Model-based Generation of Assertions for Pre-silicon Verification

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

Doktor der Ingenieurswissenschaften (Dr.-Ing.)

genehmigte Dissertation von

Keerthikumara Devarajegowda
geboren in Varadalapura, Karnataka, India

D 386

Datum der Einreichung:	12.Januar.2021
Datum der mündlichen Prüfung:	25.March.2021
Dekan des Fachbereichs:	Prof. Dr.-Ing. Ralph Urbansky
Vorsitzender der Prüfungskommission:	Prof. Dipl.-Ing. Dr. Gerhard Fohler
Gutachter:	Prof. Dr.-Ing. Wolfgang Kunz and
	Prof. Dr.-Ing. Wolfgang Ecker

Acknowledegments

The work conducted in this thesis documents research work carried out at Infineon Technologies AG, Germany in collaboration with the EDA Chair at Technische Universität Kaiserslautern, Germany. Also, the work on processor verification involved collaboration with the Robust Systems Group at Stanford University, United States chaired by Prof. Subhasish Mitra.

I would like to express my gratitude to Prof. Wolfgang Kunz, Prof. Wolfgang Ecker and Stefan Litzenberger for giving me an opportunity to pursue this doctoral work.

Most of all, I am immensely thankful to Prof. Wolfgang Ecker for the incredible support, constructive discussions and for guiding me through this doctoral work. Also, I am extremely thankful to Prof. Wolfgang Kunz and Prof. Dominik Stoffel for the technical discussions, constructive suggestions and for guiding me through different aspects of this doctoral work. My special thanks to Prof. Stoffel for his in-depth review of the thesis and for the constructive feedback. I would like to express my gratitude to Prof. Mitra for giving me an opportunity to work with his group. I would like to thank Prof. Gerhard Fohler for agreeing to chair the examination committee.

I am thankful to friends and colleagues who contributed in many different ways: Johannes Schreiner, Heimo Hartlieb, Lorenzo Servadei, Mohammad R. Fadiheh, Sebastian Simon, Alexander Rath, Jeroen Vliegen, Basavaraj Naik, Somanagouda Patil, Tobias Ludwig, Eshan Singh, Valentin Hiltl, Endri Kaja, Sebastian Prebeck, Zhao Han, Ye Ding, Andreas Neumeier. Special Thanks to Lorenzo for all the help and support, for interesting discussions, and Sebastian Prebeck for helping me with the German summary.

Finally, I would like to mention my family who are always a great support: My parents, Prad and Shiv, Trishu, and Shruthi, who is a constant motivation and thanks for proof reading all my papers.

Keerthikumara Devarajegowda
25.March.2021
Germany

Contents

Chapter 1

Introduction

An important milestone in the history of semiconductors is the invention of a point-contact transistor in 1947 by John Bardeen and Walter Brattain of Bell Laboratories. In the following year, William Shockley invented the junction transistor and marked the arrival of the transistor era [93]. The semiconductor industry grew rapidly following the invention of transistors and in 1959, the bipolar Integrated Circuit (IC) was invented by Jack Kilby and Robert Noyce of Texas Instruments [93]. The technologies continued to advance with the advent of large-scale, very large-scale and ultra large-scale ICs. The capacity of integrated circuits grew rapidly in the last decades and it became possible to integrate all essential functions of an end-to-end system on a single chip, commonly referred to as System-on-Chip (SoC). As the semiconductor industry progressed towards high performance, the application field has expanded vastly, and today, the electronic components perform billions of highly complex computations with 100% accuracy requiring only often micro seconds of computation time.

1.1 Growing Complexity of SoCs

Today most electronic components are developed through an SoC design paradigm. That is, a complete end-to-end system is developed by integrating several individual hardware and software blocks. These individual blocks are commonly referred to as design Intellectual Properties (IPs). The functionalities of an intended system or an IP are designed and validated by highly skilled engineers with the aid of sophisticated Electronic Design Automation (EDA) tools.

An integral part of the system development flows is design verification [11, 20, 23, 81, 95]. The main objective of verification is to ensure the functional correctness of the design implementation by detecting all design errors or by proving their absence. Design verification is a well researched field over the last decades and there are well established verification methodologies followed across the industrial design flows [1, 23, 57, 81], which utilize different verification techniques such as simulation, formal verification or emulation. Design verification is a resource intense activity and is known to consume more than 50% of the development time [47].

The exponential growth in the complexity of SoCs contributed to an extremely complex development process. The situation is aggravated by rapidly changing ecosystem as the semiconductor industry moves towards Autonomous Vehicles (AV), Internet-of-Things (IoTs) and smart cities. The design errors that go undetected during verification can have disastrous consequences if they are detected during system operation in the field, especially for safety-critical domains such as automotive or aerospace. Furthermore, due to the reliance of human lives on

the electronic devices, the SoCs work with extremely delicate information. As a result, ensuring the correct functioning and proving the absence of loopholes become the central part of design validation. Thus, building such SoCs requires sophisticated and proven development processes.

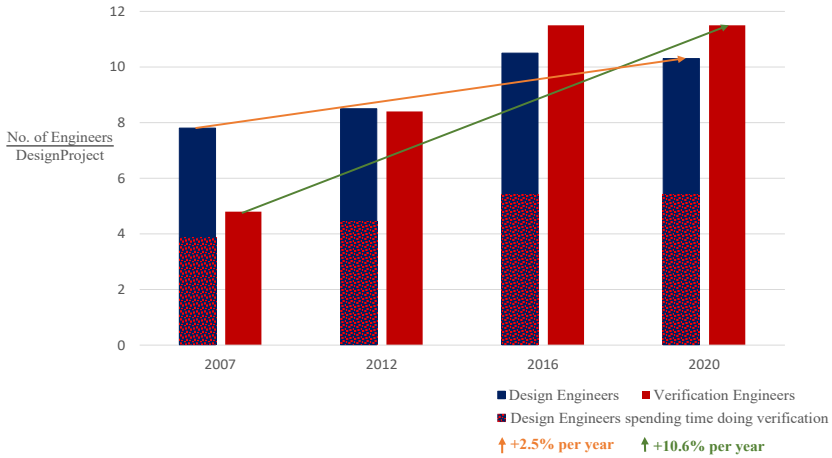


Figure 1.1: Mean peak number of design and verification engineers in design projects [79]

Despite significant advances made in the field of design verification, there is a notable gap between the capabilities of the existing verification methods and the verification needs of modern SoCs. Further, the ecosystem created by the modern SoCs resulted in a demand for developing large number of complex devices in a short turnaround time, satisfying the decreasing size of Time-To-Market (TTM) windows. To address short TTM requirements and growing verification needs, the number of verification engineers working on a design project has constantly increased over the last decade as depicted in Fig. 1.1. The chart shows the mean peak number of design and verification engineers per design project. The mean peak number of engineers doing verification has increased by 10.6% every year, whereas the mean peak number of engineers doing design has increased by 2.5%. Additionally, design engineers spend approximately 50% of their time doing verification. This only shows the rising complexity of designs and the efforts needed for design verification. However, this trend of increasing the number of resources is not feasible as the Non-Recurring Engineering (NRE) costs have to be small for the products to be economically effective.

