;
                  nextsection = frame_data;
              }
          } else if (section == frame_data) {
              ready = b_in→nb_read(msg);
              if (!ready) {
                  m_out→write(msg.data);
                  if (cnt == 0) {nextsection = idle; }
                  cnt = cnt - 1;
              }
          }
      }}};
```

Figure 4.11: Fig. 3.2 from Chap. 3

ules it communicates with. For example, the blocking *read*() on port ***b_in*** in the section named ***idle*** introduces the abstract state idle. It has a self-loop corresponding to the waiting operation (cf. Def. 25) that is executed until the synchronization signal ***sync*** becomes true. Such synchronization signals are provisioned automatically by DeSCAM. They are generated as functions/macros in the property suite and need to be refined/mapped by the developer to the corresponding features in the RTL design as explained in Sec. 5.3. Again, this enforces correct implementation and establishes the soundness of the refinement. For this example, the SystemC source code, the generated properties, and a possible RTL implementation are available for download [50]. Chap. 5 elaborates on this example in more detail and explains, in depth, how PPA models are specified in SystemC-PPA.

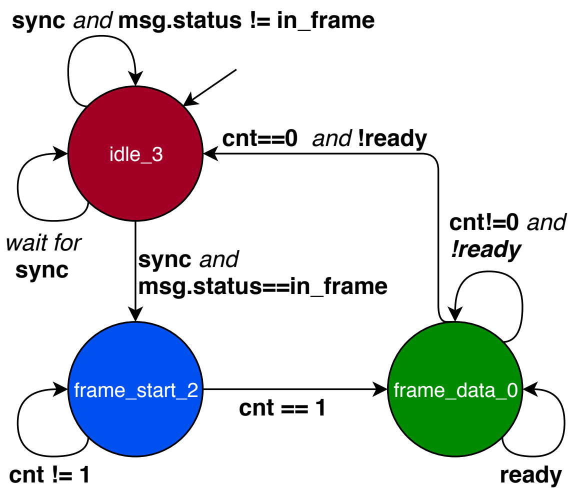


Figure 4.12: PPA for Figure 3.2

4.4.1 Interfaces

In this section, we talk in more detail about the supported interfaces of a SystemC-PPA model. Exchanging a message between two modules is done using synchronization via a handshake. At system level handshaking is implemented using built-in SystemC events. At the RTL, such message passing may translate to a handshake-based transaction, which might bear unnecessary overhead in case message loss is acceptable. SystemC-PPA addresses this issue by providing three different types of communication interfaces called ports (standard SystemC ports could not be adopted because none of them provides pure *Rendezvous* communication).

In most cases, these three supported types suffice to implement any kind of hardware communication. Each type of port results in a different PPA. Regarding the PDD methodology, use of different ports affects the generated property suite and resulting hardware design as presented in Chap.5. In our proposed RTL generation flow, a difference in PPA directly affects the generated hardware. Hence, it is important to select proper communication interfaces between modules during the ESL design process to produce an efficient RTL equivalent. More information about the resulting port implementations at the generated RTL can be found in Sec. 5.3.2. The SystemC-PPA supported ports are *Blocking*, *MasterSlave*, and *Shared*.

Blocking

The blocking interface guarantees that a message sent by the sender is received by the receiver module. As the name suggests, the blocking interface blocks the sender/receiver until both sides are ready for the message exchange. In other words, the blocking interface forces synchronization by implementing *Rendezvous* communication, where both communicating sides have to inform each other once they are ready for a message exchange.

Here the blocking communication relates to the asynchronous communication scheme presented in Sec. 4.3.2. Currently, the generated properties are only supporting synchronous communication, meaning that the four phase handshake is reduced to a two phase handshake. The methods that initiate blocking communication are `port_name->write(val)` and `port_name->read(var)`.

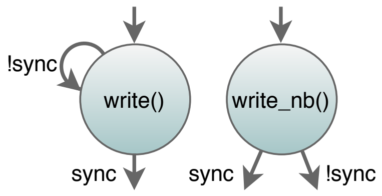


Figure 4.13: Different wait automata for `write()` and `nb_write()` of a blocking interface.

In addition, the blocking interface also offers non-blocking sending and receiving of messages. By calling the method `port_name->nb_write(val)` a write is initiated regardless of whether the receiver is ready or not. If the message is successfully delivered the method returns `true`, otherwise it returns `false`. The same applies to `port_name->nb_read(var)`. Fig. 4.13 shows the different wait automata for the regular write and the non-blocking write. The incoming synchronization signal *sync* is *true* if the counterpart module is ready for communication, or *false* otherwise. Fig. 4.13 shows the difference in the blocking behavior of the two methods.

MasterSlave

The MasterSlave interface can be only used for modeling synchronous hardware systems, e.g., system using a common clock. It connects two modules of master and slave type. The interface is based on the assumption that the slave module is ready to communicate at any given moment. This allows to send a message without waiting for the slave's synchronization signal.

The master interface offers two possible methods, `port_name->write(val)` for writing and `port_name->read(var)` for reading messages. The master can communicate with the slave at any time point with the guarantee that the slave is ready to either receive or send a message. As a result, these methods are non-blocking and no message loss can occur from the perspective of the master.

The slave interface provides similarly named methods as the master interface. A slave is always ready to communicate with the corresponding master either by repeatedly sending messages (in case the master initiates a read) or by constantly checking for arriving messages (in case the master initiates a write). It is accepted that messages sent from the slave can get lost. On the other hand, capturing a message should only happen if the corresponding master did actually send one, which is indicated by the return value of the method call. Both methods are non-blocking. A slave module must not contain any blocking communications within it to comply with the "always ready" requirement.

Shared

The shared interface does not involve any handshaking mechanism and therefore can model the behavior of volatile memory. Hence, it is useful for modeling input data like sensor values, that can change at any moment. It offers two methods, `port_name->set(val)` for sending a message, and `port_name->get(var)` for retrieving a message.

4.4.2 Components of SystemC-PPA

A brief description of each of the supported components of a SystemC-PPA description is provided in the following sections.

Data Types and Variables

SystemC-PPA supports the built-in data types: `bool`, `int` and `unsigned int`, as well as `enums` and custom data types. The last two have to be defined within the scope of the module, as shown in Listing 3.2 in lines 2 and 3. Custom data types are called *compound* and their members can consist of built-in data types and `enums`, as demonstrated in line 3. (Constructors and methods within compounds are not supported). Variables needed for the behavioral description have to be declared, as illustrated in line 18, within the SystemC module definition and not within the behavioral description itself.

Constructor

If the model is written with the help of sections (which is optional, but highly recommended), the constructor must contain section initialization. It can also include other variable initialization if needed (variable initialization during the declaration, as well as initialization of the sub-variables of a compound data type are not supported in the current version of SystemC-PPA). The SystemC macro `SC_THREAD(fsm)` registers the method `fsm()` as a thread that describes the behavior of the module, as shown in line 9.

Ports

The supported ports are derived from SystemC class template `sc_port` with custom interfaces defined by SystemC-PPA. The possible interfaces are *blocking*, *master*, *slave* and *shared*, all of which implement a different blocking mechanism. Allowed directions are *in* for receiving and *out* for sending messages. Message types may be of any of the SystemC-PPA supported data types. A port forwards the calls of interface methods in a module to the channel that it is bound to. Example port declarations can be seen in lines 11 to 13.

Functions

Functions are not allowed to change any state variables of the module since they are meant to model combinational logic. This is ensured by declaring the function as `const`. The function may only return built-in and `enum` data types.

FSM

A SystemC-PPA module must have a single method that describes its behavior and is registered as a SystemC thread as required also by the synthesizeable subset of SystemC. The use of `SC_THREAD` is necessary for simulation of blocking behavior because `SC_PROCESS` and `SC_METHOD` cannot be blocked. An example structure of such a method is illustrated in Fig. 3.2 line 12 to line 38.

The method must contain one infinite outer loop written as `while(true){..}` with the body describing the behavior. This is necessary to precisely define the behavior as an

FSM and model the continuous operation of digital hardware. The ESL model describes untimed behavior and all notions of time are not allowed.

The only allowed control structures within the `while(true)`-loop are *if-then-else* (ITE) constructs, because each path from a communication function call to the next one has to consist of a finite number of C statements. Bounded while and for-loops are currently not allowed. This is only due to technical reasons. The code can be structured with the help of *Sections* to better represent individual stages. *Sections* can be defined by an `enum` type that is declared in combination with two variables *section* and *nextsection*.

Chapter 5

DeSCAM approach

This chapter demonstrates *Property-Driven Design* from a practical point of view. As shown in Fig. 3.1, the desired flow starts with an executable system-level model. Any language with clear semantics w.r.t. a PPA can be used as a design entry language. In the scope of this work, we are going to focus on SystemC-PPA as introduced in Sec. 4.4.

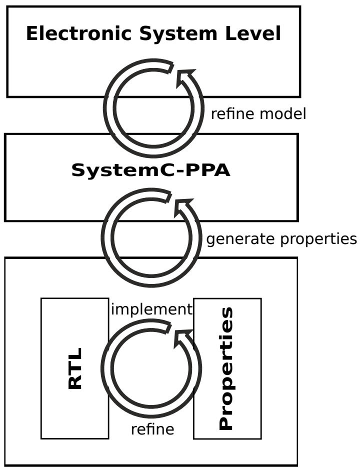


Figure 5.1: Implementation and refinement

The overall flow is depicted in Fig. 5.1 and consists of three major steps:

1. Analysis of the SystemC description for compliance with the SystemC-PPA subset,
2. generation of the properties and
3. design of the hardware with PDD design methodology.

We developed a tool supporting this flow. It generates the operation properties from the SystemC-PPA description and it can generate a RTL template as a starting point for the design process. The approach and the accompanying tool is called ***De****sign from* ***S****ystem****C*** ***A****bstract* ***M****odels* (DeSCAM). A public version of the tool is available as open source on GitHub [50].

In Sec. 5.1, we explain how we parse and analyze a given SystemC-PPA description for compliance with the SystemC subset defined in Sec. 4.4. Furthermore, we explain how a PPA is extracted from a SystemC-PPA description by providing a “coloring” algorithm.

It applies an operational coloring (cf. Sec. 4.1) to the control flow graph of the SystemC-PPA module and thereby computes the PPA.

Afterwards, in Sec. 5.2, we discuss how the parsed SystemC-PPA description is transformed into properties and, lastly, in Sec. 5.3, we show how the properties are refined along a hardware design process by an example.

5.1 From ESL to PPA

The first step of the DeSCAM approach is to parse a SystemC-PPA source file and create a data structure, called *Abstract Model*, that stores the information relevant for property generation. Fig. 5.2 shows the flow for this step. For parsing, we use the open-source compiler framework LLVM/Clang [53]. The parser produces a representation of the SystemC-PPA input as an *Abstract Syntax Tree* (AST). An AST represents the entire program code, including the SystemC scheduler or C libraries, in a tree-like structure.

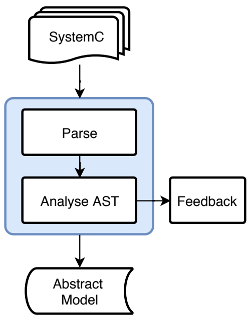


Figure 5.2: Abstract model generation

In the next step, the AST is analyzed and all information required for property generation is stored. This analysis is a static source code analysis. This means that only information that is available during compile time is extracted. Details that are relevant only for C++ analysis or SystemC simulation, like the SystemC scheduler, are stripped away. The remaining information, describing the module in a PPA view, is stored in a new data structure called *Abstract Model* (AM). The AM is explained in full detail in Sec. 5.4. It contains structural information in the form of an abstract syntax tree and behavioral information as a control flow graph (CFG). Initially, a CFG is automatically generated by Clang for the class method *void fsm()* of a SystemC-PPA module. The source code of this method needs to have a certain structure (cf. Sec. 4.4) such that (1) its behavior, when simulated by the SystemC framework, is that of a finite state machine (FSM), and (2), the FSM in terms of states and transitions can be extracted from the control flow graph of the SystemC code.

In fact, this FSM extraction is straightforward. The CFG nodes can serve directly as nodes of the state transition graph of the FSM, with a few exceptions, such as the outer *while (true)* loop, the *if-then-else (ITE)* structure for the sections and any *nextsection* assignment (if present). DeSCAM removes the respective CFG nodes and produces a CFG that represents the FSM of the PPA, as shown in Fig. 4.12.

The nodes are labeled with the line numbers of the statements in Fig. 3.2. The dashed boxes indicate the sections to which the individual CFG nodes belong. The tool analyzes

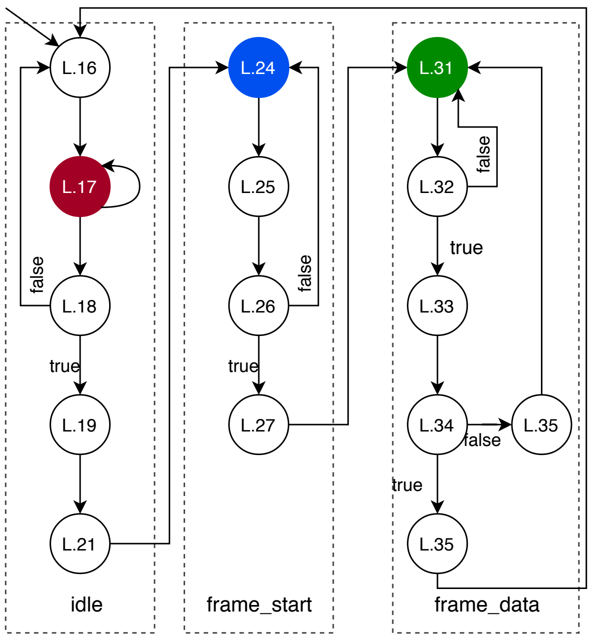


Figure 5.3: Generated colored CFG

each object of the SystemC/C++ program and checks for compliance with SystemC-PPA. Violations are reported in form of warnings and errors and the affected objects are rejected. ESL models written in SystemC often serve several purposes at the same time, e.g., to support early firmware development, to help in integration testing and to generate tests. SystemC models may therefore contain code that is meaningless in the PPA view but should still remain in the source. When DeSCAM warns about code that is not SystemC-PPA compliant the user needs to analyze the diagnostic output and make a decision on whether the affected code is relevant for the abstraction and should stay included in the model or not. For example, take the following C++ statement:

```
std::cout <<"Hello" <<std::endl;
```

This piece of code produces the DeSCAM error:

```
-E- Unknown error:  Stmts can't be processed;
```

This statement may have side effects that could change the behavior of the modeled component, for example, if the stream pipes into a variable that is used inside the module. The designer must then refine the SystemC code to remove the error. Otherwise, if the designer is certain that no side effect exists, for example, because *std::cout* has its default behavior of writing to the console, the error can be flagged and ignored.

5.2 From PPA to Properties

In the next step DeSCAM generates a PPA from the Abstract Model. This step has three phases, as illustrated in Fig. 5.4.

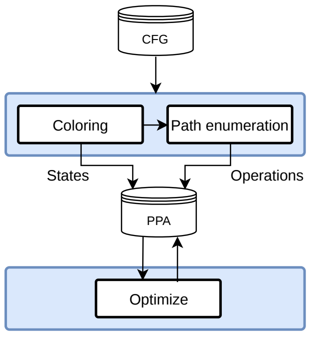


Figure 5.4: Overall flow of model generation

- Coloring: Operational coloring is applied on the initially uncolored CFG. In case of PDD the coloring is trivial, because the important states are implicitly provided by the communication primitives. Each communication with synchronization results in an important state.
- Enumeration of operational paths: Every path between two important states (see Def. 12) in the CFG results in an operation (see Def. 10) for which an operational property is generated.
- Optimization: The PPA is optimized with respect to structural complexity, in order to enhance conciseness and readability. Sec. 5.5 presents the details of the optimization procedures.

Prior to applying the operational coloring, the CFG is extended by additional operations that are defined implicitly by the SystemC-PPA model:

- **reset operation**: It is defined by the initialization of the *nextstate* variable in the constructor (*SC_CTOR*) of the *SC_MODULE* (cf. Fig. 3.2). In the current version, DeSCAM supports only SystemC-PPA modules with a single constructor. The reset operation ensures the correct initialization of RTL registers and guarantees that the design is in the correct important state after reset. This is necessary for fulfilling the completeness criterion (see Def. 2.3.2).
- **wait operation**: Wait operations are generated for communication over *blocking ports*. They correspond to the wait operation described in Def. 25. If *read*() or *write*() is called and the counterpart of the communication is not ready, then the module blocks, i.e., it needs to wait. A wait operation is added to the PPA that forces the module to remain in its state until the counterpart is ready for communication.

After the insertion of implicit operations, operational coloring, as required for generating the PPA, is applied. First, every node resulting from a blocking communication is colored (i.e., red and green). A communication primitive implementing the *master* interface is colored only if condition 1 of Def. 15 is violated. The condition states that any cyclic path within a graph must be broken by a colored node. This is the case for line 24, thus *L. 24* is colored blue. The communication primitive at line 33 does not violate the

```
property red-to-blue is
assume:
    -- starting states
    at t+0: state == L. 17;
    -- trigger sequence
    at t+0: sync == true;
    at t+0: msg.status == in_frame;
prove:
    -- output sequence
    at t+n: s_out == true;
    at t+n: cnt == 3;
    -- ending states
    at t+n: state == L. 24;
end property;
```

Figure 5.5: Operation property: red to blue

condition because there is no path without a colored node starting from *L. 33*. The paths end in either green or red.

Fig. 5.5 shows one of the resulting operation properties. This property specifies the transition from red to blue. It verifies that after a successful handshake and detecting a new frame in control state *L. 17*, the operation will always end up in control state *L. 24*, with *counter* set to 3 and with the shared port set to *true*.

In contrast to Fig. 5.3, DeSCAM names the states not by the line of code in which they are declared. Instead, as Fig. 4.12 shows, states are named by the section in which they are declared, extended by a unique ID generated by DeSCAM. If multiple communication calls occur within one section they are distinguished by the unique ID. In our example, *L. 17* is renamed to *idle_3*, *L. 24* to *frame_start_2* and *L. 33* to *frame_data_0*. The operations of the PPA are labeled with the trigger conditions. The effects of different port interfaces on the RTL implementation become apparent in Fig. 4.12. The state *idle* has a wait operation because the communication counterpart may not be ready when the module enters this state. On the other hand, the state *frame_data_0* does not have a wait operation because the communication call uses the non-blocking *nb_read()* method. The state *frame_start_2* requires no handshaking, because the respective port implements the *master* interface.

Initially, each variable specified at the system level leads to the implementation of corresponding registers in the RTL. If a variable is used only for the sake of utility, e.g., storing intermediate computation results, an RTL register is not needed. In the given example, this is the case for the variable *ready*. The variable is assigned the status value indicating success of communication via port *b_in*. It is used only once in the if-then-else at line 32 where it can be safely replaced by the status value itself, in all operations containing line 32. By contrast, the variable *cnt* at line 35 is assigned its previous value, decreased by one. The new value depends on the previous one and thus the variable is a necessary part of the state space of the PPA resulting in an RTL register.

5.3 From Properties to RTL

The final, manual, step of PDD is the actual implementation of the RTL from a PPA. Its workflow is depicted in Fig. 5.6. The starting point is the complete set of properties as generated from the PPA by DeSCAM, where each property represents one operation that needs to be implemented at the RTL. For each abstract object of the system level (variables, data values, etc.) a *macro* or *function* definition (in SVA) is generated. The operation properties are built using these macros. For example, the macro *cnt* in Fig. 5.5 results from the system-level variable with the same name, *cnt*.

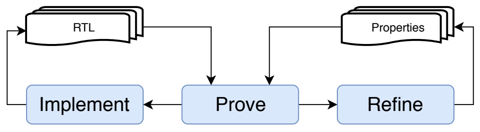


Figure 5.6: Implementation and refinement

The designer implements the RTL code matching the properties, one property at a time. Each property is refined concurrently with the implementation, i.e., timing information is added as well as detail concerning the datapath implementation. If the chosen property holds on the design, the hardware implements the operation correctly. Otherwise the RTL code needs to be corrected (or the timing adjusted). The design process is finished once all properties can be successfully verified on the design.

The property suite provides a valuable verification IP for the finished design. It serves as a formalized data sheet precisely documenting the implemented functionality.

Sec. 5.3.1 explains in more detail what a macro is and how it relates an abstract object of the system level to a concrete RTL implementation.

5.3.1 Macros

Macros are an important vehicle for abstraction. By encapsulating references to RTL objects at different time points they decouple RTL implementation details from their abstract representation in a property. The property definitions themselves remain unchanged during the design process. Only the macro bodies are edited. This approach has two benefits. Since the generated set of properties is complete, independently from the actual content of the macros, it is not possible to introduce a verification gap during the refinement process; also, instead of the whole property text only a small part of the generated lines of code has to be adapted by the designer. We evaluated the associated work effort. Results can be found in Sec. 7.

In this and the following section we illustrate the abstraction constructs using the commercial property language *ITL* [25]. Note that DeSCAM also supports SVA printout for which it generates *defines* and *functions* instead of *macros*. The interested reader can find the RTL implementation as well as the refined properties (both in ITL and SVA) for the example of Fig. 3.2 on our GitHub site.

Fig. 5.7 shows an abstract definition of a macro, consisting of a name, return type and a body as it is initially generated by DeSCAM. In the *macro body* the designer specifies the implementation details of the abstract object. Sometimes, a word-level RTL variable has the same name and an equivalent type as the system-level variable it implements.

