Technische Universität München
Department of Electrical Engineering and Information Technology
Assistant Professorship of Hybrid Control Systems

Generation of Properties for Design Verification from an Abstract FSM model

Master's Thesis

Author: Valentin Hiltl

Technische Universität München
Department of Electrical Engineering and Information Technology
Assistant Professorship of Hybrid Control Systems

Generation of Properties for Design Verification from an Abstract FSM model

Master's Thesis

Author: Valentin Hiltl
Supervisor: M.Sc. Keerthikumara Devarajegowda
Advisor: Prof. Dr. Majid Zamani
Submission date: 12.04.2019

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 12.04.2019 Valentin Hiltl

Acknowledgments

First of all I would like to thank my supervisor at Infineon Keerthikumara Devarajegowda who is currently doing his PhD in the fields of formal verification at the Technische Universität Kaiserslautern. Besides his PhD he is heaviliy involved in Infineon's projects but he has always had time for me when I had questions about my research or when I needed encouragement which I appreciate a lot. I would like to mention that I always felt as a valuable part of his research and I had the freedom to set my own priorities with respect to this thesis which motivated me to start my own PhD as well. Many thanks also to thank Prof. Dr. Majid Zamani and Mahmoud Khaled of the Assistant Professorship of Hybrid Control Systems for giving me the chance to do my thesis in industry at Infineon.
Finally I would like to mention that I feel honored because of my parents' and friends' ongoing support throughout all my years of study. They always provided guidance as well as encouragement when I needed it and this achievement would not have been possible without them.

Many thanks,
Valentin Hiltl

Abstract

Various approaches for increasing the design productivity have been proposed, e.g hardware generators or High-Level Synthesis to cope with the ever increasing chip complexity which is driven by consumer demands. This leads to the fact that the verification gap widens although even today approximately 50% of the project time has to be spent for pre-silicon verification. This thesis contributes a property automation framework which aims at increasing the verification productivity to reduce the verification gap. By providing a formal specification in a defined design entry language, the property automation framework generates a complete set of properties to effectively verify the RTL implementation. The design entry language is a system-level modeling language precisely matching the abstraction level of Path Predicate Abstraction which defines a sound relationship between RTL level implementations and their system-level models. The flow follows the Model-Driven Architecture and is embedded in Infineon's automation environment which enables automation through the concept of metamodeling. In order to demonstrate the effectiveness of the developed approach, a complete set of properties is automated for a real life design, namely an I^2C-bus protocol implementation. The generated properties are formally proven by a commercial formal verification tool to completely verify the RTL implementation.

Contents

1. Introduction

The complexity of hardware designs has grown due to consumer demands during the last decades which include rich functionality, security, high performance and robustness. This is particularly true for System On Chip (SoC) designs where the used number of cores and Intellectual Properties (IPs) continues to grow [27]. In addition the complexity of IPs used in SoC designs itself increases at a rapid pace. This is because, on the one hand an IP needs to support complex protocols like AMBA or PCI Express but on the other also has to be configurable such that it can be reused for multiple applications [28]. Increased hardware design complexity means higher integration of transistors and shrinking paths which is known as Moore's Law. A side effect of decreasing transistor sizes is that older technologies like 65 nm or 130 nm become cheaper which leads to the fact that semiconductor solutions can be applied for new areas at a reasonable price. This development is known as More-than-Moore trend which especially drives the terminal nodes of the Internet-of-Things (IoT).
Keeping time-to-market of a certain product as short as possible plays a key role as it describes the period of time where a product creates costs but no income for a company. Time-to-market can be divided into two parts, namely development time needed for designing a specific hardware implementation and time which needs to be spent for pre-silicon verification which consumes about 50% of the overall development time[27]. Short time-to-market and ever increasing chip complexity are intrinsically contradictory and the trend of billions of IoT devices further intensifies the problem.
Companies tackle the loss of design productivity by constantly increasing their head count as well as their research and development expenditures [7]. Another promising approach to increase design productivity is using hardware generation languages (HGLs) like Chisel [2] or MetaRTL [9] which raise the level of abstraction from Register Transfer Level (RTL) description to hardware generation. High-level languages like Python or Scala are utilized to describe the generation intent. Widely used in the field of signal processing algorithms is a methodology called High-Level-Synthesis (HLS) which uses the same concept of raising the abstraction level. The hardware design is modeled on Electronic System Level (ESL) for design exploration and optimization at an early design stage and by gradually refining the model a hardware implementation is generated in an automated manner. As a consequence, even complex hardware systems are designed in short devolopment cycles due to the increased design productivity of the former methodologies. The time needed for pre-silicon verification increases due to increased complexity such that the verification gap, the difference between verification productivity and design productivity, widens.

To reduce the verification gap an effective methodology for increasing verification productivity is introduced. We propose an extension of the property automation framework presented in [8] which adopts the concept of raising the abstraction level for verification. The structure of the framework is inspired by Object Management Group's (OMG) Model-Driven-Architecture (MDA) principle for code generation [34]. MDA specifies three different viewpoints on a system: computation independent, platform independent and platform specific. Each viewpoint is represented as one or more models at a specific abstraction layer where the computation independent model operates on the highest level of abstraction followed by the platform independent model and the platform specific model. Following the MDA approach model-to-model transformations are used to cast between the models of different abstraction levels. The computation independent model represents a formal model of the informal specification which specifies the desired behaviour of the hardware system as it focuses on how the system is expected to behave but hides all technology related information. In the extended property automation framework proposed in this thesis the computation independent model is written in a system-level modeling language which operates at an abstraction level called Path Predicate Abstraction (PPA) [36]. PPA means a well-defined formal relationship between the abstract system-level model and its concrete RTL implementation. In other words, assuming the computation independent model represented as PPA is correct and complete with respect to the informal specification it can be trusted to be a sound abstraction of the RTL design model in the same way the RTL design model is trusted to be a sound abstraction of the underlaying gate level model. The system-level modeling language is simulation capable such that correctness and completeness with regards to the informal specification can be evaluated easily by the Verification Engineer. The platform independent model as well as the platform specific model are reused from the original property automation framework presented in [8]. A model-to-model transformation is used to convert the computational independent model into the platform independent models. Each platform independent model represents a specific abstract temporal Boolean expression specifying how the RTL implementation is intended to behave. By applying a further model-to-model transformation these abstract temporal Boolean expressions are translated into the platform specific models, namely interval properties specified in an arbitrary target language. The extended property automation framework supports multiple target languages like SystemVerilog Assertions (SVA) or Interval Language (ITL). Using the extended property automation framework speeds up property generation and therefore increases verification productivity because properties can automatically be derived from a formal specification. The thesis is structured as follows. Chapter 2 presents an overview over state-of-the-art concepts which are relevant for the contribution of this thesis. We present different available hardware verification techniques, the concept of metamodeling and how it can be used for automation as well as the abstraction technique PPA. Furthermore we explain the property automation framework proposed in [8] which combines both the concept of metamodeling with formal verification. Chapter 3 provides an overview

over the contribution of this thesis, namely an extension of the property automation framework shown in [8]. In Chapter 4 we introduce the former in detail and explain every intermediate step of the methodology based on a working example. In order to demonstrate the effectiveness of the developed approach, we apply the extended property automation framework for an existing I^2C-bus implementation in chapter 5. Chapter 6 provides an outlook over further improvements of the concept of PPA as well as the extended property automation methodology.

2. State of the Art

2.1. Hardware Verification

Each hardware design has to be verified with respect to a given specification. Hardware verification describes the process of gaining confidence that the design's behaviour matches the given specification. The specification expresses the desired behaviour of the design in a natural language, e.g. describes all desired features wanted from a customer. There are mainly three different techniques to perform hardware verification for a given specification and design, namely Simulation-based Verification, Hardware Emulation and Formal Verification which are explained in detail in the following chapters. In general to perform a full verification of a system, or in other words to gain full confidence in the system, **all** possible behaviours of the specific system have to be taken into consideration. However today's hardware designs are too complex to perform a complete verification as the former is not feasible in terms of consumed time. Using coverage metrics is a technique to remove redundant design behaviours without reducing the level of confidence. Two types of coverage metrics are widely used:

- **Code Coverage** is a technique to measure the implementation's executed lines of source code using one of the verification methods mentioned above.
- **Functional Coverage** is an approach track which functionalities specified in the specification are covered using one of the verification methods mentioned above.

2.1.1. Simulation-based Hardware Verification

To be able to perform simulation the design has to be expressible in a simulation capable description as a precondition. Hardware Description Languages (HDL) like Verilog [14] or VHDL [15] are widely used. In the context of simulation the specific system can be understood as a function mapping from the product of input space and state space, in the following called input-state space, to the output space. An input sequence produces an output sequence when fed to the system at a specific state as a stimulus. Note that, if the system is purely combinatorial the state space is empty which implies that output sequences do not depend on the current state but on input stimuli only. Simulation means the process of providing the input sequence to the system and to decide whether the produced output sequence violates the specification. To completely verify a given system using simulation all combinations of inputs and states have to be evaluated and the corresponding output sequences need to be validated with respect to

the specification. As the number of combinations grows exponentially with the number of states and possible inputs, this approach is not practical for circuits of even only moderate size. As a consequence the input-state space needs to be reduced to a subset of reasonable size to cope with the complexity of today's designs. This reduction leads to the fact that design errors remain potentially undetected because it may happen that the input-state combination triggering the erroneous output sequence is removed from the reduced input-state space.
A widely used methodology for simulation-based verification in industry is called *Universal Verification Methodology* (UVM) [13]. The UVM provides a base class library written in SystemVerilog [12], an object oriented Hardware Description and Verification Language (HDVL) which is an extension of the original Verilog HDL. A classical UVM testbench consists of reusable verification components (VCs). Each of those VCs has a defined architecture containing the following elements:

- A **Sequencer** is an advanced input stimulus generator responsible to generate input sequences.
- The **Driver** adds control signals to the input stimuli provided by the Sequencer.
- The **Monitor** samples the output sequences transmitted from the Device Under Test (DUT) to perform validation with regards to the specification. The monitor also collects coverage information.

Sequencer, Driver as well as Monitor are in-built classes of the UVM library.

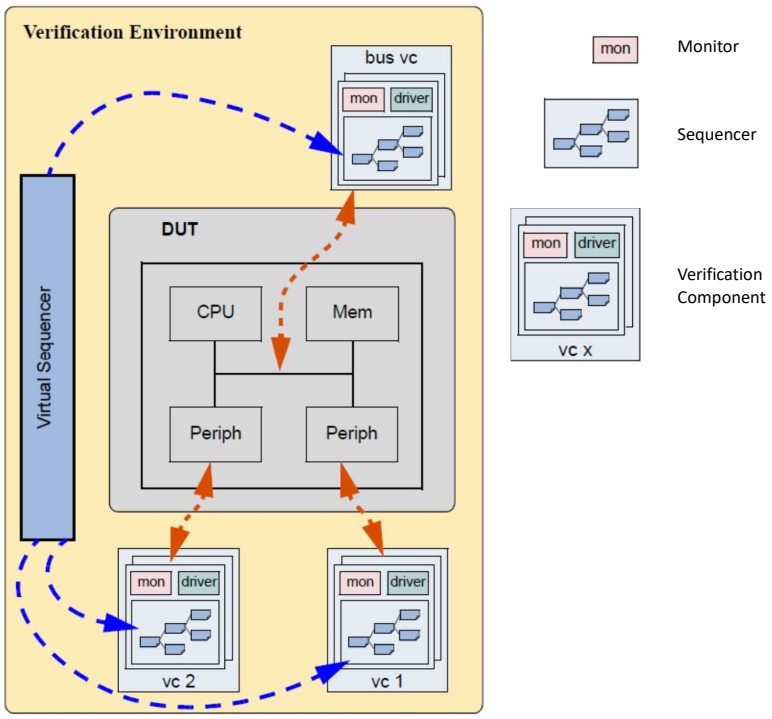


Figure 2.1.: Example for UVM testbench composed of three verification components [16]

Figure 2.1 shows an example of a UVM testbench composed of three different VCs each of them with the intent to verify the component they are associated with (vc 1, vc 2, bus vc). UVM is used to reduce effort and time caused by creating test cases manually through automatic test generation: The Sequencer automatically generates Constraint Random Tests (CRTs) which are random input sequences associated with constraints to make these stimuli valid inputs [37]. These input sequences are fed to the Driver which adds control signals needed for the system's interface protocol and passes the input sequence updated with this information to the device under test (DUT). The Monitor collects coverage information and performs validation whether the output sequence violates the specification. Although CRTs are generated automatically there is still a need for writing directed tests by hand to cover specific corner cases of the given design. Note that, the virtual sequencer shown in figure 2.1 is used to synchronize the different Sequencers of the VCs.
UVM is a *coverage-driven* simulation-based verification framework which means it evaluates the status of the verification process with respect to a coverage model [4]. A Coverage model consists of a set of metrics like functional coverage or code coverage, each of them associated with a coverage goal which has to be met during the verification process. To verify a design using UVM the same verification component is executed repeatedly using different random seeds for the Sequencers until the coverage goals are met. To reduce the time needed to meet the coverage goals the purpose of constraints is

expanded: By gradually tightening them the coverage metrics is forced to converge faster to the intended coverage goal.

2.1.2. Hardware Emulation

In times of System-on-Chips (SoCs) verification becomes even a greater challenge due to the chip's complexity. Classical software-based simulators are not capable of verifying these designs as they lack performance to execute testbenches in suitable time when running on general purpose computers, in the following called workstation. Hardware emulation is a technique which is used to deal with the increased complexity and there exist various different approaches for applying the former [23] [11] [19].
The classic methodology is to generate a bitstream from the DUT's RTL description which is then transferred to a target device, most likely a Field Programmable Gate Array (FPGA), to mimic the design behaviour in hardware. The workstation itself has to run the simulation environment or testbench only but does not simulate the complete design which heavily reduce the computational effort for the workstation. The bottleneck of this methodology however is mostly due to the communication overhead between DUT running in hardware and testbench running in software on the workstation [20] [26]. There exist mainly two approaches to circumvent this bottleneck: First, data transmission is abstracted to a transaction-level transmission where information of several clock cycles is compressed into a message to reduce channel frequency as well as quantity of transmitted data [23]. Problematic is the need of building such a transaction-level message generator which is time consuming and application specific. The second technique is to move the testbench still running in software into hardware which is called synthesizable testbench [11] [19]. To apply this technique the simulation environment is separated into two parts: Frequently used parts are transformed into synthesizable constructs as far as possible which are then moved together with the DUT to the target device. The parts of the testbenches which are not synthesizable, e.g. components of the UVM class library, are still executed in software on the workstation or on an embedded processor which is also transferred to the target device. This approach can lead to high requirements in terms of hardware resources of the target device as well as long compilation times.
Alternatively to the classic hardware emulation methodology another approach was introduced in 1997 by Quickturn Design Systems, namely processor-based emulation [32]. The concept of processor-based emulation means a certain number (~1000 to 100000) of arithmetic logic units (ALUs) that are correctly scheduled emulating all the Boolean equations of the DUT. Currently the two leading platforms are Palladium (Cadence) and Veloce (Mentor Graphics) which are considered to provide low compilation time, performance gain, and can fit even highly complex SoC designs but both systems are also very expensive in terms of costs [21].

2.1.3. Formal Hardware Verification

Formal verification is a notion for a collection of different approaches and techniques which mathematically analyzes the space of possible behaviours of a design by the means of a formal verification tool, rather than computing output sequences for particular input stimuli [22]. The formal verification tool uses intelligent mathematical techniques to take the design's complete space of possible behaviours into consideration or in other words, formal verification tools are exhaustive by nature. In comparison simulation-based verification methods look at individual points in the space of possible tests. To perform any kind of formal verification a formal description or model of the design needs to be provided as an input to the formal verification tool. Implementations written in HDLs like Verilog or VHDL are examples for valid formal descriptions accepted by formal verification tools. In the following the formal verification techniques *Equivalence Checking* as well as *Property Checking* are presented. Special interest is paid on *Interval Property Checking* because this technique is used within the scope of this thesis.

Equivalence Checking

As the name already implies the intention of equivalence checking is to show whether two designs are equivalent. We call two designs equivalent if for all input combinations the output is exactly the same. Equivalence checking is needed in mostly all synthesis design flows when moving vertically down in terms of abstraction layers, in particular to show whether the implementation is equivalent on RTL and gate level. But it is also needed for proving that horizontal transformations, namely optimization steps, are valid transformations and did not alter the system's behaviour.

Property Checking

The intention of property checking, also known as model checking, is to prove a model's compliance with given temporal logic expressions using fully automated methods. For the sake of complete automation of the prove the expressive power of temporal logic expressions is restricted to decidable formalisms. There exist mainly two decidable formalisms to characterize temporal logic expressions, namely *Computational Tree Logic* (CTL) [5] and *Linear Temporal Logic* (LTL) [31]. In CTL the classic Boolean operators are extended with two path quantifiers, **A** ('along all paths') and **E** ('along at least one path'), which need to be followed by one of the following temporal operators: **X** ('next'), **G** ('globally'), **F** ('finally'), **U** ('until') or **W** ('weak until'). LTL however provides the usual Boolean operators as well as the same temporal operators extended with the operator **R** ('release') but no path quantifiers.
In addition to providing automation the use of decidable formalism has another advantage: Assume the model conflicts with a temporal logic expression specified in a decidable formalism, the formal verification tool is able to provide a counterexample violating the temporal Boolean expression which eases tracing the source of the conflict. A temporal

logic expression is referred to as property and the set of all properties is called formal specification.
Generally speaking a model checking algorithm implemented in a formal verification tool performs a search over all possible future states starting from an initial or reset state of a specific design. For each state the algorithm determines whether it violates any of the given temporal logic expressions which represent the design's desired behaviour. As the number of state variables in the system increases, the size of the state space to be searched grows exponentially which is called *state explosion problem* and leads to computationally infeasible problems. In the following an approach to make model checking applicable for systems of reasonable size called *Bounded Model Checking* (BMC) [6] is examined in detail.

Bounded Model Checking

In bounded model checking the state search problem is mapped to a Boolean Satisfiability (SAT) problem to cope with the state explosion problem. SAT solvers require less memory usage and show increased performance such that bounded model checking can handle systems with a higher number of state variables. Additionally bounded model checking performs checks on the fly which can reveal counterexamples earlier, namely before state explosion becomes a problem.
To explain the concept of bounded model checking in detail we take a look at a general model of an arbitrary sequential circuit illustrated in Figure 2.2.

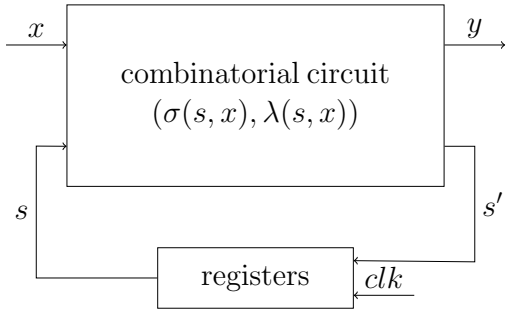


Figure 2.2.: General model of a sequential circuit

The top box contains the circuit's complete combinatorial logic which is basically the output function $y = \lambda(s, x)$ and the transition function $s' = \sigma(s, x)$ where x denotes the current input, y denotes the current output and s denotes the current state. After a discrete time interval t, at the clock event, the next state s' is stored in the register and used as current state for the next time interval $t + 1$. In bounded model checking properties do have a fixed time interval, in the following refered to as **bound** n as well as a defined starting state (often the initial state). This means that the property triggers in a certain state and expands for n number of clock periods where n is a finite number. In

classical model checking the algorithm starts to search the full state space starting from the specific state. To map this state search to a SAT problem the model gets unrolled iteratively n times, moving exactly one time step ahead with each iteration. Figure 2.3 shows an example where the general sequential circuit of figure 2.2 is unrolled for a bound of $n = 3$.

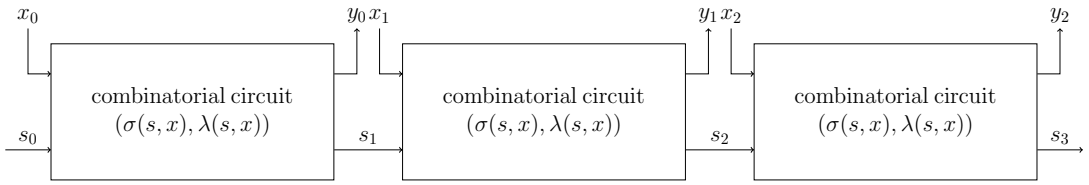


Figure 2.3.: Bounded circuit model unrolled for $n = 3$ clock cycles

Unrolling means copying the combinatorial logic of the circuit shown in figure 2.2 for n times where n denotes the bound of the specific property. Each copied circuit represents the combinatorial circuit at the corresponding timepoint. The next state output of the current circuit is connected to the current state input of the circuit corresponding to the following timepoint. In this way the entire time interval is represented as combinational circuit in which the state at a particular cycle is represented as a Boolean vector, $s_t = [s_{0,t}, s_{1,t}, s_{2,t}, s_{3,t}]$. Based on the unrolled model the property can be mapped to a Boolean satisfiability problem where the inputs at each timepoint are taken as free variables restricted only by the assumption of the property itself. The proof succeeds if the negated property cannot be satisfied.
Bounded model checking produces bounded proofs which can be useful in some situations, e.g. to find counterexamples. Verification however requires unbounded proofs which are globally valid and not restricted to a certain time interval. Bounded model checking is able to produce such globally valid proofs if it is guaranteed that the unrolled model covers the entire reachable behaviour of the design. This can be ensured if the combinatorial circuit from figure 2.2 is unrolled **at least** k times where k denotes the sequential depth of the original sequential circuit. For larger designs this approach leads to infeasible computational complexity due to the resulting SAT problem.

Interval Property Checking

Interval Property Checking (IPC) [29] is a variant of BMC which produces globally valid unbounded proofs for a specific type of properties. To apply IPC we restrict the temporal logic expressions describing the design's intended behaviour to a subset called *interval properties*. An interval property is a temporal logic expression in LTL syntax of the form $G(A \rightarrow C)$, where A is called *assumption* and C is referred to as *commitment*. The only temporal operator of the LTL syntax which is allowed to be used for the sub-formulas A and C is the next operator $\mathbf{X}$. This implies that an interval property describes behaviour

over a finite time interval.
IPC is a SAT-based model checking technique such that the procedure is almost equivalent to BMC. The only difference to BMC is that in IPC properties do not start from an initial state but **any** state which does not conflict with the assumption A. In other words the starting state is left as an free input in the SAT problem restricted only by the sub-formula A. This implies that IPC provides globally valid unbounded proofs for properties which specify behaviour over a finite time period.

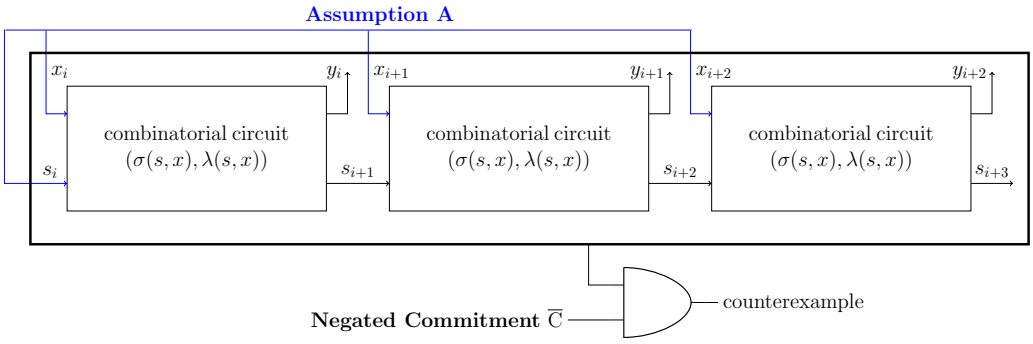


Figure 2.4.: Proof computation for interval property encompassing a time interval of $n = 3$

Figure 2.4 illustrates the proof computation for the arbitrary sequential circuit of figure 2.2 and an interval property which encompasses a time interval of three clock cycles. In other words the next operator **X** is stacked at most three times in the interval property which implies that the bounded circuit model has to be unrolled for $n = 3$ times. The formal verification tool executing the proof is free to choose any arbitrary value for the starting state s_i, as well as for the inputs x_i, x_{i+1} and x_{i+2} which does not conflict with the assumption A of the property under consideration. Equivalent to BMC, the proof of the interval property succeeds if the negated property cannot be satisfied. Applying that for an arbitrary interval property leads to: $\overline{G(A \rightarrow C)} = \overline{G(\overline{A} \vee C)} = G(A \wedge \overline{C}) =$ counterexample. As a consequence the interval property holds if counterexample never evaluates to `true`.
The extended property automation framework which is proposed in this thesis generates a complete set of interval properties from a formal specification such that IPC is the formal verification technique which is used in the scope of this thesis. In the following we introduce the completeness criterion which defines under what circumstances we call a property suite complete.