To cope with the rising complexity of designs, novel verification methods that are highly effective in detecting design errors are required. In other words, there is a great demand for effective and productive verification methods to tackle the needs of next generation hardware designs. This thesis is motivated by the observation that there are opportunities for automating certain steps of verification such as generating the properties required for formal verification. Automating property development has the potential to increase the overall verification productivity. By ensuring the correct-by-construction paradigm, the quality of generated properties can be improved. Further, code generators are built once and can be reused later for many design instances. Additionally, the formal verification technique is exhaustive by nature and provides a high verification quality. However, formal verification is generally seen as a supplementary technique to simulation. The main reasons are difficulty in developing "good properties" from design specifications, required expertise to apply formal methods, and the absence of effective formal verification methodologies. This thesis attempts to propose novel formal verification strategies for effective verification of hardware designs with generated properties.

1.2 Functional Verification in Industrial Practice

In today's development flows, the designs are validated at various development stages [20]. An IP or an SoC undergoes different types of verification such as functional verification, timing analysis, performance evaluation, safety verification, etc. In particular, the designs are first verified with respect to the implemented functions before they are manufactured. This process is commonly referred to as pre-silicon verification. Pre-silicon verification is the major resource intense activity that takes place at the Register Transfer Level (RTL). Pre-silicon verification provides high observability and controllability compared to post-silicon validation, which is performed after a chip is manufactured. This thesis focuses on pre-silicon verification techniques and methods.

In current industrial settings, IPs are developed individually and are assembled together to realize the required functionalities of an SoC. The individual IPs are developed such that they follow a standard communication protocol to interact or communicate with other IPs in the system. In the context of SoCs, the verification follows two separate flows, one for verifying the correct functionality of individual IPs ("module verification") and another for the integrated system ("integration verification"). In other words, an individual IP is verified separately to ensure that an IP on its own behaves as expected. Next, the IPs are integrated to an SoC model and a so-called system-level verification is performed targeting the interaction between interconnected IPs. Due to the complexity of today's IPs, both verification flows are significantly complex consuming on an average 50% to 70% of the overall development cycle.

Although different verification techniques exist in practice, the industrial verification flows largely rely on simulation-based methods. Formal verification is often used as a supplementary technique to simulation, employed for specific use cases such as connectivity verification or reachability analysis of coverage holes remaining after simulation runs. In some cases, formal verification is also used as a bug hunting engine to search for corner case bugs that may have escaped simulation. This thesis proposes novel verification strategies based on formal verification. A detailed outline on the state of the art in formal verification is provided in Chapter 2. An overview on simulation and emulation-based verification is provided in the following.

1.2.1 Simulation-based Methods

In state-of-the-art industrial flows, simulation is the primary choice for SoC verification mainly because, compared to other techniques, simulation-based methods scale better with the design size. To address the growing complexity of designs to be verified, the industrial verification flows have adopted standard verification methodologies [1, 57, 48, 60]. Universal Verification Methodology (UVM) has become the de-facto standard simulation-based methodology currently followed in the industry. UVM has been developed by a consortium that includes major EDA vendors and chip manufacturing companies. UVM succeeds previous methodologies such as Open Verification Methodology (OVM) and e-Reuse Methodology (eRM).

In general, the application of simulation-based verification can be segregated to three stages: verification planning, simulation run or execution and coverage or result analysis. The verification progress and the verification sign-off[1] are analyzed based on the metrics such as functional or code coverage collected during the simulation runs.

[1] Verification sign-off is a milestone in the development cycle, which is reached when the expected results are achieved by the verification and the DUV is qualified for further steps in the system development flow.

Verification Planning

For a given Design Under Verification (DUV), a verification plan is created that serves as a road map for all verification activities to be performed. A verification plan defines "what" functions of the DUV are verified and "which" strategies are used for verifying a specific function. It is important to note that the verification plan itself does not include the tests (or test cases), instead it only specifies the approach that is followed. For a set of functions to be verified, the verification plan specifies a list of tests that are created. The completion of this list defines the completion of the verification.

Although a verification plan may exist in different forms such as spreadsheet or a series of text documents, commercially available simulators provide special support for creating, refining and maintaining verification plans. The verification plan is also used to measure the verification progress. The results of simulations runs, i.e., coverage data are collected and mapped back to the verification plan to visualize the progress of the verification.

Execution

Verification execution forms the substantial part of the manual effort required during the verification step. It includes test generation, monitoring, checking and coverage collection. The execution step involves creating a testbench, which instantiates the DUV and provides input stimuli. The input stimuli are either manually developed or generated by a test generator. The stimuli are applied to the simulation environment and the behavior of the DUV is monitored and checked against the expected values. Here, the DUV is treated as a black box, i.e., the verification is performed by analyzing the response pattern of the DUV for a given input stimuli pattern without insight into the internals of the design implementation. Further, a testbench may describe its own behavior model of the function to be verified and compare the response pattern of the behavior model against the response pattern of DUV during simulation. This behavior model is often called scoreboard.

Test generation is the most important aspect of simulation-based verification methodologies. Generation of high quality tests is the pre-requisite for effective verification, i.e., to expose design errors. Directed and constrained random stimuli are the two types of tests that are typically employed. In IP verification, constrained random test generation has been widely adopted. Here, a verification engineer implements test templates that are used in the test generators to produce tests. The test generation infrastructure is typically reused from one version of an IP to its next version. In SoC verification, the test generation again uses the constrained random stimuli approach. However, since the test generation is bound to the use-cases (targeted application) of an SoC, the stimuli generation may not be reusable when the use-cases change between different SoC versions. As a result, the degree of reuse is smaller. The quality of tests generated is measured by the coverage results collected during the simulation run. As a rule of thumb, a high quality test generates a scenario that reveals a design error or excites difficult to hit coverage events. It is often necessary to provide constraints for the random stimuli generator to guide the test generation, which requires manual intervention and good understanding of the design implementation as well as potential consequences of constraints.

Also, simulators play an important role in the simulation-based verification environments. Many commercial simulators exist from different EDA vendors [19, 78, 108]. The simulators are made up of complex software algorithms, which are executed on a workstation. After reading a testbench that instantiates a DUV, a simulator compiles the source description into

a simulation model that is then evaluated to produce responses to given input stimuli. The simulator technologies have constantly improved over the last decades and are driving the performance in simulation-based verification methods. In recent years, more advances are seen in the capabilities of simulators to handle extremely complex designs and additional feature needs such as mixed-signal verification, power-aware verification and safety verification.

Coverage Analysis

The verification of a DUV by simulation is complete when all the inputs are exhaustively covered by the stimuli. However, exhaustive simulation even for designs of moderate size is not realistic as it requires significant manual efforts, huge computation resources and weeks of simulation runs. To work around this shortcoming, different metrics are defined which guide the verification engineer in deciding whether the DUV is analyzed for a reasonable combination of input stimuli that are expected to exercise all design functionalities at least once. That is, to measure the progress of verification with simulation, coverage metrics are collected during simulation runs. Metrics such as *functional coverage* and *structural/code coverage* are collected for quantifying the design functionalities that have been exercised during simulation runs.

Functional coverage metric shows which functions of the design are exercised by the input stimuli. As a result, measuring the functional coverage is specific to the functions implemented by the design. During simulation runs, when the test case that was written to verify a certain design function is triggered by the stimuli, the functional coverage point — that is created during the verification planning stage — is marked as covered. When a simulation regression run is complete, the reached functional coverage is analyzed and the regression runs are repeated until the expected coverage numbers are met.

Structural or code coverage metric is a quantitative measure of the RTL code that has been exercised during simulation. Code coverage in turn is measured as the sum of line coverage, FSM coverage, expression coverage, block coverage, toggle coverage and branch coverage. Each of these code coverage metrics considers a specific aspect of the RTL code written to realize the required functionalities. Code coverage provides a useful feedback on the exercised RTL code during simulation runs and requires no manual efforts. However, 100% code coverage does not imply a complete verification. Instead it only suggests that 100% of the RTL statements have been executed at least once during simulation.