```
macro MACRO-NAME: RETURN-TYPE :=
    MACRO BODY
end macro
```

Figure 5.7: Macro definition

```
macro cnt: int :=
    instance/RT_signal
end macro
```

Figure 5.8: Simple refinement for *cnt*

For example, Fig. 5.8 shows such a straightforward refinement for the abstract system-level variable *cnt*. A variable can also be mapped to a more complex RTL data structure, as shown in the example of Fig. 5.9 where "cnt" is represented by a composition of two slices named "foo" and "bar".

```
macro cnt: int :=
    instance/foo[31 downto 16]
    & instance/bar[15 downto 0]
end macro
```

Figure 5.9: Non-trivial refinement for *cnt*

For refining the macros there are only three rules:

- Sequences may be formulated only over finite time windows (of arbitrary length).
- Functions characterizing abstract inputs may be expressions over only input signals of the RTL design.
- Functions characterizing abstract outputs may be expressions over only output signals of the RTL design.

Ports are modeled by a combination of three specialized macros: *notify*, *sync* and *datapath*. The combination of these macros is determined by the type of communication interface the port implements. A port with a *shared* interface requires no synchronization because it models a volatile access. Hence, only a *datapath* macro is generated for it. The datapath macro describes the message and is named *portname_sig*. The operation properties guarantee that the correct message is sent by specifying a value for the datapath macro. Ports with a *master/slave* interface require, generally, no synchronization. The output port with a master interface is complemented by *notify* indicating validity of a new message for the respective slave input. Conversely, the slave input has a *sync* macro in order to evaluate the validity of the incoming message. The macros are named *portname_notify* and *portname_sync*, respectively.

For a port implementing a *blocking* interface both macros, *notify* and *sync*, are required to implement a handshaking mechanism. The four-phase handshake starts with raising the outgoing *notify* flag. The module is blocked until the incoming *sync* flag evaluates to *true*, indicating readiness of the counterpart. At the end of the transmission both flags evaluate to *false*. The correct handshaking is enforced by the generated operation properties. The design has to fulfill these properties, resulting in a correct-by-construction handshaking. The evaluation of the macros is not necessarily restricted to a single signal changing its value in a single clock cycle. The macros may describe an arbitrary protocol spanning multiple cycles and different signals.

Macros for datapath registers result from variables at the system level. The macros for compound types are split into separate macros for each subtype. For example, the variable *msg* is separated into two macros *msg_data* and *msg_status*. The same idea applies to port macros. The provided example implementation has three datapath registers: one for the variable *cnt* and two for the variable *msg*, namely *msg_data* and *msg_status*. As explained earlier, the variable *ready* is not required for the RTL implementation.

Important states, derived from the communication calls at the system level, each result in a macro. Fig. 4.12 shows the resulting PPA with three important states, *idle_3*, *frame_start_2* and *frame_data_0*. Each important state results in a macro of *boolean* return type. If the hardware is in an important state the macro evaluates to *true*, otherwise to *false*.

5.3.2 RTL Interfaces

In the following, we explain how SystemC-PPA-supported communication interfaces are interpreted at the RTL. As described in Section 4.4, there are three supported interface types: Blocking, MasterSlave, and Shared.

Blocking

In order to implement a blocking communication (write and read), a handshake as explained in 4.3.2 is required. This handshake ensures a correct synchronization between the modules. As demonstrated in Fig. 5.10, the handshake is implemented at the RTL using `sync` and `notify` signals. The data is exchanged by the data signal.

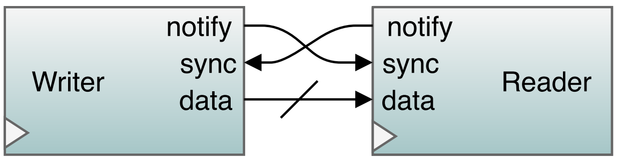


Figure 5.10: Connection between blocking reader and blocking writer modules

Fig. 5.11 depicts a timing diagram of a transaction over the blocking interface (shown from the writers perspective) when blocking read and write methods are used. This waveform demonstrates how the handshake presented in Fig. 4.5 is implemented. In this case, we assume a synchronous communication and thus the four-phase handshake is simplified to two phases. However, extending the presented protocol to a four-phase handshake only requires implementing the missing de-assertion phase of the notify signals. The handshake is transformed into the following protocol: The blocking ports have to initiate reading (or writing) by asserting the notify signals; only then can the transaction occur. The communicating modules are waiting for each other as shown with arrows t_1

and t_2 until both ports are ready. As soon as both ports are ready the message (i.e., value of data port) is exchanged and both sides lower their respective notify signal.

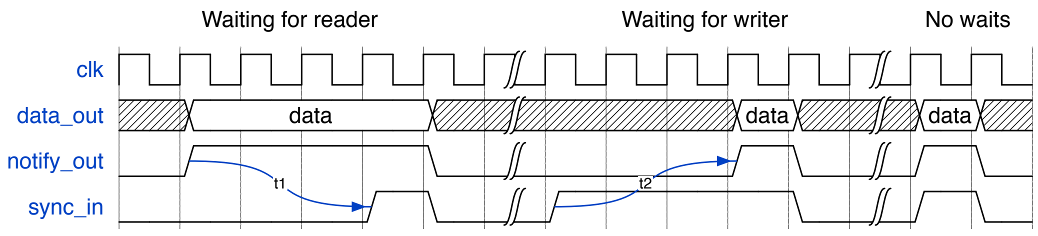


Figure 5.11: Blocking communication timing diagram from writer's perspective for various scenarios

As mentioned in Sec. 4.4, the blocking interface also supports non-blocking read and non-blocking write. If the designer uses non-blocking methods the module does not block in case that the counterpart is not ready for transaction. These methods can only be used when message loss is accepted.

Timing diagrams of different combinations of blocking and non-blocking communications are shown in Fig. 5.12. Transitions marked with arrows t_3 and t_4 show waits that occur when blocking read and blocking write is used. The success signal is asserted when the transaction was successful. This signal is added to the waveform for demonstration purposes. At the ESL, success is given when the respective thread continues execution.

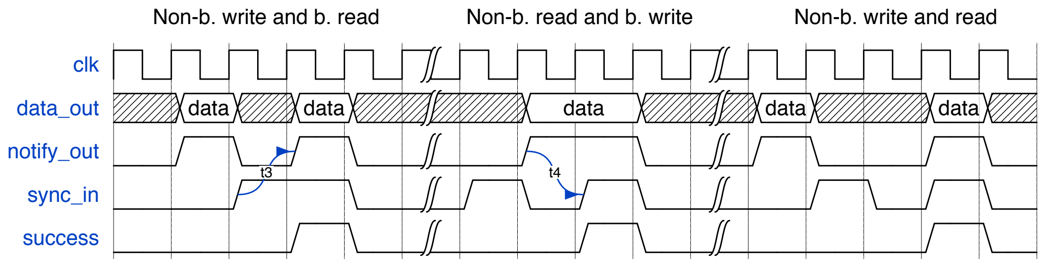


Figure 5.12: Blocking communication timing diagram from writer's perspective for various scenarios with non-blocking read and write usage (when message loss is accepted)

We can see that the `notify` signals in the generated RTL implementation are deasserted one clock cycle after both notify signals were asserted. Unlike in four-phase handshake communication, there is no blocking wait until the counterpart module deasserts its `notify` signal. All manually written RTL modules have to follow this rule in order to be compatible with the generated ones in case a combined setup is used. Compliance with this rule is checked if the PDD methodology is used for the manual implementations by proving the generated properties on the designs. In practice, the notify signal is usually implemented by a one-bit port. However, the designer is not required to implement a distinct port. In Sec. 5.3.4, we provide other options to specify the notify signal.

MasterSlave

The MasterSlave is a simplified version of the blocking interface. The interface requires the slave to be always ready to communicate. Fig. 5.13 shows the structure of this interface.

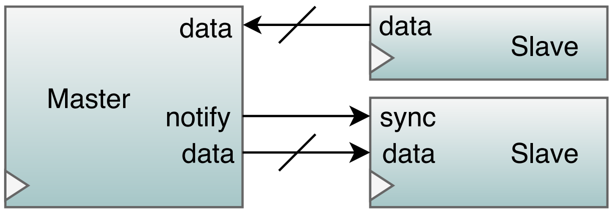


Figure 5.13: Connection between master and slave modules

Due to the fact, that the slave is required to be always ready no notify signal is required for the slave. The master module, on the other hand, communicates only when there is a demand. For this reason, it has a `notify` signal to inform the slave module about the intended transaction. Fig. 5.14 shows the timing diagram of the MasterSlave communication.

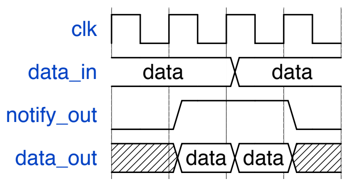


Figure 5.14: MasterSlave communication timing diagram from the master's perspective

Shared

The shared interface does not require any handshaking between modules, and therefore, it does not introduce any wait states. The shared interface results in the generated RTL code in an internal variable that is forwarded to the outside of the module so that other modules can access it. The connection between modules of the shared interface can be seen in Fig. 5.15 and the timing diagram in Fig. 5.16.

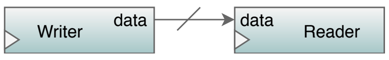


Figure 5.15: Connection between two modules using shared interface

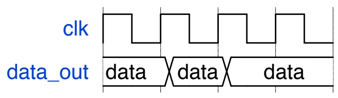


Figure 5.16: Timing diagram of a communication over shared interface from writer's perspective

5.3.3 RTL Skeleton

As a starting point, the PDD methodology requires, at the least, a minimal RTL description that provides structural information about the implementation, such as the ports and the needed datapath registers. We call it *skeleton* in the sequel. However, a behavioral description is not needed for the first iteration of PDD. The designer can either use the skeleton provided by DeSCAM or create a custom one. It is even possible to start with a pre-existing design and then only edit the properties for refinement. This is a promising solution for dealing with legacy design code, as will be shown in Sec. 7.2.

In the following, we show how to create an RTL implementation for the PPA example of Fig. 3.2. We begin with the skeleton as it is generated by DeSCAM and show how the corresponding properties are refined. We explain refinement-of-macro definitions at the example of the macro generated for port *b_in*. We show how to implement RTL code at the example of the reset operation. Fig. 5.17 shows the RTL skeleton generated by *DESCAM*. A VHDL package declaring the needed data types is generated alongside the skeleton. The shown code is simplified for demonstration purposes. The full example is available on GitHub.

The skeleton initially declares a port *b_in* (line 5) that transports a message of type *msg_t*. It consists of a 32-bit integer for *msg.data* and a 1-bit *boolean* for *msg.status* (cf. Fig. 3.2). In practice, a system-level data object can usually not be simply "copied" to the RTL, but needs to be refined to a low-level representation according to specific data formats and communication protocols. In our example, the 33-bit wide input port data, as generated by DeSCAM, is represented on the RTL as a serial bit stream received in 33 beats over a 1-bit input. In Fig. 5.17 we demonstrate a manual design decision. A 1-bit input port called *data_in* (line 8) is added and *b_in* (line 5) is removed. The untimed word-level exchange of a message at the system level is transformed into a cycle- and bit-accurate exchange. DeSCAM generates two macros for the port *b_in*: *b_in_sig_status* and *b_in_sig_data*. In the following, we describe how these macros are refined for this specific protocol.

```
entity Example is
port(
    clk: in std_logic;
    rst: in std_logic;
    - - b_in: in msg_t;
    b_in_sync: in bool;
    b_in_notify: out bool;
    data_in: in signed(0 downto 0);
    m_out: out int;
    m_out_notify: out bool;
    s_out: out bool )
end Example;

architecture Example_arch of Example is
    signal section: Example_SECTIONS;
    signal cnt_signal:int;
    signal msg_signal:msg_t;
begin
    control: process(clk)
        if (clk='1' and clk'event) then
            if rst = '1' then
                section <=idle;
                cnt_signal <= 0;
                s_out <= false;
                b_in_notify <= true;
                m_out_notify <=false;
                msg_signal.data <= to_signed(0,32);
                msg_signal.status <= in_frame;
            else
                - - Implement the control behavior
            end if;
        end if;
    end process

end Example_arch;
```

Figure 5.17: RTL implementation

5.3.4 Refinement

Fig. 5.18 illustrates how the macro for the datapath *b_in_sig_data* is refined in order to implement the protocol. The property checker proves the property for an arbitrary starting state (shown by the arbitrary time point t). The other time points referred to in the property are finite offsets from t, i.e., the property covers a finite time interval of behavior.

```
1: macro b_in_sig_data : int :=
2:     data_in & next(data_in,1) & next(data_in,2) & next(data_in,3)
3:     & next(data_in,4) & ... & next(data_in,31)
4: end macro
```

Figure 5.18: Refinement of macro *b_in_sig_data*

The first bit is received at timepoint *t*, the last bit at timepoint *t+31*. The macro describes the reception of this serially transmitted message. The method *prev(signal, n)* provides the value of *signal* *n* cycles before *t* and the method *next(signal, n)* returns the signal value at *n* cycles after *t*. Lines 2–3 describe the sequential behavior using the *next* function. The protocol for the refinement of *b_in_sig_status* is the following:

- The abstract bit is evaluated over the last four input bits of *data_in*.
- If the sequence is equal to *'1111'*, then the status bit evaluates to *in_frame* and otherwise to *oof_frame*.

Fig. 5.19 shows the resulting refinement and Fig. 5.18 describes a 4-bit integer composed of the value of the last four cycles. In this case, the RTL evaluates four bits to determine the status, whereas at the system level only one bit is required. The helper function *frame_detected* evaluates this integer and returns the required value.

```
macro b_in_sig_status: int :=
            frame_detected( prev(data_in,4) & prev(data_in,3)
            & prev(data_in,2) & prev(data_in,1) )
end macro
```

Figure 5.19: Refinement of macro *b_in_sig_status*

Refining the important states boils down to specifying which state bits of the global state vector describe the important states. In general, the designer is free to describe the important states to his/her convenience.

In our experiments two approaches showed the best results:

- *Output-based refinement:* The *notify* macros are used to describe the current state. Each important state implements a communication and thus the respective *notify* signals are set and unset, involving setting and resetting of the associated *notify* signals. This enables describing the important state with a one-hot encoding of the *notify* signals. If there are multiple system-level communication calls to the same port the encoding is extended by additional conditions in order to distinguish these calls.
- *State-based refinement:* The important state is described depending only on internal state variables.

The state-based refinement is explained by an example for *idle_3*. The presented approach works well if the designer adheres to the best-practice of reserving separate sections for each communication, as detailed in the user manual available on GitHub. The initial skeleton generated by DeSCAM provides a *section* signal which can be used to identify the important states. We utilize this section signal and refine macro *idle_3* to *section==idle*. Instead of such state-based refinement, users may also use *output-based refinement*, as described above. In this case, the macro *idle_3* could, for example, specify *b_in_notify==true* and *cnt==0*. As explained above, it is not sufficient to use the *notify* flag of port *b_in*, because state *frame_data_0* may also result from a communication through port *b_in*. Using the *notify* flag again would result in an equivalent refinement for both states which is prohibited. Macros must describe states uniquely. In this example, an additional condition is needed to resolve the ambiguity. Here, we use the datapath register *cnt*, which is known to be zero in state *idle_3* and not zero in *frame_data_0*.

5.3.5 Implementation

In this section we describe how PDD is used to progressively implement designs, property by property. The process always starts with the reset property. Fig. 5.20 shows the reset property of our example. Line 3 calls a macro named *reset_sequence*. As the name suggests, the macro defines the sequence of signal values that reset the circuit. In the simplest case, the macro describes an assertion of a signal called *reset* or the like. (The macro *reset_sequence* is not part of the PDD methodology. We use it here only to hide the details of circuit initialization.)

```
 1: property reset is
 2: assume:
 3:     reset_sequence;
 4: prove:
 5:     - - output sequence
 6:     at t:     cnt = 0;
 7:     at t:     msg_data = 0;
 8:     at t:     msg_status = in_frame;
 9:     at t:     s_out_sig = false;
10:     at t:     m_out_notify = false;
12:     at t:     b_in_notify = true;
13:     - - ending states
14:     at t:     idle_3;
15: end property;
```

Figure 5.20: Reset operation

The property specifies that when the reset is triggered the design has to fulfill the commitments in lines 5 to 14. The datapath registers must be initialized correctly (lines 6 to 8). The output *s_out* is required to become *false*. As mentioned in Sec. 5.3.1 the properties ensure a correct handshaking. The reset property proves that after reset the hardware is in state *idle_3*. This state has been generated for the communication through port *b_in*. The read from this port is initiated by asserting its *notify* signal. The port *m_out* is not used and thus *m_out_notify* evaluates to *false*. Fig. 5.17 contains the implementation of the reset operation between lines 22 and 28. Now, after the reset operation has been implemented, the PDD process continues with the operations starting in important state *idle_3*.