Completeness Criterion

Completeness can be understood as an absolute functional coverage metric which is met if and only if the formal criterion for completeness is proven for a property suite consisting of *operation properties* [3]. An operation property is an interval property with additional semantics. The whole design behaviour is considered as consecutive execution of different operations which are operating over a finite number of cycles. Each operation property describes the intended behaviour of such an operation which implies: Assume each operation has an operational property associated, by proving all operational properties on a given implementation the complete design behaviour is covered. In detail the operation properties are required to specify the output sequences with respect to determination requirements which state which visible registers and outputs in the design have to be determined under what circumstances. The completeness criterion is a technique which can be used to automatically prove that the operation properties in sum completely describe the output behaviour.
The formal verification tool OneSpin 360 [30] which is used in the scope of this thesis provides a collection of tests, namely *Case Split Test*, *Successor Test*, *Determination Test* and *Reset Test* to automatically prove or disprove the completeness criterion for a given set of properties. We say a given property suite is complete if and only if **all** tests mentioned above do hold for that given set of properties. Note the important fact that these tests reason on the behaviour specified in the properties only but do **not** consider the implementation of the design. To be able to run these tests the user needs to provide a *property graph* as a precondition which specifies the sequence of operations. In other words the property graph defines the succeeding operations for each property.
The mentioned test are briefly introduced in the following. For more information and examples, please refer to [30].

- **Case Split Test:** The case split test checks whether, given some operation property (A_P, C_P), there exists a succeeding operation (A_S, C_S) for **all** possible scenarios. To do so the successors $(A_{S,i}, C_{S,i})$ of the given operation are extracted from the property graph and it is checked for all possible input traces if any of the former is able to trigger. In other words it is examined if any of the successors' assumption parts $A_{S,i}$ can be satisfied for any of the input traces assuming that the commitment part of the predecessor C_P holds. If this is the case the case split tests holds for that given operation property otherwise a gap in the verification has been identified. The above needs to be repeated for each operation property of a certain property suite such that the case split test succeeds for the full set of properties.

- **Successor Test:** The successor test checks for every pair of predecessors and successors operations $((A_P, C_P), (A_S, C_S))$ defined in the property graph whether the successor's assumption part A_S **only** depends on inputs and visible registers determined by the commitment part of the predecessor C_P. To verify the former the verification tool examines two arbitrary traces which satisfy A_P such that

the predecessor operation triggers and therefore both traces share common values determined by the predecessor's commitment part C_P. The test checks then whether it is possible to fulfil the assume part of the successor operation property A_S for one trace but violate A_S for the other. This implies that A_S does depend on more than inputs and visible registers determined by C_P which leads to a failing successor test.

- **Determination Test:** The determination test checks whether each operation determines all outputs and visible registers with respect to the defined determination requirements. A holding determination test implies that both the visible registers as well as the outputs are proven to always have a value which is determined by the commitment part of a certain operation property.
- **Reset Test:** The reset test refers to the *reset property* which is a special property specifying the design's behaviour after reset. The reset test is constructed similarly to the tests mentioned above but applied for the reset property and without any predecessor property.

2.2. Automation through Metamodeling

2.2.1. Concept of Metamodeling

The term *meta* originates from Greek and means *above* or *beyond*. In the context of models this means a metamodel is a model of a model and metamodeling is the notion for the process of generating such a metamodel. In the scope of this thesis metamodels are used to define the structure of models, namely the model's components, their attributes as well as the relations between each other.

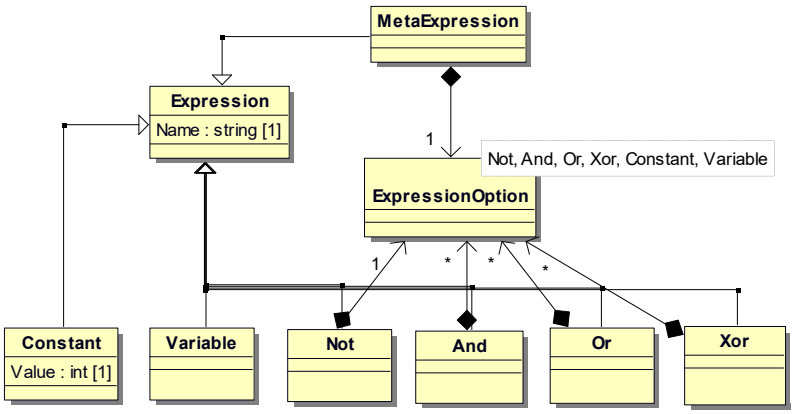


Figure 2.5.: Metamodel `MetaExpression` for bitwise expressions

Figure 2.5 shows an example metamodel called `MetaExpression` defining the structure of models for bitwise expressions specified graphically as an UML class diagram. To be consistent with the concept of object oriented programming we call the metamodel's components classes which are shown as yellow boxes in figure 2.5. Instances of these classes are referred to as objects. Relations between classes are illustrated as arrows (except the primitive relation) and in the scope of this thesis four different types of relations are supported:

- **Primitive Relation:** A class can have certain properties called attributes which we refer to as primitive relation. Note that each attribute has a datatype associated. For example the class `Constant` has a primitive relation to `Value` of datatype `int`.
- **Composition Relation:** The composition relation models a parent-child relation which means the parent object is composed of its child objects. For example an object of the class `And` can have an arbitrary number of child objects which themselves are instances of the classes `Not, And, Or, Xor, Constant` or `Variable`.
- **Generalisation Relation:** The generalisation relation is equivalent to concept of inheritance in object oriented programming. For example the classes `Constant`, `Variable`, `Not`, `And`, `Or` and `Xor` are all inheriting from the `Expression` class which means they inherit the primitive relation (attribute) `Name`.
- **Association Relation:** The association relation models a classic reference or pointer to another object of a certain class. Note that this relation is not illustrated figure 2.5.

The rootnode `MetaExpression` consists of exactly one `ExpressionOption` which is a placeholder for all possible expressions `Not`, `And`, `Or`, `Xor`, `Constant` and `Variable`. This is shown as a compositional relation starting from the class `MetaExpression` and ending in `ExpressionOption`. The fact that it consists of exactly one of the options mentioned above is indicated by the *Multiplicity*, the symbol corresponding to the arrow of the composition relation. Valid multiplicities are: 0..1 (zero or one), (1 exactly one), * (arbitrary), 1..* (arbitrary but minimum one). The classes `And`, `Or`, `Xor` can have an arbitrary number of expressions as arguments (*) where `Not` can only have one (1) and `Constant` or `Variable` are not allowed to have any.
Note: `And`, `Or` or `Xor` expressions with only one or zero arguments are not reasonable such that further consistency checks have to be included to prevent that from happening.

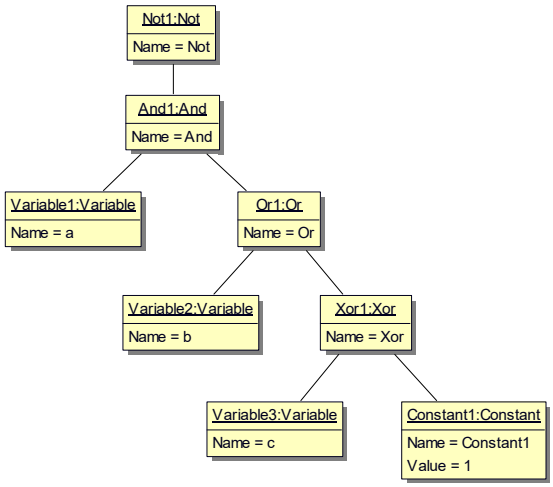


Figure 2.6.: Sample instance of the metamodel `MetaExpression`

Figure 2.6 shows an object diagram which represents a model defined by `MetaExpression`. The model represents the bitwise expression $\neg(a \wedge (b \vee (c \oplus 1)))$. Any other expression containing the operators specified in the metamodel `MetaExpression` can be expressed as an instance of the former.

2.2.2. Metamodel-based Automation Framework Metagen

Infineon uses *Metagen*, a metamodeling and code generation framework implemented in Python, successfully in the fields hardware and firmware generation as well as verification for the last eight years [9]. The structure of the Metagen framework is shown in Figure 2.7.

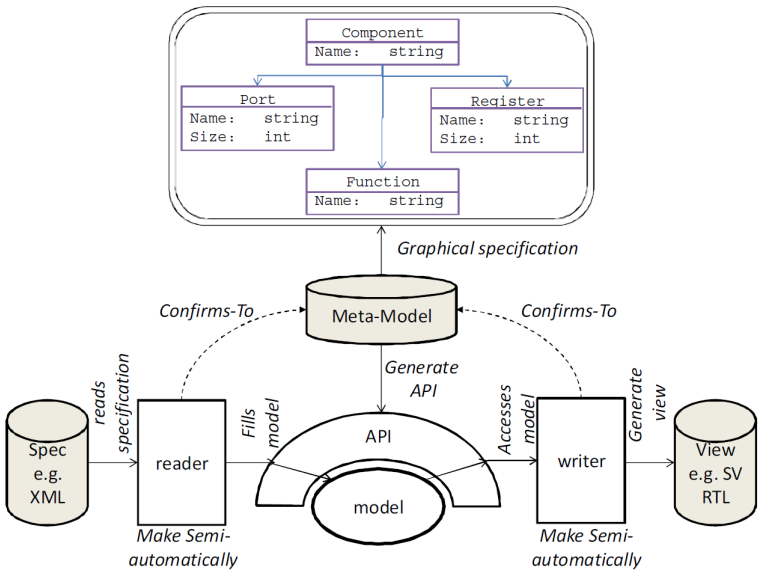


Figure 2.7.: Structure of the Metagen framework [9]

To make use of the Metagen framework one has to start off by specifying a metamodel. Assume a certain metamodel is given, shown in figure 2.7 as graphically specified. The UML diagram of the metamodel is parsed by the Metagen framework which then automatically generates an Application Programming Interface (API) for this specific metamodel. The intention of the API is simplifying the access to models by providing a set of clearly defined methods, e.g. getter and setter methods. To enter data or in other words to populate the model Metagen provides two procedures: First, the Engineer implements a reader which parses information given in the specification and fills the model automatically. Or second, the Engineer enters data directly over a Graphical User Interface (GUI) which is also automatically provided by Metagen for the underlaying metamodel. The writer accesses the model through the API and generates target code, a so-called view. The view can be of any arbitrary target language, e.g. RTL Code, SVA, ITL or XML but also a documentation. To do so the writer uses the template engine *Mako* which uses pythonic control structures (loops and conditionals) and other high-level generation pragmas to generate target code in a fast and compressed way. We call reader and writer semi-automatic as their implementation is largely assisted by the automatically generated API but still need to be written manually by the Engineer.
Using the *Metagen* framework for code generation provides advantages which are explained in the following:

- Once the reader is implemented for a specification it can be reused for altered specifications with the same syntax. This becomes clear if we reconsider the metamodel `MetaExpression` which is used to defined models representing bitwise expressions. Assume a reader is given capable of reading in specifications written in

a mathematical syntax like $\neg(a \wedge (b \vee (c \oplus 1)))$. The same reader is able to parse any arbitrary bitwise expression in the same syntax, e.g. $a \wedge b$ or $\neg a$, without further effort. As noted above the implementation of the reader can be achieved in short development time as it is largely assisted by the APIs.

- It is possible to implement multiple writers, each of them generating views in different target languages. Assume a writer for VHDL RTL code is given, the view for the specification mentioned above $\neg(a \wedge (b \vee (c \oplus 1)))$ looks like `not(a and (b or (c xor 1)))`. By adding a further writer which is for example used to generate Python code, the Python target code can be generated without further changes in the framework.
- As already explained the Engineer has to provide his own metamodel, reader as well as writer which leads to the fact that Metagen can be integrated smoothly into design flows.

So far the Metagen framework has been applied for synthesizing digital hardware as well as in the areas of analogue hardware and firmware, also covering the interfaces between these areas. Verification and documentation views have already been addressed as well. Savings of up to 95% in a single design step and up to 70% in the overall implementation of a chip has been measured [9].

2.2.3. Application of Metagen for Formal Hardware Verification

This section explains the property automation framework proposed in [8], in the following called *Metaprop*, in detail because the thesis' contribution is to extend the former. The intention of the Metaprop flow is to increase verification productivity such that the verification gap is reduced. To do so the concept of hardware generation languages which has already shown promising results is adopted for verification, namely by raising the abstraction level from property description to property generation. The other benefit besides verification productivity gains is that Metaprop is able to generate properties in multiple target languages, e.g. SystemVerilog Assertions (SVA), Property Specification Language (PSL) or Interval Language (ITL).
The Metaprop flow combines the OMG's MDA principle for code generation with the Metagen framework. The key concept of MDA are model-to-model transformations between the computational independent model, the platform independent model and the platform dependent model which denote the different viewpoints of abstraction. The structure of each of these models is defined by a metamodel in the Metagen environment to benefit from the advantages provided by the Metagen framework.

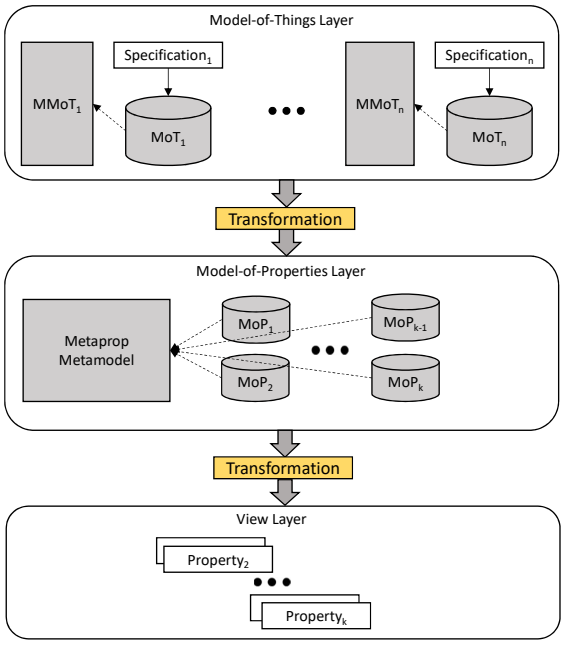


Figure 2.8.: Metaprop: A model-driven property automation framework [8]

Figure 2.8 provides an overview of the proposed flow. The Metaprop flow consists of the three viewpoints of abstraction layers defined by MDA from top (high) to down (low) and two model-to-model transformations illustrated as yellow boxes.
The upmost layer is called Model-of-Things (MoT) layer and corresponds to the computational independent model in terms of abstraction. In this layer the informal specifications $(\text{Specification}_1, \ldots, \text{Specification}_n)$ written in natural languages are captured formally as a model in the Metagen environment which are in the scope of this thesis referred to as MoTs. The structure of the MoT is defined by the underlaying Meta-Model-of-Things (MMoT). The MMoT captures high-level attributes and acts as close as possible to the informal specification. Different MMoTs are needed to capture different types of specifications. For example specifications for bitwise expressions can be captured with the MMoT from figure 2.5 but for capturing specifications describing hardware designs the former cannot be used but different MMoTs have to be generated. The Verification Engineer analyzes the informal specification written in a natural language like German or English and generates a suitable MMoT for this specification. Afterwards an instance of this MMoT is generated and gets populated with data from the informal specification which then results in the MoT.
By applying the first model-to-model transformation the MoT is converted to the Model-of-Properties (MoPs) which correspond to the platform independent models with respect to the MDA approach. The transformation can be understood as refinement as further information about the hardware, e.g. time intervals or signals names, are added which lead to a lower abstraction level. The MoPs are the core of the flow and act as an

abstract and platform independent way of describing temporal logic expressions. Each MoP is a model for one temporal logic expression where the MoP's strucutre is defined by the Metamodel-of-Properties (MMoP). Please refer to appendix A.1 for a full graphical represenation of the MMoP.
By invoking the second model-to-model transformation the MoPs are refined to the corresponding views or properties in the desired target language. Each temporal logic expression represented as MoP is mapped to exactly one interval property. The Metaprop flow is complete as interval properties written in a specific target language which correspond to the platform dependent models with respect to the MDA principle are generated.

2.2.4. Automated Property Generation for Example Design

As the contribution of this thesis is to extend the existing Metaprop flow this section applies the Metaprop framework to an example design called 'Cascaded Counters' to provide an in-detail understanding of the former. The informal specification of the design is given in the following:

> *The design consists of four counters which are connected as shown in figure 2.9. Each counter has a maximum value 'max_value' which is an input but constant. Every time the input 'rst_cnt' is HIGH the output 'cnt_out' is increased by the input 'add_in' in the next clock cycle. If the counter reaches the maximum value 'max_val' the output 'Counter_X_OUT' should be resetted to '0' the next time input 'ovf_out' is HIGH. As the first counter does not have a previous counter driving it the output 'Counter_A_OUT' should increment with every clock cycle. If the reset signal 'rst' is HIGH all counters should be resetted and outputs should be zero.*

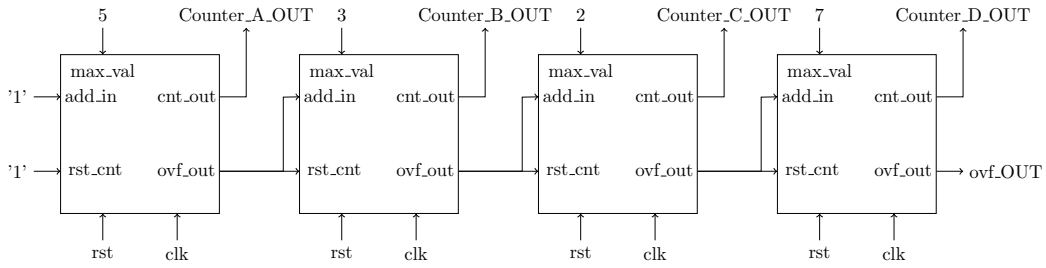
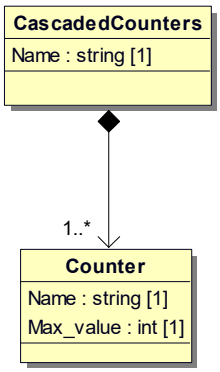


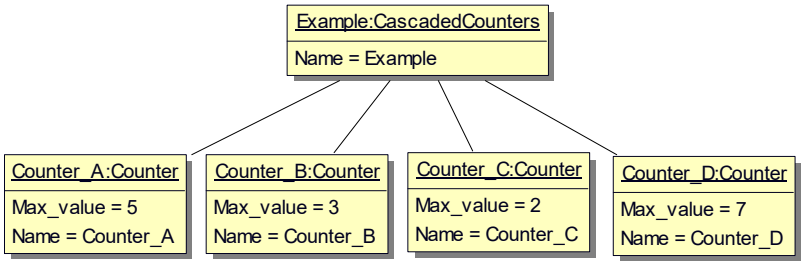
Figure 2.9.: Block diagram of design 'Cascaded Counters'

As a first step the informal specification has to be captured formally as a MoT. To do so the MMoT shown in Figure 2.10a is created. The root node `CascadedCounters` can have an arbitrary number of `Counter` objects but at least one indicated by the composition relation with multiplicity 1..*. Each `Counter` itself has a `Name` and a `Max_value` as

attributes. The MoT is generated by entering the needed parameters from the informal specification into the GUI provided by Metagen. Note that we rather use the GUI in this situation than implementing a reader due to lower effort. The resulting MoT for the given informal specification can be seen in figure 2.10b. As a next step the model-to-model transformation from the MoT to the MoPs has to be implemented. As discussed, the transformation adds further information from the informal specification because the MMoT captures only structural high-level attributes of the counters, e.g. their maximum value. The MMoT however does not provide any information about how these counters are connected with each other. This additional information needs to be provided during the model-to-model transformation. As soon as the transformation is implemented the Metaprop flow is complete in the sense that interval properties can be generated from a given MoT. Metaprop currently provides writers for ITL as well as SVA syntax. The generated properties are appended in appendix B in both target languages



(a) MMoT 'Cascaded Counters'



(b) MoT for given informal specification

Note: Assume parameters of the informal specification are changing, e.g. the counters' maximum value or the number of counters in general, the only thing which needs to be adjusted to generate a new set of properties for the altered specification is to recreate the MoT. This task is of low effort because the process is assisted by the GUI provided by the Metagen framework.

2.3. Abstraction Technique Path Predicate Abstraction

In a general design flow there exists a trusted chain of sound transformations between the involved abstraction layers, namely from transistor level to gate level and from gate level to the Register Transfer Level (RTL). A transformation being sound means in this context that the model at one abstraction layer produces output sequences for all possible input sequences identical to the output sequences produced by the model at the next lower abstraction level. The former can be verified for example by formal equivalence

checking, e.g. applied for the RTL representation and gate level representation of a certain implementation. Such a notion of soundness does **not** exist between design models at ESL and RTL. At the ESL, communication between modules is modeled abstractly using untimed messages whereas communication at the RTL is specified in a bit as well as clock-cycle accurate manner. The concept of Path Predicate Abstraction (PPA) describes a well-defined formal relationship between abstract system-level models and their concrete RTL implementations which changes the role of a system-level model fundamentally. By defining system-level models as PPAs of RTL implementations they may be trusted as sound abstractions of the former in the same way RTL implementations are trusted to be sound abstractions of the underlaying gate level.
The idea of the PPA is introduced by defining such a PPA for directed graphs. The generation of a PPA for a directed graph is based on *operational graph coloring* which means finding a *coloring function* to assign specific colors to a set of nodes of the directed graph. These colored nodes are considered to be the important states of the PPA whereas the uncolored nodes get abstracted. To aid the definition of operational graph coloring we also define *operational path segments*.

Definition 1 (Operational Graph Coloring [36]) *Consider a directed graph* $G = (V, E)$*, a subset* $W \subseteq V$ *of the graph vertices called colored nodes, a set of colors* $\hat{W} = \{\hat{w_1}, \hat{w_2}, \dots\}$ *and a surjective coloring function* $f_c \colon W \mapsto \hat{W}$*.*

- *A path segment* $(v_0, v_1, \dots, v_n)$ *such that* $v_0, v_n \in W$ *and* $v_1, \dots, v_{n-1} \in V \setminus W$ *is called operational path segment in* G*.*
- *The set* W *must be chosen and colored such that:*
  1. *every cyclic path in* G *contains at least one node from* W *(no cycles with uncolored nodes in the graph),*
  2. *for every operational path segment* $(v_0, v_1, \dots, v_n)$ *and* $u_0 \in W$ *such that* $f_c(u_0) = f_c(v_0)$ *there must exist an operational path segment* $(u_0, u_1, \dots, u_n)$ *in* G *with* $f_c(u_n) = f_c(v_n)$*.*

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

Figure 2.11 shows an example of an operationally colored graph according to definition 1.