Both functional and code coverage metrics are analyzed after the completion of simulation regression runs. When the expected coverage numbers are not met, the simulation regression is re-run with different seeds set for the random stimuli generator. It is also practiced that directed stimuli are enforced on the DUV to cover specific parts of RTL code or functionality during simulation. This step requires additional manual efforts and is typically used only as a last resort to meet the expected coverage numbers. It is important to note that coverage metrics followed in simulation-based verification are quantitative in nature and do not guarantee a high quality verification. As a consequence, the quality of verified designs depends on the type of test cases implemented to verify the design functionalities.

1.2.2 Emulation

Verification using emulation or FPGA prototyping is seen as the bridge between pre-silicon verification and post-silicon validation. In emulation, the RTL model of the DUV is mapped

to a programmable architecture such as FPGA or specialized emulators and accelerators [3, 44, 121]. The emulation platform is much faster compared to RTL simulators. However, the observability and controllability are limited during emulation. That is, only a select set of predefined RTL signals can be traced during emulation. This prevents from sufficient coverage analysis. Further, it is necessary to regenerate the FPGA bit-streams when there is a requirement for tracing different set of RTL signals. Although recent advancements within FPGA technology address some of the shortcomings of the technique, the observability remains an issue in emulation platforms [20].

1.2.3 Formal Verification

Formal verification is another pre-silicon verification technique that is gaining industry acceptance in the last decade. Formal verification has traditionally suffered from tool scalability and user friendliness, and was limited to a small range of applications. In recent years, significant advances are seen both in terms of scalability to much larger designs and usability of formal verification solutions [19, 54, 78, 81, 85, 119]. Advances in Satisfiability (SAT) solvers and SAT Modulo Theories (SMT) are driving this paradigm shift [13, 75, 41, 118]. Further, commercial formal tools have improved debugging features and offer predefined verification solutions such as connectivity verification, control and status register verification or security path verification [85, 19, 108, 78]. For these type of use cases, explicit development of properties is not needed as the tools automate the verification tasks from metadata description provided to the tool.

1.3 Motivation and Envisioned Approach

Regardless of the advancements made in the domain of formal verification, the industrial verification flows still largely rely on simulation-based methods [20, 47, 52]. A major hurdle to the widespread adoption of formal verification has been the required expertise to apply the technique effectively on real life designs. First, a set of properties that capture the intent of the design is required to verify the DUV. These properties are mostly developed manually from specifications of the DUV. Developing a set of properties from design specifications that completely verify the DUV is a known complex problem. Next, the set of properties are evaluated in a formal verification tool to determine the correctness of the DUV with respect to its design specifications. During this step, it is important to ensure that the properties converge with an outcome i.e., pass or fail and do not result in false positives. Obtaining a convergence of all the properties is not straightforward and may require state space reduction techniques depending on the specifics of the DUV. Furthermore, ensuring the absence of false positives requires high formal verification expertise. Therefore, in addition to a set of properties, a suitable and effective formal verification strategy is needed to completely analyze the DUV.

A series of functional verification surveys conducted in [47, 79] reveal that at least 50% of the design projects require 2 spins before production. The study also shows that the major cause of respins is the functional flaws in the RTL design. Respins are highly expensive both in terms of time and resources, and may have huge economic impact on the product. A chart is shown in Fig. 1.2 that depicts the root causes of functional flaws in design projects. The *x-axis* shows different root causes, while the *y-axis* shows the percentage of design projects

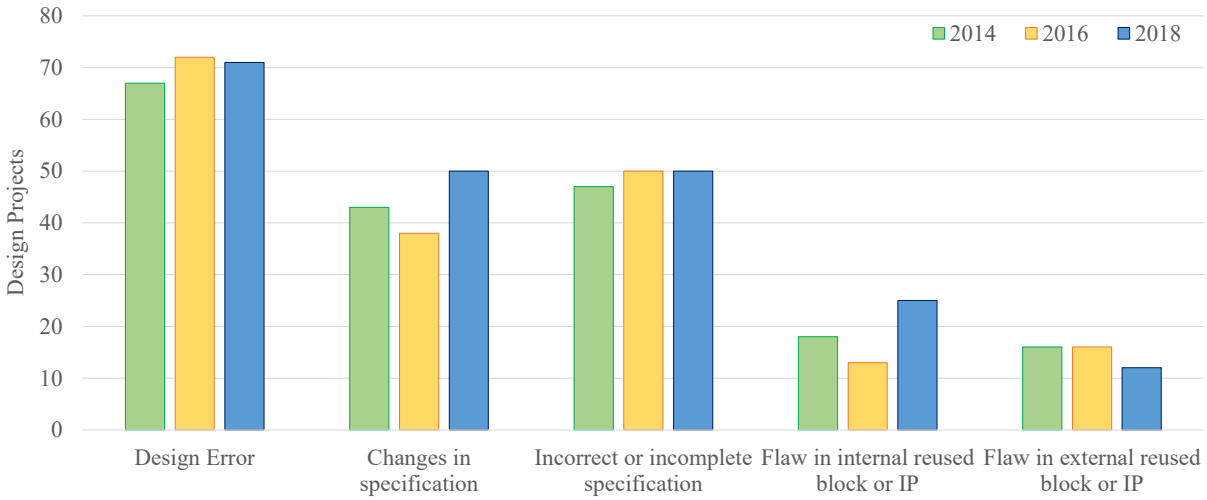


Figure 1.2: Root cause of functional flaws [79]

that are affected by the specific type of functional flaw. In 70% of the design projects, the functional flaws originated from design errors. These type of design errors can be attributed to the incorrect specification interpretation by the design engineers. The chart also shows that, 50% of the projects are affected by flaws due to changing specifications and 50% of the projects are affected by incomplete or incorrect specifications.

The numbers shown in Fig. 1.2 point to a very important drawback of the design flows, which is the usage of informal specifications to describe the functionalities of a design. Informal specifications are ambiguous and incomplete, and are the main source of functional flaws. Developing properties from informal specifications can lead to a set of incomplete or incorrect properties and eventually leads to low verification quality. Further, manual coding of properties requires substantial rework when the specifications of the DUV change constantly causing difficulties in maintaining the workflow.

1.3.1 Requirements for a "Good Property Set"

The manual coding of properties shall be replaced with an automated flow for developing properties. The property generation helps to improve the verification productivity, enables reuse and, more importantly, addresses constant changes in specifications. Since a set of properties are crucial for the application of formal verification methods, a set of requirements for a "good" property set is outlined in the following.

$\mathbb{R}_1$: A property shall be generated from formal specifications

- The specifications of a design shall be translated to formal specification models. Further, the formal specification models shall be complete w.r.t. the informal specifications. Finally, the properties shall be generated from formal specification models. This increases specification–property consistency and reduces the effort for rewriting properties in case of specification changes.

$\mathbb{R}_2$: A generated property shall be precise and complete w.r.t. each specification item

- For each specification item in the formal specification model, a property shall be generated. The generated property shall be precise and capture a specific behavior of the design. In other words, the generated property shall not cover more than one behavior of the design. This enables backtracing from properties to formal specifications and reduces, in most cases, proof complexity for the formal proof engines.

$\mathbb{R}_3$: A set of properties shall capture all specification items

- A set of properties shall be generated from the formal specification models such that they capture the complete behavior of a design. In other words, a set of generated properties shall be complete w.r.t. the design specifications. This guarantees the complete verification of the design.