```
 1: property idle_3_read_5 is
 2: for timepoints:
 3:     t_end = t+32;
 4: freeze:
 5:     b_in_sig_data_at_t=b_in_sig_data@t,
 6:     b_in_sig_status_at_t=b_in_sig_status@t;
 7: assume:
 8:     - - starting states
 9:     at t:    idle_3;
10:     - - trigger sequence
11:     at t:    b_in_sig_status=in_frame;
12:     at t:    b_in_sync;
13:     at t:    cnt==0;
14: prove:
15:     - - output sequence
16:     t_end: cnt = 3;
17:     t_end: m_out_sig = 3;
18:     t_end: msg_data=b_in_sig_data_at_t;
19:     t_end: msg_status=b_in_sig_status_at_t;
20:     t_end: s_out_sig = true;
21:     during[t+1, t_end]: b_in_notify=false;
22:     during[t+1, t_end-1]: m_out_notify=false;
23:     t_end: m_out_notify = true;
24:     - - ending states
25:     t_end: frame_start_2;
26: end property;
```

Figure 5.21: Regular operation

In our example, we continue with the operation leading from *idle_3* to *frame_start_2*, as described in lines 17 to 24 in Fig. 3.2. The corresponding operation property is shown in Fig. 5.21. There are two new ITL language features in this property (line 2 and line 4) that were not discussed so far. The first feature, *for timepoints*, is used to define the length of an operation by providing a value for the timepoint *t_end* at line 3. Note that the minimum length of an operation is one clock cycle. In case *t_end* is not defined, DeSCAM generates the property with the default value *t+1*. The property specifies that the operation receives a message at port *b_in* and modifies the datapath registers accordingly. The timepoint *t_end* of macro *msg.data* is changed to *t+32*. The operation has a length of 32 cycles, because it takes 32 cycles to receive the 32 serial bits. For example, the evaluation of macro *b_in_sig_data* at *t_end* results in the values from *t_end* to *t_end+32*.

The second ITL language feature, *freeze* (line 4), allows to associate the value of an expression at a specific timepoint *t* with a name so that it can be referenced later. For example, the "freeze variable" *b_in_sig_data_at_t* is assigned the evaluation of macro *b_in_sig_data* (the value of the received message data) at timepoint *t*. The property verifies (line 18) that the datapath register *msg_data* stores the message, received at timepoint *t*, at timepoint *t_end*, correctly. The property ensures the correct handshaking by checking that the associated *notify* flags only change value at the end of the operation, proven by line 21 and line 22. The important state entered after receiving the message is *frame_start_2*. This state results from the port *m_out*. The macro of this port has to evaluate to *3* (line 17) and the according *notify* is set (line 23).

The designer is entirely free on how to implement this operation. The provided example on GitHub is one possible implementation. Theoretically, there is an infinite number of sound refinements for the same PPA. If all properties hold on the design it is guaranteed with mathematical rigor that the implementation is sound w.r.t. the PPA and thereby sound w.r.t. the SystemC-PPA. It is, however, important that the designer is only allowed to change the length of the operation and the bodies of the macros. The property descriptions calling the macros must remain as generated by *DeSCAM*.

5.4 Programming view of the Abstract Model

In this section, we explain the idea of the *Abstract Model* (AM) in more detail. The AM is designed to capture the behavior of ESL models, as introduced in Sec. 2.1.1. The data structure stores the results of the first step of the DeSCAM approach, as presented in Sec. 5.1. DeSCAM strips away irrelevant model detail, such as the SystemC scheduler, which are only required in other applications, e.g., C++ analysis or SystemC simulation. The AM stores structural information and behavioral information in the form of an abstract syntax tree (AST) [54].

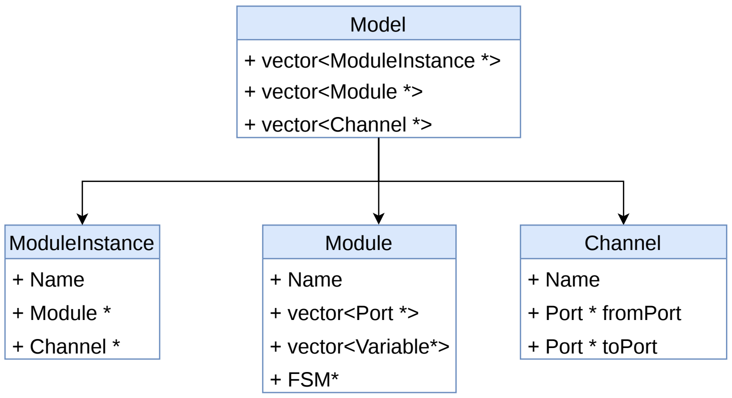


Figure 5.22: Overview of the structure of a Model

Fig. 5.22 shows a UML-like overview of the AM. Each node of the tree represents a `C++` class. An arrow indicates that the predecessor node contains a reference to the successor node. A $*$ succeeding the name of a class indicates that the predecessor can contain multiple references to instances of this class.

The root class of the AM is called *Model*; it contains three elements: *Module*, *ModuleInstance* and *Channel*. The *Module* class describes a module with its structural and behavioral information. A *ModuleInstance* is an instance of a Module. It contains a reference to the Module it instantiates, the name of the instance and a reference to the respective channel for each port. The *Channel* class implements the idea of a channel as discussed in Sec. 2.2.1 and stores a connection between an output port of a ModuleInstance and the respective input port. This information is extracted automatically from the SystemC simulation model by DeSCAM. The behavior of the entire Model results from the the product of the FSMs of each ModuleInstance and the connections between the instances. Figure 5.23 shows how a model is instantiated in SystemC, by the example

```
 0:  main(){
 1:  Master master("master");
 2:  Slave slave0("slave0");
 3:  Slave slave1("slave1");
 4:  Interconnect interconnect("interconnect");
 5:
 6:  //Connecting Master and Interconnect through two channels
 7:  Blocking<interconnect_req_t> MasterToBus("MasterToBus");
 8:  master.interconnect_req(MasterToBus);
 9:  interconnect.master_in(MasterToBus);
10:  Blocking<interconnect_resp_t> MasterAgenToMaster("BusToMaster");
11:  interconnect.master_out(MasterAgenToMaster);
12:  master.interconnect_resp(MasterAgenToMaster);
13:
14:  //Connecting Slave0 and Interconnect through two channels
15:  Blocking<interconnect_req_t> BusToSlave0("BusToSlave0");
16:  slave0.interconnect_req(BusToSlave0);
17:  interconnect.slave_out0(BusToSlave0);
18:  Blocking<interconnect_resp_t> SlaveToBus0("SlaveToBus0");
19:  interconnect.slave_in0(SlaveToBus0);
20:  slave0.interconnect_resp(SlaveToBus0);
21:
22:  //Connecting Slave1 and Interconnect through two channels
23:  }
```

Figure 5.23: Instantiation of SystemC-PPA model within the main function

of a bus with single master and two slaves. Currently, DeSCAM does not support structural hierarchy. This means that all components have to be instantiated and connected by channels in the main function of the SystemC model.

Lines 1 to 4 show how four ModuleInstances are instantiated. The first instance – named *master* of the module *Master* – is instantiated in line 1. Lines 2 and 3 instantiate two instances of the module *Slave*, (i.e., *slave0* and *slave1*), respectively. The next step is to connect the ports of each ModuleInstance to their counterpart. The *Master* module has two ports, one request port and one response port, both of type blocking. We instantiate two channels of type blocking in line 7 and line 10. The instance *master* is connected to the instance *interconnect* in line 8 and line 9, and vice versa in line 11 and line 12. The same idea applies for connecting the slaves to the interconnect.

As explained in Sec. 4.3, the soundness of the compositional model relies on a correct implementation of the modules and defined communication schemes between them. This is guaranteed if the modules are implemented using the DeSCAM approach as presented in Sec. 5.1 to Sec. 5.3.5.

In the following we focus on the data structure of the module. This structure is the entry point for the generation of the operation properties. For this step it is useful to store the behavior, originally described by the thread of the module, in such a way that it is possible for it to be efficiently analyzed.

Model of a Module

As Fig 5.24 shows, the *Module* element is composed of ports, variables, functions and the FSM containing the PPA and the CFG. The Module for the example in Fig. 3.2 has three port elements, three variables and no functions. However, the most interesting part of the *Module* is the data structure for the PPA and the CFG. The main difficulty here is to implement the behavior described by the program code.

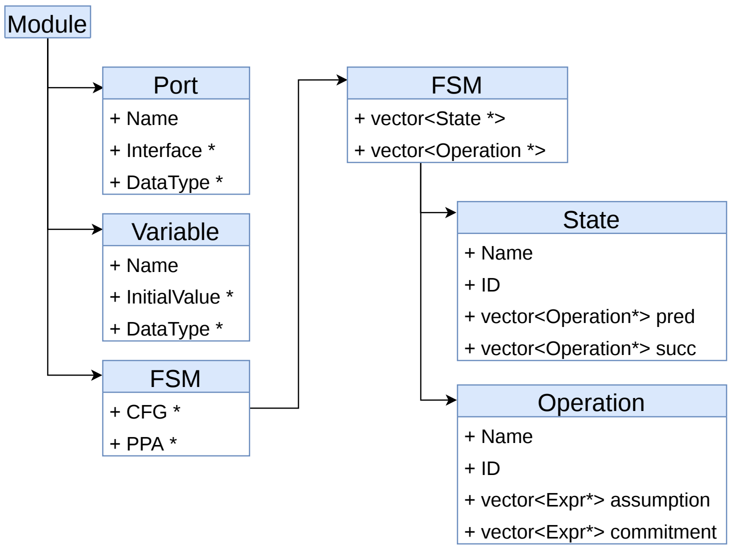


Figure 5.24: Overview of the structure of a Module

Statements and Expressions

In order to do so, we borrowed core ideas from the field of compiler programming. The behavior is described through the code of the SystemC thread. When analyzing the code we distinguish between *statements* and *expressions*. A statement is a complete line of code that performs some action, while an expression is any section of the code that evaluates to a value. Expressions can be combined "horizontally" into larger expressions using operators, while statements can only be combined "vertically" by writing one after another. For example, `var = x + 2;` is a statement composed of the expressions `var` and `x+2`. Expressions can be defined recursively. `x+2` is an arithmetic expression built from the expressions `x` and `2`. The return value of the an arithmetic expression is a signed or unsigned integer number. Another example of an expression is `x==2;` – a logical expression with *Boolean* return type.

In the domain of compilers code is usually represented as an *Abstract Syntax Tree* (AST). The AST is a (directed) tree with variables and constant values as leaf nodes. The root node of an AST can either be a statement or an expression. For example, Figure 5.25 shows the AST for the statement `var = x + 2;`. The root node is the `=` statement. It is composed of a left-hand side expression that is a reference to the variable *var* and the right-hand side being an arithmetic expression. This arithmetic expression `+` is the root node of the subtree defining this expression. Figure 5.26 provides an UML-like

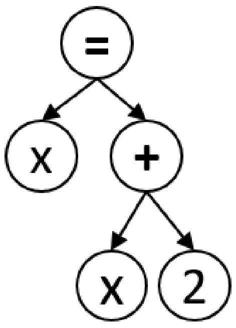


Figure 5.25: Example of an AST for `x=x+2;`

overview of statements and expressions available within our model. There are only three types of statements: *Assignment*, *Branch* and *Read / Write*.

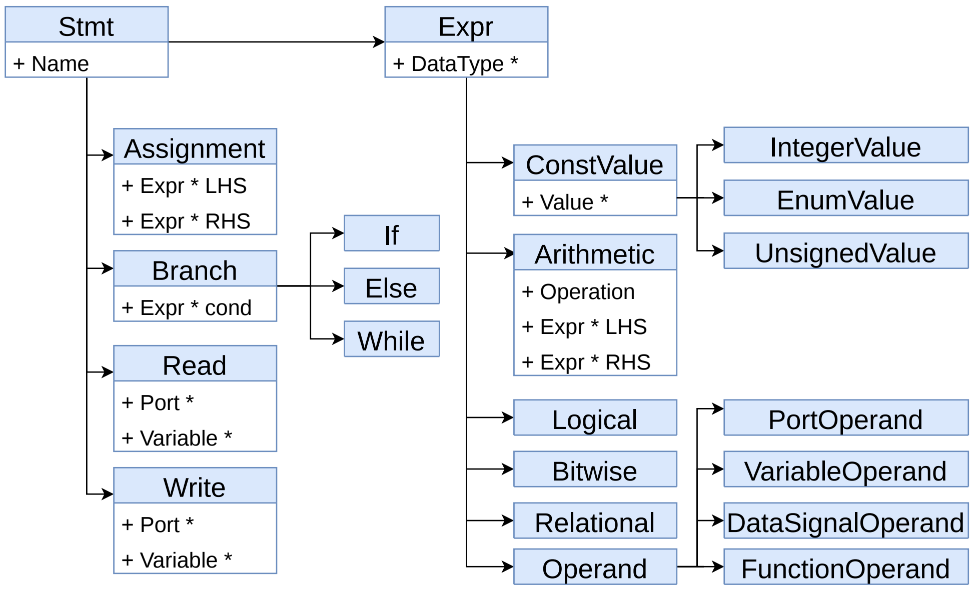


Figure 5.26: Example of an AST for `x=x+2;`

We do not explain all *Expr* in detail; for a more comprehensive discussion we refer the reader to [50]. Statements are composed of structural elements or expressions. For example, an assignment is constructed by two expressions, a left-hand side (LHS) expression representing the variable that is assigned a value and a right-hand side expression (RHS) representing the value to be written. A branch, e.g., `if(x==2))`, has a reference to the expression `(x==2)` representing the condition. This condition is always of type *Boolean*.

In the expression there is the group of binary operations. Each of these operations requires a LHS and RHS expression:

- Arithmetic: `+`, `-`, $\div$, $\times$, `modulo`
- Relational: `==`, `!=`, `<`, `>`, `<=`, `>=`
- Logical: `and`, `or`, `xor`, `nand`, `nor`, `xnor`
- Bitwise: `&`, `|`, `<<`, `>>`, `^`

Lastly, there is a Unary operation requiring only an operator (`not, ++, --`) and an expression.

We also have the notion of operands. If a structural element is referenced (e.g., variable, port) this element is referenced through an *Operand*. This is due to two facts: First the structural definition and the statement tree do not share the same base class. Second, the expression is always required to evaluate to a value. This information is missing in the structural tree. In order to transform the structural element to an expression we encapsulate it in an *Operand*. For example a variable `var` is encapsulated by a *Variable-Operand* element. The same idea applies to ports. However, depending on the interface the port may provide a datapath signal encapsulated by the *DataSignalOperand* or a synchronization signal encapsulated by the *SyncSignal* operand.

Model of the PPA

Fig. 5.24 provides an overview of the data structure of a PPA. As introduced in Sec. 2.3 it is possible to visualize a set of operations as directed cyclic graph. The important states form the nodes of the graph and the operations are the edges of the graph, with the edge starting in the start state and ending in the end state. The graph view of the PPA allows for optimizations, as explained in Sec. 5.5. This is why we also store the PPA in a graph data structure.

The model of the PPA is composed of *States* and *Operations*. The state element stores references to all incoming and outgoing operations. A special state is the reset state, because it has no incoming edges. An operation is composed of a list of assumptions and a list of commitments. The assumptions are expression of *Boolean* return type and the commitments are assignments. When the operations are transformed to operation properties, as explained in Sec. 5.2 the operations are extended by a notion of time. The operation properties are stored in a separate data structure called *PropertySuite*, which is explained in [50]. In the following section we are going to explain why and how the operations are optimized.

5.5 Optimizations

The goal of this section is to explain how redundancies are removed from the operations and how they are made easier to read for a human. The redundancies result from the specific way the operations are generated. For example, the algorithm enumerates all execution paths between two important states to generate the operations. However, not every combination is reachable in the design. Also the computation of the commitments results in redundancies, e.g. an integer variable *var* is increased by *one* three times along the execution path. Without an optimization the commitment results into *var = var+1+1+1* instead of *var = var+3*. For finding optimizations we transform the operations into a representation amenable to SMT solving (cf. Sec. 2.4). In the scope of this work we use the open source solver *z3* [55]. We implemented a translator that allows to translate the statements and expression to z3-SMT instances.

Fig. 5.27 shows how the abstract model, after it has beeen generated, is subjected to optimization procedures. The optimizations are carried out directly on the internal PPA data structure. We introduce three different types of optimizations:

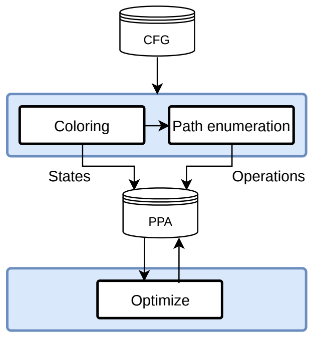


Figure 5.27: Overall model generation flow, repeated here from Fig. 5.4

1. Simplification: The assumptions and commitments of the operations are simplified while preserving the meaning of the operation
2. Redundancy removal: Operations with contradicting assumptions are removed from the PPA
3. Unreachable removal: Operations that are unreachable in the PPA are removed. An operation is unreachable if there exists no trace from reset that fulfills the operation's assumptions.

Candidates for optimizations are detected by formulating properties and proving or disproving them on the PPA. In general, it is possible to translate the property to a regular *Boolean* SAT problem. However, the PPA operates on the word level and translating the problem to a *Boolean* SAT problem removes the high-level view of the PPA and increases proof complexity. As explained in Sec. 2.4, the more efficient way is to use a SMT solver and remain at the word level view of the PPA.

In the following, we explain the core ideas of each optimization on selected examples. In practice, the properties used to find the optimizations become more complex than the ones shown in our examples. However, using SMT allows to perform thousands of checks during the construction of the PPA without them becoming a bottleneck for the performance of the DeSCAM tool.

Simplification

In this step, the properties are simplified with respect to readability and proof complexity. The main goal is to remove redundancy within expressions and statements and thereby increase the readability of the properties and improve the documentation character of the properties. The optimization targets the assumption and the commitment of each property individually. Both are represented as a tree of sub-expressions. The problem of finding a simpler representation of an expression is formulated as an SMT problem. As these expressions are usually not very complex, the corresponding SMT instances are ease to solve with short run times.

In the following we use the example in Fig. 5.28. It contains two communication statements (line 1 and line 8) resulting in two important states. The operations result from possible execution paths between the important states. For creating the operations, the tool iterates through the list of statements between the important states. An *if-then-else* construct (e.g., line 2), creates two operations: One operation represents the

```
port_in→read(var) //important state: read
if(var > 0){
    result = var * 2;
}
if(var > 10){
    result = 0;
}
port_out→write(result); //important state: write
[...]
```

Figure 5.28: Snippet of code for explaining the idea of simplification

then branch and is assigned the positive condition from the ITE statement. The other operation represents the *else* branch and is assigned the negated condition. In our example here, there are four different operations with the assumptions:

$\{$(`var>0` $\wedge$ `var>10`) , (`var>0` $\wedge$ `var<=10`) , (`var<=0` $\wedge$ `var>10`) , (`var<=0` $\wedge$ `var<=10`)$\}$

Consider, now, the execution path: `var>0` $\wedge$ `var>10`. This results in an operation with two assumptions. One may immediately see that having two conditions is not necessary because `var > 10` implies that `var > 0`. In order to interpret the assumptions the tool translates them to clauses. The used SMT solver z3 provides an API that allows to formulate arbitrary logic propositions. The tool composes a formula of the assumptions: `var > 0` $\wedge$ `var > 10`. Afterwards, the solver is used to prove that: `var > 0` $\wedge$ `var > 10` $\implies$ `var > 0`. If successful, the operation is simplified by removing one assumption from the assumption list. This does not affect the validity of the property, because the original and the simplified assumption describe the same invariant.

In the following, we explain how the commitments of an operation can be simplified using *z3*. The code shown in Fig. 5.29 results in a single operation, starting in the important state *read* and ending the important state *write*. The important states results from the communication calls of line 1 and line 5.

```
1: port_in→read(var) //important state: read
2: ++var;
3: ++cnt;
4: result = cnt + var;
5: port_out→write(result); //important state: write
6: [...]
```

Figure 5.29: Code snippet for explaining the idea of redundancy removal

The resulting property is shown in Fig. 5.30. The property has only two assumptions (cf. line 9 and line 10), assuming that the hardware is in the important state *read* and that the counterpart is ready for communication. The property proves that the hardware ends in the correct important state (cf. line 12), that the data path register *cnt* is increased by one (cf. line 13) and that output register is assigned the correct value (cf. line 14). The value of the output register is the sum of the received messages and the value of *cnt* increased by two. The figure shows the optimized version of the addition.