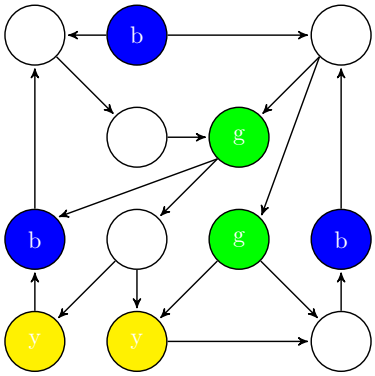
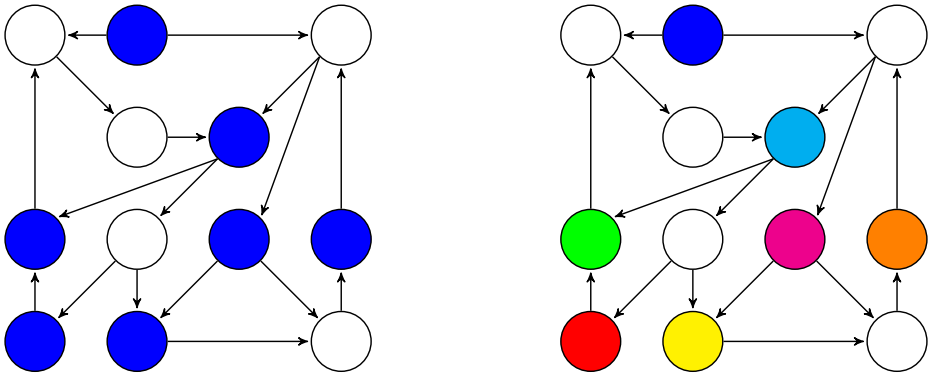


Figure 2.11.: Operationally colored graph

The nodes colored in 'blue' (b), 'green' (g) and 'yellow' (y) are elements of the set $W \subseteq V$, whereas the uncolored nodes displayed as white do belong to $V \setminus W$. To verify that the directed graph is correctly operational colored we need to check if the two conditions of definition 1 are fulfilled. The first condition is satisfied as there are no cyclic paths containing only uncolored nodes. The second condition can be checked easily as well. Consider **all** nodes of a specific color $v_i \in W$ as well as all operational path segments $(v_i, v_{i,1}, \ldots, v_{i,n})$ starting from the node v_i. For each of these nodes v_i the set of reachable colors $\hat{W}_{v_i} = \{w_j \mid w_j = f_c(v_{i,n})\}$ needs to be equivalent $\hat{W}_{v_i} = \hat{W}$ to satisfy condition 2. To make it more clear the above is applied for the color 'blue'. The operationally colored graph in figure 2.11 contains three nodes which are colored in 'blue'. **Each** of these three nodes v_i with $f_c(v_i) =$ 'blue' has operational path segments to nodes colored in 'green' as well as 'yellow'. This means that the set of reachable colors is equivalent for all nodes of color 'blue' v_i, $\hat{W}_{v_i} = \hat{W} =$ {'green', 'yellow'}, such that condition 2 holds for all operational path segments starting from a 'blue' node. Repeating the above for nodes colored in 'green' and 'yellow' leads to a satisfaction of condition 2 and therefore the graph is correctly operationally colored according to definition 1.
Note: The operationally colored graph from figure 2.11 is not unique due to the fact that the operational coloring function is not unique itself. There exist two parameters which can be altered: $k = |W|$ the number of colored nodes and $l = |\hat{W}|$ the number of used colors. For the parameter k any value in the range $k_{min} \leq k \leq |V|$ can be provided where k_{min} is the minimum number of nodes to be colored such that the first condition holds. By inspection of figure 2.11 k_{min} can be determined as $k_{min} = 2$, namely coloring the nodes shown in 'green' only. The parameter l needs to be in the range $1 \leq l \leq k$ which does **not** mean that an operational coloring function f_c does exists for each specific l. Figure 2.11 shows an operational graph coloring for the parameters $k = 7$ and $l = 3$. Figure 2.12 shows two trivial cases of valid operational graph colorings where in (a) all colored nodes are of same color and in (b) each node has its unique one. For both cases definition 1 can be checked easily in the same procedure as explained above.

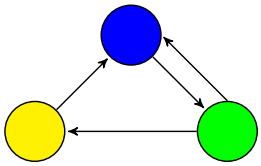


(a) $k = 7, l = l_{min} = 1$ (b) $k = 7, l = l_{max} = k = 7$

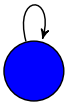
Figure 2.12.: Operational graph colorings for different $l = |\hat{W}|$

For the operational colored graphs from figure 2.11 and 2.12 more abstract graphs can be drawn considering the colored nodes only. To do so a single node for each color and an edge for each operational path segment is added to the respective abstract graph. Hence, every operation, such as blue $\mapsto$ green, represented by several edges in the original graph is represented by a single edge in the abstract graph. These abstract graphs are called path predicate abstractions and can be seen in figure 2.14.

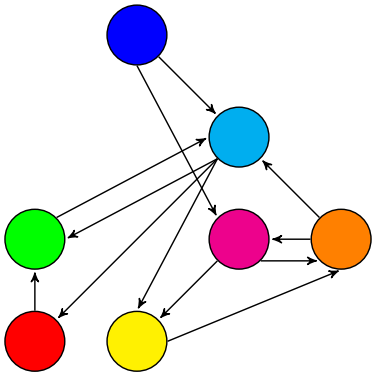
Definition 2 (Path Predicate Abstraction [36]) *We consider a graph $G = (V, E)$ with a set of colored nodes $W \subseteq V$, a set of colors $\hat{W}$, an operational coloring function $f_c \colon W \mapsto \hat{W}$. A directed graph $\hat{G} = (\hat{W}, \hat{E})$, such that for any two nodes $u, w \in W$, it is $(f_c(u), f_c(w)) \in \hat{E}$ if and only if there is an operational path segment $(u, \ldots, w)$ in G, is called a graph path predicate abstraction of G.*



(a) For figure 2.11



(b) For figure 2.12a



For figure 2.12b

Figure 2.14.: Graph predicate abstractions for operational graph colorings.

The following soundness theorem states the preservation of color sequences which means connecting the paths in the concrete graph with the paths of the abstract graph. First we defined the term color sequence:

Definition 3 (Color Sequence [36]) *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 $f_c : W \mapsto \hat{W}$. A color sequence produced on a path $(s_0, s_1, s_2, \dots)$ in G is the sequence of colors $(\hat{w}_0, \hat{w}_1, \hat{w}_2, \dots)$ on the path where the i-th color in the sequence $\hat{w}_i = f_c(s_j)$ if and only if $s_j \in W$ is the i-th colored node on the path. Note that $i \leq j$ since uncolored nodes $s \notin W$ on the path do not contribute to the color sequence.*

Theorem 1 (Soundness with regards to color sequences [36]) *Let $G = (V, E)$ be a graph and $\hat{G}$ be the path predicate abstraction induced by an operation graph coloring $f_c : W \mapsto \hat{W}$ with a set of colored nodes $W \subseteq V$.*

1. *Given an arbitrary finite (infinite) concrete path $(v_0, v_1, v_2, \dots)$ in G with $v_0 \in W$. There exists a finite (infinite) abstract path $(\hat{w}_0, \hat{w}_1, \hat{w}_2, \dots)$ in $\hat{G}$ that represents the color sequence produced on the concrete path.*

2. *Given an arbitrary finite (infinite) abstract path $(\hat{w}_0, \hat{w}_1, \hat{w}_2, \dots)$ in $\hat{G}$. For every node $v_0 \in V$ in G such that $f_c(v_0) = \hat{w}_0$ there exists a finite (infinite) concrete path $(v_0, v_1, v_2, \dots)$ such that the color sequence produced on that path is $(\hat{w}_0, \hat{w}_1, \hat{w}_2, \dots)$.*

Proof: The first part of theorem 1 is satisfied by construction. By applying the operational coloring function f_c to the nodes of the conrete path $(v_0, v_1, v_2, \dots)$ every colored node $v \in W$ is mapped to its color $\hat{w} = f_c(v)$ whereas all uncolored nodes get abstracted. This leads to the abstract path $(\hat{w_0}, \hat{w_1}, \hat{w_2}, \dots)$ in $\hat{G}$ that represents the color sequence produced on the concrete path.
To prove the second part of the theorem we consider a single edge $(\hat{w_i}, \hat{w_{i+1}})$ of an arbitrary abstract path $(\hat{w_0}, \hat{w_1}, \hat{w_2}, \dots)$. According to requirement 2 of definition 1 there exists an operation path segment $(v_i, v_{i,1}, v_{i,2}, \dots, v_{i,l-2}, v_j)$ in the concrete graph for **all** colored nodes v_i of the same color $f_c(v_i) = \hat{w_i}$ to a node of some color $v_j = \hat{w_{i+1}}$. Note that the nodes along the operational path segment $v_{i,1}, v_{i,2}, \dots, v_{i,l-2}$ are of no color. Any arbitrary abstract path $(\hat{w_0}, \hat{w_1}, \hat{w_2}, \dots)$ is composed of its single edges $(\hat{w_0}, \hat{w_1}), (\hat{w_1}, \hat{w_2}), \dots$. By concatenating the corresponding operational path segments of the former a concrete path $(v_0, v_1, v_2, \dots)$ which produces the color sequence $(\hat{w_0}, \hat{w_1}, \hat{w_2}, \dots)$ is generated.
The theory of PPA can be extended to make it applicable for FSMs which means that a PPA can be generated for an arbitrary FSM. As the behaviour of any sequential circuit can be represented as FSM it follows that any arbitrary sequential circuit can be abstracted to a PPA. Note that we do not explain the full formalism shown in [36] but rather give a short overview how the PPA can be constructed from an arbitrary FSM.
The approach of constructing a PPA for FSMs is equivalent to the one shown for directed graphs. As a first step the concept of operational coloring is adopted to FSM theory namely by defining operational coloring for FSMs. This operational coloring consists of three seperate individual coloring functions, namely a *State Coloring* function which assigns a *state color* $\hat{s} \in \hat{S}$ to a subset of the FSM states, an *Input Coloring* function mapping input sequences to *input colors* $\hat{X}$ and an *Output Coloring* function mapping output sequences to *output colors* $\hat{Y}$. Abstract transitions are defined according to a transition function $\hat{\delta} : \hat{S} \text{ x } \hat{X} \mapsto \hat{S}$ where each transition relates to an abstract operation. Abstract operations are triggered by a concrete input sequences which correspond to the according input color of the set $\hat{X}$ and produce a concrete output sequence represented as an output color of set the $\hat{Y}$ which is defined in the output function $\hat{\lambda} : \hat{S} \text{ x } \hat{X} \mapsto \hat{Y}$. Combining the elements described above as a tuple results in the abstract FSM $\hat{M} = (\hat{S}, \hat{s_{reset}}, \hat{X}, \hat{Y}, \hat{\delta}, \hat{\lambda})$ which is also referred to as PPA for FSMs.
In the same way as done for directed graphs it can be shown that the abstract PPA is a sound abstraction from the concrete FSM with respect to color sequences. The preservation of color sequences can be extended to a preservation of LTL formulas, please refer to [36] for the complete formalism and proofs. Note that in theory this enables performing formal verification techniques directly on the abstracted PPA which makes complex proofs become computationally feasible due to the fact that they are performed on an easier and more abstract model. In practice however there exists no formal verification tool which accepts PPAs as design input. To circumvent that we produce a complete set of operation properties describing the full behaviour of the PPA which is shown in chapter 4 in detail.

3. Specification of the Tasks

The goal of hardware verification is to prove that a given implementation of a design conforms with its specification. The formal verification technique IPC uses interval properties to capture the informal verification in a formal way. The formal verification tool takes these properties and the implementation itself as an input and proofs whether the properties hold for that certain implementation. Metaprop as a novel property automation framework presented in chapter 2.2.3 shows verification productivity gains through property automation to reduce the verification gap. But there exist drawbacks which we explain in the following. After showing these drawbacks we introduce the contribution of this thesis, namely an extended version of the Metaprop framework which solves the problems mentioned in the following.

- **Multiple MMoTs for different specification classes:** As a first step to make use of the Metaprop flow the Verification Engineer needs to analyze the informal specification, extract its necessary properties and cast them to a MMoT. The MMot 'Cascaded Counters' shown in chapter 2.2.4 is suitable for specifications describing inter-connected counters with altering number of counters but cannot be used to capture properties of a CPU for example. Creating the MMoT is a manual process needed to be done for every specification class which can be time consuming.

- **Incomplete set of properties:** As described in the previous point the MMoT is specific for a class of specifications. This leads to the fact that the model-to-model transformation from MoT to MoP layer of the Metaprop framework needs to be specific for this class of specifications as well as it takes the MoT as an input. This transformation takes data stored as MoT as well as further information from the informal specification needed for refinement as inputs and generates abstract temporal Boolean expressions which we refer to as MoPs. This transformation has to be manually implemented for each MMoT which can be a challenging task to extract all necessary information from the informal specification such that no functionality is missed during this process. If necessary information is missed during the transformation step it leads to incomplete abstract temporal Boolean expressions which then lead themselves to an incomplete set of properties with respect to the specification. In other words, desired functionality may not be covered in the generated set of properties and therefore cannot be proven on the design's implementation.

- **MoTs are not simulation capable:** As already mentioned, the first step to

make use of the Metaprop framework is to create a MMoT to capture the informal specification formally as MoT. Simulation of the formal specification is used to gain confidence whether the formal specification as executable model exactly represents the informal specification. As the MoT is a data structure only defined by the MMoT it is **not** simulation capable.

We propose an extended version of the property automation framework Metaprop which circumvents the drawbacks explained above. An overview over the former is shown in figure 3.1.

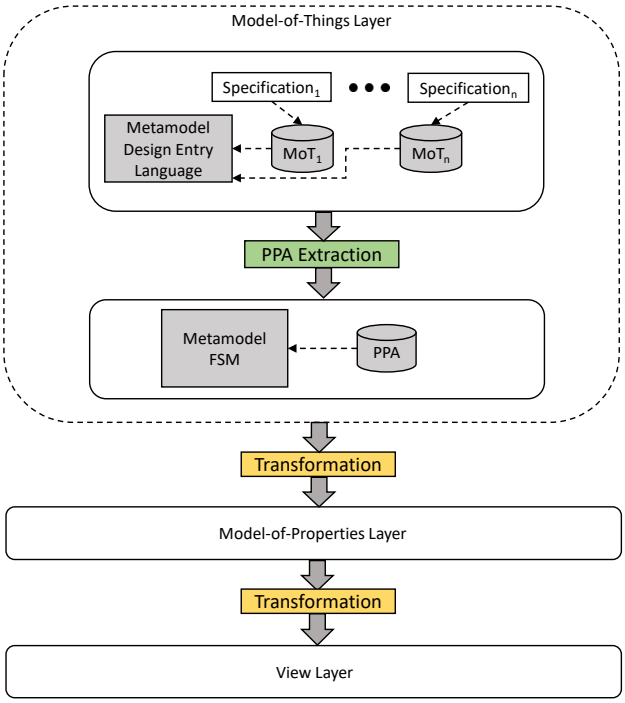


Figure 3.1.: Extended version of the property automation framework Metaprop

Equivalent to the original Metaprop flow there exist the same three layers of abstraction derived from Model-Driven-Architecture. The Model-of-Properties layer as well as the View layer stay untouched with respect to the original framework. The Model-of-Things layer however is represented by two different metamodels, namely Metamodel Design Entry Language (MDEL) and Metamodel FSM (MFSM). The verification engineer reads the informal specification ($\text{Specification}_1, \ldots, \text{Specification}_n$) and converts it to formal specification written in a simulation capable language which is refered to as design entry language. This step may take several iterations as the Verification Engineer needs to verify by simulation that the informal specification conforms the executable model. The executable models which represent the formal specifications are captured as instances of MDEL as $\text{MoT}_1, \ldots, \text{MoT}_n$. MDEL is a EBNF like description of the design entry

language which defines the syntax of the former. Note that any formal specification written in the syntax defined by the MDEL can be mapped to a MoT of this metamodel which implies that only one single metamodel is needed to capture all informal specifications. The MDEL matches precisely the abstraction level of Path Predicate Abstraction which means that the executable model is an implicit representation of the PPA. To extract the explicit PPA description corresponding to definition 2 from the executable model we interpret this model as Control-Flow-Graph (CFG) of successive statements defined in the syntax of the design entry language. As a CFG is a directed graph, we define an operational graph coloring function for CFGs which matches definition 1. By applying the operational graph coloring function to the CFG, the states and transitions of the PPA are identified (shown as green box) which means the PPA can be understood as abstract FSM. Therfore we capture the PPA as an instance of a general metamodel for FSMs (MFSM). The model-to-model transformation from MFSM to Metamodel-of-Properties (MMoP) becomes simple as every transition of the PPA is mapped to exactly one abstract temporal Boolean expression in the Model-of-Properties layer, in other words every transition maps exactly to one MoP. By selecting a view the following model-to-model transformation refines the MoPs to operation properties in the specified target language. The extended version of the Metaprop flow solves the drawbacks mentioned before. First, there exist one unique metamodel, namely MDEL, to capture all types of informal specifications formally. The Verification Engineer describes the behaviour of the system in the syntax of the design entry language which implies that the executable model is an instance of the MDEL. Second, a subset of SystemC called SystemC-PPA is chosen as design entry language which is simulation capable. In other words, occuring mismatches between informal and formal specification or missing functionalities can be spotted early in the verification process by simulating the executable formal specification. Third, the generated set of properties is complete by design. As explained above the formal specification written in SystemC-PPA syntax implicitly defines the PPA. We assume the executable model representing the formal specification to be bug-free as well as complete as it can be verified by simulation with regards to the informal specification. The PPA is extracted and contains all possible paths of the executable model and is therefore complete as well as bug-free by design as well. As noted in chapter 2.3 the PPA can be understood as an abstract FSM which is a sound abstraction of the concrete FSM capturing the behaviour of any arbitrary sequential circuit. This implies that this specific PPA which is assumed to be complete and bug-free is a sound abstraction of the concrete FSM describing the sequential behaviour of the complete and bug-free RTL implementation. The set of operation properties derived from this PPA is therefore complete by design.

To measure the quality and check the completeness of the generated set of properties we apply the extended property automation framework for the I^2C-bus protocol. As a first step we analyze the informal specification of the I^2C-bus protocol and converted it into an executable formal specification written in SystemC-PPA syntax. The executable

formal specification is fed into the extended Metaprop framework to generate a set of operation properties. Afterwards we use the set of generated properties and an exisiting I^2C-bus protocol implementation which is used within Infineon as inputs for the formal verification tool OneSpin [30] to perform IPC. To measure the quality of the generated properties we inspect the structural metric code coverage. To prove completeness of the property suite we use the proposed completeness criterion which can be automatically be proven by performing successor test, case split test, determination test and reset test.

4. Extended Property Automation Framework

With this chapter we give an in detail explanation of the extended property automation framework with special interest on the updated Model-of-Things layer because the Model-of-Properties layer as well as the View layer stay unchanged with respect to the original Metaprop flow from chapter 2.2.3. To run the extended Metaprop flow a formal specification which is parsable by the Metagen environment is needed. Chapter 4.1.1 demonstrates the problems of informal specifications and why these cannot be used for automated property generation. To solve the demonstrated problems we introduce a system-level modeling language in chapter 4.1.2 called SystemC-PPA which is used as design entry language for modeling the informal specification formally as executable model. SystemC-PPA is a subset of SystemC associated with special semantics which we refer to in detail in this chapter. In chapter 4.2 we present a metamodel for the design entry language called MDEL which defines the available syntax of the SystemC-PPA language to model the formal specification. In chapter 4.3 we explain how the Path Predicate Abstraction can be extracted from the MoT which is the formal specification represented as an instance of the MMoT MDEL. Finally we present in chapter 4.4 a second MMoT MFSM, a general model for FSMs which is used to store the previously extracted PPA. By applying the model-to-model transformation from MFSM to the Model-of-Properies layer the extended Metaprop framework is complete in the sense that a set of operation properties can be generated. This is possible due to the fact that we reuse the MoP layer as well as the View layer from the original Metaprop framework without any changes.

4.1. Modeling the Formal Specification in the Design Entry Language SystemC-PPA

In this chapter we describe the first step to run the proposed extended property automation framework, namely getting from the informal specification to a formal specification which is parsable by the Metaprop framework. In general, a specification describes the desired behaviour of a system. The most important goal besides being correct in terms of the described functionalities is to be complete with regards to the system's intended behaviour. Missing some functionalities in the specification means that these functionalities cannot be proven on the system's implementation.

4.1.1. Problems of Informal Specifications

Generally the informal design specification is provided in any natural language, e.g written English or German. The problem of informal languages is that their syntax is not well defined which leads to several problems which are illustrated with the following informal example specification [10].

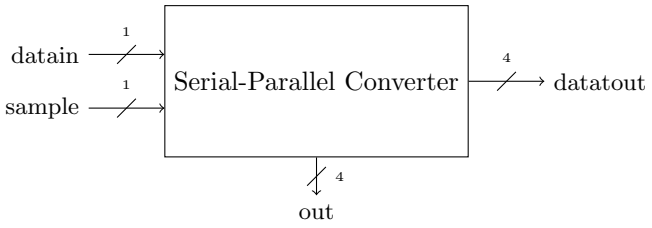


Figure 4.1.: Blockdiagram 'Serial-parallel Converter'

> *The input 'datain' accepts a stream of bits and the output 'dataout' emits the same stream delayed by four clock cycles. The output 'out' is four bits wide. If the input 'sample' is the value LOW then the 4-bit word at 'out' is the last four bits input at 'datain'. If 'sample' is HIGH then the word at 'out' is zero or in other words four 'LOWs'.*

This informal specification is not suitable as an input for the extended Metaprop framework or for formal hardware verification in general because it is

- **vague:** For example it is not properly specified whether the 'last four bits input' include the current bit.
- **incomplete:** The specification does not determine the 'dataout' during the first three clock cycles.
- **not executable:** The specification cannot be compiled into code for simulation.
- **not parsable:** The text is unstructured and therefore hard to analyze.

For the sake of automation a parsable specification written in a formal language is a precondition for the proposed methodology such that the formal specification can be read as an input in a easy and automated way.

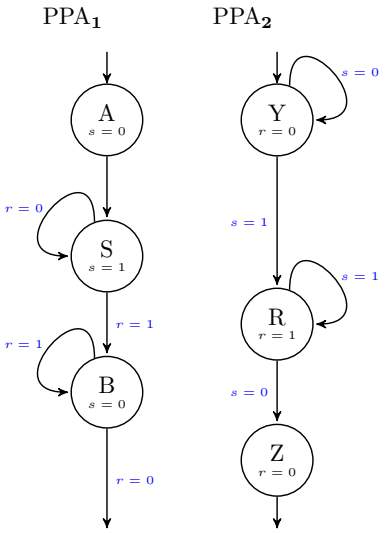
Definition 4 (Formal Language) *A language is called formal if it is a mathematically-based language with a clearly defined syntax as well as semantics. As it has clearly defined syntax there exists a parser to analyze the formal language.*

In the scope of this thesis SystemC is chosen as design entry language to describe the specification formally. SystemC is a system-modeling language which is implemented as a C++ class library for system and hardware design [17]. Using the additional SystemC library makes it possible to describe concurrent processes whereas the basic C++ language is purely sequential. There are mainly three reasons for choosing SystemC as formal design entry language. First, as SystemC is a formal language it solves the problems caused by specifications written in informal languages noted above. A SystemC model is simulation capable and therefore can be verified with regards to the informal specification to spot incompleteness or mismatches between formal and informal specification (*executable, complete*). As SystemC uses C++ syntax its syntax as well as its semantics are well defined (*explicit, parsable*). Second SystemC is widely used in industry due to the already known C++ syntax. And third, SystemC uses the concept of Transaction-Level Modeling (TLM) which is a high-level approach to model communication between modules. Communication is modeled as time-abstract passing of messages where the communicating modules synchronize with each other using events. As unnecessary communication details are hidden inside these events simulation speed increases because the simulation runs at a higher level of abstraction.