$\mathbb{R}_4$: A set of properties shall be traceable from the specification items

- In Requirement-Driven Development Flows (RDDF), every feature of the design and its corresponding property developed to verify the feature shall be traceable from design specifications or requirements. Therefore, a set of properties generated from formal specification models shall carry a specification tag such that they can be tracked back to their specification items. This fulfills the quality requirement for example, for safety-critical designs.

$\mathbb{R}_5$: The property generation flow shall obey the 4-Eyes Principle

- A fundamental requirement of hardware design flows is to separate the design development from design verification tasks. This rule is commonly referred to as 4-Eyes Principle (4EP). According to this rule the design and verification tasks shall be carried out by two different engineers (4 eyes). 4EP is necessary to avoid design errors resulting from similar mistakes in both design and verification code.
- In frameworks that automate the generation of both RTL and verification code (properties, testcases, etc.), the generation flows must take separate paths to the target code from the specifications. Ensuring 4EP reduces the probability of masking of design bugs.

$\mathbb{R}_6$: The property generation shall support multiple property languages

- Commercial formal verification tools support different property description languages such as SystemVerilog Assertions (SVA), Property Specification Language (PSL) and In-Terval Language (ITL). To support various needs such as legacy code and special feature support by different formal tools, the properties generation shall be supported for multiple property languages. This allows to use the formal verification tool and methodology best suited for a specific problem.

$\mathbb{R}_7$: A generated set of properties must support different verification techniques

- The properties can be employed in different verification techniques such as simulation, formal verification and emulation. However, the properties for different techniques may

slightly vary. For example, in formal verification, due to its exhaustive nature additional constraints are needed to enforce legal signal behavior. Such explicit constraints are not necessary in simulation-based verification as the properties are evaluated only for certain input stimuli that are triggered in the testbench. Therefore, to enable high reusability the properties shall be generated such that they are applicable in different verification techniques. An example use-case is to use the properties in a simulation environment if the formal tool does not converge.

$\mathbb{R}_8$: A set of properties shall be human-readable

- The properties capture a certain design intent and are evaluated against the design implementation with a chosen verification technique in a chosen EDA tool. An important aspect of design verification is debugging the property failures. For facilitating easy debug, the generated properties shall be human-readable. Further, the readability of properties ensures reuse of properties for different design instances and eases property debug when needed.

1.3.2 Envisioned Approach

As illustrated in Fig. 1.2, the informal specifications and manual coding are the major source of functional flaws. They are also the main drawbacks to the effective application of formal verification on real-life designs. Several requirements for a "good" property set are outlined in the previous section. Considering these requirements for a property set and the need for a formal verification strategy suitable for the DUV, an approach is envisioned in Fig. 1.3 that is addressed in this thesis.

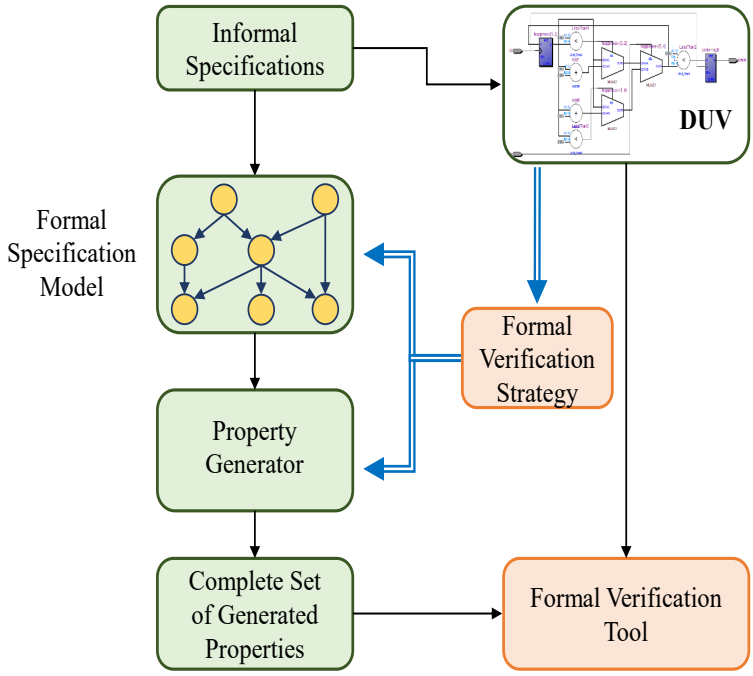


Figure 1.3: Envisioned approach

In the envisioned approach shown in Fig. 1.3, the first step is to capture the informal specifications in formal models. Formal specifications avoid ambiguity and define the intended functions of a design with clear semantics. These formal specification models are used as the single source for developing properties. Due to the diverse nature of designs, different specification modeling methods are supported.

Manual coding of properties is replaced with an automatic property generator such that the changes in the specification are automatically incorporated in the generated properties. Additionally, the generation framework improves the overall verification productivity, ensures the specification to property consistency and enables high degree of reusability. The specification modeling facilitates simpler definition of code generators such that only a small effort is required to define the properties for a given DUV. For this purpose, the specification modeling supports different types of design implementation and offers a suitable specification modeling technique (for example, structural or state transition notations).

Finally, the generated set of properties is complete with respect to the formal specifications and is used in a formal verification tool to exhaustively verify the DUV.

1.3.3 Thesis Overview

The remainder of this thesis comprises seven chapters which are organized as follows. In Chapter 2, an overview on the state-of-the-art in formal verification is provided. Relevant formalisms for bounded and unbounded model checking are also introduced. As this thesis proposes a property generation flow, relevant background details on a metamodel-based automation framework is outlined in Chapter 3. The contributions of this thesis are presented from Chapter 4 and onwards.

The property generation framework adopts the Model-Driven Architecture (MDA) principles for developing code generators and considers all requirements of a "good" property set. These aspects of the property generation are elaborated in Chapter 4.

A suitable formal verification strategy or methodology is required to effectively apply formal verification on various designs. In Chapter 5, formal verification of pipelined processors is considered. A formal verification method called C-S^{2}QED for complete verification of processor cores is proposed. A completeness proof for the proposed approach based on Complete Interval Property Checking (C-IPC) is discussed in detail.

In Chapter 6, the applicability and effectiveness of the property generation is demonstrated by generating a compete set of properties for various industry-strength designs. We describe how the generation flow has been applied to verify RISC-V ([117]) processor core variants with the proposed processor verification method C-S^{2}QED. Further, different aspects of applying the property generation to other industry-strength designs such as AHB-to-APB bridge, programmable interrupt controller and bus matrices are elaborated. The results are tabulated and observations from the experimental results are discussed.

A summary of the work concludes the main contributions of the thesis in Chapter 7. During the course of this work, a novel approach for formally verifying error detection and correction codes has been proposed. Further, an approach to extract the control signals of a processor decoder using formal verification tools has been proposed. For both approaches, the property generation flow developed during this doctoral work is applied to improve the overall productivity and quality. These approaches describe other applications of property generation beyond C-S^{2}QED. The published papers are listed in Appendix E.