```
 1: property read_to_write is
 2: dependencies: no_reset;
 3: for timepoints:
 4:     t_end = t+1;
 5: freeze:
 6:     cnt_at_t = cnt@t,
 7:     port_in_sig_at_t = port_in_sig@t;
 8: assume:
 9:     at t: read;
10:     at t: port_in_sync;
11: prove:
12:     at t_end: write;
13:     at t_end: cnt = (1 + cnt_at_t);
14:     at t_end: port_out_sig = ((2 + port_in_sig_at_t) + cnt_at_t);
15:     [...]
17: end property;
```

Figure 5.30: Operation property generated for Fig. 5.29

The commitments of an operation are computed during the processing of the statements of the execution path. After processing all statements of an execution path, the tool knows the respective symbolic value of each register at the end of the path. Not using a variable on an execution path results in a commitment verifying that its value remains the same.

In the following, we explain how the resulting value of the output register of the blocking port *port_out* is computed by walking through the statements of the execution path of Fig 5.29 backwards. The outgoing message from line 5 is defined by the value of the temporal variable *result*. This value results from the sum of the current values of the variables *cnt* and *var* (cf. line 4). The value of variable *cnt* is defined by line 3 that is the value at the beginning of the operation increased by one. The variable *var* holds the value of the message received at the beginning of the operation in line 1 increased by one in line 2. Without simplification, the commitment to the output register in line 14 would be of the form: `cnt+1+port_in_sig+1`. We simplify this statement to: `cnt+port_in_sig+2`. This technique unleashes its full potential if there are more complicated data path operations with many constants.

Redundancy removal

In order to explain the idea of redundancy removal, we consider the example from Fig. 5.28 again. As stated above, this piece of code results in four possible operations. The assumption {`var <= 0` $\wedge$ `var > 10`} of one of the generated operations consists of contradicting conditional expressions. We translate the assumption into an SMT instance: `var < 0` $\wedge$ `var > 10`. The solver finds this expression to be unsatisfiable and the property is, consequently, removed from the PPA. The remaining three operations have the following, optimized, assumptions: {`var > 0`} or {`var > 0` $\wedge$ `var <= 10` } or {`var <= 10` }.

In this example, the advantage of SMT solving over pure Boolean SAT becomes apparent. If we had to simplify the above assumption expressions using a regular SAT solver, run times could easily explode because variables would have to be "bit-blasted" to 32-bit

Boolean representations, leading to large SAT problems and long proof times.

Removal of unreachable operations

In the last step, we remove unreachable operations. Fig. 5.31 shows a piece of code resulting in an unreachable operation. Although this example seems contrived, in practice, unreachable operations may naturally occur due to the abstraction of PPA, and may result in unnecessarily complex sets of properties. In general, the generated properties serve as an implementation guide for the hardware designer. However, if a generated property describes an unreachable operation, the designer is burdened with the extra effort of detecting this and removing the property from the property suite.

```
1: port_in→read(var) //important state: start
2: check = 0;
3: port_in→read(var) //important state: read
4: if(check == 0){
5:     ++check;
6: } else --check;
7: port_out→write(check); //important state: write
9: [...]
```

Figure 5.31: SystemC-PPA resulting in unreachable operations

Applying the basic coloring idea, as presented in Sec. 4.1, has the effect that information from predecessor operations does not transfer to the successor operation. The successor operation can have assumptions that are unsatisfiable given the commitment of the predecessor operation. DeSCAM propagates, depending on the desired optimization level, information over multiple operations and removes operations that are unreachable.

The example in Fig. 5.31 results in three operations, one starting in *start* (cf. line 1) ending in *read* (cf. line 3) and two starting in *read* and ending *write* (cf. line 5). The operation starting in *start* proves that the variable *check* is equal to zero at the ending state *read*. However, this information is not available to the subsequent operations. The operation for the path `check != 0` is actually unreachable in the design. The optimization *Redundancy removal* does not remove this operation, because the assumptions are, by themselves, satisfiable.

The issue is resolved by building an extended proposition of the assumption of the current operation and the commitments of predecessor operations. Let us consider the operation starting from important state read. In order to know whether the assumptions are satisfiable, we add the commitment of the predecessor operation as condition to the assumption list. This constructs the following propositions: `check == 0` $\wedge$ `check == 0` and `check == 0` $\wedge$ `check != 0`. The solver will find that the second expression is not satisfiable. This allows us to remove the operation from the PPA.

This approach is only a heuristic and not every unreachable operation can be found. However, it is a conservative approach because an operation is only removed if it is found to be truly unreachable. Transforming this optimization to an exhaustive approach would require to implement an algorithm that finds all sequences of operations starting with the reset operation and ending in the operation being tested for reachability. Then, for each sequence, a property is formulated verifying that if the assumptions of all predecessor operations are fulfilled, the operation under test cannot be triggered (i.e., its assumption cannot be fulfilled). A counterexample to this property is a witness for reachability. Since for each possible sequence such a SAT-based property check is needed, finding all unreachable operations in this way exhaustively is computationally infeasible. However, in our experiments, we found that considering only the predecessor of the operation under test is sufficient to find most of the unreachable operations. To improve these results it is possible to consider predecessors with a distance >1.

Chapter 6

Extension for Pipelining

In this chapter we describe an extension to the PDD approach for pipelined designs. We show that the extended approach guides the design process by helping to resolve data/structural hazards while preserving benefits of the regular PDD approach. In Sec. 6.1 we explain why it is necessary to have a special approach for pipelining. We show in Sec. 6.2 how this problem is solved by combining a specialized property generation approach with a simplified S^2QED technique. Lastly, in Sec. 6.3, we explain the details of this approach and present results based on two examples to show the feasibility.

6.1 The need for an extension

The ESL, respectively the SystemC-PPA model, is a sequential model. It describes transitions between important states of the design as operations. The operations extracted from the ESL are used to generate formal properties proving that the RTL is sound w.r.t. the ESL. An operation property describing an algorithm may span multiple clock cycles if the algorithm is implemented with pipelining or re-timing registers. Otherwise, the algorithm is executed in one clock cycle.

For example, implementing a multiplication at the system level requires one statement. At the RTL such an operation may be implemented as a combinational logic block (no pipelining: *t_end == t+1*) or with sequential steps (pipelining: *t_end == t+n*). The property proves in both cases that the multiplication is implemented correctly, independent of the chosen implementation.

The step-wise execution of an algorithm in more than one clock cycle is, in the context of this work, called *non-overlap pipelining*, because only one operation is active at the same time in the design and the executions of the individual operations never overlap with each other. This means that there is no concurrent behavior in the implemented design that is not covered by the PPA of the ESL. However, in practice, the executions of the operations overlap with each other in time to increase the data throughput of a design or to enable resource sharing. This form of pipeling is denoted as *overlap pipelining*. An overlapping is only possible if there is concurrent behavior in the design.

In the following, we explain why the properties that are generated from the sequential model cannot hold on a design with overlap pipelining and what are possible solutions to this problem within the current PDD flow. We use Fig. 6.1 to demonstrate the difference between non-overlap pipelining and overlap pipelining. Figs. 6.2–6.5 are used to provide an intuition why the sequential properties hold for a non-overlap pipelining design and fail for an overlap pipelining design. Afterwards, we explain how this problem can be solved, in principle, in the existing flow and why we think an extension of this flow is beneficial in case of overlap pipelining.

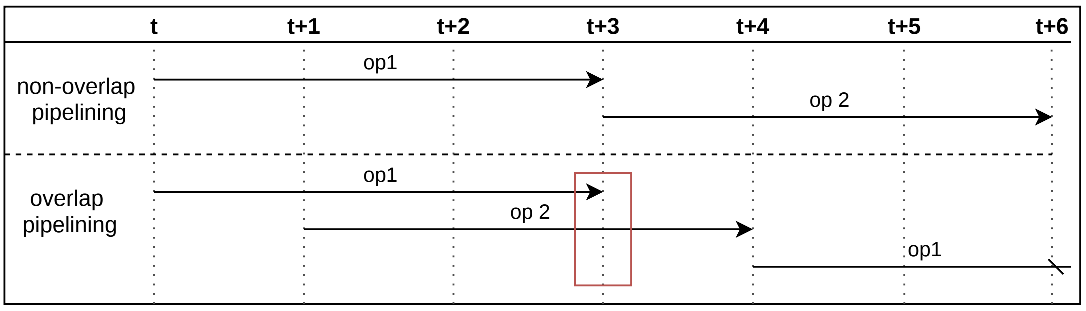


Figure 6.1: non-overlap pipelining vs. overlap pipelining

At the top of Fig. 6.1 is a timeline with abstract timepoints from *t* to *t+6*. The operations *op1* and *op2* each span three clock cycles ($t\_end == t + 3$). The second operation starts when *op1* ends and the important end state (cf. Sec. 4.3) of *op1* is the important start state of *op2*. This sequencing of operations is equivalent to the sequencing of operations at the ESL. The operation property proves that the sequential model is correctly refined into a pipelined hardware.

The bottom of the figure shows the sequencing in case of overlap pipelining. The operation describes the same behavior as in non-overlap pipelining except that the second operation starts at timepoint *t+1* instead of *t+3*. The important start state of *op2* is not the important end state of *op1*. The commitments of *op1* and *op2* overlap at timepoint $t + 3$ as indicated by the red box.

If the concurrent execution of the operations is not described by the ESL, the PPA generated from the ESL does not match the PPA implemented by the RTL. The RTL does not fulfill the system-level specification and the two models are not sound w.r.t. the same PPA. To demonstrate this issue we provide an RTL design as shown in Fig. 6.2. The design has one blocking input port with the respective synchronization signals and one blocking output port. Fig. 6.3 shows the PPA extracted from the sequential ESL. The ESL specifies the following behavior: In state A the design waits for a new message to arrive. Upon arrival *op1* is triggered and the design transitions to state B. In state B the design waits for a message to be transmitted. The transmission is successful if *op2* is triggered.

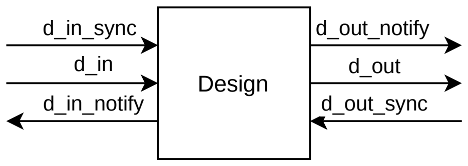


Figure 6.2: Structural view of the example

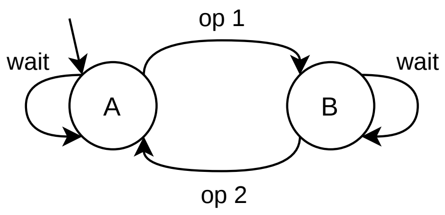


Figure 6.3: Sequential PPA of Fig. 6.2

The properties in Fig 6.4 and Fig 6.5 describe the extracted behavior. They fulfill the timing of the non-overlap pipelining case in Fig. 6.1. If *op1* is triggered the value at the input is multiplied with an internal counter. At the end of the operation, a handshake is offered (*d_out_notify* == *true*) and the output port is set to the computed value. For reasons of simplicity we do not explicitly demonstrate the respective ESL implementation and the wait operation properties.

In order to ensure soundness, the property needs to prove all outputs at all times as well as the correct value of all visible registers. For example, op1 proves that the internal counter does not change its value (see line 14) and op2 proves that the counter is correctly increased by 1 (see line 13).

```
 1: property op1 is
 2: assume:
 3:     - - starting states
 4:     at t+0: state == A;
 5:     - - trigger sequence
 6:     at t+0: d_in_sync == true;
 7: prove:
 8:     - - output sequence
 9:     during[t+1,t+2]: d_out_notify == false;
10:     at t+3: d_out_notify == true;
11:     at t+3: d_out == d_in*prev(cnt, 3);
12:     during[t+1,t+3]: d_in_notify == false;
13:     - - ending states
14:     at t+3: cnt == prev(cnt, 3);
15:     at t+3: state == B;
16: end property;
```

Figure 6.4: Operation property op1

```
 1: property op2 is
 2: assume:
 3:     - - starting states
 4:     at t+0: state == B;
 5:     - - trigger sequence
 6:     at t+0: d_out_sync == true;
 7: prove:
 8:     - - output sequence
 9:     during[t+1,t+3]: d_out_notify == false;
10:     during[t+1,t+2]: d_in_notify == false;
11:     t+3: d_in_notify == false;
12:     - - ending states
13:     at t+3: cnt == prev(cnt, 3)+1;
14:     at t+3: state == A;
15: end property;
```

Figure 6.5: Operation property op2

In lines 9 and 10 of Fig. 6.4, the property verifies that the notify of the output port remains low until the end of the operation and that the value is set to the correct value in line 11. This results directly from the sequential model. The input port is not used after the message is received and the *d_out_notify* is deasserted for timepoints $t + 1$ to $t + 3$ (see line 12). Proving a correct value of the notify signals at all times is a requirement for soundness. For example, removing line 12 could lead to a design that requests a new data message without storing it. Such an RTL model would drop messages and would not be a sound refinement.

Let us consider the overlap pipelining example of Fig. 6.1, with the behavior of the operations as specified in Figs. 6.4 and 6.5. The pipelining works as follows: If the message is received *op1* starts computing the value of the output. The computation is based on the value of the counter at timepoint *t*. At *t+1* *op2* begins to compute the next value of the counter. At *t+3* the output is set to the correct value and at *t+4* the value of the counter is computed correctly and the design is ready to receive the next instruction.

In a PDD flow the designer must prove all properties on the design. At timepoint $t + 3$ the operations overlap such that *op1* proves that *d_out_notify* is set to *true* and *op2* proves that *d_out_notfiy* is set to *false*. Obviously, there is no design that can fulfill these conditions at the same time. This problem propagates to any sequential specification with an overlap pipelining implementation. For example, consider two operations *ADD* and *LOAD* proving the correct execution of an instruction. The *ADD* instruction does

not access data memory and thus proves that the respective notify is set to *false* for the entire operation. On the other hand, the *LOAD* operation proves the respective notify is set to *true* during memory phase. This results, as in the above example, in contradicting commitments of the properties.
In the current PDD-flow there are two approaches to solve this issue:

1. The designer computes the FSM of the pipelined design that is the product machine of the implemented stages. The ESL description is redesigned such that it implements the pipelining. The downside of this approach is that large parts of the RTL are duplicated at the ESL and the desired abstraction is lost.

2. The designer refines the pipelining within the macros. We used this approach in the results shown in Sec. 7.1. The ESL remains abstract. However, with this approach parts of the RTL are duplicated in the refinement of the macros.

The two above approaches require an additional effort by the designer. The goal of PDD is to aid the designer and accelerate the design process. Especially, the case of overlap pipelining results in an overhead when PDD is used. Furthermore, the current approach does not help the designer to implement the pipeline, especially it does not help to indicate correct resolution of structural and data hazards.

What is the solution? In the remainder of chapter we are going to propose an extension to PDD called “Property Driven Development for Pipelining” (PDD-P). With PDD-P we are going to achieve the following three goals:

1. The ESL remains abstract and sequential.
2. No complex and time consuming refinement of the macros are needed.
3. Hazards are correctly resolved.

6.2 PDD-P explained

In this section we explain how PDD is extended to achieve the goals above. Fig. 6.6 provides a comparison of the current flow, depicted on the left hand side, and the extended flow on the right hand side with the new steps colored in blue.

The first novelty is the new design step *Insert Stages* allowing the designer to add information about the desired pipeline to the ESL model. In practice, a new stage is added to the model by calling `insert_state()` at the respective position in the SystemC-PPA module. From a theoretical point of view a new important state is created for each function call in the PPA. The behavior of the stage is described by the execution path starting from this new important state and ending in the next important state. The details of this step are discussed in Sec. 6.3.1.

The “augmented” SystemC-PPA module is parsed by DeSCAM (see Sec. 5.2). In the PDD flow an operation describes an execution path between two important states. This results in a distinct operation for each stage of the design. Take, for example, an ADD instruction in a processor with a 5-stage pipeline (IF – Instruction Fetch, ID – Instruction Decode, EX – Execute, MEM – Memory access, WB – Writeback). In standard PDD, the instruction execution is decomposed into five operations, one for each pipeline stage.

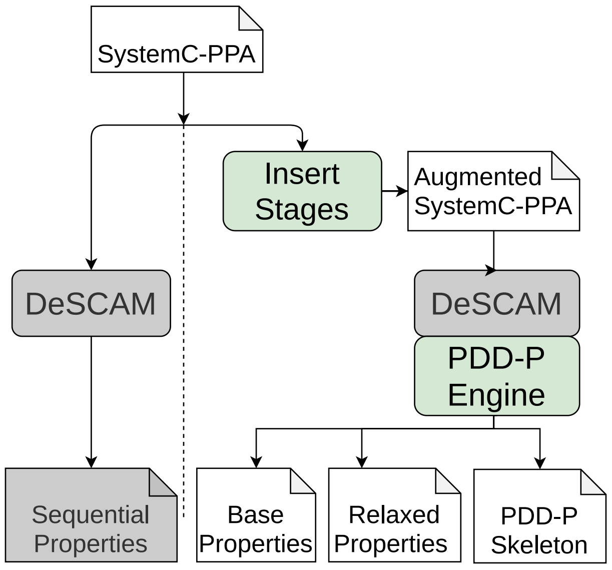


Figure 6.6: Current vs. extended flow

This results in two problems: First, this is not how a designer thinks about the ADD operation. It is more natural to see the ADD instruction as one operation, spanning over all stages. Second, having these *micro operations* results in a large overhead, when signal macros are refined. To solve this issue, we introduce the PDD-P engine to compute *macro operations*. They cover a sequence of micro operations. E.g., the ADD instruction is the macro operation, covering the micro operations for each stage (*IF* to *ID*, *ID* to *EX*, ..., *WB* to *IF*). Each macro operation starts and ends in the same important state (*IF*), but it can cover different sequences of micro operations. The algorithm for computing macro operations is described in full detail in Sec. 6.3.2.

Due to the introduction of macro operations the problem of pipelining is reduced to the problem of non-overlap pipelining versus overlap pipelining as presented in the previous section. The **core idea** to solve this issue is to generate two sets of properties. A *base property* set describing the non-overlap pipelining case of the macro operations and a *relaxed property* set describing the overlap pipelining cases of the design.

The base property set assumes that the pipeline is empty (or flushed) at the beginning of the operation and thereby proves a first execution of an operation in the pipeline. This is equivalent to the behavior of the ESL, because a new operation may only start if the previous one is finished. The base property is convenient for the designer to prove non-pipeline related behavior of the design. However, the base property set is not sufficient to prove soundness w.r.t. the PPA. The case-split test (see Sec. 2.3.4) fails for this set of properties, because the operations do not cover the case that the pipeline is not empty.

Proving these cases is the task of the relaxed property set. The name of this set results from the fact that the assumptions are relaxed compared to the base property set. The initial state of the pipeline is unconstrained and the properties prove that the design produces correct results independent of the state of the pipeline and thereby covers the missing cases from the case-split test.

To prove the relaxed property set we need an S^{2}QED setup, as presented in Sec. 2.5, with two instances of the design. The PDD-P skeleton is generated for the user to automatically create this setup. We will provide more details in Sec. 6.3.2 and Sec. 6.3.3.

Having the two sets of properties requires also a change of the "refine and implement" phase of the PDD flow. It is split into two phases for PDD-P:

1. The goal of the first phase is to implement the sequential behavior part of the design, specified by the base property set. In this phase the designer focuses on a correct functional implementation. There is no need to resolve any hazards, because the properties all assume that their operations are executed starting with an empty pipeline.

2. The goal of the second step is to prove with the relaxed property set that the design behaves correctly if the pipeline is not empty. This requires that all possible hazards are resolved. If a hazard is not resolved correctly the property will fail and the designer has to fix the design.

The implementation is correct if and only if both sets hold on the design. This is due to the fact that the properties are designed such that the combination of these two sets fulfill the completeness criterion, as defined in Sec. 2.3.2. An informal proof is provided in Sec. 6.3.3.

Let us analyze whether PDD-P fulfills the goals defined in the previous section.

1. Abstract and sequential ESL: The benefit of PDD-P is that the ESL design remains a sequential model. The insertion of the stages does not affect the I/O behavior of the model and only requires adding one line of code for each stage to the ESL. However, there is no need to model the product machine of the stages at the ESL.
2. Quick refinement: PDD-P removes the overhead for refining the properties to a pipelined design. For proving the base property only a trivial refinement is needed. The refinement of the relaxed property is based on the base property and thus only requires a trivial refinement, too.
3. Hazard resolution: The relaxed property set only holds if all hazards are correctly resolved. If a hazard is missed by the designer, the property fails and a counter example is provided that can be analyzed to resolve the issue.

6.3 Details of PDD-P

In this section we discuss the details of the extension by analyzing the individual steps of the PDD-P flow. The major novelty of this approach compared to [31] is that PDD-P is not restricted to processors and that the properties are generated from a behavioral description instead of a structural description. Additionally, we are able to discover opportunities for forwarding in pipelines. The S^2QED property is generated for the user by default.

The example in Fig. 6.7 is used in the following section to explain the details of PDD-P. It shows a SystemC-PPA of a RISC-V 4-stage processor. The figure is simplified and contains only operations relevant for explaining the *Insert Stage* step. This ESL