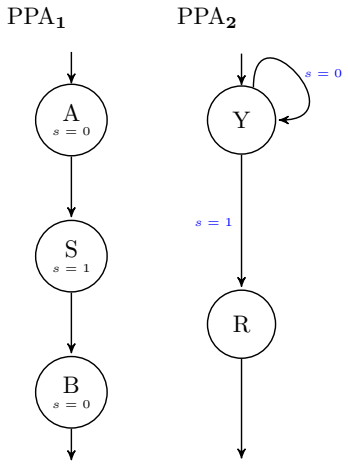
4.1.2. Semantics of the SystemC-PPA subset

The informal specification is modeled formally as a SystemC model which means specifying the desired hardware behaviour in SystemC syntax. As SystemC is primarily a software programming language some high-level objects of the SystemC class framework lack a clear semantics with respect to the abstract hardware system. The C++ `std::vector` class is an example for such an object as it is a container for arrays of dynamically adjustable size. A high-level object dynamically allocating memory does semantically not exist as an abstract hardware construct. By restricting the SystemC syntax to a subset called SystemC-PPA [25] these objects lacking precise semantics in hardware are excluded.
In the following we explain the semantics of SystemC-PPA. In SystemC one describes functionality hierarchically separated as modules which interact with each other on system-level using communication events based on TLM. Each module contains processes which run concurrently and model the module's functionality. In SystemC-PPA the semantics of modules is extended in the sense that each module describes a PPA which is a time-abstract FSM consisting of states and transitions. The states of the PPA correspond to the communication events of the SystemC-PPA module and the transitions to the paths connecting these events. Combining several modules to model system-level behaviour results in an asynchronous product of time-abstract FSMs where each FSM is defined by its SystemC-PPA description as a module [35]. As each PPA is time-abstract, every PPA can have its own clock speed associated. This implies on system-level where these PPAs need to communicate with each other the need of synchronisation mechanisms.

In SystemC-PPA synchronisation is implemented via three different types of handshaking mechanisms which are explained in the following:



(a) Four-phase handshake for asynchronous communication



(b) Easied handshake for unilateral synchronisation

Figure 4.2.: Handshaking mechanism provided in SystemC-PPA for synchronisation

- **Blocking:** The blocking communication inferface implements a full four-phase handshake which ensures that messages are never lost. To explain the four-phase handshake in detail we use figure 4.2a which illustrates state sequences of two communicating PPAs (inputs are shown in blue and outputs inside the states). Assume PPA_1 and PPA_2 want to exchange some data. To do so they need to synchronize with each other. Assume initially PPA_2 waits in state Y when PPA_1 moves from state A to state S where the synchronisation signal s is asserted, possibly together with some data. The synchronisation signal s to be true triggers PPA_2 which moves into state R where the received signal r is set, again possibly together with some data. The fact that PPA_1 needs to wait in its sending state S, which implies s as well as the data to stay asserted until r is asserted, means that this message from PPA_1 to PPA_2 is never lost. Vice versa PPA_2 waits in state R (data and r stay set) until s is de-asserted (PPA_1 is in state B and has received the message) which means the message from PPA_2 to PPA_1 was successfully exchanged. Note that the waiting condition $r = 1$ in state B for PPA_1 is necessary otherwise a new message sent during some state sequence $(B, \ldots, A, S, B)$ could remain unrecognized if PPA_2 stays in state R during this time. This is called a four-phase handshake because two events for PPA_1 $(s = 0, s = 1)$ and PPA_2 $(r = 0, r = 1)$ needs to happen until the next transaction can be processed.

- **Master/Slave:** There are cases where a four-phase handshake is unnecessary effort, e.g if one side is always ready to communicate which is called slave. This is illustrated in figure 4.2b where PPA_1 acts as a master and PPA_2 acts as a slave. For unilateral synchronisation the slave does not have a synchronisation signal r as it is assumed to be always available for communication. This eases the handshake mechanism as shown in figure 4.2b
- **Shared:** The shared communication interface does not provide any handshaking mechanisms and models a simple hardware port.

To make it more concrete we use an example formal specification consisting of exactly one module such that the system-level behaviour is equivalent to the module's behaviour. Figure 4.3 contains the formal specification of an 8-bit serializer as the SystemC-PPA model. We use this working example throughout the whole chapter to explain every intermediate step of the extended Metaprop flow in detail.
Lines 1 to 18 from figure 4.3 refer to structural information about the SystemC-PPA module, namely the module header, the constructor as well as datatype, port or variable declarations. Note that the ports `data_in` and `bit_out` use the blocking communication interface which implements the four-phase handshake mechanism described above. The module's behavioural information is contained in the lines 20 to 36, namely in the `fsm()` process. Note that, as already indicated by the name of the process, it describes the desired hardware behaviour in a FSM like manner. The process is composed of two sections `get_data` and `serialize_data`. In the `get_data` section an integer value is read from the port `data_in` and stored in the variable `data_reg`. As the port uses the blocking communication interfaces the execution of the model blocks as long as data is available at this port. Data from the `data_reg` is serialized by shifting with regards to the value of the `bit_counter` starting with the most significant bit. Again the port `bit_out` uses the blocking interface which implies that the execution blocks until the value of `bit` is successfully written. After shifting eight times (`bit_counter == 8`) execution starts again with reading in new data to be serialized.
Applying the SystemC-PPA semantics to this model leads to the PPA shown in figure 4.4 consisting of two states because there exist two communication events in the description (line 25 and line 31 of figure 4.3). State S_1 corresponds to the communication event `data_in->read(data_reg)` and S_2 belongs to communication event `bit_out->write(bit)`. Paths between these communication events are compressed as transitions between the former. Both ports use the blocking interface which implements the four-phase handshake shown in figure 4.2a. The blocking behaviour defined by the interfaces of the ports `data_in` and `data_out` in the SystemC-PPA description is included in the PPA with the presence of `sync` and `notify` signals. The `sync` signals can be understood as inputs from the communication counterpart which when asserted imply that the communication partner is ready to communicate such that a communication transaction is started. For example, the serializer stays in the state S_1 until

`data_in_sync` signal is asserted by the communication partner implying data is available at that port. At this point in time the execution stops blocking and reads the available data from the `data_in` port and stores it in the variable `data_reg`. The `notify` signals is the counterpart to the `sync` signal showing the communication opposite that the module is ready for communication at that port. As long as the PPA is in state S_1 the `data_in_notify` is asserted signaling the communication partner which is connected to the port `data_in` on system-level that data can be safely written to this port. The `sync` and `notify` signals implement the four-phase handshake of the blocking communication interface which guarantees that a message is never lost.

```
1  class Serializer : public sc_module {
2  public:
3  SC_CTOR(Serializer) :
4    nextsection(get_data) {SC_THREAD(fsm)};
5  enum Sections {
6    get_data, serialize_data
7  };
8  Sections section, nextsection;
9
10 /* Ports */
11 blocking_in<int> data_in;
12 blocking_out<bool> bit_out;
13
14 /* Variables */
15 int data_reg;
16 bool bit;
17 int test;
18 int bit_counter;
19
20 void fsm() {
21   while (true) {
22     section = nextsection;
23     if (section == get_data) {
24       /* Communication Event */
25       data_in->read(data_reg);
26       bit_counter = 0;
27       nextsection = serialize_data;
28     } else if (section == serialize_data) {
29       bit = ((data_reg << bit_counter) >> 7) == 1;
30       /* Communication Event */
31       bit_out->write(bit);
32       bit_counter = bit_counter + 1;
33       if (bit_counter == 8) nextsection = get_data;
34     }
35   }
36 }
37 };
```

Figure 4.3.: SystemC-PPA description of a serializer

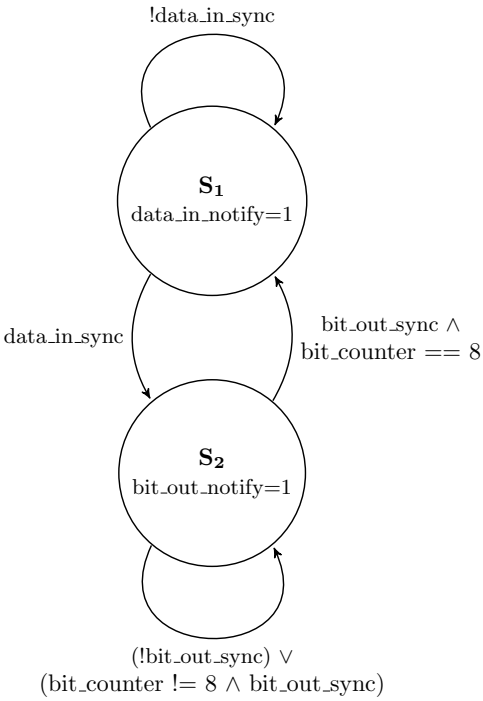


Figure 4.4.: PPA of figure 4.3

The formal specification written in SystemC-PPA could be used directly as an input for the extended Metaprop framework but the Metagen environment where each layer of the Metaprop framework is embedded in does not provide a parser for SystemC syntax intrinsically. To circumvent the need for implementing a SystemC parser we use an open-source tool called SCAM [33] which parses the executable formal specification written in SystemC-PPA based on the LLVM/Clang framework [24] and transforms the model into an abstract syntax tree. SCAM's plug-in system is extended with a plug-in

for Extensible Markup Language (XML) which intents to transform the abstract syntax tree which was generated from the formal specification into XML format. The reason for creating an intermediate representation of the formal specification in XML format is that the Metagen environment provides an XML parser for every generated metamodel automatically which can be used to read in data of the metamodel's structure (see chapter 2.2.2). To make use of this intrinsically provided parser we generate a metamodel called design entry language (MDEL) which defines the available SystemC constructs which can be used to model the formal specification. The SCAM plug-in for XML is adopted such that the generated XML output matches the needed data structure defined by the metamodel MDEL. This circumvents the extra effort to write a SystemC parser and makes it possible to read in the formal specifications in XML format with the intrinsic Metagen XML parser. This also implies that the formal specification in XML format is used as the input for the extended Metaprop framework. The complete available syntax of SystemC-PPA for modeling formal specifications is shown in following chapter when we define the metamodel for the design entry language SystemC-PPA called MDEL.

4.2. Metamodel-of-Things Design Entry Language

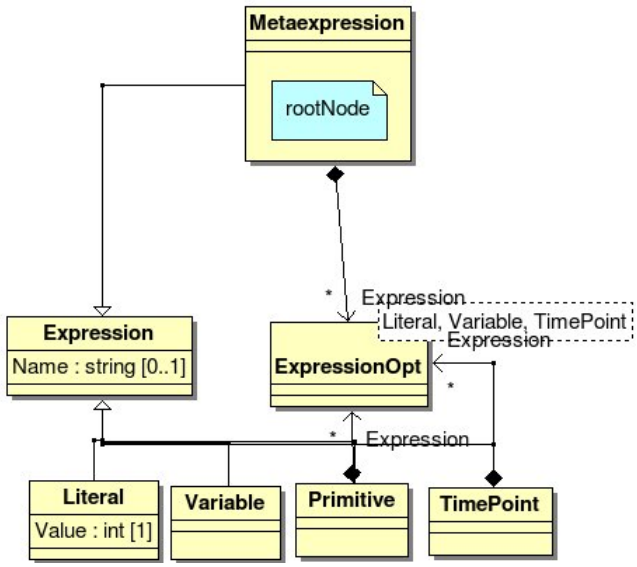
In the Metagen environment the classic approach to specify a metamodel is to a create UML class diagram, please refer to chapter 2.2.4 for an example. But Metagen also provides the possibility to define a metamodel in EBNF like syntax as the EBNF-description of a formal language can be understood as metamodel of the former because it defines the language's structure. As the intention of the Metamodel Design Entry Language (MDEL) is to define the syntax of SystemC-PPA it is reasonable to use the EBNF like description to express the metamodel in the Metagen environment. Note that any metamodel specified as textual description can be transformed into an UML representation and vice versa as the same types of relations (primitive relation, composition relation, association relation, generalisation relation) are supported by both, graphical and textual, metamodeling techniques. Figure 4.5 shows the MDEL specified in EBNF format which defines all available constructs of the design entry language SystemC-PPA which can be used to formally model the informal specification. This implies that, assuming that a given model only uses these available constructs and the model has been converted by the SCAM tool into XML format, the XML parser automatically generated by Metagen for the MDEL is able to parse that given model. In other words there is only one unqiue metamodel, namely MDEL, which can be used to capture all different kinds of specifications written in SystemC-PPA syntax. Note that this metamodel is adapted to SystemC syntax with respect to the naming as well as available constructs but can be extended easily with additional constructs to support any design entry language, e.g. Python.

```
⟨ModuleDescription⟩       ::= Name {⟨DataTypeDeclaration⟩} {⟨PortDeclaration⟩}
                              {⟨VariableDeclaration⟩} {⟨Section⟩}

⟨DataTypeDeclaration⟩     ::= ⟨BuiltInType⟩
                            | ⟨EnumType⟩
                            | ⟨CompoundType⟩
                            | ⟨CompoundSubType⟩
⟨BuiltInType⟩             ::= 'Int' | 'Unsigned' | 'Boolean'
⟨EnumType⟩                ::= Name {⟨EnumElement⟩}
⟨EnumElement⟩             ::= Name
⟨CompoundSubType⟩         ::= Name {⟨CompoundSubType⟩}
⟨CompoundSubType⟩         ::= Name Ref2DataTypeDeclaration

⟨PortDeclaration⟩         ::= Name ⟨Interface⟩ ⟨Direction⟩ Ref2DataTypeDeclaration
⟨Interface⟩               ::= 'Blocking' | 'Master' | 'Slave' | 'Shared'
⟨Direction⟩               ::= 'In' |'Out'

⟨VariableDeclaration⟩     ::= Name Ref2DataTypeDeclaration [Init]

⟨Section⟩                 ::= Name {⟨StatementOpt⟩}
⟨StatementOpt⟩            ::= ⟨CommunicationCall⟩
                            | ⟨Shared⟩
                            | ⟨ChangeSection⟩
                            | ⟨Assignment⟩
                            | ⟨ITE⟩
⟨CommunicationCall⟩       ::= ⟨Blocking⟩
                            | ⟨NonBlocking⟩
⟨Blocking⟩                ::= ⟨CommunicationType⟩ ⟨Expression⟩ Ref2PortDeclaration
⟨NonBlocking⟩             ::= ⟨CommunicationType⟩ ⟨Expression⟩ Ref2PortDeclaration
⟨CommunicationType⟩       ::= 'Read' | 'Write'
⟨Shared⟩                  ::= ⟨CommunicationType⟩ ⟨Expression⟩ Ref2PortDeclaration
⟨ChangeSection⟩           ::= Ref2Section
⟨Assignment⟩              ::= Ref2VariableDeclaration '=' ⟨Expression⟩
⟨ITE⟩                     ::= ⟨Expression⟩ ⟨IfPart⟩ [⟨ElsePart⟩]
⟨IfPart⟩                  ::= {⟨StatementOpt⟩}
⟨ElsePart⟩                ::= {⟨StatementOpt⟩}
```

Figure 4.5.: Textual description of the Metamodel Design Entry Language MDEL

The rootnode is called `ModuleDescription` and has composition relations with multiplicity 1..* to `DataTypeDeclaration, PortDeclaration, VariableDeclaration` and `Section` expressed in EBNF notation of curly brackets. SystemC-PPA supports `Int`,

`Unsigned` and `Boolean` as built-in datatypes, arbitrary enumeration datatypes as well as compound types of the formers. As explained in the previous section SystemC-PPA introduces special semantics for the ports' interfaces with regards to TLM of communciation between modules. This information is stored in the node `PortDeclaration` as it can have any of the previously discussed interface types `Blocking, Master, Slave` or `Shared`. A `PortDeclaration` node also stores information whether it is an input or output port as attribute `direction` and has an association relation `Ref2DataType` to the used datatype, indicated by the keyword `Ref2`. The behaviour of the module or in other words the content of the specification is described in multiple `Sections`. A `Section` is composed of several statements where the `StatementOpt` class provides the available constructs for modeling the system behaviour. The `CommunicationCall` statements are of special interest as they form the communication events which translate to the states of the PPA described by the SystemC-PPA module. A `CommunicationCall` can either be of `Read` or `Write` type, does have a association relation to the corresponding port `Ref2PortDeclaration`, indicated by the keyword `Ref2`, as well as a composition relation to `Expression`. Note that two types of `CommunicationCalls` are supported, namely `Blocking` and `NonBlocking`. As already indicated by the name the `Blocking` communication event blocks the execution of the model until the transaction is successfully processed, whereas the execution continues for the `NonBlocking` communication event. The `Expression` class maps to a general abstract expression metamodel `Metaexpression` which is shown in figure 4.6, a subset for bitwise expressions was already shown in figure 2.5.




```
Supported Primitives = [NOT,
    CABS, CMINUS, CMULT, CMOD,
     CREM, SLICE, INDEX,
    REVERSE, HEAD, TAIL, ROR,
    RAND, RNAND, RNOR, RXOR,
    RXNOR, BAND, BNAND, BOR,
    BNOR, BXOR, BXNOR, LAND,
    LNAND, LOR, LNOR, LXOR,
    LXNOR, SIGNEDCAST,
    UNSIGNEDCAST, UNSIGNEDCONV,
     SIGNEDCONV, HWPLUS, HWMUL,
     HWMINUS, RSL, RSA, LS,
    CONCAT, MUX, ISNEG, ISPOS,
     LT, LTEQ, GT, GTEQ, EQ,
    NEQ, REPEAT, PAST, NEXT,
    WITHIN, ROSE, FELL, STABLE
    ]
```

Figure 4.6.: Expression metamodel

This metamodel is used to define arbitrary expressions as abstract expression trees. Note

that both metamodels from the figures 2.5 and 4.6 do have exactly the same structure. For further explanation about the structure of the metamodel please refer to chapter 2.2.1. The only differences are the `TimePoint` operator class as well as the the `Primitive` class in figure 4.6. The `TimePoint` is a special timing operator which evaluates its arguments at a specific offset whereas the `Primitive` class is a placeholder for all remaining supported operators. A list of all supported operators can be seen on the right. Note that the bitwise expressions metamodel from figure 2.5 is included as operators `NOT`, `BAND`, `BOR` and `BXOR` where the `B` indicates that it is about the bitwise operators. Note as well that the abstract expression model defined by the metamodel from figure 4.6 can be reused by merging it with other metamodels. This technique is used for all metamodels in the extended Metaprop flow to provide a common abstract expression model throughout the complete flow.

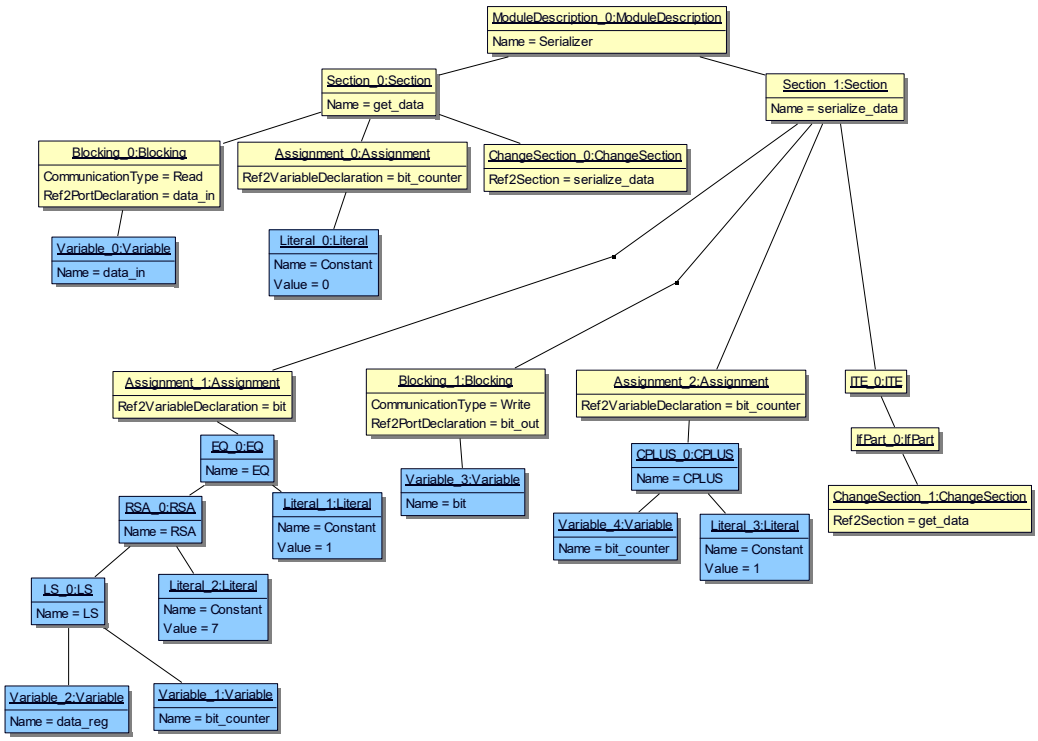


Figure 4.7.: Serializer from figure 4.3 as instance of MDEL

To give a better understanding how the formal specification looks like after it has been parsed into the Metagen environment as instance of the MDEL metamodel we reconsider our working example of the serializer from figure 4.3. Figure 4.7 illustrates a simplified view of the object diagram of the serializer's formal specification. In other words Figure 4.7 shows the MoT of the serializer which is an instance of the MMoT MDEL. Omitted are the declarations of ports, variables and types to ease the view. Objects colored in yellow correspond to the objects defined in the MDEL metamodel whereas objects of color blue

belong to the abstract expression metamodel. The data structure of the MoT represents the abstract syntax tree of the formal specification. The model does have two sections, the `get_data` section modeling the parallel input data and the `serialize_data` section modeling the serialized output. Each section has composition relations to the statements which happen inside the respective section. For example section `get_data` of figure 4.3 contains the execution sequence of the statements `[data_in->read(data_reg)]` ⇒ `[bit_counter=0]` ⇒ `[nextionsection=serialize_data]` illustrated in figure 4.7 as leaf nodes `Blocking_0`, `Assignment_0` and `ChangeSection_0` of `Section_0`.

4.3. Extraction of Path Predicate Abstraction from Model-of-Things

This section explains in detail how a PPA is extracted from a given MoT, namely by applying a specific operational graph coloring function to the objects of the MoT. As we have introduced the PPA as abstraction technique we use the term *extract* to emphasize that the formal specification written in SystemC-PPA matches precisely the abstraction layer of PPAs. This implies that no abstraction or refinement happens during the process of generating the PPA from the MoT. To extract the PPA from the MoT an operational graph coloring function which follows definition 1 needs to be defined. As a recap, definition 1 states the properties for a coloring function to be called operational based on directed graphs. As shown in the previous chapter the MoT represents the abstract syntax tree of the formal specification which is **not** a directed graph such that definition 1 can not be applied. To circumvent that we add additional semantics to the MoT. We interpret the objects of the MoT which are visited when the SystemC-PPA description is simulated as Control-Flow-Graph (CFG) which is directed graph by design.

Definition 5 (Control-Flow-Graph) *A Control-Flow-Graph (CFG) is a directed graph* $G = (V, E)$*, where* V *is the set of the graph's vertices and* E *is the set of edges which are pairs* (v_1, v_2) *of elements of* V*, where* v_1 *is the starting vertex of edge and* v_2 *is the ending vertex of edge. A Control-Flow-Graph is a representation of all possible paths which might be traversed throughout the execution of a SystemC-PPA model. The set of vertices* V *correspond to the objects of the MoT which are visited while the SystemC-PPA model is executed and the edges* E *to the possible paths between these objects.*

To interpret the MoT as CFG not all objects of the MoT need to be considered because the type -, variable - or port declarations are not relevant for the execution of the SystemC-PPA description but store additional information. Objects inside the subtrees of `Section` objects however are relevant as these sections store information about the desired behaviour of the hardware design. In other words, objects appearing in the subtrees of `Section` objects get executed if the formal specification represented as SystemC-PPA model is simulated and therefore need to be considered in the CFG. Figure

4.8 shows how the MoT's `Section` objects including their subtrees are interpreted as CFG based on the MoT of the serializer from 4.7.