1.4 Publication List

Large parts of this thesis have already been pre-published. The publications are listed chronologically as follows:

1. K. DEVARAJEGOWDA, J. SCHREINER, R. FINDENIG AND W. ECKER, Python based framework for HDSLs with an underlying formal semantics. In *proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Irvine, CA, 2017, pp. 1019-1025.
2. K. DEVARAJEGOWDA AND W. ECKER, On Generation of Properties from Specification. In *proceedings of the 2017 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, Santa Cruz, CA, 2017, pp. 95-98.
3. K. DEVARAJEGOWDA AND W. ECKER, Meta-model Based Automation of Properties for Pre-Silicon Verification. In *proceedings of the IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Verona, Italy, 2018, pp. 231-236.
4. W. ECKER, K. DEVARAJEGOWDA, M. WERNER, Z. HAN AND L. SERVADEI Embedded Systems' Automation following OMG's Model Driven Architecture Vision. In *proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Florence, Italy, 2019, pp. 1301-1306. (*doi: 10.23919/DATE.2019.8715154*).
5. E. SINGH, K. DEVARAJEGOWDA, S. SIMON, R. SCHNEIDER, K. GANESAN, M. FADIHEH, D. STOFFEL, W. KUNZ, C. BARRETT, W. ECKER AND S. MITRA, Symbolic QED Pre-silicon Verification for Automotive Microcontroller Cores: Industrial Case Study. In *proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Florence, Italy, 2019, pp. 1000-1005.
6. K. DEVARAJEGOWDA, W. ECKER AND W. KUNZ, How to Keep 4-Eyes Principle in a Design and Property Generation Flow. In *proceedings of the MBMV 2019; 22nd Workshop - Methods and Description Languages for Modelling and Verification of Circuits and Systems*, Kaiserslautern, Germany, 2019, pp. 1-6.
7. K. DEVARAJEGOWDA, L. SERVADEI, Z. HAN, M. WERNER AND W. ECKER, Formal Verification Methodology in an Industrial Setup. In *proceedings of the 22nd Euromicro Conference on Digital System Design (DSD)*, Kallithea, Greece, 2019, pp. 610-614.
8. K. DEVARAJEGOWDA, M. R. FADIHEH, E. SINGH, C. BARRETT, S. MITRA, W. ECKER, D. STOFFEL, AND W. KUNZ, Gap-free processor verification with S^2QED and property generation. In *proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Grenoble, France, 2020.
9. K. DEVARAJEGOWDA, E. KAJA, S. PREBECK, W. ECKER, ISA Modeling with Trace Notation for Context Free Property Generation. In *proceedings of the Design Automation Conference (DAC)*, San Francisco, USA, 2021.

Chapter 2

Formal Verification Techniques

A brief outline to the formal verification in industrial practice is provided in Section 1.2.3. This thesis shows how the model-driven software development can be leveraged for automatic generation of properties for effective application of formal methods. In addition, this thesis shows how the principles of S^2QED together with the property generation can be used to reduce the manual efforts required to "completely" verify a processor core. A detailed overview of the formal verification techniques is outlined in the following sections.

2.1 Introduction

Formal verification can be defined as a process of checking the correctness of a design implementation against the design specification using mathematical theories. A specific behavior of the design is captured in a temporal formula and the temporal formula is checked exhaustively on the mathematical model of the design for all legal input values. In industrial practice, commercial formal verification tools support languages which have an extended syntax to simplify the definition of temporal formulas. The temporal formulas are manually derived from specifications and are commonly referred to as *properties*. When the mathematical model does not hold for the temporal formula, an appropriate error trace (also called a counterexample) is provided. When the model holds for the formula, it proves that the design behaves as described by the specifications. Formal verification techniques can be broadly classified into *Theorem Proving*, *Equivalence Checking* and *Model Checking*. Fig. 2.1 shows the categorization of formal techniques.

2.2 Theorem Proving

Theorem Proving is a sub-field of formal verification that deals with the mechanization of formal reasoning following the laws of logic. In theorem proving, the system is represented as a set of mathematical definitions using the laws of mathematical logic. The desired or expected properties of the system are derived as theorems that conform to the mathematical definitions. Theorem provers use first-order and higher-order logic provers to validate the system behavior [114]. Some of the well known theorem provers based on higher-order logic are $HOL-4$ [80], PVS [86] and $ISABELLE/HOL$ [88]. Examples for theorem provers based on first-order logic are *ACL2* [62] and $ISABELLE/FOL$ [88]. First-order logic is considered as the

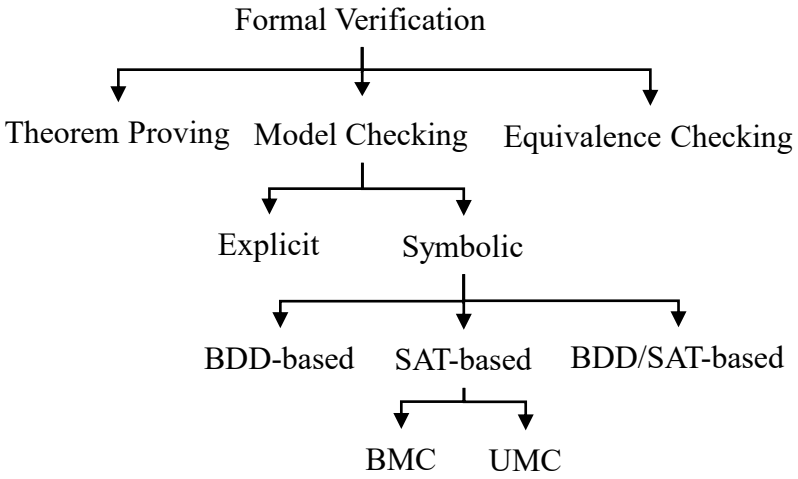


Figure 2.1: Formal verification techniques

most expressible logic which is semi-decidable. Any logic that is more expressive than first-order logic is not decidable. For practical usage of first-order logic for hardware verification, it is necessary to model the time (for sequential circuits) using natural numbers. However, first-order logic lacks mechanisms to provide complete formalisms for natural numbers [65]. Higher-order logics have been demonstrated for their applicability for hardware verification [63, 89]. Since the validity of higher-order logics is undecidable, the proof systems are interactive in nature and are typically employed as proof assistants. As a result, although theorem provers can be used for verifying reactive digital systems, due to their lack of complete automated methods, they are used in combination with other formal techniques.

2.3 Equivalence Checking

Equivalence Checking or *Formal Equivalence Checking* (FEC) can be defined as determining the equivalence of two model representations using mathematical reasoning. The model representations can be derived from the same design with two different platforms (e.g., VHDL or Verilog), at different abstraction levels (e.g., RTL or gate level) or from two different designs expected to produce the same set of outputs at all time points. Combinatorial equivalence checking and sequential equivalence checking are two FEC types that are commonly used in the industry [99].

The combinatorial equivalence checking is used to compare two versions of the same design (or circuit) at different abstraction levels, for example, to determine whether the synthesized net-list is equivalent to its RTL description [67, 17, 85, 19, 78]. In Combinational equivalence checking, the equivalent state variables of the two different circuits are identified by state matching[1] and checked for equivalence. The state matching procedure involves collecting all state variables into one equivalence class, proving non-equivalence of some state variables and splitting equivalence classes accordingly. For proving the non-equivalence of state variables, different approaches based on satisfiability (SAT), automatic test pattern generation (ATPG), binary decision diagram (BDD) and structural and logic simulation are used. The BDD-based

[1] State matching is a process of identifying equivalent state variables, also known as latch mapping.

approaches do not scale well for large designs. The SAT-based approaches for equivalence checking are considered hard due to huge re-convergent fan-out structures [68].