```
 1: SC_MODULE(VCORE ){
 2:         //INSTR Interface
 3:         master_in<unsigned int> instr_in;
 4:         master_out<unsigned int> instr_req;
 5:         //MEM
 6:         blocking_out<MemReq> mem_write;
 7:         shared_in<unsigned>mem_read;
 8:         //Variables [...]
 9:         void fsm() {
10:                 while(true) {
11:                         insert_state("IF"); //Fetch
12:                         instr_in->master_read(instr);
13:                         instr_req->master_write(dec_pc + 4);
14:                         insert_state("ID"); //Instruction decode
15:                         if(getOpCode(instr) == 1) { //add
16:                                 ex_write_reg = true;
17:                                 ex_dest_reg = getDest(instr);
18:                                 ex_result = regfile[getOpA(instr)] + regfile[getOpB(instr)];
19:                         } else if(getOpCode(instr) == 4) { //load
20:                                 ex_write_reg = true;
21:                                 ex_dest_reg = getDest(instr);
22:                                 memRequest.addr = regfile[getOpB(instr)] + getImm(instr);
23:                                 memRequest.write = false;
24:                                 mem_write->write(memRequest, "MEM_LOAD");
25:                                 [...]
26:                         } else if( ...) { //other instructions
27:                         insert_state("EX"); //Execute
28:                         wb_dest = ex_dest_reg;
29:                         wb_result = ex_result;
30:                         wb_write_reg = ex_write_reg;
31:                         insert_state("WB"); //write back
32:                         regfile[0] = (wb_write_reg && wb_dest == 0) ? wb_result : regfile[0];
33:                         [...]
34:                         regfile[7] = (wb_write_reg && wb_dest == 7) ? wb_result : regfile[7];
35: }}}
```

Figure 6.7: Simplified core

description is a sequential description and does not describe any pipeline-related behavior, except for the function calls that indicate the desired pipeline stages.

Lines 11, 14, 27 and 31 demonstrate how to use the `insert_state()` function. The execution of an *add* instruction is demonstrated in lines 15 to 19 and lines 19 to 25 show the execution of a *load* instruction. The main difference between these two instructions is that the *load* instruction implements a blocking communication call in line 24. This results in an important state that is not present for the *add* instruction. The generated PPA is depicted in Fig. 6.9.

Lines 32 to 34 of Fig. 6.7 show how the ternary operator (`cond?true:false`) is used in the context of SystemC-PPA. For example, line 32 states that if the condition is true, `regfile[0]` is assigned the computed value from execute stage, otherwise it keeps the old value. In the following we explain the core ideas of the new steps, provide detail on special cases and limitations and, lastly, we conclude the section with results.

6.3.1 Insert Stage

The main goal of the *Insert Stage* step is to add information about the desired pipeline structure to the ESL. This information is then used in the PDD-P engine, as shown in Sec. 6.3.2, to compute the macro operations. In order to introduce the information of the pipeline the ESL calls `insert_state(‘‘name’’)` at the respective position in the code. Calling this function results in a colored node in the CFG that is not related to a communication. This colored node translates to an important state (see Sec. 5.1). The string “name” allows the user to specify a name for this state instead of using a random name. For example line 14 of Fig. 6.7 introduces an important state named *ID*.

From a control-flow point of view calling the function `insert_state()` splits the current execution path and enforces a step in the simulation as well as in the design. Due to the fact that the step is not related to a communication call, it does not affect the I/O behavior. From a theoretical point of view there is no need for this important state, because it does not break a cycle of uncolored nodes. Not removing this colored node results in a less abstract PPA. In order to explain the main differences between regular PPA and the augmented PPA, we use the following two figures.

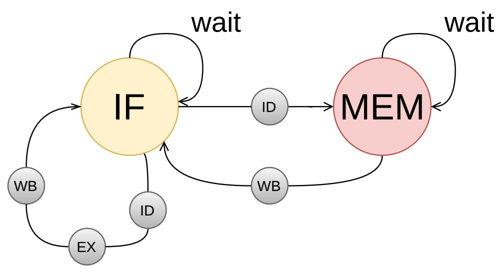


Figure 6.8: Regular PPA from PDD flow

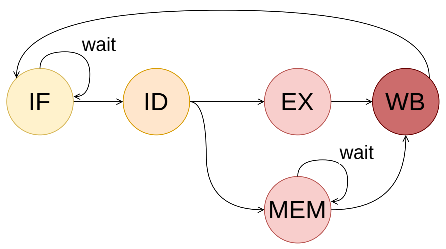


Figure 6.9: Augmented PPA

Fig. 6.8 shows the PPA generated from the simplified core without using `insert_state()`. The grey nodes indicate the stages the operations are traversing. It is only an indication, because introduced stages do not directly appear within the properties. From a PPA point of view, there is no need to color these states, because there is no cyclic path without a colored node. For example, the operation that starts and ends in important state IF and passes through ID, EX and WB describes any operation without access to data memory. This operation is a good example for a non-overlap pipelining operation with a length of three cycles and it covers the entire execution of the instruction, i.e., it is a macro operation.

Any instruction with access to data memory is split into two micro operations (plus a wait operation). The execution starts in important state IF, passes through the ID stage and ends in MEM. From MEM, the operation transition through WB back to IF. Having two micro operations for one instruction is not how a designer thinks about a load or store instruction. Fig. 6.9 shows the resulting PPA for the case that explicit states are inserted. Its easy to see that every grey colored node is now colored. In this case any instruction is described by four micro operations (not including the wait). For this PPA DeSCAM generates a property for each transition between an important state.

6.3.2 PDD-P engine

If the tool is invoked with `--pipelined` the *PDD-P-engine* is enabled. The task of this engine is to compute the macro operations from a given PPA. The algorithm includes three major steps:

1. Find the first important state after the reset, denoted as *root*.
2. Find all cyclic paths starting and ending in *root*, each such path describes a cyclic sub-graph of the PPA.
3. Translate cycles to macro operations: Every operation in the cycle, respectively every transition between two important states, is considered as a micro operation. The micro operations are merged into a single macro operation covering the entire cycle.

We use Fig. 6.9 to explain the individual steps of the algorithm and to point out possible limitations of the current PDD-P engine.

Step 1

The first step is to find the *root* node within the PPA that is used as a starting point for finding the cycles. The current engine assumes that the *root* is the first important state from reset. This is not necessarily the case and in theory it is possible to use a different node as root.

Step 2

The tool implements a special version of a *depth-first-search* (DFS) algorithm that starts in the root node and traverses the graph until the root node is reached again. In other words, the algorithm finds all paths starting and ending in root. However, if the PPA has cyclic subgraphs other than the ones beginning and ending in root then the algorithm may never terminate. A regular DFS algorithm terminates if a node is visited twice. The implemented one is special because it always tries to find a path to the root and may iterate in the cyclic subgraph infinitely. The solution to this problem is explained in the remainder of this section by means of an example.

Fig. 6.9 shows the augmented PPA. The root node of this PPA is the important state *IF*. The algorithm traverses the graph until *IF* is reached again. By a manual inspection we identify three cyclic paths: two with a finite length (*IF-ID-EX-WB-IF* and *IF-ID-MEM-WB-IF*) and one with an infinite length *IF-ID-MEM-MEM-...-MEM-WB-IF*. The path with the infinite length poses a problem for the algorithm, because it may never reach the root node again.

How can we modify the search such that the algorithm terminates? This depends on the type of the cycle. The first type is a wait operation resulting from a blocking communication. This operation leads to an infinitely long path if the communication is blocked forever. To solve this issue we assume a bounded wait for the input which is added as a constraint to the property. The wait operations are verified implicitly by including them into the macro operation. As explained in Sec. 6.3.4, an induction-based proof is used to close the verification gap resulting from the constraint.

The second type of cyclic sub-graph is not related to a blocking communication. Fig. 6.10 shows an example of a PPA with a cyclic sub-graph composed of the states SLV, DBG and INP. There exists a path *IF*-*ID*-DBG-INP-SLV-...-DBG-INP-SLV-*ID*-*EX*-*WB*-*IF* with an infinite length. In this case it is not sufficient to apply a liveness constraint on an input, because the design remains in this sub-graph until the condition is met and one of the operations marked with *yes* and *no* are triggered. The engine throws an error for this case, because macro operations cannot be generated. Future research should work towards finding solutions for this problem. For now, the standard PDD flow with the complex refinement can be used.

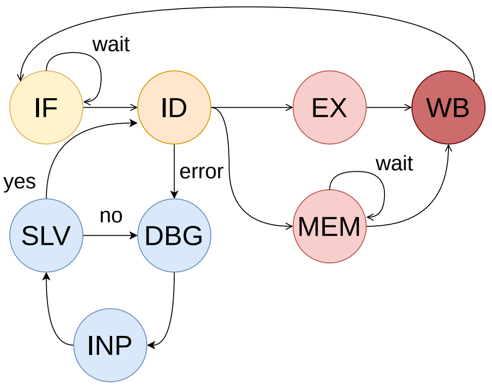


Figure 6.10: PPA with cyclic subgraph

Step 3

The main goal of this step is to compute the macro operations from the micro operations and to generate both the base property and the relaxed property. Each macro operation translates directly into a base property. The generation of the relaxed property set is based on the macro operations as well, but also incorporates ideas from S^{2}QED. The following sections assume a basic understanding of the S^{2}QED approach. For details to this approach we refer to Sec. 2.5 or [31, 30].

Merging the micro operations into a macro operation requires the introduction of multiple timepoints. In regular PDD each operation has only two timepoints *t* and *t_end*. This is not sufficient for macro operations, because not every commitment is proven at *t_end* of the macro operation. Instead, the timepoint is relative to the *t_end* of the micro operation. The engine introduces a timepoint for each micro operation covered by the macro operation. This way, the macro operation ensures the same sequencing of commitments as the micro operations. The same idea applies to the assumption part of the macro operation property.

6.3.3 Properties

In this section, we explain, by means of an example, how a base property and a relaxed property are structured. As stated earlier, we use the relaxed property set to prove the gaps resulting from the base property set. How do we ensure a complete verification?

Idea

The core idea relies on an extended S^2QED approach, called *Complete-S^2QED* (C-S^2QED), presented in [31]. It extends the S^2QED consistency proof with a proof of functionality to achieve complete formal verification. The major contribution of [31] is a mathematical proof stating that it is sufficient to verify functionality correctness on the constrained design, because all other uncovered context related bugs (initially uncovered due to the constraining) are detected by the consistency proof. This means that it is not necessary to prove functionality for the unconstrained instance, resulting in a simplified functional verification.

However, the original approach presented in [31] does not fit well in a top-down design flow due to three reasons:

- The C-S^2QED property is too complex for a top-down design, because it combines two proofs in one property. A failing property does not tell the user if the problem is within the functionality or within consistency.
- The designer has to find possible forwarding cases by himself. This is a manual effort and a source of errors we are able to avoid with PDD-P.
- The properties are only generated from structural description instead of the behavioral descriptions. A structural description cannot be used for design sign-off and is not functionally abstracted.

The idea of having separate base property and relaxed property sets aims to decompose the complete verification problem into verifying functional design behavior and consistency check. This reduces proof complexity, allows more properties to run in parallel and provides for easier debugging. This separation also makes properties more clear and self-documenting, especially the base property set due to the abstracted pipeline behavior. Another benefit, compared to C-S^2QED, is that the base property set is proven only for the DUT and not the S^2QED setup with two instances. The smaller computational model results in a less complex proof of the base property as the results provided in Sec. 7.5 demonstrate.

Base property

An example for a base property is shown in Fig. 6.11. The respective macro operation describes an ADD instruction. This macro operation is derived from one of the cycles of Fig. 6.9. Lines 3 to 7 show the introduced timepoints. They result directly from the important states introduced into SystemC-PPA as shown in Fig. 6.7. In this case the refinement of the timepoints is trivial because the operation is executed from an empty (or flushed) pipeline. There are no delays due to stalling or hazard resolution.

The only state macro refinement required for this approach is the state macro of the root, denoted as *IF_1*, in line 9. The macro is refined such that it relates to the important state *IF_1* of the design. The second macro *empty_pipline* is the constraint assuming an empty pipeline. Line 10 results from the assumption of the micro operation starting in *ID* and ending in *EX*. The assumption is different from the assumptions in PDD, because *t_ID* is not the start of the operation. It relates to timepoint *t+1*. There is no need for

```
 1: property add is
 2: for timepoints:
 3:         t_IF = t,
 4:         t_ID = t+1,
 5:         t_EX = t_ID+1,
 6:         t_WB = t_EX+1,
 7:         t_end = t_WB+1;
 8: assume:
 9:         at t_IF: IF_1 and empty_pipeline;
10:         at t_ID: (getOpCode(instr) = ADD;
11: prove:
12:         at t_ID: dec_pc = (2 + dec_pc@t_IF);
13:         at t_ID: instr = instr_in_sig@t_IF;
14:         at t_ID: instr_req_sig = (4 + dec_pc@t_IF);
15:         at t_EX: ex_op_b = reg_in_sig@t_ID(getOpB(instr@t_ID));
16:         at t_EX: ex_result = (reg@t_ID(getOpA(instr@t_ID)) + reg@t_ID(getOpB(instr@t_ID)));
17:         at t_WB: wb_dest = ex_dest_reg@t_EX;
18:         at t_WB: wb_value = ex_result@t_EX;
19:         at t_end: regfile[0] = (wb_dest@t_WB = 0)? wb_value@t_WB : regfile[0]@t_WB;
20:         at t_end: regfile[1] = (wb_dest@t_WB = 1)? wb_value@t_WB : regfile[1]@t_WB;
21:         [...];
22:         at t_end: regfile[7] = (wb_dest@t_WB = 7)? wb_value@t_WB : regfile[7]@t_WB;
23:         during[t_ID,t_WB]: memory_notify = false;
24: end property;
```

Figure 6.11: Base property for an ADD instruction

an assumption on the important state at *t_ID* because the execution of it is known to be one clock cycle.

The commitments of the property are shown in lines 12 to 23. It appears that there is no commitment concerning the important state. This is not required because it is known that the operation executes *ID* at t+1. If this is not the case one of the commitments related to the stage (lines 12 to 14) fails. Lines 15 to 23 ensure a correct propagation of the control and data signal through the pipeline. The architectural state of the design is updated in lines 20 to 23. Lastly, in line 23 we prove that the notify related to the memory port remains deasserted for the duration of the operation.

Relaxed property

As mentioned earlier, proving the base property is not sufficient for a complete coverage of the design behavior. This is due to the second condition *empty_pipeline* in line 9. The macro includes assumptions on the initial state of the pipeline. There are two possibilities to fulfill the case split test: First, each base property proves at *t_ID* – start of a new operation – that *empty_pipeline* holds. This is not possible because the current instruction is already in the pipeline and, thus, the pipeline is not empty (contradiction to *empty_pipeline* constraint). Second, there is a property assuming the case *not(empty_pipeline)*. Such a property is equivalent to a property that proves all pipelining cases, resulting in the complex refinement we aim to avoid. Instead we use our extended C-S^{2}QED technique and cover this case with the relaxed property set.

We use Fig. 6.12 to explain the structure of a relaxed property. It shows the respective relaxed property for the base property shown in Fig. 6.11. Lines 4 to 14 show two sets of timepoints. The first set is related to the *base instance* and can be directly derived from the base property. The second set (lines 10 to 14) is related to the instance that has no constraint on the initial state of the pipeline. Here, the refinement is not as simple as in the base property case.