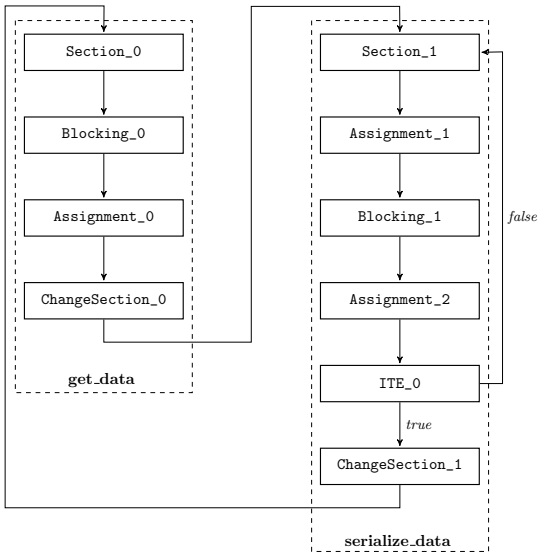


Figure 4.8.: MoT of the serializer interpreted as CFG

The figure above contains all objects from the simplified view of the MoT of figure 4.7 which correspond to the MDEL metamodel (colored in yellow). Note again that the simplified view of the MoT leaves out the declaration objects which do not need to be considered for creating the CFG as explained above. Objects corresponding to the abstract expression metamodel (colored in blue) are not explicitly shown in figure 4.8 either, because they are children of the objects belonging to the MDEL metamodel and therefore implicitly present. The CFG is another representation of the MoT's abstract syntax tree which expresses the sequence of executed objects when the SystemC-PPA description of figure 4.3 is simulated. In the `get_data` section (`Section_0`) data is read from the `data_in` port (`Blocking_0`), the `bit_counter` is resetted to zero (`Assignment_0`) and the section is changed to `serialize_data` (`ChangeSection_0`). In the `serialize_data` section (`Section_1`) the bit to be sent is extracted from the received data (`Assignment_1`) and written to the `bit_out` port (`Blocking_1`). If the received byte has been completely serialized the execution goes back to `get_data` section (*true*) otherwise it iterates over the `serialize_data` section again (*false*).
The MoT with extended semantics fulfills the precondition of definition 1 of being a directed graph because a CFG is the former by definition. This makes it possible to apply operational graph coloring following definition 1 for the MoT and by extracting the colored states the PPA is generated from the MoT. Directly implementing operational graph coloring following defintion 1 is not feasible due to two reasons. First, as already shown in chapter 2.3 there are different coloring functions which fulfill the properties of an operational coloring function leading to different PPAs. In other words the PPA is not

unique for a given directed graph. Second, algorithmically computing such an operational coloring from definition 1 is a computationally complex task which belongs at least to the *NP-Complete* complexity class. NP-Complete describes a group of computational problems which share the property that assume a solution is given this solution can be verified in polynomial time whereas finding a possible solution is not feasible in polynomial time. Richard M. Karp stated in 1972 21 general problems which belong to the NP-Complete complexity class [18]. The reason for operational graph coloring being at least in NP-Complete complexity class becomes clear if we take a closer look at the definition 1 again. The first condition states that every cyclic path in the operational colored directed graph $G = (V, E)$ contains at least one colored node $v \in W$. Note that the explicit color is not relevant for the following consideration but only the distinction between the set of colored nodes W and the set of uncolored nodes $\overline{W} = V \backslash W$. This can be reduced to the following satisfiability problem: Is it possible to color $k = |W|$ nodes such that every cycle in the directed graph G contains a node $v \in W$ which is called the *Feedback Node Set* computation problem [18]. This is one of the 21 general problems mentioned above which is shown to be NP-Complete.
To circumvent these problems we define a trivial *operational coloring function for MoTs* which fulfills the requirements of definition 1 to be called operational.

Definition 6 (Operational Graph Coloring for MoTs) *Consider a MoT represented as a CFG, $G = (V, E)$. Furthermore consider a subset $C \subseteq V$ of the graph's vertices called communication events and a set of colors $\hat{W} = \{\hat{w}_1, \hat{w}_2, \dots\}$. Assume that every cyclic path in the CFG contains at least one communication event $c \in C$. The operational coloring function f_c is defined as $f_c \colon C \mapsto \hat{W} : \forall\ u, v \in C \implies f_c(u) \neq f_c(v)$. Then Operation Graph Coloring for MoTs is a valid operational graph coloring according to definition 1.*

To show whether this definition is a valid operational graph coloring the two conditions of definition 1 need to be satisfied. The first condition evaluates to true by definition as it is assumed that every cyclic path in the CFG contains at least one communication event $c \in C$. This is referred to as a design rules for writing formal specifications in SystemC-PPA syntax. Ignoring this design rule while writing the formal specification lead to an error which is spotted by the SCAM tool which parses the SystemC-PPA description. The proof of the second condition is trivial. The operational coloring function f_c assigns a unique color to each element $c \in C$. It follows instantly that assuming two nodes $u, v \in C$ of same color $f_c(u) = f_c(v)$ need to be the same node $u = v$. Condition two always evaluates to true if applied for the same nodes $u_0 = v_0$.
The reason for coloring communication events is that the execution of the formal specification as SystemC-PPA model might block infinitely long if the underlaying communication event uses the blocking communication interface which implements the four-phase handshake. The execution is only allowed to continue if the according synchronisation signal is received such that the communication transaction can be processed. In other words

the PPA needs to have a state which models this blocking behaviour, namely modeling the wait for synchronisation with a self transition in the specific state in the PPA. We apply operational graph coloring for MoTs on the serializer's CFG from figure 4.8 which results in figure 4.9.

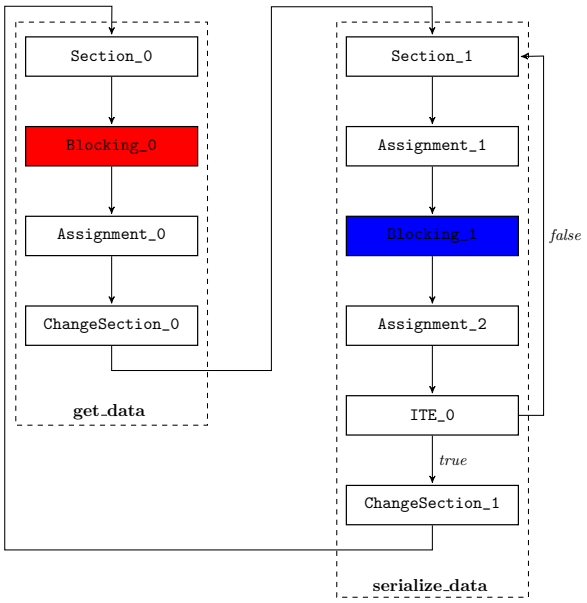


Figure 4.9.: Operationally colored CFG of the serializer

The colored communication events `Blocking_0` and `Blocking_1` are identified to be the states of the serializer's PPA which has already been shown in figure 4.4. The CFG shows all possible paths which can be traversed if the SystemC-PPA model is executed. To identify the transitions of the PPA we take a closer look at specific paths, namely all possible paths connecting the already colored objects. For each of these paths exactly one transition is added to the PPA of the serializer. For example, `Blocking_1` has a path to itself as well as to the colored object `Blocking_0`. This results in two transitions, a self transition in state `Blocking_1` and a transition from state `Blocking_1` to state `Blocking_0` (see figure 4.4 which shows the complete PPA of the serializer). Uncolored objects which occur along the paths connecting colored objects are **not** abstracted but stored inside the according generated transition as they contain relevant information for the property generation from the PPA. This implies that no abstraction or refinement happens during this process as no information is added or removed from the original CFG. The CFG from figure 4.9 shows in total three different paths between colored objects which are transformed into transitions of the PPA. To receive the complete PPA from figure 4.4 further transitions need to be added due to the SystemC-PPA semantics which are not explicitly represented as paths in the CFG:

- **Reset transition:** The reset transition refers to the constructor of the SystemC-

PPA module which assigns an initial value for the `nextsection` variable. This defines which communication event is visited first if the simulation of the SystemC-PPA model is initiated. In other words the reset transitions expresses the initial state of the PPA.

- **Wait transition:** Wait transitions are generated for communication events using the blocking communication interface. Wait transitions are self transitions which guarantee that the state of the PPA stays unchanged unless a synchronisation signal from the communication counterpart is received.

We have shown how the explicit description of the PPA is extracted from a given MoT. The explicit description of a PPA consists of states and transitions between the former such that we map it to a general model of a FSM which is shown in the following chapter.

4.4. Path Predicate Abstraction as Finite State Machine

In this chapter we introduce the second metamodel of the Model-of-Things layer called MetaFSM (MFSM), a generic metamodel for finite state machines, to store the explicit model of the PPA after it has been extracted from the MoT. As a recap, the states of the PPA correspond to the communication events of the SystemC-PPA model whereas the PPA's transitions refer to the paths connecting these states in the CFG. Each transition representing a path in the CFG describes an operation of the overall design behavior which can be understood as consecutive execution operations. Later in this chapter we explain in detail how transitions are translated into operational properties, namely by the model-to-model transformation to the Model-of-Properties layer. In addition this metamodel includes further information about the specific RTL implementation of the design, e.g. signal or register names which is needed for transformation to the Model-of-Properties layer. This information is used to refine the PPA stored as instance of MFSM to abstract temporal Boolean expressions which are stored as instances of the Metaprop metamodel (see appendix A.1). Before going into detail about this model-to-model transformation we present the metamodel MetaFSM in the following.

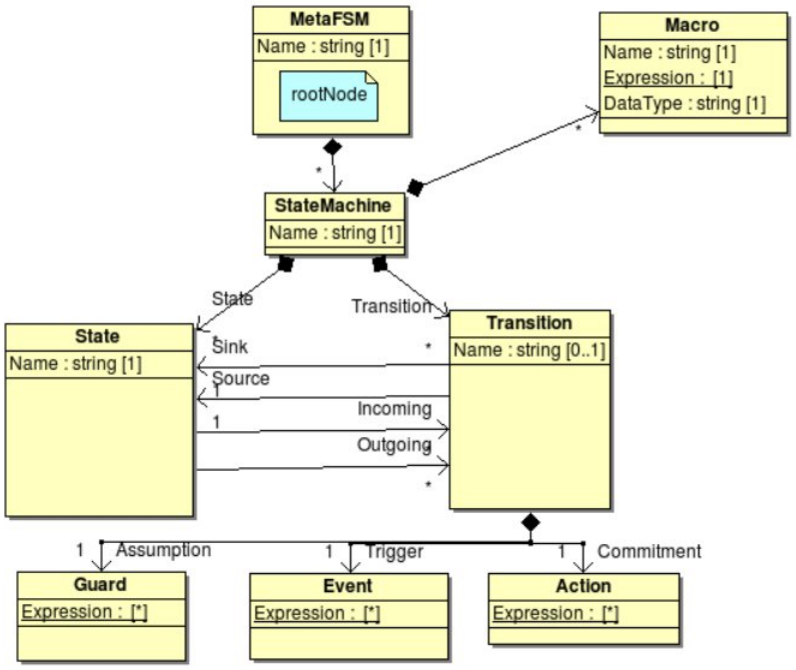


Figure 4.10.: Generic metamodel for FSMs

The root node is called `MetaFSM` and has a composition relation to `StateMachine` with multiplicity *. The fact that an instance of `MetaFSM` can consist of several instances of the `StateMachine` class as indicated by multiplicity * is due to the fact that theoretically a general FSM can be decomposed into multiple submachines which interact with each other. During the scope of this thesis however the PPA is stored as exactly one instance of the `StateMachine` class. Each `StateMachine` has an arbitrary number of states which refer to the communication events of the SystemC-PPA model as well as transitions which correspond to the paths connecting these communication events. A `State` has a `Name` which indicates by convention the section where the communication event takes place. In FSM theory a transition triggers at specific events, has guard conditions which need to evaluate to true as a precondition such that the transition is able to be taken as well as action items which get executed when the transition is triggered. This is represented as `Transition` class in the MetaFSM metamodel as it has composition relations to `Guard`, `Event` as well as `Action` class which correspond to the former with respect to their theoretical semantics. The `Event` class covers the handshaking mechanisms, in detail stores information about the synchronisation signals which need to evaluate to true such that the transition can be triggered. The `Guard` class holds other expressions appearing as if conditions in the SystemC-PPA model which need to be true such that the underlaying path of the transition is taken. The `Action` class however does not store action items to be executed like in general FSM theory but expressions which need to be proven on the specific design when the transition reached its sink. Note that the classes `Guard`,

`Event` and `Action` have primitive relations called `Expression`. This is a placeholder which refers to the universal abstract expression metamodel of figure 4.6 which is used as common expression model throughout the complete extended Metaprop flow. Note as well that the `StateMachine` also has a composition relation to a class called `Macro`. In software engineering the term macro is used for grouping a sequence of instructions to execute them with a single call only. In terms of PPA this semantics slightly changes because macros are used to map the PPA description to the specific implementation. In detail, all identifiers which are used in the SystemC-PPA, e.g. variables or ports like `data_reg` or `bit_out`, are understood as placeholders. For each of these identifiers or placeholders an object of the `Macro` is automatically generated with the intention to map this identifier to its hardware equivalent. The `Name` attribute of class `Macro` corresponds to the identifier used in the SystemC-PPA model whereas the information of the hardware equivalent is stored as an expression tree of the abstract expression metamodel.
The example illustrated in figure 4.11 explains the role of guard, event and action based on the concrete working example of the serializer. On the left one can see the PPA extracted from the SystemC-PPA description of the serializer. The transition which we take a closer look at is highlighted in blue.

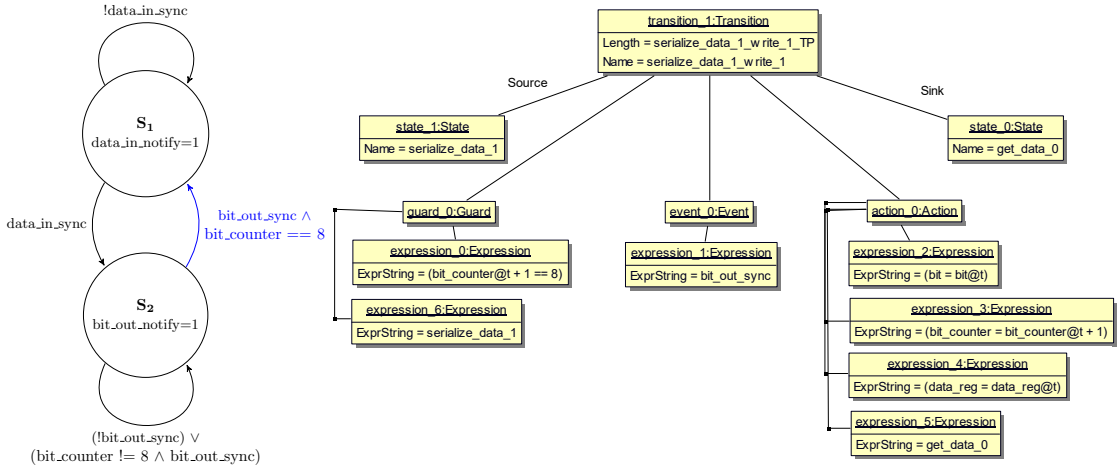


Figure 4.11.: Example transition as instance of MetaFSM's `Transition` class

The highlighted transition corresponds to the path of the CFG where the last bit has been successfully serialized and the execution goes back to `get_data` section to fetch another byte to be serialized. On the right one can see the highlighted transition as instance of the `Transition` class illustrated as object diagram. The `Transition` object is called `serialize_data_1_write_1` indicating the source state as well as that the source state is a write communication event. The `Guard` object holds all expressions which need to evaluate to true such the transition is triggered except expressions related to synchronisation. For this transition to trigger we need to guarantee that the PPA is in the

correct state `serialize_data_1` (`expression_6`) as well as the `bit_counter` equals eight which implies the whole byte has already been successfully serialized (`expression_0`). Synchronisation related conditions are stored as expressions of the `Event` object which triggers the transition. The communication event in the CFG which corresponds to the source state `serialiaze_data_1` uses the blocking communication interface such that the transition should only trigger if and only the synchronisation signal `bit_out_sync` evaluates to true (`expression_1`). The `Action` class stores expressions which need to be proven when the sink state is reached. This includes on the one hand datapath operations specified in the SystemC-PPA model which are executed when the underlaying path is traversed by the execution, namely if new values get assigned to the defined variables. If a variable is **not** mentioned while traversing the underlaying path it needs to be proven that the value of the variable stays unchanged. While traversing the path in the CFG which corresponds to the highlighted transition the `bit_counter` is incremented by one (`expression_3`) but the other variables `bit` and `data_reg` do not get assigned new values. This implies that they keep the values they had when the operation has been triggered (`expression_4`,`expression_5`).
Following the MDA approach for the extended Metaprop flow the next step is to perform a model-to-model transformation from the PPA as instance of the MetaFSM metamodel to the Model-of-Properties layer. Each object of the `Transition` class maps to exactly one abstract temporal Boolean expression expressing the intended operation behaviour which is captured as MoP. An abstract temporal Boolean expression object models an interval property which consists of an assumption part A and a commitment part C. The objects corresponding to the `Guard` and `Event` class form the assumption part A because the expressions of these objects need to evaluate to true to trigger the transition as explained in the semantics of the MetaFSM metamodel above. Expressions of the `Action` object form the commitment part C.
Now as the model-to-model transformation from the MetaFSM metamodel to the Model-of-Properties layer is defined the extended Metaprop flow is complete. Complete in this context means that a property suite in a specific target language can be automatically generated starting from a MoT because the extended Metaprop flow reuses the Model-of-Properties layer, the View layer as well as the model-to-model transformation between the former without any changes, please refer to chapter 2.2.3 for further information about these layers.
Figure 4.12 shows the operation property generated by the extended Metaprop flow for the transition from figure 4.11 in ITL syntax. It can be easily identified that the assumption part is composed of expressions which are contained in the `Guard` and `Event` objects whereas the commitment part consists of the expressions stored in `Action` object (see figure 4.11).

```
1  property serialize_data_1_Write_1 is
2
3  for timepoints:
4    t_end = t+serialize_data_1_Write_1_TP;
5
6  assume:
7    at t: ((bit_counter + 1) = 8) and serialize_data_1 and bit_out_sync;
8  prove:
9    at t_end: (bit = prev(bit,serialize_data_1_Write_1_TP));
10   at t_end: (bit_counter = (bit_counter + 1));
11   at t_end: (data_reg = prev(data_reg,serialize_data_1_Write_1_TP));
12   at t_end: get_data_0;
13 end property;
```

Figure 4.12.: Generated interval property for transition `serialize_data_1_write_1` from figure 4.11

To feed this property together with an RTL implementation to a formal tool to prove whether this property holds on the given implementation of the design this property needs to be mapped to the implementation which is done by using macros. Macros are automatically generated by the extended Metaprop flow as instance of the `Macro` class of the MetaFSM metamodel. Depending on the object which a macro is generated for its semantics differ:

- **Variable macros:** For each variable declaration in the SystemC-PPA model a macro is generated. A variable is used to store data which implies that variables in the SystemC-PPA model refine to RTL registers which is the hardware equivalent for storing data. For example the variable `data_reg` refines to the actual name of the RTL register in the implementation wich stores the data to be serialized. Generating macros for variables is fully automated in the extended Metaprop flow.
- **Port macros:** A port declaration may introduce multiple macros depending on the used communication interface. Each port generates a *message* macro which describes the received or sent message depending on whether it is an input or output port. In addition to the *message* macro, macros for *sync* and *notify* signals depending on the implemented handshaking mechanism need to be included. A port using the **shared** interface does no implement any handshaking mechanism such that only a *message* macro is generated. Ports with a **Master/Slave** interface introduce, additonally to the *message* macro, a *sync* macro for the slave and a *notify* macro for the master. For ports using the **blocking** interface both *notify* and *sync* for both communication parts need to be included to implement the four-phase handshake. Macro generation for ports is also fully automated in the extended Metaprop flow.
- **State macros:** Each communication event refers to a state of the PPA which

introduces a macro for this specific state. This macro refines the name of the PPA state, e.g. `serialize_data_1` to a specific state of the hardware implementation which can be any arbitrary encoding over the available RTL registers in the design. Automatic macro generation for states is included in the extended Metaprop flow.

- **Timepoint macros:** For each transition a macro specifying the length of the interval property which is described by the transition is introduced. This macro states after how many clock cycles the commitment part of the interval property should be evaluated. The identifier `serialize_data_1_Write_1_TP` is an example for a timepoint macro generated for the interval property of figure 4.12. Note that timing is not automated by the extended Metaprop flow such that timepoint macros are automatically generated but initialized by default with the value 1. This implies that the Verification Engineer needs to update the timepoint macros accordingly after the property suits has been generated.

5. Application of the Methodology to a Real-World Design

5.1. I^2C as Real-World Design

I^2C is a standard bidirectional 2-wire bus protocol developed by NXP Semiconductors in 1982 [1] which is used for inter-integrated-circuit communication. The I^2C bus consists of a serial data wire (SDA) and a serial clock wire (SCL) which are carrying information between the devices connected to the bus. Each device has an unique address associated and can act as receiver as well as transmitter (bidirectional). A device can either be a **master** or a **slave** which is not predefined but dynamically allocated. We call the device which initiates a transaction and controls the bus as master whereas all the other devices act as slaves at that time. The I^2C-bus is a multi-master bus which means that an arbitrary number of devices capable of controlling the bus can be connected to the it. This can lead to situations where two of these devices are trying to initiate data transfers at the same time. To avoid data loss and guarantee a controlled sequence of transactions, the I^2C-bus protocol provides arbitration, a process to determine which device gets access at the bus.

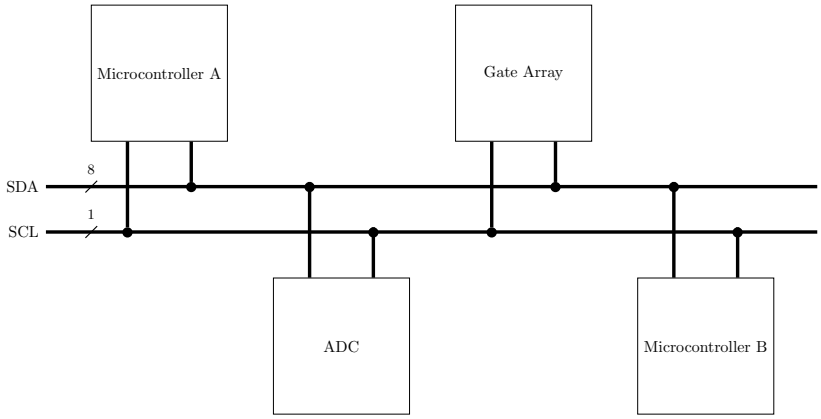


Figure 5.1.: Example of an I^2C-bus multi-master configuration

Figure 5.1 shows an example of I^2C-bus configuration consisting of two microcontrollers and two peripherals, namely an Analogue-Digital-Converter (ADC) and a Gate Array. In general every device is capable of becoming a master and therefore has the ability

to invoke a transaction. Usually only microcontrollers have the need to initiate data transfers such that in figure 5.1 the simplest multi-master configuration consisting of two possible masters is shown. To make the master-slave as well as the receiver-transmitter relationships more clear the following two scenarios are considered:

1. **Suppose microcontroller A wants to transfer data to microcrontroller B:** Microcontroller A (master) addresses microcontroller B (slave) with its associated unique address. The other peripherals act as slaves as well but are not relevant as they are not addressed. After successfully addressing microcontroller B, microcontroller A (master-transmitter) sends data to B (slave-receiver) and terminates the transfer when the transaction is done.