The designs that are expected to be equivalent display structural similarities. This key aspect of designs is exploited in early approaches that included identifying equivalent points, partitioning the circuits and treating these points as primary inputs. However, partitioning the circuits and treating internal nodes as primary inputs may result in false failures. A new approach to detect and exploit internal equivalence by using the ATPG methods is proposed without introducing the cut-points in [67]. This approach, called *HANNIBAL*, nullified the problem of false failures and established an effective and efficient approach for formal (combinational) equivalence checking. This work inspired further improvements [69, 107] on the approach and led to a wide acceptance of combinational equivalence checking on the industrial designs. Today combinational equivalence checking is widely used in the industrial practice. It is helpful in finding unexpected behaviors after the synthesis process, for example, deviation due to critical path optimizations.

The sequential equivalence checking is used to determine if two models generate the same set of outputs at all time points for an equivalent set of inputs. The sequential equivalence does not require the internal nodes of the designs to be equivalent. This type of equivalence checks are explored for determining if a set of properties determine the correct values for the outputs of a design for all time points [81].

2.4 Model Checking

Model checking, also known as *property checking* is an algorithmic way of proving that a sequential system's behavior conforms to its specification. Model checking is the primary technique used by the FV tools to analyze the behavior of a design implementation. A model checker verifies the validity of a system's implementation with respect to its specifications by checking the validity of temporal formulas on the mathematical model of the implementation.

Definition 1 [Model Checking]:
Let us consider a sequential system $\mathcal{M}$ with a finite set of states $\mathcal{S}$. Let ϕ be a temporal formula capturing an expected behavior of the system $\mathcal{M}$. Model checking is defined as an algorithmic way of proving or disproving if $\mathcal{M}$ models ϕ, i.e., $\mathcal{M} \models \phi$. □

For verifying the validity of design implementation against the design specification, a model checker requires the following ingredients [68, 65]:

1. a mathematical model of the implementation with appropriate expressiveness,
2. a suitable specification language to define an expected design behavior, and
3. an effective proof method (algorithm).

The type of model checking where all reachable states of a design are explicitly represented is called *explicit model checking* [65, 55]. Explicit model checking is practically not feasible for designs of moderate to high complexity due to state space explosion. In *symbolic model checking*, reachable states of a system are implicitly represented using Boolean functions [76, 65]. Symbolic model checking applies binary decision diagrams (BDD)-based and satisfiability (SAT)-based proof methods to determine the validity of a design's behavior w.r.t. the specification.

2.4.1 Modeling Hardware Behavior

Sequential circuits (or designs) are composed of logic gates and storage elements such as flip-flops, registers or RAMs. The presence of storage elements introduces a temporal relationship between the input and output signals. That is, at an arbitrary time point t, the values of output signals not only depend on the input signals at t, but also on the previous input signal values (time points $< t$). As a result, Boolean functions are insufficient to model the behavior of sequential circuits. For modeling the sequential behavior of hardware designs different formalisms for automata are used.

Definition 2 [Finite automaton]:
A *finite automaton* $\mathcal{M}$ is a 6-tuple $\mathcal{M} := (\mathcal{S}, \mathcal{S}_i, I, O, \delta, \lambda)$, where

- $\mathcal{S}$ is a finite set of states,
- $\mathcal{S}_i \subseteq \mathcal{S}$ is a set of non-empty initial states,
- I is a finite set of input symbols,
- O is a finite set of output symbols,
- δ is a state transition function with $\delta : \mathcal{S} \times I \mapsto \mathcal{S}$,
- λ is an output function with $\lambda : \mathcal{S} \times I \mapsto O$. □

A finite automaton is also known as a *finite state machine* (FSM), a *Mealy automaton* or a *transductor*. Temporal structures are another formalism used to model the behavior of hardware designs. Temporal structures are also referred to as Kripke structures and are typically used to describe the temporal logic expressions [64].

Definition 3 [Kripke structure]:
A Kripke structure is a quintuple $\mathcal{K} := (\mathcal{S}, \mathcal{S}_i, \mathcal{R}, \mathcal{A}, \mathcal{L})$, where

- $\mathcal{S}$ is a finite set of states.
- $\mathcal{S}_i \subseteq \mathcal{S}$ is a set of non-empty initial states,
- $\mathcal{R}$ is a transition relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ such that $\mathcal{R}$ is left-total, i.e., $\forall s \in \mathcal{S}\ \exists\ s'$ such that $(s, s') \in \mathcal{R}$.
- $\mathcal{A}$ is a set of Boolean atomic formulas that are formulated in terms of the input and output signals of the design.
- $\mathcal{L}$ is a labeling or valuation function $\mathcal{L} : s \mapsto 2^{\mathcal{A}}$. An atomic formula $a \in \mathcal{A}$ has value *true* in a state $s \in \mathcal{S}$ if $s \in \mathcal{L}(a)$. □

A Kripke structure can be seen as a state transition graph in which reachable states represent the nodes and state transitions represent the edges of the graph. A Kripke structure does not include input and output alphabets of a system, instead it includes a labeling function $\mathcal{L}$, which defines for each state $s \in \mathcal{S}$ a set of atomic formulas $\mathcal{A}_s \in \mathcal{A}$ that are valid in the state s ($\mathcal{L} : s \mapsto \mathcal{A}_s \in \mathcal{A}$, where $\mathcal{A}_s = \{a \mid s \mapsto a = true\}$).

2.4.2 Temporal Logics and Proof Methods

To describe the behavior of sequential circuits over a finite time interval, it is essential to model the "time interval". In the context of hardware digital designs, discrete time can be modeled in two different ways: linear time and branching time. In linear time model there exists exactly one successor time point for each time point. In contrast, branching time model allows more than one successor time points for an arbitrary time point. Although linear model is well suited to model physical time, the branching model is appropriate to capture computations in which

different execution traces are selected at a certain time point. Linear time can be considered as a special case of branching time [65].

The discrete time model is formalized by temporal structures. Temporal structures can be based on Kripke structures defined in Section 2.4.1. A temporal structure is a graph with nodes $\mathcal{S}$ (set of states with $\mathcal{S}_i \subseteq \mathcal{S}$ as the set of initial states), where the edges are defined by $\mathcal{R}$ (transition relation). The labeling function $\mathcal{L}$ labels the nodes with atomic formulas which are valid for the corresponding nodes. Beginning from a starting state $s_0 \in \mathcal{S}_i$, a temporal structure is traversed according to the successor states given by the transition function $\mathcal{R}$. The outcome of this computation is an infinite branching tree with the starting state s_0 as the root, called computation tree. Computation Tree Logic (CTL) and Linear Time Logic (LTL) are two commonly used temporal languages for capturing specific behavior of sequential circuits over a finite time interval.

Computation Tree Logic

The temporal language Computation Tree Logic (CTL) was first proposed with an accompanying proof method in [42]. In computation tree logic, the propositional operators are extended with modal operators *"always"* (A) and *"exists"* (E). These modal operators are used in combination with the temporal operators: X (next), G (globally), F (finally), U (until) and W (weak until) to express the behavior of sequential circuits over time.

Definition 4 [CTL Syntax]:
Let us consider a set of atomic formulas $\mathcal{A}$. A CTL formula ϕ is a state formula, syntactically defined as follows:

- If $\phi \in \mathcal{A}$, then ϕ is a state formula,
- If ϕ_1 and ϕ_2 are state formulas, then $\neg\phi_1$, $\neg\phi_2$, $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, $EX\phi_1$, $EX\phi_2$, $EG\phi_1$, $EG\phi_2$, $EF\phi_1$, $EF\phi_2$, $AX\phi_1$, $AX\phi_2$, $AG\phi_1$, $AG\phi_2$, $AF\phi_1$, $AF\phi_2$, $E(\phi_1 U \phi_2)$, $E(\phi_1 W \phi_2)$, $A(\phi_1 W \phi_2)$ are also state formulas. □

Definition 5 [CTL Semantics]:
The semantics of CTL are defined as follows: Let $\mathcal{K} := (\mathcal{S}, \mathcal{S}_i, \mathcal{R}, \mathcal{A}, \mathcal{L})$ be a Kripke structure, $s \in \mathcal{S}$ be a state, $a \in \mathcal{A}$ be an atomic formula, ϕ_1 and ϕ_2 be state formulas over $\mathcal{A}$, then $\mathcal{K}$,$s \models \phi$ denote that ϕ holds in state s of structure $\mathcal{K}$.