```
 1: property S2QED_add is
 2: for timepoints:
 3:         -- constrained instance
 4:         t_if_i1 = t,
 5:         t_id_i1 = t_if_i1+1,
 6:         t_ex_i1 = t_id_i1+1,
 7:         t_wb_i1 = t_ex_i1+1,
 8:         t_end_i1 = t_wb_i1+1,
 9:         -- unconstrained instance
10:         t_if_i2 = t,
11:         t_id_i2 = t_if_i2+1..5 waits_for (cpu2/dmem_enable_o==0 || cpu2/dmem_valid_i),
12:         t_ex_i2 = t_id_i2+1..5 waits_for (cpu2/dmem_enable_o==0 || cpu2/dmem_valid_i),
13:         t_wb_i2 = t_ex_i2+1,
14:         t_end_i2 = t_wb_i2+1;
15: assume:
16:         -- constraints on CPU1
17:         at t_IF: IF_1 and empty_pipeline;
18:         at t_ID: (getOpCode(instr) = resize(1,32));
19:         -- same instruction for IUV
20:         at t: cpu1/imem_addr_o = cpu2/imem_addr_o;
21:         -- I/O should be the same (no stalling)
22:         at t: (cpu2/dmem_enable_o==0 || cpu2/dmem_valid_i);
23:         -- QED consistent registers
24:         at t_wb_i2: cpu2/regfile == regfile@t_wb_i1;
25:         -- Flushing
26:         at t_id_i2: cpu2/ex_mispredict = ex_mispredict@id_i1;
27: prove:
28:         at t_ID: cpu1/imem_addr_o = cpu2/imem_addr_o;
29:         -- general registers
30:         at t_end_i2: cpu1/regs@t_end_i1 = cpu2/regs@t_end_i2;
31: end property;
```

Figure 6.12: Relaxed property for an ADD instruction

As we can see in line 11, *t_id_i2* depends on the evaluation of the condition. If another instruction is blocking the decode stage (e.g., load or store) then the instruction is stalled until the decode stage is free. The design is unrolled for up to 5 clock cycles (indicated by `..5`). This is an overconstraint on the design, because it is assumed that after 5 cycles the decode stage is free. Of course this may not always be the case, e.g., if *dmem_valid_i* remains low forever, the processor is stalled forever. We explain in Sec. 6.3.4 how this problem is solved in PDD-P.

At line 17 we use the assumptions from the base property to assume that instance 1 executes from an empty pipeline state. The assumption at line 18 ensures that the fetched instruction is an ADD instruction. As shown in [31], it is possible to prove the consistency with one S^{2}QED property. However, in a hardware design flow it is more efficient to split the operation in order to ensure ease of debugging. Furthermore, splitting the S^{2}QED properties allows parallelization of the proof. We, therefore, generate one relaxed property for each base property.

The second instance is free on its execution, except of the fact that both instances need to execute the same macro operation. This is ensured by assuming that the inputs and outputs of the designs are equivalent in line 20 and line 22. Intuitively, line 20 means that both designs fetch the same instruction. It is important to remember that PDD-P is not limited to processor designs. The generalization of this assumption is the fact that both instances receive the same inputs and the initial state is equivalent. Every time a design reads an input the two instances are required to read the same value.

Next we are going to focus on the assumption of QED consistency, because this assumption is really important for the entire approach. The base instance executes the instruction starting from an empty pipeline. In case of a processor, this means that the register file does not change until write back stage. However, the relaxed instance is free on its initial state and the register file can change during the execution. If this is the case, a correct forwarding has to be implemented to fulfill the consistency check. This consistency assumption (see line 24) is automatically computed from the base property. Every time a hazard can occur (e.g., write-after-read) a QED consistency assumption is necessary. The fact that the engine is able to compute the timepoint for assuming QED consistency is a major advantage over the approach of [31].

The assumption in line 26 ensures that both instances show the same behavior related to flushing. From an abstract point of view flushing can be described as an abortion of a macro operation. This is usually not described by the sequential model and cannot be computed by the engine. In Sec. 6.3.5, we explain how flushing can be introduced without altering the ESL behavior and thereby fulfilling the goals defined in Sec. 6.1.

In line 28 we prove that both instances fetch the next instruction and in line 30 we prove that the architectural registers are updated correctly. Its important to note that in contrast to [31] we do not prove a specific value for outputs or registers. Instead the proof is reduced to verifying the consistency of the two instances as in classic S^{2}QED. This reduces the complexity of a property and makes debugging easier.

Consider, for example, a design that implements functionality correctly for one macro operation but there is a bug in the forwarding. Then the base property holds on the design but the relaxed property fails. This removes the need to inspect errors related to functionality while debugging. However, just proving the relaxed property is not sufficient, because a functional bug will not be discovered.

6.3.4 Wait Properties

In this section we are going to discuss how a wait operation is modeled in PDD-P. Modeling an infinite wait is a backbone for the sound modeling of abstractions with PPA (see Sec. 4.3.3). The PDD methodology models a wait operation by a special wait property verifying that the design remains in its important state while waiting for the synchronization signal.

In PDD-P, multiple micro operation are merged into one macro operation. As explained in Sec. 6.3.2, the wait operations are not directly included to avoid infinitely long macro operations. A wait operation, in PDD, results from a blocking communication call at the ESL. However, neglecting the wait operation for PDD-P results in a verification gap. For example, a sender transports a message via a blocking port at the ESL. The module is blocked until the message is received by the receiver. It is never lost. Consider the case that there is a bug in the stalling unit of the pipelined RTL implementation and the design aborts sending the message after two clock cycles. The execution continues and the message is lost. This destroys the sound relationship of the ESL and the RTL. To ensure soundness we need to prove the correct behavior for an infinite wait. We propose an induction-based proof. The infinite wait operation is split into an *induction base* and an *induction step*.

The induction base is not a distinct operation; it is already described by the macro operation generated by the PDD-P engine. Each macro operation automatically includes the proof that the the base is reachable. The base case is defined as the respective important state and the corresponding synchronization signals. Additionally, the macro operation verifies a bounded wait with the bound defined by the designer. The induction step proves that if the wait operation is triggered then the design remains in this state. This step function is similar to the original wait operation with the difference that the proof only verifies signals related to the stage rather than the entire design.

Induction base

Fig. 6.13 shows an example for a macro operation covering a blocking communication. This macro operation is computed from the micro operations resulting from the path *IF, ID, MEM, WB* and *IF* of Fig. 6.15. Compared to the macro operation depicted in Fig. 6.11 it has an additional timepoint in line 6, due to the blocking communication. This timepoint is used to prove a bounded wait along with the macro operation.

For proving the induction base we use the timepoint *t_MEM*. In order to prove that we reach the important state we prove the commitments resulting from the micro operation *ID→MEM*. The commitments are shown in lines 13 to 15. By proving these commitments it is verified that the important state is reachable.

```
 1: property mem is
 2: for timepoints:
 3:         t_IF = t,
 4:         t_ID = t+1,
 5:         t_MEM = t_ID+1,
 6:         t_MEM_wait = t_MEM+0 .. BOUND waits_for mem_sync,
 7:         t_WB = t_MEM_wait+1,
 8:         t_end = t_WB+1;
 9: assume:
10:         [...]
11: prove:
12:         [...]
13:         at t_MEM: mem_notify;
14:         at t_MEM: mem_addr = addr;
15:         at t_MEM: mem_data = data;
16:         during[t_MEM+1,t_MEM_wait]: mem_notify;
17:         during[t_MEM+1,t_MEM_wait]: mem_addr = prev(addr);
18:         during[t_MEM+1,t_MEM_wait]: mem_data = prev(data);
19: end property;
```

Figure 6.13: Induction base property for a memory instruction

In theory it is not necessary to prove a bounded wait along with the induction base, because we have the induction step to prove this. In practice, it makes sense to include a bounded wait, because it allows the designer to implement a correct waiting mechanism without the need to run the step property. In most cases, a design does not wait forever and it is sufficient to choose a realistic upper bound for the waiting period.

Line 6 shows a finite wait from zero to $BOUND$ cycles. The value for $BOUND$ is determined by the designer. In most cases, a sufficient value for $BOUND$ is between three and five. In order to avoid the case that *mem_sync* remains low, the user has to add a liveness constraint to the property. This constraint assumes that if *mem_notify* is set then *mem_sync* evaluates to true within the provided bound. This constraint introduces a gap in the verification that is closed by the induction step. Lines 16 to 18 prove the behavior of the design in case the wait operation is triggered. If complexity of the induction base becomes an issue, the designer may reduce the bound to zero and thereby indirectly remove the commitments. These commitments are proven also in the step property as as shown in lines 6 to 8 of Fig. 6.14. However, having these commitments directly indicates to the designer that this operation needs to implement a correct waiting mechanism.

Induction step

The goal of the induction step operation is to prove that if a wait operation is triggered the design remains in its state. Fig. 6.14 shows the step property. The induction step assumes to start from the base (see line 2) and proves that the design remains in its state in the next step (lines 6 to 8). A step is defined as one clock tick.

```
1: property mem_wait is
2: assume:
3:         at t: mem_notify;
4:         at t: not(mem_sync);
5: prove:
6:         at t+1: mem_notify;
7:         at t+1: mem_addr = prev(addr);
8:         at t+1: mem_data = prev(data);
9: end property;
```

Figure 6.14: Induction step for a memory instruction

The assumption in line 3 ensures that the induction step is only triggered if the respective sync signal remains low. As lines 5 to 7 show, the property only proves the signals related to the stage. Other outputs or data path registers are ignored, because they are covered by a different overlapping macro operation. This still results in a complete verification, because any pipeline-related bugs not covered by the these two properties are covered by the relaxed property set.

6.3.5 Flushing

In a pipelined design it is sometimes necessary to cancel the execution of a macro operation. This is called *flushing*. For example, the misprediction of a branch decision, a timeout or an overwriting of an operation may result in an abortion of in-flight operations and a flushing of the pipeline. The sequential ESL usually does not describe the abortion of an operation. Let us consider the execution of a branch instruction at the ESL. The ESL executes in a sequential manner. There is no instruction fetch happening while the branch is executed. Hence, there is no need to describe a flushing. At the RTL, depending on the design decisions, an instruction can be fetched and if the branch is mispredicted the execution of this instruction is aborted.

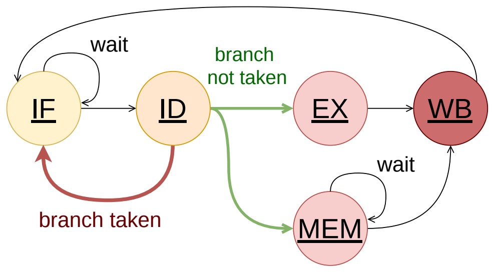


Figure 6.15: PPA that is extended for flushing

How do we augment the SystemC-PPA with this information? The easiest way to describe such a flushing is to introduce a shared port transporting a Boolean message. When the augmented PPA is created the designer decides at which stage an operation can be aborted. The introduced port is read at the respective stage and depending on the evaluation of the message the execution is continued or aborted. In order to not affect the simulation behavior the port is implemented such that it always yields *false*. However, the PDD-P engine creates two operations, one for the case the message evaluates to *true* and one for *false*. At the RTL the generated macro is refined to the internal register (e.g., branch_decision). Depending on the value of the register one of the operations is triggered.

Fig. 6.15 extends the augmented PPA from Fig. 6.9. A new edge (indicated as red arrow) from important state *ID* to *IF* is introduced describing the abortion of an operation. This results in a new cycle and thereby in a new macro operation for the design. Each macro operation now has an assumption at *t_id* either assuming that the value is low or high depending on the fact whether the branch is taken or not.

6.3.6 Limitations

In this section we briefly describe the current limitations of PDD-P:

- PDD-P is limited to designs with a static in-order pipeline. Modeling an out-of-order pipeline remains an open issue. Most likely, multiple communicating PPAs are needed to describe such a pipeline.

- As already discussed, we don't allow cyclic sub-graphs within the PPA. This contributes to a restriction to the approach for general purpose hardware. Communication designs, e.g., a bus, sometimes operate in different modes. This can result in a cyclic subgraph for each mode. A workaround for this is to write the SystemC-PPA such that these cycles do not occur.

- PDD-P is not completely automated. The identification of the empty pipeline state remains a manual task for the designer. Aside from this, PDD-P has the same limitations as the regular S^{2}QED approach. The proof complexity grows non-linearly with the complexity of the computational model. However, due to the compositionality of PPA it is possible to divide the design into smaller subsets with better manageable complexity.

Chapter 7

Experimental Results

The proposed methodology has been evaluated by means of five case studies. Two case studies were conducted in an industrial setting. Four case studies have been conducted in our academic environment so that all related data can be made available in the public domain [50].

All experimental results were obtained on an Intel Core i7 @ 3 GHz with 32 GB of RAM. All property checking experiments were conducted with the commercial property checker OneSpin 360 DV-Certify™.

7.1 Case Study: RISC-V Processors

The first case study comprises multiple implementations of a RISC-V CPU, each being a sound refinement of the same system-level model. The system-level model is a SystemC-PPA-compliant *Instruction Set Simulator* (ISS) implementing the *RV32I Base Integer Instruction Set*, as specified in [56], excluding interrupts. In the sequel we refer to the SystemC-PPA model as ISS. The presented results are obtained with the regular PDD methodology (see Sec. 3) and not the extended methodology for pipelining PDD-P. PDD is elaborated on three different implementations:

- Simple sequential CPU. The design is implemented with two modules: a CPU module and a register file. Datapath computations are described mostly by combinational functions. The goal of this implementation is to provide an RTL design requiring as little refinement effort as possible.

- Complex sequential CPU. The functionality of the processor is divided into four modules. The core is composed of a decoder, an arithmetic logic unit (ALU), a register file and a control unit. All communication between the modules is realized by *master/slave* interfaces. This CPU demonstrates how a complex communication structure implemented at the RTL can be abstracted at the system level.

- Pipelined CPU: The CPU consists of a control unit, a datapath and a register file. The processor is implemented as a five-stage static pipeline with forwarding. The control unit orchestrates the pipelining and sets the respective control signals. The datapath implements the computation and communicates with the register file.

The complete set of properties was generated for the ISS and refined for each implementation, resulting in three different property suites.

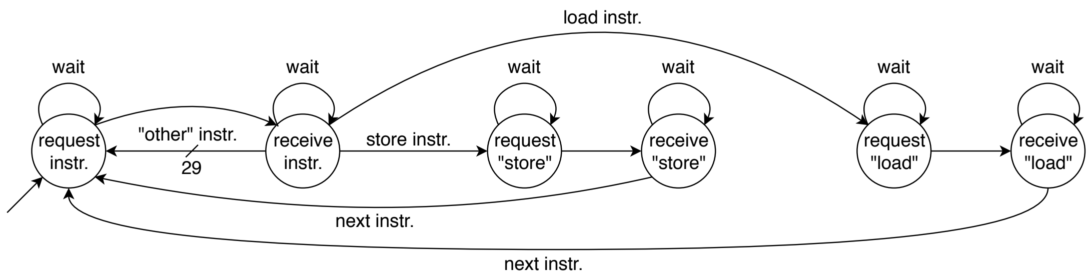


Figure 7.1: PPA of the ISS

Fig. 7.1 shows the PPA of the ISS. The ISS is connected to a memory by a *blocking* output port for sending a new memory request and a *blocking* input port for receiving the response. As we explained in Sec. 4.4, each blocking communication results in an important state. If the handshake fails, the system waits in its current state, represented by the wait operation at each important state. An instruction cycle starts in *request instr*, requesting an instruction from the memory. The ISS transitions to state *receive instr*, if the request is successful. Upon receiving the instruction the execution continues, depending on the decoding of the instruction. Two cases need to be distinguished: In case of a load or store instruction being received the ISS needs to make another communication to memory, resulting in state *request load* for a load instruction and *request store* for a store instruction. The state *request load (store)* is followed by another state *receive load (store)* reading the response from the memory. Execution then continues with fetching the next instruction from the memory. In case of any other type of instruction (e.g., R-type, I-type or B-type) the operation “other instructions”, subsuming the operations for each of these opcode classes, is triggered. The operations verify that the correct values are written into the right registers and that the program counter is set to the correct value. The operation ends in state *request instr* and a new cycle begins.

Fig. 7.2 shows the property generated for R-type instructions. It implements all register-to-register instructions like *add*, *or* and *shift*. The property starts in state *receive instr* (line 9); it is assumed that a new instruction is available (line 12) from the memory. The encode type of the instruction is evaluated by a combinational function *getEncType()*. A combinational function is not allowed to change any state variables of the module. It returns a value as a function of the input parameters. When the operation is triggered it is guaranteed that the program counter is set to the correct value (line 15). Lines 17 to 19 set up the request for the next instruction. During the operation all *notify* flags are set to *false* and the memory is notified that there is a new request by raising the *notify* flag at *t_end*. The result of the computation is returned by the combinational function *getALUresult()* and stored to the register file. Lines 21 to 24 sets up the communication with the register file. In line 16, the property ensures that the hardware does not accidentally initiate a read from the memory.

The refinements for the pipelined processor are, naturally, more complex. The property describes a single instruction, but, due to the pipelining, the processor executes other instructions in parallel. This results in a more complex refinement of properties