2. **Suppose microcontroller A wants to receive data from microcontroller B:** Again microcontroller A (master) initiates the transaction by addressing microcontroller B (slave). Afterwards Microcontroller A (master-receiver) receives data from microcontroller B (slave-transmitter) and terminates the transaction.

Note: As shown in the scenarios above the receiver-transmitter relationship is not permanent but only depend on the direction of data at that time. Also note that the master always intiates the transaction which does **not** dependent on whether the master wants to receive or transmit data.
Referring to the I^2C-bus specification [1] the protocol has a number of features which are either mandatory (M), optional (O) or not applicable (n/a), depending on the configuration of the I^2C-bus. We distinguish between two different configurations, namely single master and multi master. The different possible setups are shown in table 5.1.

Feature	**Single-Master**	**Multi-Master**
START Symbol	M	M
STOP Symbol	M	M
Acknowledge	M	M
Synchronization	n/a	M
Arbitration	n/a	M
Clock stretching	O	O
7-Bit slave address	M	M
10-Bit slave address	O	O
General Call address	O	O
Software Reset	O	O
START byte	n/a	O

Table 5.1.: I^2C-bus features depending on the configuration [1]

In the scope of this thesis a basic single-master setup is considered. This is due to the fact that Infineon's I^{2}C-bus implementation is used in single-master configuration **only**. This leads to the fact that the features arbitration, synchronisation as well as START byte do not have to be considered in the scope of this thesis. For further explanation of the mandatory features as well as clock stretching, please refer to appendix C. The optional features, namely 10-Bit slave address, general call address and software reset are not considered due to complexity. The formal specification of the I^{2}C-bus protocol which we present in the following chapters can be extended with these optional features such that they are also supported.

5.2. I^{2}C-Bus Protocol Specification as SystemC-PPA model

For running the extended Metaprop flow we need to capture the I^{2}C-bus protocol specification formally as a SystemC-PPA model. Due to the fact that an I^{2}C-bus implementation can either act as slave **or** master we split the formal specification into two separate SystemC-PPA models. One model covers the master behaviour and the other model captures the behaviour of the slave respectively. Both models operate on byte nor on bit level which leads to a more abstract description of the formal specification which is more easy to understand and model.

5.2.1. SystemC-PPA model for Slave Behaviour

A graybox of the slave SystemC-PPA model can be seen in figure 5.2 which visualizes the declared ports and variables of the SystemC-PPA description (please refer to appendix D.1 for the complete model). Edges illustrated on the left side refer to declared ports which are used for communication between the slave module and an arbitrary device, e.g. a microcontorller, which makes use of the I^{2}C-bus. Edges shown on the right are used by the slave module for transmitting to or receiving from the I^{2}C-bus. All ports use the blocking communication interface except the `data_to_device` port colored in blue is defined with a shared communication interface. All declared variables are shown as dashed boxes inside the slave module box. The function of the variables is indicated by their assigned name, e.g. the `data_from_device_reg` variable is used to store data which is received from the device and sent over the I^{2}C-bus afterwards. Note that all variable names do have the suffix `reg` to indicate that variables map to registers in the RTL implementation of the design. Note as well that the `status_reg` defined as compound type `status_t` in the SystemC-PPA description translates into two separate variables `status_reg_start` and `status_reg_stop`. Both variables are of Boolean type and contain information whether the slave has received a START or STOP symbol over the I^{2}C-bus respectively.

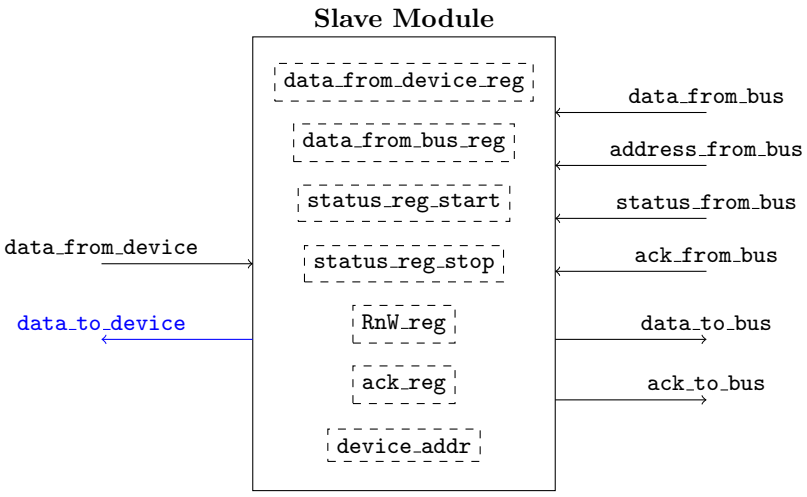


Figure 5.2.: Graybox model of slave SystemC-PPA description

The SystemC-PPA description modeling the I^2C slave behaviour is divided into four sections, `idle`, `get_addr`, `transmit_data` and `receive_data`, which we explain in the detail in the following.

Section `idle`

```
1 if (section == idle) {
2   status_from_bus->read(status_reg);
3   if (status_reg.start) {
4     nextsection = get_addr;
5   }
6 }
```

Figure 5.3.: Code extract section `idle` from slave SystemC-PPA model

In the `idle` section the execution blocks until status information from the bus is available at the port `status_from_bus` (line 2). Only if a START symbol is detected execution changes the section to `get_addr` (line 4). If a STOP symbol or neither a STOP or a START symbol is provided from the bus, the execution iterates over the `idle` section again and waits for the next synchronisation signal to read status information from the I^2C-bus.

Section `get_addr`

```
1  if (section == get_addr) {
2    address_from_bus->read(data_from_bus_reg);
3    if ((data_from_bus_reg >> 1) == device_addr) {
4      data_to_device->set(data_from_bus_reg);
5      RnW_reg = (data_from_bus_reg << 7) == 128;
6      ack_to_bus->write(true);
7      if (RnW_reg) {
8        nextsection = transmit_data;
9      } else {
10       nextsection = receive_data;
11     }
12   } else {
13     nextsection = idle;
14   }
15 }
```

Figure 5.4.: Code extract section `get_addr` from slave SystemC-PPA model

The `get_addr` section models receiving the 7-bit address extended with the R/$\overline{\text{W}}$-bit which indicates whether it is a write or read transaction. The address is always the first byte to be sent over the I^{2}C-bus after the START symbol. This byte is received by the means of the port `address_from_bus` and stored in the variable `data_from_bus_reg` (line 2). The seven most significant bits of this variable represent the received address which is compared with the `device_addr` variable which stores the unique ID of the device (line 3). Assume that they do **not** match which implies that the slave is not addressed by the master. Given this scenario, the execution returns to the `idle` section to wait for the next transaction (line 13). This behaviour is important for a single-master/multi-slave setup of the I^{2}C-bus because an address is always distributed by the master to **all** connected slave. Each slave checks independently whether it is addressed and if this is case continues with processing the transaction. Assume that the received address matches the content of the `device_addr` variable, the received byte consisting of address and R/$\overline{\text{W}}$-bit is forwarded to the device (line 4). Afterwards the R/$\overline{\text{W}}$-bit is extracted as least significant bit from the `data_from_device_reg` and stored in the variable `RnW_reg` (line 5). To signal the master the address has been successfully received an acknowledgement signal is sent over the bus by the means of the `ack_to_bus` port (line 6). Based on the content of the `RnW_reg` the section is changed to `transmit_data` (line 8) or `receive_data` (line 10).

Section `transmit_data`

```
1  if (section == transmit_data) {
2    data_from_device->read(data_from_device_reg);
3    status_from_bus->read(status_reg);
4    if (status_reg.start && !status_reg.stop) {
5      nextsection = get_addr;
6    } else if (status_reg.stop && !status_reg.start) {
7      RnW_reg = false;
8      nextsection = idle;
9    } else {
10     data_to_bus->write(data_from_device_reg);
11     ack_from_bus->read(ack_reg);
12     if (!ack_reg) {
13       nextsection = idle;
14     }
15   }
16 }
```

Figure 5.5.: Code extract section `transmit_data` from slave SystemC-PPA model

The `transmit_data` section models the transmission of a single byte assuming that the received device address matches with the unique device ID and the R/$\overline{\text{W}}$-bit being asserted indicating that the master wants read data from the slave. To be able to transmit data over the bus data needs to be read from the device which uses the I^2C-bus for communication. This is modeled by reading from port `data_from_device` and storing the data in the according register `data_from_device_reg` (line 2). This communication event implements the I^2C-bus feature of clock stretching. As the port `data_from_device` uses the blocking communication interface, the execution blocks until data is received from the device forcing the communication counterpart, namely the master, into a wait state. Before data can actually be sent over the I^2C-bus the slave needs to check whether a repeated START or STOP symbol has been sent by the master. For a START symbol being detected (line 4) the execution returns to the `get_addr` section (line 5) to restart the transaction and receive a new address. For a STOP symbol (line 6) the transaction is terminated and the execution goes back to the `idle` section (line 8). If none of the former is detected (both symbols cannot be detected at the same time as SDA line is not able to fall and rise at the same time) the byte stored in the `data_from_device_reg` is written to the bus by the means of the port `data_to_bus` (line 10). After data has successfully been sent the slave reads from the port `ack_from_bus` whether the master has acknowledged the transmitted data (line 11). If this is the case the execution returns to the beginning of the `transmit_data` section to send another byte. Otherwise execution returns to `idle` section (line 13).

Section `receive_data`

```
1  if (section == receive_data) {
2    status_from_bus->read(status_reg);
3    if (status_reg.start && !status_reg.stop) {
4      nextsection = get_addr;
5    } else if (status_reg.stop && !status_reg.start) {
6      nextsection = idle;
7    } else {
8      data_from_bus->read(data_from_bus_reg);
9      data_to_device->set(data_from_bus_reg);
10     ack_to_bus->write(true);
11   }
12 }
```

Figure 5.6.: Code extract section `receive_data` from slave SystemC-PPA model

Assuming the received address matches the device ID and the R/$\overline{\text{W}}$-bit **not** being asserted the slave receives data from the master which is modeled in the `receive_data` section. Before the slave is able to receive data from the I^2C-bus it needs to check for a START or STOP symbol appearing on the bus. To do so the slave reads from the port `status_from_bus` and stores it to the according variables `status_reg_start` and `status_reg_stop` (line 2). If a repeated START symbol is detected the execution goes to section `get_addr` to restart the transaction by receiving the address (line 4). For a STOP symbol appearing on the bus the transaction is terminated and the execution returns to `idle` section (line 6). For the case that neither START nor STOP symbol is received the slave continues with reading data from the I^2C-bus by the means of the port `data_from_bus` and storing it in the according variable `data_from_bus_reg` (line 8). After the data has been successfully read it outputs the byte directly to the device by setting the according output port `data_to_device` (line 9). The slave sends an acknowledgment to the bus to signal the master that the byte has been received successfully (line 10). The execution returns to the beginning of section `receive_data` for receiving a new byte.

5.2.2. SystemC-PPA model for Master behvaviour

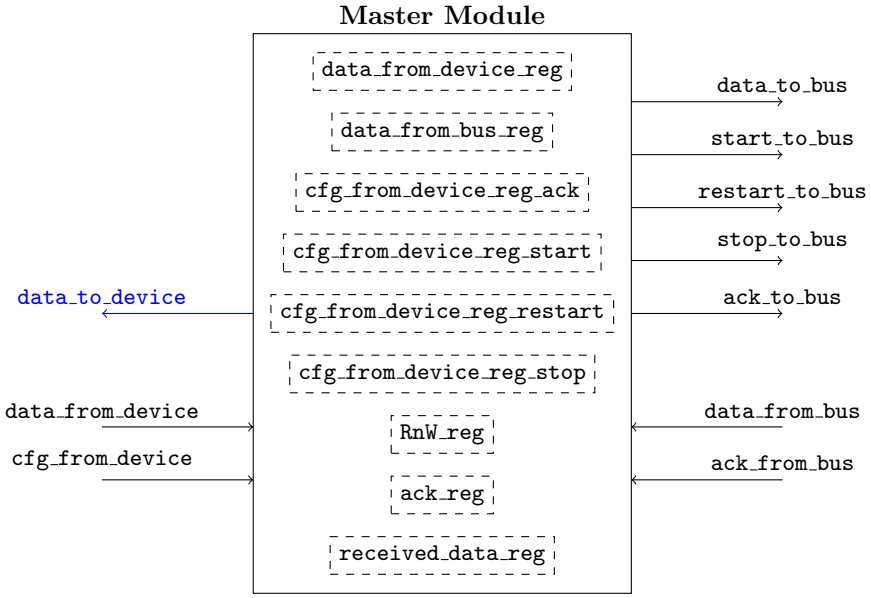


Figure 5.7.: Graybox modle of master SystemC-PPA description

This chapter explains the SystemC-PPA description capturing the master behaviour of the I^{2}C-bus in more detail. A graybox model is given in figure 5.7 which shows the inputs ports, outputs ports and variables of the model. Edges drawn on the left correspond to the model's ports which are related to communication between device and the SystemC-PPA module. An edge shown on the right however does imply that the corresponding port is used in the SystemC-PPA description for modeling communication over the I^{2}C-bus. The port `data_to_device` is colored in blue to highlight that this port uses the shared communication interface whereas all the other ports are defined with blocking communication interface. The registers `data_from_device_reg`, `data_from_bus_reg`, `Rnw_reg` and `ack_reg` do have the equivalent function as is in the slave SystemC-PPA description. The master starts, restarts or stops the transmission by sending the corresponding symbol over the I^{2}C-bus. To model this behaviour the master SystemC-PPA description provides a variable `cfg_from_device` which is defined as a compound type consisting of four subtypes. The device using the master interface for data transmission sets the variables `cfg_from_device_reg_start`, `cfg_from_device_reg_restart` and `cfg_from_device_stop` accordingly to initiate sending these control symbols over the I^{2}C-bus. The `cfg_from_device_reg_ack` variable is used to indicate whether the device wants to acknowledge the received data. The register `received_data_reg` is needed to indicate whether data is received from the device over the `data_from_device` port.

Section `idle`

```
1  if (section == idle) {
2    cfg_from_device->read(cfg_from_device_reg);
3    if(cfg_from_device_reg.start) {
4      data_from_device->read(data_from_device_reg);
5      cfg_from_device_reg.start = false;
6      RnW_reg = (data_from_device_reg << 7) == 128;
7      start_to_bus->write(true);
8      nextsection = send_addr;
9    }
10 }
```

Figure 5.8.: Code extract of section `idle` from master SystemC-PPA model

The `idle` section models the master's behaviour when waiting for a new transaction to be processed. To initiate a transaction the device needs to assert the variable `cfg_from_device_start`. To do so the port `cfg_from_device` is read (line 2) and it is checked whether the `cfg_from_device_start` variable has been set (line 3). Only if the variable `cfg_from_device_start` is asserted a new transaction is initiated. To be able to send data over the I^2C-bus certain data needs to be provided by the device. With regards to the I^2C-bus protocol the first byte to be sent over the I^2C-bus is encoded as 7-bit address extended with the R/$\overline{\text{W}}$-bit. By reading from the port `data_from_device` the master module fetches the provided data and stores it in the variable `data_from_device_reg` (line 4). The variable `cfg_from_device_start` gets resetted (line 6), the R/$\overline{\text{W}}$-bit is extracted and stored in the variables `RnW_reg` (line 7) and the START symbol is sent over I^2C-bus (line 8). After the START symbol has been transferred to the slave over the I^2C-bus successfully execution changes to the `send_addr` section (line 9).

Section `send_addr`

```
1  if (section == send_addr) {
2    data_to_bus->write(data_from_device_reg);
3    ack_from_bus->read(ack_reg);
4    if(ack_reg && RnW_reg) {
5      nextsection = receive_data;
6    } else if (ack_reg && !RnW_reg) {
7      nextsection = transmit_data;
8    } else {
9      nextsection = send_status;
10   }
11 }
```

Figure 5.9.: Code extract of section `send_addr` from master SystemC-PPA model

In the `send_addr` section the 7-bit address extended with the R/$\overline{\text{W}}$-bit is sent over the I^{2}C-bus (line 2). This message is distributed to all slaves which are connected to the I^{2}C-bus but only the slave with the matching device address acknowledges the received address. This acknowledgment is read over the port `ack_from_bus` and stored in the `ack_reg` variable (line 3). Based on this variable and the content of the `RnW_reg` it is determined how to proceed. Assuming that the slave has successfully acknowledged the address (`ack_reg = true`), execution changes either to section `receive_data` if `RnW_reg` is asserted (line 5) otherwise to section `receive_data` (line 7). If none of the connected I^{2}C modules acting as slaves acknowledged the sent address (`ack_reg = false`) the master can either restart or terminate the transaction which is captured in section `send_status` (line 9).

Section `send_status`

```
1  if (section == send_status) {
2    cfg_from_device->read(cfg_from_device_reg);
3    if (cfg_from_device_reg.restart) {
4      data_from_device->read(data_from_device_reg);
5      nextsection = send_restart;
6    } else if (cfg_from_device_reg.stop) {
7      nextsection = send_stop;
8    }
9  }
```

Figure 5.10.: Code extract section `send_status` from master SystemC-PPA model

As none of the connected slaves have acknowledged the sent address the device using the I^{2}C module provides information about how to proceed (line 2). If the variable `cfg_from_device_reg_restart` is asserted by the device the transaction is restarted by resending an address (line 3). The address information is read from the device (line 4) and the execution changes to section `send_restart` (line 5). Assuming `cfg_from_device_reg_stop` being asserted asserted the transaction is terminated by changing the section to `send_stop`. If none of the former variables have been asserted by the device the execution iterates over the `send_status` section and blocks until new information is provided by the device.

Section `receive_data`

```
1  if (section == receive_data) {
2    data_from_bus->read(data_from_bus_reg);
3    data_to_device->set(data_from_bus_reg);
4    cfg_from_device->read(cfg_from_device_reg);
5    if (cfg_from_device_reg.ack) {
6      ack_to_bus->write(true);
7    } else if (cfg_from_device_reg.restart) {
8      received_data_reg = data_from_device->nb_read(data_from_device_reg);
9      if (received_data_reg) {
10       nextsection = send_restart;
11     } else {
12       nextsection = send_stop;
13     }
14   } else {
15     nextsection = send_stop;
16   }
17 }
```

Figure 5.11.: Code extract section `receive_data` from master SystemC-PPA model

The slave with the matching address has acknowledged the address and the R/$\overline{\text{W}}$-bit is asserted which implies that the master reads data from the slave. A byte is read from the I^{2}C-bus by the means of the `data_from_bus` port, stored in variable `data_from_bus_reg` (line 2) and forwarded to the device over the `data_to_device` port (line 3). After one byte has been successfully received the device has three different options how to proceed the transaction, namely requesting the next byte from the slave, restarting or terminating the transaction. To determine the desired option the I^{2}C master module reads the configuration from the port `cfg_from_device` (line 4). If the variable `cfg_from_device_reg_ack` is asserted by the device the I^{2}C master module acknowledges the received byte (line 6) and the execution returns to the beginning of the `receive_data_` section to read another byte from the I^{2}C-bus. If the `cfg_from_device_reg_restart` variable is asserted the transaction is restarted by retransmitting a new address extended with a R/$\overline{\text{W}}$-bit. This

information needs to be provided from the device and is read in line 8. Note that this read communication event uses `nb_read` which means execution does **not** block until data is available but immediately reads from a certain port. `nb_read` returns the success to the variable `received_data_reg` whether the data has been read successfully. Only if data is provided from the device which is modeled with `received_data_reg` being asserted the transaction is restarted and the execution changes to section `send_restart` (line 10). Otherwise no data is supplied by the device which terminates the transaction by changing the section to `send_stop` (line 12). The transaction is also terminated if the device sets the variable `cfg_from_device_reg_stop` in line 4.

Section `transmit_data`

```
1  if (section == transmit_data) {
2    cfg_from_device->read(cfg_from_device_reg);
3    if (cfg_from_device_reg.restart) {
4      data_from_device->read(data_from_device_reg);
5      nextsection = send_restart;
6    } else if (cfg_from_device_reg.stop) {
7      nextsection = send_stop;
8    } else {
9      data_from_device->read(data_from_device_reg);
10     data_to_bus->write(data_from_device_reg);
11     ack_from_bus->read(ack_reg);
12     if (!ack_reg) {
13       nextsection = send_status;
14     }
15   }
16 }
```

Figure 5.12.: Code extract section `transmit_data` from master SystemC-PPA model

This section models data transmission from the I^2C master module to the slave module. Before data can be transmitted the device using the master module needs to determine whether the transaction should be restarted or terminated by asserting the according variables `data_from_device_reg_restart` or `data_from_device_reg_stop` in line 2. If none of the former is asserted data is read from the device (line 9) by the means of the port `data_from_device_reg` and sent over the I^2C-bus in line 10. After the byte has been transmitted successfully the master module reads the acknowledgement from the slave module (line 11). If the slave has acknowledged the transmitted data execution returns to the beginning of section `transmit_data` to send another byte. Assume the case that the slave has not acknowledged the transmitted byte execution changes to section `send_status` which means that the transaction can either be restarted or terminated.

Section `send_restart`

```
1 if (section == send_restart) {
2   RnW_reg = (data_from_device_reg << 7) == 128;
3   cfg_from_device_reg.restart = false;
4   restart_to_bus->write(true);
5   nextsection = send_addr;
6 }
```

Figure 5.13.: Code extract section `send_restart` from master SystemC-PPA model

When the section `send_restart` is reached the transaction is restarted by sending a START symbol over the I^2C-bus. The previous section ensures that the new address is stored in the variable `data_from_device`. The R/$\overline{W}$-bit is extracted (line 2), the variable `cfg_from_device_reg_restart` is resetted (line 3) and the repeated START symbol is sent over I^2C-bus (line 4).

Section `send_stop`

```
1 if (section == send_stop) {
2   cfg_from_device_reg.stop = false;
3   stop_to_bus->write(true);
4   nextsection = idle;
5 }
```

Figure 5.14.: Code extract section `send_stop` from master SystemC-PPA model

By reaching the section `send_stop` the transaction is terminated with sending a STOP symbol to the slave module. First the variable `cfg_from_device_reg_stop` is resetted (line 2), second the STOP symbol is transmitted over the I^2C-bus (line 3) and third, execution changes to section `idle` to be able to process the next transaction.

5.3. Evaluation of the Results

We feed both SystemC-PPA models describing the master and slave behaviour respectively to the extended property automation framework as inputs. Two separate set of interval properties are generated in either SVA or ITL syntax as the extended Metaprop framework provides writers for both languages. We use the formal verification tool OneSpin 360 - Version 2017_06 [30] to prove the individual properties on the given I^2C implementation. First, we examine the resources consumed by OneSpin to perform the proofs. To do

so we run the formal verification tool with the following setup: Intel Xeon E5-2690 v3 @ 2.6GHz with 32GB RAM. The resource consumption is summarized separately for master and slave properties in table 5.2.