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

Linear Temporal Logic

The temporal language linear temporal logic was proposed with an accompanying proof method [96]. In contrast to CTL (branching time logics), the semantics of LTL is given by considering non-branching paths [43]. LTL defines a set of temporal operators to express the behavior of sequential circuits over time : X (next), G (globally), F (finally), U (until), W (weak until) and R (release).

Definition 6 [LTL Syntax]:
Let us consider a set of atomic formulas $\mathcal{A}$. The syntax of an LTL formula is defined as follows:

- Every atomic formula $\phi \in \mathcal{A}$ is an LTL formula.
- If ϕ_1 and ϕ_2 are LTL formulas, then $\neg\phi_1$, $\neg\phi_2$, $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, $X\phi_1$, $X\phi_2$, $G\phi_1$, $G\phi_2$, $F\phi_1$, $F\phi_2$, $(\phi_1 U \phi_2)$, $(\phi_1 W \phi_2)$ and $(\phi_1 R \phi_2)$ are also LTL formulas. □

In contrast to CTL, the semantics of LTL is defined with regard to paths, i.e., the underlying model of time in LTL is linear. An LTL formula ϕ is valid, if for all linear structures $\mathcal{K}$, it holds that $\mathcal{K}, s \models \phi$. Similarly, an LTL formula ϕ is satisfiable, if there exists a structure $\mathcal{K}$ such that $\mathcal{K}, s \models \phi$ holds. That is, every structure $\mathcal{K}$ with $\mathcal{K}, s \models \phi$ is a model of ϕ.

Definition 7 [LTL Semantics]:
The semantics of LTL are defined as follows: let $\mathcal{K}$ be a given Kripke structure, ϕ_1 and ϕ_2 be LTL formulas, $\pi_i := (s_i, s_{i+1}, \ldots)$ be an infinite path, $a \in \mathcal{A}$ be an atomic formula, and let $\pi \models \phi$ denote that the LTL formula ϕ is satisfied by path π.

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

The standard property specification languages (e.g., PSL, SVA, ITL) supported by commercial FV tools can be mapped to the formal temporal languages like CTL or LTL [85, 78, 19]. The languages supported for industrial applications have extended syntax to simplify the definition of design specifications.

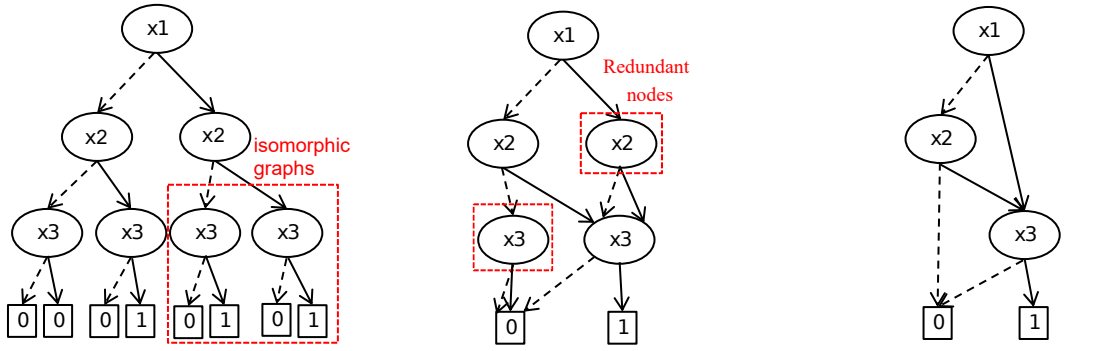
Proof Methods

Different proof methods have been proposed to exhaustively verify whether the temporal formulas are satisfiable on the sequential design's model. CTL model checking, LTL model checking, symbolic model checking, bounded model checking and interval property checking are some of the well established proof methods [43, 42, 96, 81]. As the number of state bits in the design's model grow, the proof complexity grows exponentially. As a result LTL- and CTL-based proof methods suffer from the state space explosion problem [65]. However, bounded model checking and interval property checking methods have shown their feasibility even for very large designs. Although both methods extensively apply SAT-based techniques, binary decision diagrams still find their usage in reachability analysis. To facilitate better understanding, we first provide a

basic outline on the working principles of binary decision diagrams and satisfiability theories. Afterwards, bounded model checking and interval property checking methods are described.

2.4.3 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are directed, cyclic graphs with one root node and one or two terminal nodes. A BDD is an ordered BDD (OBDD) if variables respect the given linear ordering on all paths ($x_1 < x_2 < x_3$, where x_1, x_2, x_3 are variables of a design) as shown in Fig. 2.2a. An ordered BDD may contain isomorphic graphs (Fig. 2.2a) and/or redundant nodes (Fig. 2.2b). A reduced OBDD is obtained by removing isomorphic sub-graphs and redundant nodes as shown in Fig. 2.2c. ROBDDs are canonical (unique), compact and easy to manipulate.



(a) Isomorphic sub-graphs (b) Redundant nodes (c) Reduced - OBDD

Figure 2.2: Reduction of a decision tree to ROBDD

A model checker begins the search by creating a state space BDD for the initial state such that all variables can attain symbolic values unless restricted by the constraints. All reachable states from the initial state are computed using system transitions. The process of finding the next reachable state continues until a fixed point is reached where no additional states are added to the state BDD. At this stage, a specific behavior of the system captured in a temporal formula is checked. When a violation is found, the model checker works backwards from the violating point to the initial state and creates a counterexample. A model checker may also run into a scenario where it cannot produce any concrete result due to computational explosion. Such results occur when the BDDs generated occupy all available memory.

Although the BDDs provide a means for fully checking whether the system's implementation conforms to the specified behavior, there are limitations associated with large designs. As the number of variables increase in a Boolean function, their OBDD representation grows polynomially which results in requiring more memory to store the BDDs. A better variable ordering can reduce the required memory to store BDDs. However, for larger designs the reduction due to better variable ordering is insignificant and still requires more computation resources.

2.4.4 Boolean Satisfiability

An alternative approach to BDDs are Satisfiability (SAT)-based approaches. The SAT problem is to find a set of values for a given set of variables in a Boolean function such that the Boolean function evaluates to 'true'.