```
 1: property property fetch_16_read_5 is is
 2: for timepoints:
 3:     t_end = t+k;
 4: freeze:
 5:     - - Register values in REG_FILE
 6:     pcReg_at_t = pcReg@t;
 7: assume:
 8:                            - - starting states
 9:     at t+0:                receive_instr;
10:                            - - trigger sequence
11:     at t+0:                (getEncType(loadedDataInstr)=ENC_R);
12:     at t+0:                fromMemoryPort_sync;
13: prove:
14:                            - - output sequence
15:     t_end:                 pcReg = (4 + pcReg_at_t)(31 downto 0);
16:     during[t+1, t_end]:    fromMemoryPort_notify = false;
17:     t_end:                 toMemoryPort_sig_addrIn = (4 + pcReg_at_t)(31 downto 0);
18:     t_end:                 toMemoryPort_sig_dataIn = 0;
19:     during[t+1, t_end-1]:  toMemoryPort_notify = false;
20:     t_end:                 toMemoryPort_notify = true;
21:     t_end:                 toRegsPort_sig_dst = getRdAddr(INSTR);
22:     t_end:                 toRegsPort_sig_dstData = getALUresult(INSTR,REGFILE);
23:     during[t+1, t_end-1]:  toRegsPort_notify = false;
24:     t_end:                 toRegsPort_notify = true;
25:     t_end:                 - - Unimportant datapath register ...
26:                            - - ending states
27:     t_end                  request_instr
28: end property;
```

Figure 7.2: R-Type instructions

and design. Take, for example, the refinement of the source registers. In the sequential implementation it is sufficient to refine the macros for the registers by referring to the datapath registers storing the actual values. In the pipelined implementation data hazards can occur, necessitating implementation of a resolution mechanisms such as forwarding. The property macros are refined by referring to the values of the forwarding unit or to the datapath registers, depending on whether a hazard was detected or not.

Lastly, the timing in line 3 is refined by providing a value for k which is different for the different implementations:

- *Simple sequential implementation:* $k = 1$. Each operation is executed in one clock cycle.

- *Complex sequential implementation:* $k = 8$. The timing here depends on the communication between the modules. Each operation has its own distinct timing. The designer has to understand the underlying communication sequences and reflect this in the timing.

- *Pipelined implementation:* This is the most complex case, because the timing of the operation depends on the pipeline state, e.g., a stall due to a data hazard increases the time until an operation finishes. In practice, there are two ways to specify this. One way is to keep the value for k static and let the macros describing the pipelining accommodate for the different pipeline states. The other way is to keep the macros static and reflect the pipeline state through different values for k. In either case, the pipeline state is reflected in the property refinement. The implementation provided in our public-domain online repository uses the first approach.

Table 7.1: Design size and LoC

Lines of code	ISS	Simple Seq.	Complex Seq.	Pipelined
Properties generated	1165	-	-	-
Properties - lines added	-	0	1	411
Properties - lines changed	-	56	68	56
Implementation	1000	1110	1626	2264
Synthesis				
Input/Output	-	36/71	36/71	36/71
Flip-Flops	-	1340	1881	2698

Tab. 7.1 provides the results for the size of the designs and the property suite. A set of 21 properties was generated from the SystemC-PPA of the ISS in less than 25 seconds. The generated properties have a total of 1165 Lines of Code (LoC). In general, only the macro refinement and the timepoints are changed and the assumption and commitment part of the properties remain unchanged. Tab. 7.1 row *"Properties – lines added"* shows how man new lines have to be added during macro refinement and row *"Properties – lines changed"* reports how many of the generated LoC are changed.

The verification effort in PDD results from the time spent on the refinement of the properties during the implementation process. Tab. 7.2 provides manual work efforts, CPU times for formal property checking and simulation times for the system-level model. The second row of the table denotes the design efforts for creating the SystemC model of the ISS and for creating from it the RTL implementations of the different RISC-V implementations. The third row shows the additional manual efforts needed for refining the generated properties during the RTL design process. The reported work efforts were those of first-time users of the methodology.

As can be expected, manual efforts grow as the designs become more complex w.r.t. inter-module communication, timing and pipelining. For example, all operations of the simple processor have a length of one cycle and the design and property refinement starting from the system-level model is nearly trivial. This keeps the work effort for refining the properties under two hours. For the more complex processor versions we exploited the SystemC-PPA communication mechanisms, as discussed in Sec. 4.4, to decompose the ISS model into several SystemC-PPA sub-models that correspond to the different processor modules. The design efforts given in Tab. 7.2 include the efforts for these decomposition steps at the SystemC-PPA level as well as for the creation of the RTL code.

Table 7.2: Design effort and simulation results for different RISC-V implementations

Design and verification results	**ISS**	**Simple Seq.**	**Complex Seq.**	**Pipelined**
Design effort	1 person week	1 person day	4 person days	3 person months
Property refinement effort	—	2 person hours	1 person day	1 person month
Property checking time total	—	2 min	5 min	4:20 h
Longest individual checking time	—	28 s	65 s	1:30 h
Max. memory usage (MB)	—	4003	5220	4628
Simulation time				
Prime numbers (s)	5	16	56	95
Fibonacci (s)	1	4	10	15
Bubble sort (s)	8	35	130	259

The property refinement of the pipelined processor required about 1 person month, due to the complex pipelining. It is important to note that a completed property refinement process implies that all properties hold on the design. Thus, further RTL simulation for verification is not required so that all efforts for traditional simulation and creation of test benches can be avoided.

As shown in Tab. 7.2, proving the actual properties on the design is very fast, especially for the smaller design. Most properties are proven in less than five minutes. The longest proof time was for the R-type instruction property of the pipelined processor. The complexity lies within the datapath operations, which are a worst-case scenario for SAT engines, due to the large state space. A common practice is to blackbox datapath-heavy components (e.g., the ALU) to reduce proof times drastically.

We simulate the designs with three computation-heavy C++ programs, compiled with the RISCV-V R32 toolchain:

- "Prime numbers", computes ten prime numbers starting from n=10000.
- "Fibonacci", computes numbers of the Fibonacci sequence.
- "BubbleSort", sorts an array with 500 integer numbers. Initially, the numbers are sorted in descending order and the algorithm sorts them in ascending order, resulting in the worst-case execution time for BubbleSort.

The results, as given in Tab. 7.2, demonstrate a simulation speedup by simulating the ISS between 4X in case of the simple design and ~32X for the more complex design, compared to the RTL implementations. Simulation of a SystemC model can, obviously, be expected to generally outperform RTL simulation. In case of our PDD methodology the SystemC model, at the abstraction level of an instruction set simulator, has the additional advantage of a sound relationship with the RTL, meaning that the RTL implementation and the ISS execute software in functionally identical ways. This is, to our knowledge, the first time that an ISS can be used as a golden model for design and even for firmware sign-off.

7.2 Case Study: SONET/SDH Framer by Alcatel-Lucent

The second case study is based on an industrial implementation of a SONET/SDH Framer circuit from Alcatel-Lucent. The circuit identifies words of data from communication lines operating in a non-ideal physical environment where, e.g., discrepancies in the clock frequencies of the communicating modules must be compensated for.

In prior work, this design has been completely verified and a system model with two PPAs has been created "bottom-up" according to [2]. In this system, one module describes the main functionality of the Framer, while the other is a monitor collecting performance data in order to determine the synchronization status of the Framer.

The efforts attributed to the bottom-up creation of the two PPAs were about six person months.

Table 7.3: SONET/SDH Framer — original design and redesign

	RTL Design			PPA		
Module	inp./out.	FFs	LoC	inp./out.	var.	states/ops.
Framer (or.)	549/280	4.2k-47k	27k	7/6	4	4/13
Monitor(or.)	20/6	30	850	3/1	2	2/9
Framer (re.)	549/280	3.9k-42k	12k	7/6	4	4/13
Monitor (re.)	20/6	425	92	3/1	2	2/9

Tab. 7.3 shows the numbers for the original design (rows 1 and 2), which contains, depending on the configuration through VHDL generics, between 132 and 549 input bits, 88 and 280 output bits, and 4054 and 47213 state bits. The major part of this large state space results from the buffering of the input stream. The table also shows the numbers for the PPAs of the Framer and the Monitor. Also in this case study a high degree of abstraction is obtained by the PPA models. For example, in this design it is exploited that the correct buffering of the input stream is verified at the RT level and does not need to be represented at the system level.

Starting from the compositional SystemC-PPA descriptions, we redesigned both circuit modules from scratch following the PDD approach. This was accomplished in less than two person months.

The generated property suites cover the behavior of all possible design configurations by VHDL generics. The property suites comprise 22 operation properties. Proving them on all possible configurations of the original design takes 23 min and only 2 min for the redesign. Proving the most complex property took less than 9 min with a maximum memory consumption of 1589 MB. Memory consumption ranges from 496 MB to 1589 MB for both designs.

Tab. 7.3 also shows the numbers for the redesign (rows 3 and 4). The state space of the redesign could be somewhat reduced. Depending on the configuration through VHDL generics, the new design counts between 3997 and 41961 state bits.

The new implementation not only represents a "clean refactoring" of the old IP block but also underwent aggressive RTL optimizations for minimizing the power consumption of the circuit, such as manifold sharing of a single counter for various purposes (as opposed to several independent instances in the original design), reduction of input buffering, as well as clock gating of large parts of the circuitry, as proposed in [57]. These measures lead to substantial reduction of circuit activity, resulting in a reduction in power consumption of 50%. Applying these intricate optimizations to the RTL design was only possible because their functional correctness could be immediately verified using the accompanying property suite.

Table 7.4: SONET/SDH Framer — simulation of 10^7 frames

Design	RTL sim.	SystemC-PPA sim.	property proofs
RTL Industrial	541 s	2 s	23 min
RTL Redesign	600 s	2 s	10 min

Another goal of the top-down methodology is to move verification to the system level. Tab. 7.4 shows the results for the simulation for 10^7 frames. Due to the strong abstraction of the system-level description the simulation times are reduced from 541 s to 2 s, while the soundness of the system model guarantees preservation of the design's I/O sequences. For illustration of our approach, we also employed PDD to design a simplified version of the industrial framer design, available with all relevant data in public domain under [50].

7.3 Case Study: Industrial FPI Bus

This case study is based on a complex on-chip bus protocol called Flexible Peripheral Interconnect (FPI), owned by Infineon Technologies. A highly optimized industrial RTL implementation of this bus was available for our experiments. We constructed a system where two peripherals act as masters and two memories act as slaves. As an implementation of this system we instantiated the FPI bus with the industrial RTL modules for the above configuration.

PPAs for these modules were created bottom-up along the lines of [2] and were verified to fulfill all conditions for compositionality as developed in Chap. 4. Tab. 7.5 shows data for the industrial designs and the PPAs extracted bottom-up.

Table 7.5: FPI Bus — original design

	RTL Design			PPA		
Module	inp./out.	FFs	LoC	inp./out.	var.	states/ops
MasterAgent	199/202	292	3568	9/3	4	5/84
SlaveAgent	199/202	292	3568	12/15	0	1/16
BCU	258/215	941	8966	4/4	12	1/6

For the RTL design, we show the number of (binary) input/output signals, flip-flops and lines of code (LoC). The industrial design has a total of 12.5k LoC in VHDL. The agents share the same HDL description and are configured to be a master or a slave through generics.

For the PPAs, we give the number of abstract inputs, outputs, data path variables as well as the number of abstract states and operation properties, i.e., the nodes and edges in the state transition graph of the abstract FSM. The effort for PPA extraction by bottom-up verification was measured to be about 1 person month per 2k LoC.

For demonstrating the feasibility of the PDD approach we redesigned the most sophisticated component of the bus, the *MasterAgent*, top-down. In order to come up with a differently structured design, we used the PPA that had been extracted bottom-up and transformed it into a compositional SystemC-PPA description consisting of a control module and a memory module. The external interfaces of the redesign are identical to the original version so that the new design could be integrated "first-time-right" into the existing industrial environment. Starting from the SystemC-PPA model, a new RTL implementation of the the master agent was obtained using the tool DeSCAM and the PDD methodology. The redesign took 3 person days and was performed by a different engineer than the person who extracted the original PPA bottom-up. The new RTL design is comparable to the old implementation in terms of hardware cost, area and performance. However, due to the existence of a complete set of operation properties holding on the design, there are now additional optimization opportunities. For example, [57] and [58] present automatic power consumption reduction techniques based on a complete set of operations.

The small amount of effort required for redesigning the master agent shows the efficiency of the PDD methodology, which benefits greatly from the automatically generated properties. The case study further demonstrates how PDD can help with the problem of refactoring legacy designs. Replacing old code components with new versions is a process highly prone to human error. PDD guarantees implementation correctness even after performing substantial design changes. The maintainability of the code is greatly enhanced by the SystemC-PPA (high-level documentation) as well as the SVA properties for the RTL (low-level documentation).

Table 7.6: FPI Bus — redesign

	RTL Design			PPA			
Module	inp./out.	FFs	LoC	inp./out.	vars.	states/ops.	proof
Control	253/110	186	553	11/10	3	6/95	58 s
Memory	100/213	425	200	2/7	15	2/47	50 s

Tab. 7.6 shows the results for the redesign of the MasterAgent. The I/O interface of the redesign is equivalent to the original one. The different number of inputs and outputs results from the decomposition of the original design into two modules, the *Control* and the *Memory.* The maximum computational effort encountered for a single property was a proof time of 1.7 s and a memory consumption of 1369 MB. As a result of this computational efficiency, the designer can work with the property checker interactively and can check design refinements repeatedly during the design process.

7.4 Case Study: Wishbone Bus

With this case study we demonstrate the flexibility of PDD when describing systems at different levels of abstraction. We considered a Wishbone bus system comprised of one master and four slaves and created two different system-level models for it. The first is protocol-independent and models the possible communications between the master and the slaves using message passing. The second one is a bit more refined and models bus agents for the master and the slaves as well as bus transactions according to the Wishbone protocol.

Both models were represented in SystemC-PPA and were subsequently refined into RTL implementations according to PDD, using DeSCAM for generating properties. Tab. 7.7 shows data for both models and their implementations. The PPAs are, again, represented in terms of the numbers of input/output predicates, data path variables, abstract states, and transitions (operations). The RTL implementations of both models have an identical I/O interface. However, the simple implementation is of significantly lower complexity than the Wishbone-compliant implementation.

Table 7.7: Wishbone Bus

	RTL Design			PPA		
Module	inp./out.	FFs	LoC	inp./out.	var.	states/ops
Simple	209/303	403	203	5/5	5	9/34
Wishbone	209/303	2023	634	5/5	17	12/45

For each model, we ran simulations of bus transactions, both on the system level as well as the RTL, cf. Tab. 7.8. As expected, system-level simulation is orders of magnitude faster than RTL simulation. Since each RTL is a sound refinement of the corresponding SystemC-PPA model, any simulation result obtained on the system level also holds for the implementation, which makes RTL simulation replaceable by SystemC simulation in many cases.

Table 7.8: Wishbone Bus — simulation of 10^7 frames

Design	RTL sim.	SystemC-PPA sim.	property proofs
Simple	144 s	4 s	34 s / 850 MB
Wishbone	322 s	90 s	410 s / 3885 MB

In a second experiment we demonstrate the versatility of PPA when it comes to modeling designs at various levels of abstraction. As mentioned, both SystemC-PPA models have the same external I/O interface. Therefore, it was possible to use the SVA properties that were generated by DeSCAM for the more abstract, protocol-independent SystemC-PPA model and refine them such that they hold also on the less abstract SystemC-PPA description modeling a system of bus components. The manual effort for this "cross-model" property refinement was less then 1 person day. As a result, the RTL implementation of the Wishbone is now also a sound refinement of the more abstract system-level model which allows for even higher verification efficiency than "its own" SystemC-PPA model.

This experiment shows that there is great flexibility when choosing the abstraction level of SystemC-PPA models.

All models and properties from this case study are publicly available in full on GitHub [50].

7.5 Case Study: PDD-P flow

In this section we report the results for the PDD-P flow. We elaborate the benefits of the extension with two examples. The results are generated on an IntelCore I7 with 32 GB of RAM, running the commercial property checker OneSpin.

The first design is a simple processor with a static 4-stage pipeline and a forwarding unit. The second design is an AHB bus with four masters and three slaves.

The two designs are pre-existing designs. We created the SystemC-PPAs models for the existing RTL designs. The aim is to show that the SystemC-PPA subset provides the required features and that the generated properties do no limit the designer. We re-implemented core functionality to mimic a design process and show the feasibility of this approach.

7.5.1 Processor

The first design is a simple 4-stage in-order processor. This processor is used as our example in the previous sections. A total of 9 macro operations are created. The generated base properties are proven in 9 sec and the relaxed properties need 2:48 min to prove. As mentioned earlier, the functionality is covered by the base property set. A bug in the data path of the ALU, e.g., implementing subtract instead of an add, results in a failing base property. The commitment in line 16 of Fig. 6.11 fails. The counterexample is found in 1 second.

The proof time for the relaxed properties appears to be long compared to the base properties. However, when a bug is introduced in the forwarding unit a counterexample is provided by the property checker in less then four seconds, allowing fast design cycles. In this case, the commitment in line 30 of Fig. 6.12 fails.

Table 7.9: Results for the processor

Property Name	**Property Name**	**Relaxed prop.**
	Time[min]	Time[min]
add	0:01	0:23
branch_not_taken	0:01	0:02
branch_taken	0:01	0:02
load	0:01	0:50
neg	0:01	0:17
nop	0:01	0:11
reset	0:01	-
store	0:01	0:40
sub	0:01	0:23

7.5.2 AHB

The second design is an AHB bus with fixed arbitration and single read/write transfers. The protocol allows to implement a pipeline with three stages (addr, data, resp). An empty pipeline is defined as a state where each master is idle. In this implementation the master with ID 3 has the highest priority and the master with ID 0 the lowest. For this example we use fixed priority-based arbitration and accept that the master with the lowest priority can starve.

Table 7.10: Results for AHB bus

Property Name	**Base property**	**Relaxed prop.**
	Time[min]	Time[min]
master0_to_slave0	0:57	4:30
master1_to_slave0	0:36	5:35
master2_to_slave0	0:29	3:14
master3_to_slave0	0:39	1:32

For this example a macro operation for each possible transfer between *master0* and a slave is generated from the ESL. Proving all base properties takes about 2:40 min with an average of less then 1 min per property. Proving the relaxed properties takes about 15 min . An interesting observation is that the lower the ID of the respective master is, the longer the proof of the relaxed property takes. This is due to the fact that more stalling cases occur if master 2 requests. Although the macro operations overlap in their execution, there is no dependency between them. Once a macro operation is in the pipeline it executes independently of the other operations. Hence, no forwarding cases can happen and no consistency assumption is necessary.

The starvation is detected by a failing property *master0_to_slave0.* By analyzing the counterexample it is easy to see that master 0 is never granted access to the bus. This issue is solved by applying a liveness constraint on the model, stating that after a certain number of clock cycles master 0 is granted access. This behavior is also visible at the system level during simulation.

Chapter 8

Conclusion and future work

In this work we presented *Property Driven Design*, a new methodology to refine transaction-level system models into RTL implementations. The theoretical basis of this methodology is compositional path predicate abstraction; its practical basis is state-of-the-art property checking along with methods to generate and refine properties starting from SystemC-PPA descriptions. Due to the established well-defined formal relationship between the transaction-level model and the implementation, a "trustworthy" system model becomes available such that global verification tasks can be shifted from the RTL to the system level.

The lack of a trustworthy and abstract system-level model in today's design process leads to a "verification gap", i.e., a difference between the increase in design complexity and improvements in verification methodologies. Due to this gap, verification becomes a bottle-neck within the design cycle. This work provides a solution to this problem by providing a trustable, abstract and executable system-level model that can be used as "golden model" for design sign-off.