Table 5.2.: Resource evaluation

	# Properties	# Hold	# Unreachable	Time in s	Memory in MB
Slave	29	28	1	349	46701
Master	50	48	2	1451	86237

The slave SystemC-PPA module translates into 29 operation properties where 28 of them do hold for the given I^2C implementation, only one operation describes unreachable behaviour. An operation property describing unreachable behaviour means in the context of formal verification that the formal verification tool is not able to find a temporal trace in the implementation such that the operation's assumption part is satisfied. In other words the specific property is never triggered and therefore the presence of this property does not add any information to the proof. The reason for this specific property being unreachable is that the assumption part contradicts the applied constraints which ensure the correct configuration of the I^2C implementation. In detail the specific property models waiting for a synchronisation signal to evaluate to `true` which is already guaranteed to be `true` by constraint. Unreachable properties in general can be removed by including a SAT solver in the extended Metaprop framework which identifies unreachable behaviour. Performing IPC on the full set of slave properties took 349s and 46701 MB of memory were used by the formal verification tool.
The extended property automation framework generates a property suite consisting of 50 operation properties when provided with the SystemC-PPA master model as input. Again two operations describe unreachable behaviour but for another reason than explained before. In this case the properties' assumption part do not contradict the constraints which are applied to guarantee a correct configuration of the I^2C implementation but sub-formulas appearing in the assumption parts do contradict each other. This is due to the fact that in general the PPA considers all possible paths in the CFG interpretation of the MoT where some of these paths may not be reachable for the execution when the corresponding SystemC-PPA module is simulated. These paths translate to operation properties whose assumption parts are not satisfiable. These properties could also be removed from the generated property suite by including a SAT solver into the extended Metaprop framework as they do not add any information to the proofs. OneSpin needed 1451 s to prove every master property using the formal verification technique IPC and used 86237 MB of storage.
As a next step we inspect functional coverage by showing that both sets of generated properties are complete with respect to the completeness criterion introduced in section 2.1.3. Satisfying the completeness criterion means that there exist no gaps in the

verification process which means that the consecutive execution of operation properties covers the whole design behaviour. In other words, there exist no design behaviour that is not uniquely described in the set of properties. The completeness criterion can be evaluated automatically by the formal verification tool Onespin by executing several tests, namely case split test, successor test, determination test and reset test. The results are summarized in table 5.3.

Table 5.3.: Completeness evaluation

	Case Split Test	Successor Test	Determination Test	Reset Test
Slave	✓	✓	✓	✓
Master	✓	✓	✓	✓

As shown in table 5.3 all tests do pass for both property suites describing master and slave behaviour respectively. This is expected as the extended Metaprop framework is implemented specifically to generate property sets which satisfy the completeness criterion by design. We have now certified by applying the completeness criterion that the set of properties completely describe the desired design behaviour. By proving the full set of properties on the I^2C implementation as shown in 5.2 it is guaranteed that the implementation mimics the desired behaviour with respect to the property suite.
Note that there still exists a possibility that there is an error left in the implementation namely if some property explicitly contradicts the specification or if the verification is over-constrained. It may also occur that some functionality is missed in the SystemC-PPA model which is then not included in the generated set of properties which leads to the fact that this functionality cannot be proven on the implementation.
As we have shown 100% functional coverage by satisfying the completeness criterion we now inspect structural coverage for both, master and slave properties separately. This can be of benefit to measure the progress of verification on the one hand as well as to find dead or redundant RTL code. By running the structural code coverage evaluation in OneSpin it identifies all code locations in the RTL implementation and checks whether they are covered by a property of the given property suite. In terms of code coverage evaluation, code locations are statements which are treated as targets. Onespin classifies these targets into different categories which are defined in the following:

- **Covered:** The target is active in some witness trace and one or more formerly holding properties fail due to the manipulation of the target.
- **Uncovered:** There is **no** witness trace in which this target is active such that properties do not fail due to the manipulation of the target.
- **Dead:** The target is proven to be unreachable.

Code coverage information is collected by OneSpin and summarized in table 5.4.

Table 5.4.: Code coverage evaluation

	Slave	Master
Covered in %	21.90	62.38
Uncovered in %	72.62	32.14
Dead in %	5.48	5.48

The property set describing the desired slave behaviour verified 21.90 % of the I^2C implementation's overall code locations. 72.62 % of the code locations remain uncovered. There are several reasons for a code location being classified as uncovered: First, as explained the I^2C implementation can act as **both**, master and slave, whereas the slave property suite just describes desired slave behaviour such that all code locations relevant for master behaviour only are not covered intrinsically. Second, the I^2C implementation provides functionalities needed for multi-master configuration only, e.g functionality to detect bus collisions, which are not covered in the formal specification intentionally due to the fact that Infineon uses the I^2C-bus in single-master setup only. And third, the I^2C implementation provides debug features which are not covered in the formal specification as slave SystemC-PPA module either such that these debug features are not reflected in the generated property suite. 5.48 % of the code locations are classified as dead code, e.g statements which can never be reached by any witness trace. This is due to the fact that the given I^2C implementation implements additional features, e.g the 10-bit addressing mode, but these features are disabled in the implementation.
The property set describing the desired master behaviour reached 62.38 % code coverage. The reasons for 32.14 % (5.48 %) of the overall code locations being classified as uncovered (dead) code are equivalent to the ones explained above for the slave properties as both sets are proven on the same I^2C implementation.
Note that the code coverage results for master and slave cannot simply be added together because there exist code locations which are covered by both sets of properties which have to be accounted only once.

6. Future Work

In this chapter we present ideas to further improve the extended Metaprop framework in general as well as the generated property suite for the I^2C-bus protocol specification.
Depending on the SystemC-PPA model the extended Metaprop flow might produce operation properties describing unreachable behaviour in the generated property suite as we have seen in chapter 5.3. A property classified as unreachable by the formal verification tool means that the tool is not able to find a trace in the implementation starting from its initial state which satisfies the assumption part of the interval property. To resolve this issue a SAT solver can be included in the extended Metaprop flow which checks the assumption parts of the interval properties for contradictions. If contradicting expressions in the assumption part are found the property is removed from the generated property suite as it covers unreachable behaviour.
As already mentioned in chapter 5.1 the I^2C-bus protocol specification includes optional features, e.g. 10-Bit slave address, General Call address, Software Reset or START byte. Furthermore Infineon uses the I^2C-bus as single master configuration only such that synchronisation or arbitration mechanism do not have to be considered. The mentioned features as well as the multi master bus setup are currently not addressed in the SystemC-PPA models for slave and master and therefore cannot be proven for the implementation. Further extending the SystemC-PPA modules to cover these configurations such that these features are reflected in the generated property suites is a future task.
The MDEL determines the available constructs which can be used by the Verification Engineer to model the informal specification as SystemC-PPA model. In other words MDEL defines the available SystemC-PPA syntax which is a subset of the complete SystemC syntax. A future task is to extend the MDEL by adding constructs which are present in the SystemC syntax to provide more expressive power for modeling the informal specification. For example SystemC provides the opportunity to use 'functions' which groups a sequence of statements to ease the view of the SystemC model as well as for reusability. By extending the MDEL with the specific construct 'function' this functionality can also be supported in the extended Metaprop framework. Thinking this idea one step further means that the MDEL is not restricted to SystemC syntax but is able to support any construct of any design entry language. Especially Python as design entry language is of special interest as the extended Metaprop framework as well as the Metagen environment is implemented in Python.
In SystemC one has the possibility to model the module's behaviour using multiple processes which are running in parallel. In SystemC-PPA however we restrict the module

to exactly one process which defines the underlaying PPA implicitly. Concurrency is modeled using compound types to express that several variables are read or written at the same time. But there exist occasions where concurrently running processes are needed to fully model the desired hardware behaviour. For example, in the I^2C slave and master module communication events define specific timepoints where the module requires data from the device as input. The implementation however is able to accept input from the device at any arbitrary point in time which leads to the fact that the communication between module and device needs to be modeled as separate, concurrently running process to fully capture the implementation behaviour. This can be achieved by extending the semantics of SystemC-PPA such that the concept of multiple processes running in parallel is well defined with respect to the underlaying PPA.

A. Metamodels

A.1. Metaprop Metamodel

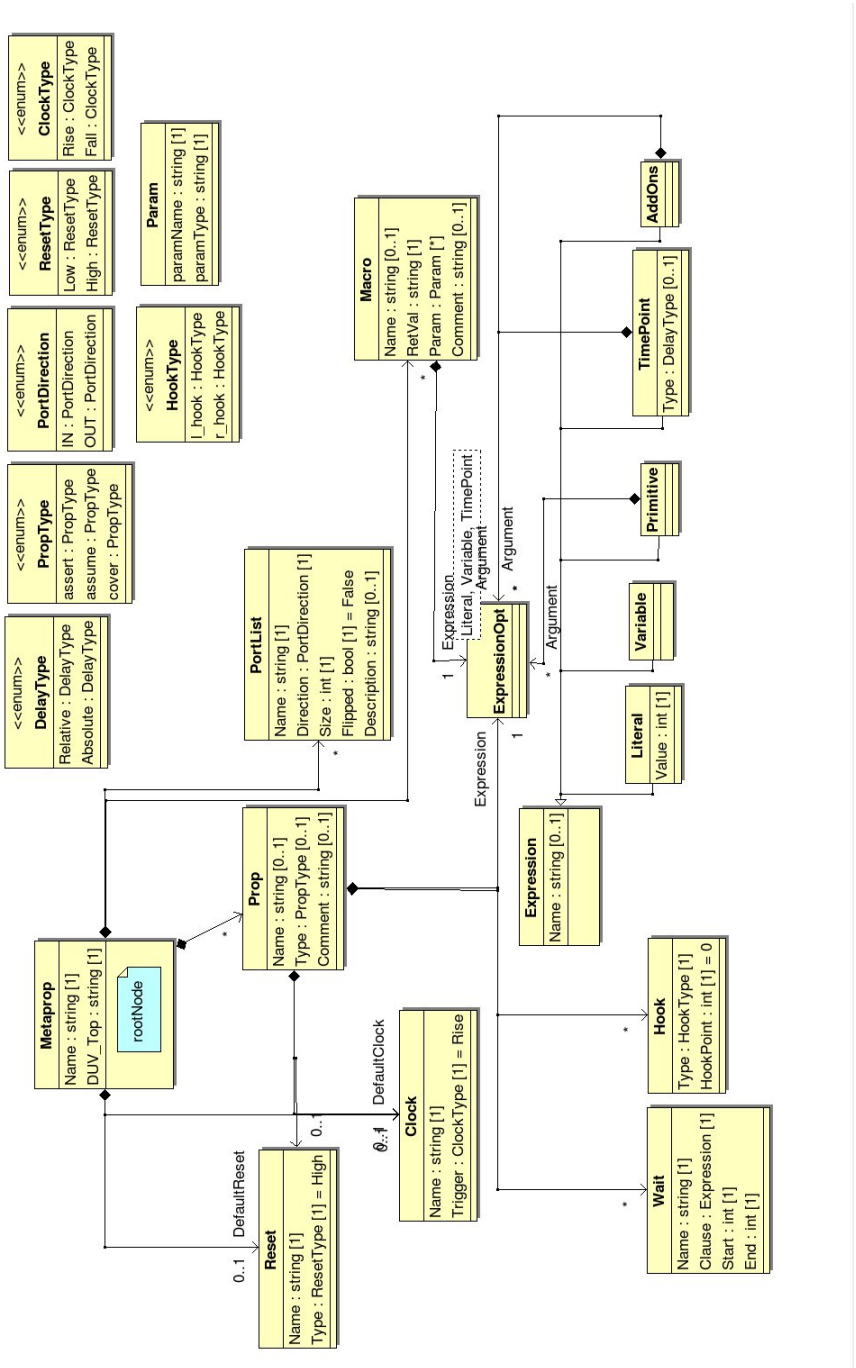


Figure A.1.: Metaprop Metamodel in the Model-of-Properties Layer

B. Generated Properties for Example Design

B.1. SVA Syntax

```
module Blinker_prop(
input [2:0] Counter_A_OUT,
input [1:0] Counter_B_OUT,
input [1:0] Counter_C_OUT,
input [2:0] Counter_D_OUT,
input ovf_OUT,
input clk,
input rst
);

//// Properties ////

property Counter_A_OUT_RST_prop;
(rst
|->
(Counter_A_OUT == 0));
endproperty

property Counter_B_OUT_RST_prop;
(rst
|->
(Counter_B_OUT == 0));
endproperty

property Counter_C_OUT_RST_prop;
(rst
|->
(Counter_C_OUT == 0));
endproperty

property Counter_D_OUT_RST_prop;
(rst
|->
(Counter_D_OUT == 0));
endproperty

property ovf_OUT_RST_prop;
(rst
|->
(ovf_OUT == 0));
endproperty

property Counter_A_OVF_prop;
@(posedge clk)
disable iff(rst)
(( (Counter_A_OUT == 0) &&
( $past(Counter_A_OUT) == 5) )
|->
##1
((Counter_B_OUT == ( $past(Counter_B_OUT) +
     1)) || (Counter_B_OUT == 0)) );
endproperty

property Counter_A_NO_OVF_prop;
@(posedge clk)
disable iff(rst)
(( !( (Counter_A_OUT == 0) &&
( $past(Counter_A_OUT) == 5) ))
|->
##1
(Counter_B_OUT == $past(Counter_B_OUT)) );
endproperty

property Counter_A_LSS_MAX_prop;
@(posedge clk)
disable iff(rst)
(1
|->
(Counter_A_OUT <= 5));
endproperty

property Counter_A_GRT_MIN_prop;
@(posedge clk)
disable iff(rst)
(1
|->
(Counter_A_OUT >= 0));
endproperty
```

```
property Counter_A_CNTR_INC_prop;
@(posedge clk)
disable iff(rst)
((Counter_A_OUT != 0)
|->
((Counter_A_OUT == $past(Counter_A_OUT)) ||
     (Counter_A_OUT == ( $past(
    Counter_A_OUT) + 1))));
endproperty

property Counter_A_CNTR_RES_prop;
@(posedge clk)
disable iff(rst)
((Counter_A_OUT == 0)
|->
((Counter_A_OUT == $past(Counter_A_OUT)) ||
     ( $past(Counter_A_OUT) == 5)));
endproperty

property Counter_B_OVF_prop;
@(posedge clk)
disable iff(rst)
(( (Counter_B_OUT == 0) &&
( $past(Counter_B_OUT) == 3) )
|->
##1
((Counter_C_OUT == ( $past(Counter_C_OUT) +
     1)) || (Counter_C_OUT == 0)) );
endproperty

property Counter_B_NO_OVF_prop;
@(posedge clk)
disable iff(rst)
(( !( (Counter_B_OUT == 0) &&
( $past(Counter_B_OUT) == 3) ))
|->
##1
(Counter_C_OUT == $past(Counter_C_OUT)) );
endproperty

property Counter_B_LSS_MAX_prop;
@(posedge clk)
disable iff(rst)
(1
|->
(Counter_B_OUT <= 3));
endproperty

property Counter_B_GRT_MIN_prop;
@(posedge clk)
disable iff(rst)
(1
|->
(Counter_B_OUT >= 0));
endproperty

property Counter_B_CNTR_INC_prop;
@(posedge clk)
disable iff(rst)
((Counter_B_OUT != 0)
|->
((Counter_B_OUT == $past(Counter_B_OUT)) ||
     (Counter_B_OUT == ( $past(
    Counter_B_OUT) + 1))));
endproperty

property Counter_B_CNTR_RES_prop;
@(posedge clk)
disable iff(rst)
((Counter_B_OUT == 0)
|->
((Counter_B_OUT == $past(Counter_B_OUT)) ||
     ( $past(Counter_B_OUT) == 3)));
endproperty

property Counter_C_OVF_prop;
@(posedge clk)
disable iff(rst)
(( (Counter_C_OUT == 0) &&
( $past(Counter_C_OUT) == 2) )
|->
##1
((Counter_D_OUT == ( $past(Counter_D_OUT) +
     1)) || (Counter_D_OUT == 0)) );
endproperty

property Counter_C_NO_OVF_prop;
@(posedge clk)
disable iff(rst)
(( !( (Counter_C_OUT == 0) &&
( $past(Counter_C_OUT) == 2) ))
|->
##1
(Counter_D_OUT == $past(Counter_D_OUT)) );
endproperty

property Counter_C_LSS_MAX_prop;
@(posedge clk)
disable iff(rst)
(1
|->
(Counter_C_OUT <= 2));
endproperty

property Counter_C_GRT_MIN_prop;
```

```
@(posedge clk)
disable iff(rst)
(1
|->
(Counter_C_OUT >= 0));
endproperty

property Counter_C_CNTR_INC_prop;
@(posedge clk)
disable iff(rst)
((Counter_C_OUT != 0)
|->
((Counter_C_OUT == $past(Counter_C_OUT)) ||
     (Counter_C_OUT == ( $past(
    Counter_C_OUT) + 1))));
endproperty

property Counter_C_CNTR_RES_prop;
@(posedge clk)
disable iff(rst)
((Counter_C_OUT == 0)
|->
((Counter_C_OUT == $past(Counter_C_OUT)) ||
     ( $past(Counter_C_OUT) == 2)));
endproperty

property Counter_D_OVF_prop;
@(posedge clk)
disable iff(rst)
(( (Counter_D_OUT == 0) &&
( $past(Counter_D_OUT) == 7) )
|->
(ovf_OUT == 1));
endproperty

property Counter_D_NO_OVF_prop;
@(posedge clk)
disable iff(rst)
((!(Counter_D_OUT == 0) && ( $past(
    Counter_D_OUT) == 7))
|->
(ovf_OUT == 0));
endproperty

property Counter_D_LSS_MAX_prop;
@(posedge clk)
disable iff(rst)
(1
|->
(Counter_D_OUT <= 7));
endproperty

property Counter_D_GRT_MIN_prop;
@(posedge clk)
disable iff(rst)
(1
|->
(Counter_D_OUT >= 0));
endproperty

property Counter_D_CNTR_INC_prop;
@(posedge clk)
disable iff(rst)
((Counter_D_OUT != 0)
|->
((Counter_D_OUT == $past(Counter_D_OUT)) ||
     (Counter_D_OUT == ( $past(
    Counter_D_OUT) + 1))));
endproperty

property Counter_D_CNTR_RES_prop;
@(posedge clk)
disable iff(rst)
((Counter_D_OUT == 0)
|->
((Counter_D_OUT == $past(Counter_D_OUT)) ||
     ( $past(Counter_D_OUT) == 7)));
endproperty

Counter_A_OUT_RST_prop_assert: assert
    property(Counter_A_OUT_RST_prop);
Counter_B_OUT_RST_prop_assert: assert
    property(Counter_B_OUT_RST_prop);
Counter_C_OUT_RST_prop_assert: assert
    property(Counter_C_OUT_RST_prop);
Counter_D_OUT_RST_prop_assert: assert
    property(Counter_D_OUT_RST_prop);
ovf_OUT_RST_prop_assert: assert property(
    ovf_OUT_RST_prop);
Counter_A_OVF_prop_assert: assert property(
    Counter_A_OVF_prop);
Counter_A_NO_OVF_prop_assert: assert
    property(Counter_A_NO_OVF_prop);
Counter_A_LSS_MAX_prop_assert: assert
    property(Counter_A_LSS_MAX_prop);
Counter_A_GRT_MIN_prop_assert: assert
    property(Counter_A_GRT_MIN_prop);
Counter_A_CNTR_INC_prop_assert: assert
    property(Counter_A_CNTR_INC_prop);
Counter_A_CNTR_RES_prop_assert: assert
    property(Counter_A_CNTR_RES_prop);
Counter_B_OVF_prop_assert: assert property(
    Counter_B_OVF_prop);
Counter_B_NO_OVF_prop_assert: assert
    property(Counter_B_NO_OVF_prop);
```

```
Counter_B_LSS_MAX_prop_assert: assert
    property(Counter_B_LSS_MAX_prop);
Counter_B_GRT_MIN_prop_assert: assert
    property(Counter_B_GRT_MIN_prop);
Counter_B_CNTR_INC_prop_assert: assert
    property(Counter_B_CNTR_INC_prop);
Counter_B_CNTR_RES_prop_assert: assert
    property(Counter_B_CNTR_RES_prop);
Counter_C_OVF_prop_assert: assert property(
    Counter_C_OVF_prop);
Counter_C_NO_OVF_prop_assert: assert
    property(Counter_C_NO_OVF_prop);
Counter_C_LSS_MAX_prop_assert: assert
    property(Counter_C_LSS_MAX_prop);
Counter_C_GRT_MIN_prop_assert: assert
    property(Counter_C_GRT_MIN_prop);
Counter_C_CNTR_INC_prop_assert: assert
    property(Counter_C_CNTR_INC_prop);
Counter_C_CNTR_RES_prop_assert: assert
    property(Counter_C_CNTR_RES_prop);
Counter_D_OVF_prop_assert: assert property(
    Counter_D_OVF_prop);
Counter_D_NO_OVF_prop_assert: assert
    property(Counter_D_NO_OVF_prop);
Counter_D_LSS_MAX_prop_assert: assert
    property(Counter_D_LSS_MAX_prop);
Counter_D_GRT_MIN_prop_assert: assert
    property(Counter_D_GRT_MIN_prop);
Counter_D_CNTR_INC_prop_assert: assert
    property(Counter_D_CNTR_INC_prop);
Counter_D_CNTR_RES_prop_assert: assert
    property(Counter_D_CNTR_RES_prop);

endmodule
bind Blinker Blinker_prop inst_Blinker_prop
    (.*);
```

B.2. ITL Syntax

```
property Counter_A_OUT_RST_prop is
assume:
at t: rst;
prove:
at t: (Counter_A_OUT = 0);
end property;

property Counter_B_OUT_RST_prop is
assume:
at t: rst;
prove:
at t: (Counter_B_OUT = 0);
end property;

property Counter_C_OUT_RST_prop is
assume:
at t: rst;
prove:
at t: (Counter_C_OUT = 0);
end property;

property Counter_D_OUT_RST_prop is
assume:
at t: rst;
prove:
at t: (Counter_D_OUT = 0);
end property;

property ovf_OUT_RST_prop is
assume:
at t: rst;
prove:
at t: (ovf_OUT = 0);
end property;

property Counter_A_OVF_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: ((Counter_A_OUT = '0') and (prev(
    Counter_A_OUT) = 5));
prove:
at t + 1: ((Counter_B_OUT = (prev(
    Counter_B_OUT) + '1')) or (
    Counter_B_OUT = '0'));
end property;

property Counter_A_NO_OVF_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: not(((Counter_A_OUT = '0') and (prev(
    Counter_A_OUT) = 5)));
prove:
at t + 1: (Counter_B_OUT = prev(
    Counter_B_OUT));
end property;

property Counter_A_LSS_MAX_prop is
local trigger:rose(clk);
```

```
disable iff:(rst);
assume:
at t: '1';
prove:
at t: (Counter_A_OUT <= 5);
end property;

property Counter_A_GRT_MIN_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: '1';
prove:
at t: (Counter_A_OUT >= '0');
end property;

property Counter_A_CNTR_INC_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: (Counter_A_OUT /= '0');
prove:
at t: ((Counter_A_OUT = prev(Counter_A_OUT)
    ) or (Counter_A_OUT = (prev(
    Counter_A_OUT) + '1')));
end property;

property Counter_A_CNTR_RES_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: (Counter_A_OUT = '0');
prove:
at t: ((Counter_A_OUT = prev(Counter_A_OUT)
    ) or (prev(Counter_A_OUT) = 5));
end property;

property Counter_B_OVF_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: ((Counter_B_OUT = '0') and (prev(
    Counter_B_OUT) = 3));
prove:
at t + 1: ((Counter_C_OUT = (prev(
    Counter_C_OUT) + '1')) or (
    Counter_C_OUT = '0'));
end property;

property Counter_B_NO_OVF_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: not(((Counter_B_OUT = '0') and (prev(
    Counter_B_OUT) = 3)));
prove:
at t + 1: (Counter_C_OUT = prev(
    Counter_C_OUT));
end property;

property Counter_B_LSS_MAX_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: '1';
prove:
at t: (Counter_B_OUT <= 3);
end property;

property Counter_B_GRT_MIN_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: '1';
prove:
at t: (Counter_B_OUT >= '0');
end property;

property Counter_B_CNTR_INC_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: (Counter_B_OUT /= '0');
prove:
at t: ((Counter_B_OUT = prev(Counter_B_OUT)
    ) or (Counter_B_OUT = (prev(
    Counter_B_OUT) + '1')));
end property;

property Counter_B_CNTR_RES_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: (Counter_B_OUT = '0');
prove:
at t: ((Counter_B_OUT = prev(Counter_B_OUT)
    ) or (prev(Counter_B_OUT) = 3));
end property;

property Counter_C_OVF_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: ((Counter_C_OUT = '0') and (prev(
    Counter_C_OUT) = 2));
prove:
```

```
at t + 1: ((Counter_D_OUT = (prev(
    Counter_D_OUT) + '1')) or (
    Counter_D_OUT = '0'));
end property;

property Counter_C_NO_OVF_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: not(((Counter_C_OUT = '0') and (prev(
    Counter_C_OUT) = 2)));
prove:
at t + 1: (Counter_D_OUT = prev(
    Counter_D_OUT));
end property;

property Counter_C_LSS_MAX_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: '1';
prove:
at t: (Counter_C_OUT <= 2);
end property;

property Counter_C_GRT_MIN_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: '1';
prove:
at t: (Counter_C_OUT >= '0');
end property;

property Counter_C_CNTR_INC_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: (Counter_C_OUT /= '0');
prove:
at t: ((Counter_C_OUT = prev(Counter_C_OUT)
    ) or (Counter_C_OUT = (prev(
    Counter_C_OUT) + '1')));
end property;

property Counter_C_CNTR_RES_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: (Counter_C_OUT = '0');
prove:
at t: ((Counter_C_OUT = prev(Counter_C_OUT)
    ) or (prev(Counter_C_OUT) = 2));
end property;

property Counter_D_OVF_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: ((Counter_D_OUT = '0') and (prev(
    Counter_D_OUT) = 7));
prove:
at t: (ovf_OUT = '1');
end property;

property Counter_D_NO_OVF_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: ((Counter_D_OUT = '0') nand (prev(
    Counter_D_OUT) = 7));
prove:
at t: (ovf_OUT = '0');
end property;

property Counter_D_LSS_MAX_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: '1';
prove:
at t: (Counter_D_OUT <= 7);
end property;

property Counter_D_GRT_MIN_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: '1';
prove:
at t: (Counter_D_OUT >= '0');
end property;

property Counter_D_CNTR_INC_prop is
local trigger:rose(clk);
disable iff:(rst);
assume:
at t: (Counter_D_OUT /= '0');
prove:
at t: ((Counter_D_OUT = prev(Counter_D_OUT)
    ) or (Counter_D_OUT = (prev(
    Counter_D_OUT) + '1')));
end property;

property Counter_D_CNTR_RES_prop is
local trigger:rose(clk);
```

```
disable iff:(rst);
assume:
at t: (Counter_D_OUT = '0');
prove:
at t: ((Counter_D_OUT = prev(Counter_D_OUT)
    ) or (prev(Counter_D_OUT) = 7));
end property;
```

C. I^{2}C-Bus Features

C.1. START and STOP Symbol

All transactions have to start with a START symbol and need to end with a STOP symbol (see figure C.1). Both symbols are always generated by the initator of the transaction which is by definition the master. The bus is free for usage for any device if the SDA as well as the SCL line are HIGH. A START symbol is encoded as a HIGH to LOW transition on the SDA wire while SCL wire is HIGH. The number of bytes that can be transmitted per transaction is unrestricted. After the complete transaction has finished (all bytes are transferred) the STOP symbol is sent by the master. The STOP symbol is defined as LOW to HIGH transition on the SDA wire while SCL is HIGH. Furhtermore the master has the opportunity to restart the transaction during an ongoing data transfer by sending a so-called repeated START symbol which is encoded equivalent as the START symbol.

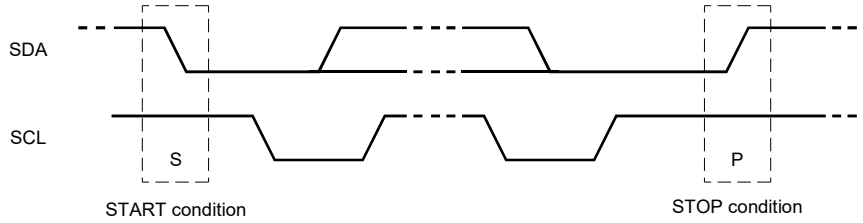


Figure C.1.: START and STOP Symbols[1]

C.2. Acknowledge

The Acknowledge (ACK) bit is used by the receiver to notify the transmitter that a byte was successfully transfered. This also implies that the receiver is ready for another byte to be sent. It is defined as follows: The transmitter releases the SDA wire during the ninth clock period after START symbol (acknowledgement clock phase) such that the receiver can set the SDA wire to LOW to indicate an ACK (see figure C.2). The receiver also has the possbility to send a Not Acknowledge (NACK) which is represented by the SDA wire being HIGH during the ninth clock period. This indicates that the transactions was not successfully processed which for example can be due to the fact that simply no receiver is present on the bus with the associated address. Given this scenario the SDA

remains HIGH during the ninth clock period because there is no driver to pull it to LOW which is decoded as NACK by the master side. After receiving a NACK the transmitter has the possibility either to restart the transfer by sending a repeated START symbol or abort the transaction by sending a STOP condition.

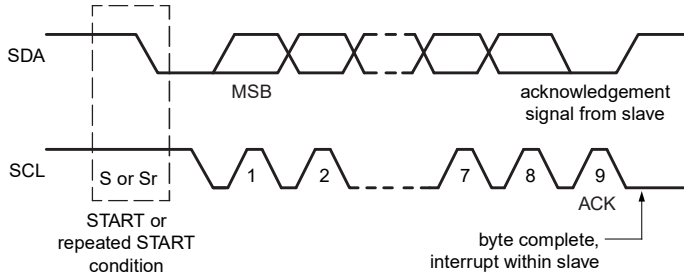


Figure C.2.: Data transfer including Acknowledge bit[1]

C.3. Clock Stretching

Clock Stretching means holding the SCL line low and therby forcing the communication counterpart of the transaction into a wait state. Clock Stretching makes use of the physical bus setup, namely the wired-AND connection between I^2C-interface and SCL/SDA line. To be connected using a wired-AND means, if a device pulls the SCL/SDA line LOW it stays LOW even if another devices tries to set it to HIGH. Figure C.3 shows a possible scenario where Clock Stretching is needed.

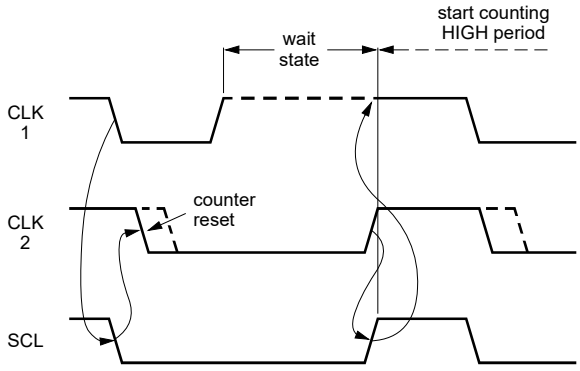


Figure C.3.: Clock Synchronization[1]

Assume CLK1 is associated with the master which transmits data to the slave whereas CLK2 corresponds to the slave which receives data from the former. In general the master drives the SCL wire with a specific frequency. In this scenario the slave is not able to store data as quickly as it is able to receive data. To solve this situation the slave

stretches the clock which means holding the SCL line LOW as long as needed to store the received byte, forcing the transmitter into a wait state. After releasing the SCL line again the receiver is ready to read the next byte. Note that this mechanism leads to the fact the I^2C-bus runs at a lower baudrate than specified.

D. SystemC-PPA models

D.1. I²C-Bus Protocol Slave Model

```
1  class Slave : public sc_module {
2  public:
3  SC_CTOR(Slave) :
4  nextsection(idle) {SC_THREAD(fsm)};
5  struct status_t {
6    bool start;
7    bool stop;
8  };
9
10 enum Sections {
11 idle, get_addr, receive_data, transmit_data
12 };
13 Sections section, nextsection;
14
15 /* Communcation from bus */
16 blocking_in<int> data_from_bus;
17 blocking_in<int> address_from_bus;
18 blocking_in<status_t> status_from_bus;
19 blocking_in<bool> ack_from_bus;
20
21 /* Communication to bus */
22 blocking_out<int> data_to_bus;
23 blocking_out<bool> ack_to_bus;
24
25
26 /* Communication with device */
27 shared_out<int> data_to_device;
28 blocking_in<int> data_from_device;
29
30 int data_from_device_reg;
31 int data_from_bus_reg;
32 status_t status_reg;
33 bool RnW_reg;
34 bool ack_reg;
35 int device_addr;
36
37
38 void fsm() {
39   while (true) {
40     section = nextsection;
41     if (section == idle) {
```

```
42        /* Read status from bus */
43        status_from_bus->read(status_reg);
44        if (status_reg.start) {
45          /* START symbol detected on the bus | Change to section 'get_addr' */
46          nextsection = get_addr;
47        } else {
48          /* No START symbol detected on the bus | Iterate over section 'idle' */
49        }
50      } else if (section == get_addr) {
51        /* Read address from bus */
52        address_from_bus->read(data_from_bus_reg);
53        /* Check whether Slave is addressed */
54        if ((data_from_bus_reg >> 1) == device_addr) {
55          /* Slave is addressed | Write data to device */
56          data_to_device->set(data_from_bus_reg);
57          /* Extract RnW bit from address */
58          RnW_reg = (data_from_bus_reg << 7) == 128;
59          /* Send acknowledgment to bus */
60          ack_to_bus->write(true);
61          /* Check whether RnW-bit is asserted */
62          if (RnW_reg) {
63            /* RnW asserted | Read transaction -> slave needs to transmit data | Change
                 section to 'transmit_data' */
64            nextsection = transmit_data;
65          } else {
66            /* RnW not asserted | Write transaction -> slave needs to receive data | Change
                  section to 'receive_data' */
67            nextsection = receive_data;
68          }
69        } else {
70          /* Slave not addressed | Change section to 'idle' */
71          nextsection = idle;
72        }
73      } else if (section == transmit_data) {
74        /* Read data from device */
75        data_from_device->read(data_from_device_reg);
76        /* Read status from bus */
77        status_from_bus->read(status_reg);
78        if (status_reg.start && !status_reg.stop) {
79          /* START symbol detected | Restart transaction | Change section to 'get_addr'
80          nextsection = get_addr;
81        } else if (status_reg.stop && !status_reg.start) {
82          /* STOP symbol detected | Stop transaction | Reset RnW-bit */
83          RnW_reg = false;
84          /* Change to section 'idle' */
85          nextsection = idle;
86        } else {
87          /* Write data to bus */
88          data_to_bus->write(data_from_device_reg);
89          /* Receive acknowledgement from bus */
90          ack_from_bus->read(ack_reg);
91          /* Check whether master sent ACK or NACK */
92          if (!ack_reg) {
```

```
93          /* Master sent NACK | Change section to 'idle' */
94          nextsection = idle;
95        }
96      }
97    } else if (section == receive_data) {
98      /* Read status from bus */
99      status_from_bus->read(status_reg);
100     if (status_reg.start && !status_reg.stop) {
101       /* START symbol detected | Restart transaction | Change section to 'get_addr' */
102       nextsection = get_addr;
103     } else if (status_reg.stop && !status_reg.start) {
104       /* STOP symbol detected | Stop transaction | Change section to 'idle' */
105       nextsection = idle;
106     } else {
107       /* Neither START or STOP symbol detected | Read data from bus */
108       data_from_bus->read(data_from_bus_reg);
109       /* Write data to device */
110       data_to_device->set(data_from_bus_reg);
111       /* Send acknowledgement to bus */
112       ack_to_bus->write(true);
113     }
114   }
115 }
116 }
117 };
```

D.2. I²C-Bus Protocol Master Model

```
1  class Master: public sc_module{
2  public:
3  SC_CTOR(Master):
4  nextsection(setup){SC_THREAD(fsm)};
5  struct cfg_t{
6    bool start;
7    bool restart;
8    bool stop;
9    bool ack;
10 };
11
12 enum Sections {idle,send_addr,send_status,receive_data,transmit_data,send_restart,
       send_stop};
13 Sections section,nextsection;
14
15 /* Communcation from bus */
16 blocking_in<bool> ack_from_bus;
17 blocking_in<int> data_from_bus;
18
19 /* Communication to bus */
20 blocking_out<bool> start_to_bus;
21 blocking_out<bool> restart_to_bus;
```

```
22 blocking_out<bool> stop_to_bus;
23 blocking_out<int> data_to_bus;
24 blocking_out<bool> ack_to_bus;
25
26 /* Communication with device */
27 blocking_in<int> data_from_device;
28 blocking_in<cfg_t> cfg_from_device;
29 shared_out<int> data_to_device;
30
31 /* Visible registers */
32 int data_from_device_reg;
33 cfg_t cfg_from_device_reg;
34 bool RnW_reg;
35 bool ack_reg;
36 int data_from_bus_reg;
37 bool received_data_reg;
38
39
40 void fsm() {
41   while (true) {
42     section = nextsection;
43     if (section == idle) {
44       /* Read configuration from device */
45       cfg_from_device->read(cfg_from_device_reg);
46       /* Start transaction only if START flag was set by device */
47       if(cfg_from_device_reg.start) {
48         /* START flag asserted from device | Read data from device */
49         data_from_device->read(data_from_device_reg);
50         /* Clear START flag */
51         cfg_from_device_reg.start = false;
52         /* Store RnW-bit */
53         RnW_reg = (data_from_device_reg << 7) == 128;
54         /* Write START symbol to bus */
55         start_to_bus->write(true);
56         /* Change section to 'send_addr' */
57         nextsection = send_addr;
58       } else {
59         /* START flag not asserted by device | Iterate over section 'idle' */
60       }
61     } else if (section == send_addr) {
62       /* Write address and RnW-bit to bus */
63       data_to_bus->write(data_from_device_reg);
64       /* Read acknowledgement from bus */
65       ack_from_bus->read(ack_reg);
66       if(ack_reg && RnW_reg) {
67         /* Slave sent ACK | RnW flag asserted | Change section to 'receive_data' */
68         nextsection = receive_data;
69       } else if (ack_reg && !RnW_reg) {
70         /* Slave sent ACK | RnW flag not asserted | Change to 'transmit_data' */
71         nextsection = transmit_data;
72       } else {
73         /* Change to section 'send_status' */
74         nextsection = send_status;
```

```
75        }
76      } else if (section == send_status) {
77        /* Read configuration from device */
78        cfg_from_device->read(cfg_from_device_reg);
79        if (cfg_from_device_reg.restart) {
80          /* RESTART flag asserted by device | Read new address from device */
81          data_from_device->read(data_from_device_reg);
82          /* Change to section 'send_restart' */
83          nextsection = send_restart;
84        } else if (cfg_from_device_reg.stop) {
85          /* STOP flag asserted by device | Change to section 'send_stop' */
86          nextsection = send_stop;
87        } else {
88          /* Read configuration from device again | Iterate over section 'send_status' */
89        }
90      } else if (section == receive_data) {
91        /* Read data from bus */
92        data_from_bus->read(data_from_bus_reg);
93        /* Forward read data to device */
94        data_to_device->set(data_from_bus_reg);
95        /* Read configuration from device */
96        cfg_from_device->read(cfg_from_device_reg);
97        if (cfg_from_device_reg.ack) {
98          /* ACK flag is asserted by device | write ACK to bus */
99          ack_to_bus->write(true);
100       } else if (cfg_from_device_reg.restart) {
101         /* RESTART flag asserted by device | Read new address from device */
102         received_data_reg = data_from_device->nb_read(data_from_device_reg);
103         /* Check whether address was provided by device */
104         if (received_data_reg) {
105           /* Address provided by device | Change to section 'send_restart' */
106           nextsection = send_restart;
107         } else {
108           /* Address not provided by device | Change to section 'send_stop' */
109           nextsection = send_stop;
110         }
111       } else {
112         /* Change to section 'send_stop' */
113         nextsection = send_stop;
114       }
115     } else if (section == transmit_data) {
116       /* Read configuration from device */
117       cfg_from_device->read(cfg_from_device_reg);
118       if (cfg_from_device_reg.restart) {
119         /* RESTART flag asserted by device | Read new address from device */
120         data_from_device->read(data_from_device_reg);
121         /* Change to section 'send_restart' */
122         nextsection = send_restart;
123       } else if (cfg_from_device_reg.stop) {
124         /* STOP flag asserted by device | Change to section 'send_stop' */
125         nextsection = send_stop;
126       } else {
127         /* Read data from device */
```

```
128         data_from_device->read(data_from_device_reg);
129         /* Write data to bus */
130         data_to_bus->write(data_from_device_reg);
131         /* Read acknowledgement from bus */
132         ack_from_bus->read(ack_reg);
133         if (!ack_reg) {
134           /* NACK sent from slave | Change section to 'send_status' */
135           nextsection = send_status;
136         }
137       }
138     } else if (section == send_restart) {
139       /* RESTART flag was asserted by device | Extract RnW-bit from data */
140       RnW_reg = (data_from_device_reg << 7) == 128;
141       /* Clear RESTART flag */
142       cfg_from_device_reg.restart = false;
143       /* Send RESTART symbol to bus */
144       restart_to_bus->write(true);
145       /* Change section to 'send_addr' */
146       nextsection = send_addr;
147     } else if (section == send_stop) {
148       /* Clear STOP flag */
149       cfg_from_device_reg.stop = false;
150       /* Send STOP symbol to bus */
151       stop_to_bus->write(true);
152       /* Change section to 'idle' */
153       nextsection = idle;
154     }
155   }
156 }
157 };
```

List of Figures

List of Tables

Bibliography

[1] *I^2C-bus specification and user manual*. English. Version v.6. NXP Semiconductors. 2014.

[2] Berkeley Architecture Research. *Chisel*. Accessed: 2019-25-03. URL: `https://chisel.eecs.berkeley.edu/`.

[3] Bormann, J. "Vollständige funktionale Verifikation". PhD thesis. Technische Universität Kaiserslautern, Germany, 2009.

[4] Chu, X. "Combination of Assertion and Coverage-Driven Verification Methodology". In: *MICROELECTRONICS & COMPUTER* (2008), pp. 1–1315.

[5] Clarke, E., Emerson, E. A., and Sistla, A. P. "Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications". In: *ACM Trans. Program. Lang. Syst.* (1986), pp. 244–263.

[6] Clarke, E., Biere, A., Raimi, R., and Zhu, Y. "Bounded Model Checking Using Satisfiability Solving". In: *Form. Methods Syst. Des.* (2001), pp. 7–34.

[7] Collett, R. and Pyle, D. *What happens when chip-design complexity outpaces development productivity?* Accessed: 2019-25-03. URL: `https://www.mckinsey.com/~/media/McKinsey/dotcom/client_service/Semiconductors/Issue%203%20Autumn%202013/PDFs/4_ChipDesign.ashx`.

[8] Devarajegowda, K. and Ecker, W. "Meta-model Based Automation of Properties for Pre-Silicon Verification". In: *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. 2018, pp. 231–236.

[9] Ecker, W., Velten, M., Zafari, L., and Goyal, A. "The metamodeling approach to system level synthesis". In: *2014 Design, Automation Test in Europe Conference Exhibition*. 2014, pp. 1–2.

[10] Görke, W. *Fehlertolerante Rechensysteme*. Munich: Oldenburg Verlag, 1989.

[11] Henftling, R., Zinn, A., Bauer, M., Zambaldi, M., and Ecker, W. "Re-use-centric architecture for a fully accelerated testbench environment". In: *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*. 2003, pp. 372–375.

[12] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language". In: *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018), pp. 1–1315.

[13] "IEEE Standard for Universal Verification Methodology Language Reference Manual". In: *IEEE Std 1800.2-2017* (2017), pp. 1–472.

[14] "IEEE Standard Verilog Hardware Description Language". In: *IEEE Std 1364-2001* (2001), pp. 1–856.

[15] "IEEE Standard VHDL Language Reference Manual". In: *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (2009), pp. c1–626.

[16] Initiative, A. S. *Universal Verification Methodology 1.1 User's guide*. English. Version 1.1. Accellera. 190 pp.

[17] John Aynsley, D. L. "IEEE Standard for Standard SystemC Language Reference Manual". In: *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)* (2012).

[18] Karp, R. M. "Reducibility Among Combinatorial Problems". In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*. 1972, pp. 85–103.

[19] Kim, Y.-I. and Kyung, C.-M. "TPartition: testbench partitioning for hardware-accelerated functional verification". In: *IEEE Design Test of Computers* (2004), pp. 484–493.

[20] Kim, Y.-I., Yang, W., Kwon, Y.-S., and Kyung, C.-M. "Communication-efficient hardware acceleration for fast functional simulation". In: *Proceedings. 41st Design Automation Conference, 2004.* 2004, pp. 293–298.

[21] Koczor, A., Matoga, Ł., Penkala, P., and Pawlak, A. "Verification approach based on emulation technology". In: *2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. 2016, pp. 1–6.

[22] Kropf, T. *Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems*. 1st. Berlin, Heidelberg: Springer-Verlag, 1999. ISBN: 3540654453.

[23] Kudlugi, M., Hassoun, S., Selvidge, C., and Pryor, D. "A transaction-based unified simulation/emulation architecture for functional verification". In: *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*. 2001, pp. 623–628.

[24] L. Foundation. *LLVM*. Accessed: 2019-02-03. URL: `http://llvm.org/`.

[25] Ludwig, T., Schwarz, M., Urdahl, J., Deutschmann, L., Hetalani, S., Stoffel, D., and Kunz, W. "Property-Driven Development of a RISC-V CPU". submitted.

[26] Mavroidis, I., Mavroidis, I., and Papaefstathiou, I. "Accelerating Emulation and Providing Full Chip Observability and Controllability". In: *IEEE Design Test of Computers* (2009), pp. 84–94.

[27] Mentor. *The 2018 Wilson Research Group Functional Verification Study*. Accessed: 2019-25-03. URL: https://blogs.mentor.com/verificationhorizons/blog/2018/11/14/prologue-the-2018-wilson-research-group-functional-verification-study/.

[28] Narayan, R. and Symons, T. *I created the Verification Gap*. Accessed: 2019-25-03. URL: http://events.dvcon.org/2015/proceedings/papers/10_1.pdf.

[29] Nguyen, M. D., Thalmaier, M., Wedler, M., Bormann, J., Stoffel, D., and Kunz, W. "Unbounded Protocol Compliance Verification Using Interval Property Checking With Invariants". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2008), pp. 2068–2082.

[30] OneSpin Solutions GmbH. URL: https://www.onespin.com/.

[31] Pnueli, A. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 1977, pp. 46–57.

[32] Rizzatti, L. *Hardware Emulation: Three Decades of Evolution—Part III*. 2015. URL: https://www.mentor.com/products/fv/verificationhorizons/volume11/issue3/hardware-emulation-3-decades-evolution-part3 (visited on 11/28/2018).

[33] *SCAM*. Accessed: 2019-02-03. URL: https://github.com/ludwig247/SCAM.

[34] Truyen, F. "The Fast Guide to Model Driven Architecture The Basics of Model Driven Architecture". In: (2006), p. 16.

[35] Urdahl, J., Stoffel, D., and Kunz, W. "Path Predicate Abstraction for Sound System-Level Models of RT-Level Circuit Designs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2014), pp. 291–304.

[36] Urdahl, J. "Path Predicate Abstraction for Sound System-Level Modeling of Digital Circuits". dissertation. 2015.

[37] Yuan, J., Pixley, C., Aziz, A., and Albin, K. "A framework for constrained functional verification". In: *ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No.03CH37486)*. 2003, pp. 142–145.