Our case studies clearly show the attractiveness of the approach: First, complex design tasks can be mastered in relatively short time while simultaneously guarantying the functional correctness of the resulting design. Second, the experimental results indicate that the abstract SystemC-PPA models execute significantly faster then the refined RTL models. Using such models for design sign-off should have a significant positive impact on time-to-market.

The obtained results are based on industrial standard languages such as SystemC, Verilog and SVA. This allows designers and verification engineers to leverage the power of already existing, commercially available tools, while benefiting from the improved design flow. Executing the case studies showed that designs with a pipeline can result in complex refinements. This motivated us to extend the existing PDD flow to PDD-P to archive a better support for pipelining.

The main contribution of this work is the open-source tool "Design from SystemC Abstract Models" (DeSCAM). It supports the PDD methodology by generating the required properties and optional HDL skeletons. First, the tool parses the provided SystemC description, checks that it adheres to the SystemC-PPA subset and provides immediate feedback to the user in case of a violation. In the scope of this work we provided a definition of the SystemC-PPA subset. The subset is designed such that it allows transaction-level modeling at the system level while still complying to semantics of the

PPA. Using DeSCAM, instead of a manual creation of the properties, results in two additional benefits:

First, the tool provides a formalized way of generating the properties from the specification. Hence, the problem of "miss-interpreting" the specification and thus generating incorrect/misguided properties does not occur. Second, the designer is not relieved from the responsibility to ensure that the set of properties is complete.

For an efficient PDD methodology, DeSCAM has to fulfill two non-functional goals. First, the SystemC-PPA subset should be as inclusive as possible, allowing the system-level engineer to use a natural way of writing SystemC code. Second, the generated properties have to be easy to read from a human point-of-view. Readability of the properties is an important aspect of PDD. Aside from ensuring soundness w.r.t. the semantics of PPA, the properties serve as a formalized documentation within the design process. As explained in Chap. 5, the tool implements optimization techniques to increase readability of the properties. However, within the scope of this work we only managed to implement a core subset of SystemC language features.

Future work should aim at extending this subset by more language features like bounded loops, switch-case statements and/or bit-slicing to provide a more intuitive design experience to the user. An industrial case study with Nordic Semiconductor [59] showed that PDD is applicable in practical settings. However, the current state of the tool lacks some PPA related features. Although the provided selection of communication interfaces allows to model any type of protocol, it sometimes burdens the designer with an unnecessary large overhead to match the SystemC-PPA to the envisioned protocol. To improve usability, a library of interfaces that allow for a more natural design process should be provided in future.

Furthermore, we will explore how to connect SystemC-PPA easier with high-level simulation models and extend DeSCAM to better support Engineering Change Orders (ECOs) at the the system level, such that incremental design steps preserve a maximum of the RTL macro refinements. Finally, we would like to note that the future development will continue as a spin-off, that aims at building as commercial version of the software.

Chapter 9

Summary

We introduce Property-Driven Design, a tool-flow that guarantees formal soundness between ESL and RTL and thus enables a shift-left of general functional verification by moving HW verification to higher abstraction layers. In addition, by generating a formal Verification IP (VIP) automatically from ESL descriptions, the entry hurdle to formal methods is reduced considerably, opening them to a wider audience, which effectively 'democratizes' them. Short feedback cycles reduce time spent on RTL verification and lead to higher-quality designs.

The proposed builds upon existing flows like Unified Verification Methodology (UVM). Starting from a set of use cases (e.g., a constraint-random stimulus or precomputed input sequences), a behavioral ESL model, e.g., specified in SystemC-TLM, is checked for correctness with a simulation-based approach. In a traditional approach, a time-costly re-verification of the RTL Design Under Test (DUT) against the same use cases is required to build confidence in behavioral equivalence of ESL and RTL. Our approach removes the need for a complete functional re-verification of the DUT, saving valuable RTL verification time and, additionally, allows to cover the entire RTL design behavior with a complete set of formal properties.

The VIP covers the design intent described by the ESL model as a complete set of formal properties (e.g., as SVA), wherein each property covers the hardware behavior in terms of a transactional operation between automatically inferred states, e.g., different stages of a pipeline. Instead of re-running the simulation-based verification with the DUT, the design is verified formally using the VIP. If all properties hold on the DUT, the design is correct-by-construction, w.r.t. the ESL model.

The basic ingredients to the new flow are:

- Path-Predicate Abstraction (PPA), a mathematical model that allows to establish formal relationship between ESL and RTL based on formal properties. Therefore, verification results obtained at the ESL also apply at the RTL.
- DeSCAM, an EDA tool that automates VIP creation for the target design in a chosen verification language, which significantly reduces required verification expertise and effort.
- Standard industrial formal verification languages, tools and verification techniques.

Chapter 10

Deutsche Kurzfassung: Eigenschaftsgetriebene Hardwareentwicklung

Bis heute ist das *Register Transfer Level* (RTL) in den meisten Methoden der Einstiegspunkt zum Entwurf eines *Systems-on-Chip* (SoC) oder eines eingebetteten Systems. Trotz aller Fortschritte im Bereich des *Elektronic System Level* (ESL), der virtuellen Prototypen oder neuer Verifikationstechniken basiert der eigentliche konzeptionelle Prozess des RTL-Entwurfs auf Sprachen wie VHDL oder Verilog. Dieser Prozess hat sich in den letzten Jahrzehnten praktisch nicht verändert. Eine Ausnahme hiervon bildet *High-Level Synthesis* (HLS), welches sich in speziellen Bereichen, wie zum Beispiel bei der Implementierung von Signalverarbeitungsalgorithmen, als anwendbar erwiesen hat.

In der industriellen Praxis wird der Großteil der RTL-Designs noch immer manuell erstellt. Ausgangspunkt für diesen Prozess sind informelle Spezifikationen, Beschreibungen, Diagramme von *Finite-State-Machines* (FSMs), Zeitdiagramme oder Flussdiagramme, welche von einem Entwickler nach seinem Verständnis in ein RTL-Design übertragen werden. Das RTL-Design wird allgemein auch als "golden model" bezeichnet, da es den wesentlichen Bezugspunkt zur Verifikation der Funktionalität darstellt. Selbst bei Nutzung von HLS bleibt die RTL-Beschreibung das "golden model" für die Verfikation der Designs und nicht die Hochsprache, welche den Ausgangspunkt der Synthese darstellt.

Modelle auf der Systemebene hingegen werden in der Regel nur als "Prototypen" betrachtet. Sie werden für die frühzeitige Bewertung der funktionalen und nicht funktionalen Designeigenschaften verwendet. Diese Trennung zwischen Systemebene und RTL-Implementierung ist eine der Hauptursachen für die wachsende Komplexität der Verifikation in den gängigen industriellen Designsflows.

Im Rahmen dieser Arbeit arbeiten wir an einem neuen Ansatz für das RTL-Design, ausgehend von einer Beschreibung auf der Systemebene. Die vorgeschlagene Methodik basiert auf einem manuellen Designprozess und schränkt den Designer bezüglich der zu entwickelnden Implementierung nicht ein. Wir stellen eine Tool zur Verfügung um sicherzustellen, dass sowohl die Hardware-IP als auch die Verifizierungs-IP systematisch und kompositorisch entwickelt werden können. Der größte Vorteil der vorgeschlagenen Methode ist, dass eine formal definierte Beziehung zwischen RTL und Systemebene hergestellt wird. Dieser Aspekt ist entscheidend, wenn versucht wird, von den heutigen Entwurfspraktiken abzuweichen und die Verwendung abstrakter Systemmodelle als goldene Designmodelle voranzutreiben.

Der vorgeschlagene Ansatz wird als *Property-Driven Development* (PDD) bezeichnet und wurde von einer weit verbreiteten Technik aus dem Software-Engineering namens *Test-Driven Development* (TDD) inspiriert. Das TDD Paradigma weicht vom klassischen V-Modell für die Softwareentwicklung ab und basiert auf der Überzeugung, dass die eigentliche Softwareentwicklung durch die Erstellung von Softwaretests im Voraus positiv beeinflusst wird. Solche Gray-Box-Tests haben gezeigt, dass sie im Vergleich zu klassischen White-Box Tests zu einer höheren Fehlerabdeckung führen. Die Übertragung dieser Idee auf die Hardwaredomäne führt zu einer speziellen Entwurfsmethodik. Die Überprüfung der "Tests" erfolgt frühzeitig und systematisch im Entwurfsprozess. Die Rolle der Softwaretests in TDD können im Hardware-Design durch formale Eigenschaften, in Eigenschaftssprachen wie der *Property Specification Language* (PSL) oder *System Verilog Assertions* (SVA), ersetzt werden.

Ein besonderes Merkmal unserer Methodik ist die systematische und automatische Erstellung der Tests (hier SVA oder PSL Properties) aus der abstrakten Beschreibung. Die automatische Generierung stellt einen wesentlichen Unterschied zum TDD aus der Software-Domäne dar. Die generierten Tests sind an die besonderen Anforderungen des RTL-Hardware-Designs angepasst. Der Einsatz moderner formaler Verifikationstechniken führt zu einem systematischen und intuitiven Designverfahren, ohne die Freiheit des Designers einzuschränken. Die spezifische Art und Weise, wie die Eigenschaften anfänglich generiert und später verfeinert werden, ermöglicht schließlich eine formale Aussage über die funktionale Korrektheit der erstellten RTL-Implementierung.

Wenn alle vom abstrakten Modell generierten Properties für das finale Design bewiesen wurden, ist bewiesen, dass das Design mit mathematischer Sicherheit eine korrekte Verfeinerung des abstrakten Modells ist. Dies wird durch eine klar definierte formale Beziehung zwischen dem abstrakten Modell und seiner konkreten Implementierung sichergestellt. Umgekehrt bedeutet dies, dass nach erfolgreichem Abschluss des Entwurfsprozesses die Systemebene eine formal korrekte Abstraktion des RTL darstellt.

Die Erstellung von ESL-Modellen, welche eine PPA des RTL-Designs beschreiben, verändert die Rolle dieser Modelle grundsätzlich. Anstatt nur ein Prototyp zu sein und keine definierte Beziehung zur Implementierung zu haben, kann die Systemebene nun als Designmodell angesehen werden, ähnlich wie RTL-Designmodelle als solide Abstraktionen der zugrunde liegenden Gate-Ebene (aufgrund der formalen Äquivalenzprüfung) gelten. Basierend auf der PPA stellt der von uns bereitgestellte theoretische Rahmen eine formale Verbindung zwischen dem abstrakten Systemmodell und dem konkreten RTL-Design her. In unserer Methodik wird die Semantik des Systemmodells durch eine kompositionale PPA definiert.

Wir zeigen in Kap. 3, wie dies in einer realistischen Entwurfsmethode, die auf Standardsprachen basiert, verwendet werden kann. Wir führen eine Teilmenge von SystemC mit dem Namen SystemC-PPA zur Beschreibung von Modellen auf Systemebene ein und präsentieren den PDD-Flow. Properties werden automatisch aus SystemC-PPA Beschreibungen generiert und anschließend während des Designprozesses verfeinert. Der Hauptbeitrag dieser Arbeit ist das Open Source Tool DeSCAM, dass die SystemC-PPA-Beschreibung automatisch analysieren und die Eigenschaften generieren kann.

In Kap. 5 stellen wir das Tool vor und beschreiben, wie die Properties correcty-by-construction erzeugt werden. Eine besonders herausfordernde Aufgabe ist die Implementierung von Designs mit einer Pipeline auf Basis eines abstrakten sequentiellen Modells. Wir haben eine spezielle Methodik für dieses Problem entwickelt, die in Kap. 6 vorgestellt wird. Diese Methodik hilft dem Designer eine korrekte Implementierung sicherzustellen und mögliche Hazards aufzulösen. In Kap. 7 präsentieren wir Fallstudien, die für verschiedene Industrie- und Open-Source Unternehmen durchgeführt wurden. Unsere experimentellen Ergebnisse zeigen, dass vollständig verifizierte RTL-Designs erstellt werden können. Alle Entwurfsschritte basieren auf Standardsprachen und verwenden nur die modernsten formalen Techniken und Tools. Unsere Experimente zeigen, dass der manuelle Aufwand für Design und Verifizierung wesentlich reduziert wird.

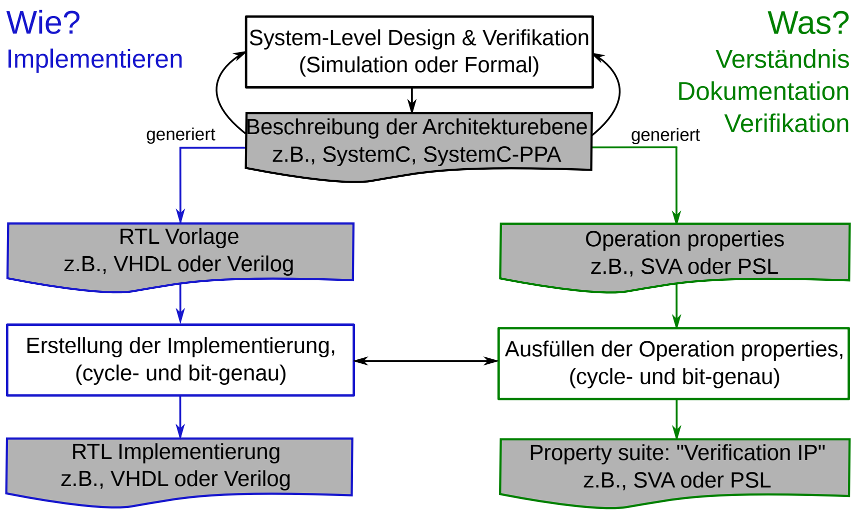


Figure 10.1: Property-Driven Design: Übersicht

In Abb. 10.1 sieht man eine Übersicht über den neu entwickelten PDD-Flow. Beginnend auf dem ESL können wir das abstrakte Design durch eine ausführbare Modellbeschreibung formalisieren. In unserer aktuellen Implementierung des Flows verwenden wir zu diesem Zweck die Sprache SystemC.

Bereits auf Systemebene werden erste Verfeinerungen auf architektonischer Ebene vorgenommen und beschrieben. Die Architekturebene (*Architectural Level*) umfasst jede Beschreibung auf Systemebene, für die eine klare Semantik in Bezug auf PPA möglich ist. Zu diesem Zweck wurde die Sprache SystemC-PPA erstellt (vgl. Sec. 4.4). Auf der Architekturebene wird das System zeitabstrakt modelliert, indem abstrakte Automaten und deren Interaktionen beschrieben werden. Es wird beschrieben, welche Berechnungen zwischen Kommunikationspunkten verschiedener Komponenten durchgeführt werden.

Dies schreibt jedoch nicht vor, wie ein bestimmter Algorithmus auf der RT-Ebene implementiert werden muss. Zum Beispiel eine Multiplikation auf Systemebene wird möglicherweise mit Multiplikationsalgorithmus von Booth, einem Wallace Tree oder ein anderer geeigneten Methode im RTL implementiert. Die formalen Eigenschaften ermöglichen das Abstrahieren von solchen Implementierungsdetails. SystemC-Beschreibungen, die auf der Architekturebene zu SystemC-PPA verfeinert wurden, können dann von unseren Tools verarbeitet werden.

Lebenslauf

Persönliche Daten	
Name	Tobias Ludwig

Ausbildung	
08/1999 – 06/2008	
	Albert-Schweitzer-Gymnasium, Dillingen
04/2009 – 01/2015	Abschluss: Abitur (2,1)
	Technische Universität Kaiserslautern
	Studium der Informationstechnik
06/2012 – 08/2012	Abschluss: Diplom Ingenieur (1,5)
	Eberspächer, Brighton, USA
	Praxisprojekt für Studienarbeit
05/2014 – 12/2015	Thema: „Manufacturing Execution System (MES) im industriellen Einsatz“ (1,0)
	Michigan State University, USA
	Diplomand im Fachbereich „Computer Science and Engineering“
	Thema: „Distance-preserving trees” (1,0)

Praktische Erfahrungen	
07/2008 – 03/2009	
	Arbeiterwohlfahrt Dillingen
04/2010 – 10/2010	Zivildienst
	Factory-IT, Diefflen
	Entwicklung einer Software zur Auswertung von Produktionsdaten
01/2011 – 12/2013	
	Eberspächer GmbH, Neunkirchen
	Werkstudent

	Aufsetzen von SIMATIC IT (MES, Siemens)
	Aufsetzen von Proficy (MES, General Electrics)
10/2012 – 05/2015	Frontend-Entwicklung (ASP.NET, C#)
	– **Technische Universität Kaiserslautern**
	– Wissenschaftliche Hilfskraft
	– Lehrstuhl „Entwurf informationstechnischer Systeme“
	Entwicklung von Algorithmen zur Hardwareverifizierung von Prozessoren (C/C++, Python)
	Erstellung von Dokumentationen (Doxygen)
05/2015 - jetzt	– OOP Reimplementierung bestehender Software (VHDL, C/C++)
	Technische Universität Kaiserslautern
Kenntnisse	Wissenschaftlicher Mitarbeiter Lehrstuhl „Entwurf informationstechnischer Systeme“ –
EDV-Kenntnisse	–
	Microsoft Office
Betriebssysteme	Webentwicklung (HTML, CSS, JavaScript)
Programmiersprachen	Windows, MacOS, Linux
FPGA Prototyping	C/C++, C#, Python, Assembler, ASP.NET, MSSQL, PHP
Sprachkenntnisse	VHDL, ONESPIN, XILINX
	Englisch (fließend)
	Französisch (Grundkenntnisse)

Kaiserslautern, 23.09.2020

Bibliography

[1] K. Beck, *Test Driven Development: By Example.* Addison-Wesley, 2002.

[2] J. Urdahl, D. Stoffel, and W. Kunz, "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)*, vol. 33, no. 2, pp. 291–304, Feb. 2014.

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

[4] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis (IEEE Cat. No.03TH8721)*, 2003, pp. 19–24.

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

[6] A. S. Initiative.

[7] F. Kesel, *Modellierung von digitalen Systemen mit SystemC: Von der RTL- zur Transaction-Level-Modellierung.* De Gruyter, 2012.

[8] D. Gajski, J. Zhu, R. D. Omer, A. Gerstlauer, and S. Zhao, *SPECC: Specification Language and Methodology.* Springer US, 01 2000.

[9] M. Aubury, I. Page, G. Randall, J. Saul, and R. Watts, "Handel-c language reference guide," *Computing Laboratory. Oxford University, UK*, 1996.

[10] Mentor Graphics, https://www.mentor.com/.

[11] A. Rose, S. Swan, J. Pierce, and J.-M. Fernandez, "Transaction level modeling in systemc," 2003.

[12] "Ieee standard for standard systemc language reference manual," Jan 2012.

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

[14] L. W. Nagel, "Spice2: A computer program to simulate semiconductor circuits," Ph.D. dissertation, EECS Department, University of California, Berkeley, 1975.

[15] T. Ludwig, J. Urdahl, D. Stoffel, and W. Kunz, “Properties first – correct-by-construction rtl design in system-level design flows,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, pp. 3093–3106, June 2020.

[16] J. Urdahl, S. Udupi, T. Ludwig, D. Stoffel, and W. Kunz, “Properties first? A new design methodology for hardware, and its perspectives in safety analysis (invited paper),” in *The IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2016.

[17] “IEEE Standard for Property Specification Language (PSL),” *IEEE Std 1850-2005*, 2005. [Online]. Available: http://www.eda.org/ieee-1850/

[18] “IEEE 1800 – standard for SystemVerilog–unified hardware design, specification, and verification language,” IEEE.

[19] A. Pnueli, “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.

[20] E. M. Clarke, E. A. Emerson, and A. P. Sistla, “Automatic verification of finite-state concurrent systems using temporal logic specifications,” *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, Apr. 1986.

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

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

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

[24] C.-J. H. Seger and R. E. Bryant, “Formal verification by symbolic evaluation of partially-ordered trajectories,” *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, 1995.

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

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

[27] J. Bormann, “Vollständige Verifikation,” Dissertation, Technische Universität Kaiserslautern, 2009.

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

[29] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, *Satisfiability Modulo Theories.* IOS Press, 2009, ch. 26, pp. 825–885.

[30] M. R. Fadiheh, J. Urdahl, S. S. Nuthakki, S. Mitra, C. Barrett, D. Stoffel, and W. Kunz, "Symbolic quick error detection using symbolic initial state for pre-silicon verification," in *Design, Automation & Test in Europe Conference (DATE).* IEEE, 2018, pp. 55–60.

[31] K. Devarajegowda, M. R. Fadiheh, E. Singh, C. Barrett, S. Mitra, W. Ecker, D. Stoffel, and W. Kunz, "Gap-free processor verification with s^{2}sqed and property generation," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE), Grenoble, France*, March 2020.

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

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

[34] S. Ray and W. A. Hunt, Jr., "Deductive verification of pipelined machines using first-order quantification," in *Proc. Intl. Conf. on Computer-Aided Verification (CAV).* Boston, MA: Springer, 2004, pp. 31–43.

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

[36] ——, "A complete compositional reasoning framework for the efficient verification of pipelined machines," in *Proc. International Conference on Computer-Aided Design (ICCAD)*, 2005, pp. 863 – 870.

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

[38] K. Hao, S. Ray, and F. Xie, "Equivalence checking for behaviorally synthesized pipelines," in *Proc. Design Automation Conference (DAC).* New York, NY, USA: ACM, 2012, pp. 344–349.

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

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

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

[42] A. Kölbl, J. R. Burch, and C. Pixley, "Memory modeling in ESL-RTL equivalence checking," in *Proc. Design Automation Conference (DAC)*, San Diego, CA, USA, June 2007, pp. 205–209.

[43] A. Kölbl, R. Jacoby, H. Jain, and C. Pixley, "Solver technology for system-level to RTL equivalence checking," in *Proc. Design, Automation Test in Europe (DATE)*, April 2009, pp. 196–201.

[44] P. Chauhan, D. Goyal, G. Hasteer, A. Mathur, and N. Sharma, "Non-cycle-accurate sequential equivalence checking," in *Proc. Design Automation Conference (DAC)*, 2009, pp. 460–465.

[45] K. Hao, F. Xie, S. Ray, and J. Yang, "Optimizing equivalence checking for behavioral synthesis," in *Proc. Design, Automation and Test in Europe (DATE)*, Leuven, Belgium, 2010, pp. 1500–1505.

[46] J. Langer and U. Heinkel, "High level synthesis using operation properties," in *Proc. of Forum on Specification Design Languages (FDL 2009)*, Sep. 2009, pp. 1–6.

[47] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, "Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SoC) verification," *ACM Trans. Des. Autom. Electron. Syst*, vol. 24, no. 1, pp. 10:1–10:24, Jan. 2019.

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

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

[50] "DeSCAM," https://github.com/ludwig247/DeSCAM.

[51] J. Urdahl, D. Stoffel, M. Wedler, and W. Kunz, "System verification of concurrent RTL modules by compositional path predicate abstraction," in *Proc. Design Automation Conference (DAC)*, 2012, pp. 334–343.

[52] G. J. Holzmann, "The SPIN model checker," *IEEE Transactions on Software Engineering*, vol. 23, pp. 279–295, 1997.

[53] L. Foundation, "Clang," http://clang.llvm.org. [Online]. Available: http://clang.llvm.org

[54] "Abstract Syntax Trees," https://en.wikipedia.org/wiki/Abstract_syntax_tree.

[55] L. D. Moura and N. Bjørner, “Z3: An efficient SMT solver,” in *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS).* Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340.

[56] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović, “The RISC-V instruction set manual volume II: Privileged architecture version 1.9,” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-161*, 2016.

[57] S. Udupi, J. Urdahl, D. Stoffel, and W. Kunz, “Dynamic power optimization based on formal property checking of operations,” in *Proc. Intl. Conf. on VLSI Design (VLSID)*, 2017.

[58] S. Udupi, J. Urdahl, D. Stoffel, and W. Kunz, “Exploiting hardware unobservability for low-power design and safety analysis in formal verification-driven design flows,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 6, pp. 1262–1275, June 2019.

[59] “Nordic Semiconductor,” https://www.nordicsemi.com.