Let us consider an example to understand how a SAT solver works. Consider a premise description where a, b and c are variables of a digital circuit:

$$implementation = \neg((\neg a \wedge c) \vee (a \wedge \neg b)) \tag{2.1}$$

$$requirement = ((\neg a \wedge \neg c) \vee (b \wedge c)) \tag{2.2}$$

The objective of the implementation is to meet the requirement as specified by the clause. The implication operator ($\Longrightarrow$) is used to express the required behavior as follows:

$$\neg((\neg a \wedge c) \vee (a \wedge \neg b)) \implies ((\neg a \wedge \neg c) \vee (b \wedge c)) \tag{2.3}$$

The implication definition $p \implies q$ can be written in a disjunctive normal form (DNF) as $\neg p \bigvee q$. Hence, the implication expression in Eqn. 2.3 can be re-written as follows:

$$((\neg a \wedge c) \vee (a \wedge \neg b)) \vee ((\neg a \wedge \neg c) \vee (b \wedge c)) \tag{2.4}$$

In order to prove that the *implementation* meets the *requirement*, it is sufficient to prove that the negation of the expression in Eqn. 2.4 does not evaluate to 'true' for any assignment of values to the variables. The DNF expression $(\neg p \vee q)$ can be reformulated in a conjunctive normal form (CNF) $(\neg(\neg p \vee q) \rightarrow p \wedge \neg q)$. The Eqn. 2.4 is re-written in CNF as follows:

$$(a \vee \neg c) \wedge (\neg a \vee b) \wedge (a \vee c) \wedge (\neg b \vee \neg c) \tag{2.5}$$

The SAT problem is to find a set of assignments for the variables a, b and c, such that the CNF expression in Eqn. 2.5 evaluates to 'true' or to prove that no such assignments exists. If there is no assignment exists such that the expression in Eqn. 2.5 evaluates to 'true', it is proven that the implementation meets the specified requirement. In general, an arbitrary expression is satisfiable if there exists an assignment to variables such that the expression evaluates to 'true', otherwise the expression is unsatisfiable. As the number of variables grow, the time required to solve the problem grows exponentially. However, several efficient SAT solver algorithms and subsequent improvements have been proposed that are applicable to large real-life designs and require reasonable computation resources [26, 58, 8, 83].

2.4.5 Bounded Model Checking

As the design size grows in terms of number of state bits, the corresponding design model $\mathcal{M}$ also grows, leading to the "state space explosion" problem in model checking. State space explosion is a scenario in which the proof complexity becomes high such that a conclusive outcome cannot be obtained with practical computation resources (memory or CPU time). To address this problem, Bounded model checking (BMC) leverages the power of model checking with satisfiability solving [23]. BMC explores the state space much faster and for some particular problems offers large performance improvement over other approaches.

A standard sequential design model is shown in Figure 2.3 with I as a set of input signals and O as a set of output signals. The transition function δ computes the next state ($\delta : \mathcal{S} \times I \mapsto \mathcal{S}$). The output function λ computes the output signal values as a function of input and current state ($\lambda : \mathcal{S} \times I \mapsto O$). In other words, the combinational part of the design computes the output signal values, while the state elements i.e., the registers, store the current state of the design.

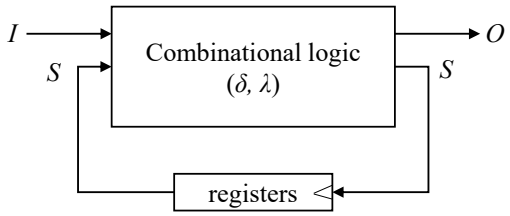


Figure 2.3: Sequential design model

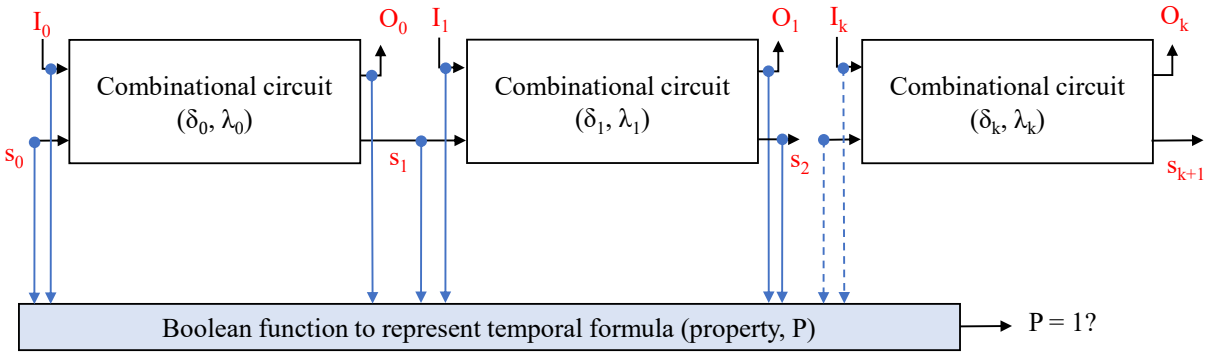


Figure 2.4: BMC iterative design model for a bound depth of k

In case of BMC, a temporal formula is built to describe an expected behavior over a finite time interval. The sequential design model is unrolled for the length of the time interval (number of clock cycles). The temporal formula is constructed as a Boolean expression in terms of the design variable instances in the different time frames of the unrolling. Figure 2.4 shows a bounded design model that has been unrolled for a length of k from the starting state s_0. The iterative design model is constructed by concatenating k copies of combinational logic for δ (transition function) and λ (output function). The next state calculated by the combinational logic representing one cycle is then connected to the current state of the logic representing the successor cycle. In this way, the sequential design is represented as a combinational logic in which the state of the design at a certain time point t is represented as a Boolean vector s_t.

Properties are written to specify enabling conditions, i.e., an *antecedent*, and to capture correspondingly reached states, i.e., a *consequent*. On this bounded model the property can be mapped to a Boolean satisfiability problem where the inputs are taken as free variables if they are not constrained by the antecedent of the property. The proof of the property succeeds if the negated property cannot be satisfied. When a property does not hold on the design model $\mathcal{M}$, a trace is created by backtracking and a counterexample is provided. Otherwise, BMC only proves the property for k clock cycles. A property is proven for all reachable states, if a value for length k is chosen that is larger than the sequential depth. The sequential depth can be defined as the minimum number of iterations or clock cycles required to reach all reachable states. Selecting values for k that are larger than sequential depth is not always practical as it may lead to a point where the model checker can no longer determine if the requirements are satisfied for given value of k.

The results obtained from bounded model checking are useful to determine if a certain behavior of the design is valid for a given depth k. Counterexamples to bounded proofs indicate bugs. However, to make a clear statement on the absence of unspecified behavior by the designs, unbounded proofs are required. Unbounded proofs are proofs that are globally valid and are not

restricted to a bounded time interval.

2.4.6 Interval Property Checking

Interval Property Checking (IPC) is a SAT-based model checking technique that provides unbounded proofs for a class of properties called *interval properties* [81, 112, 111]. In case of interval properties, the temporal formula or the property is captured in an implication format such that both assumptions and commitments of a property are described over a finite time interval. Although IPC is based on a bounded circuit model like BMC, the proof results are globally valid on the reachable state space because they are based on a symbolic initial state, as discussed below.

Definition 8 [Interval Property]:
An interval property ϕ is a LTL formula of the form $G(A \implies C)$, where A (assumptions) and C (commitments) are sequence predicates that describe the behavior of a design over a finite time interval. □

Definition 9 [Sequence Predicate]:
A sequence predicate is a LTL formula in which the only temporal operator that is allowed is the *next* operator X. □

In practice, interval properties are also referred to as *operation properties*, since the intention of an interval property is to capture a specific *operation* of the design in a finite sequence of behavior over a finite time interval. A set of interval properties for a design can be built to capture every operation performed by the design. The complete set of interval properties is described in more detail in Section 2.5.

The computation model of the interval property checking is similar to the BMC computation model (cf. Sec. 2.4.5) except that there is no assumption on the starting or initial state. This allows the SAT solver to consider any state as the starting state during satisfiability solving. As a result, if the negation of a certain property is unsatisfiable (i.e., the property holds on the design implementation), the property is valid for any starting state. In other words, the property holds unbounded on the design and globally valid for all reachable states starting from any state.

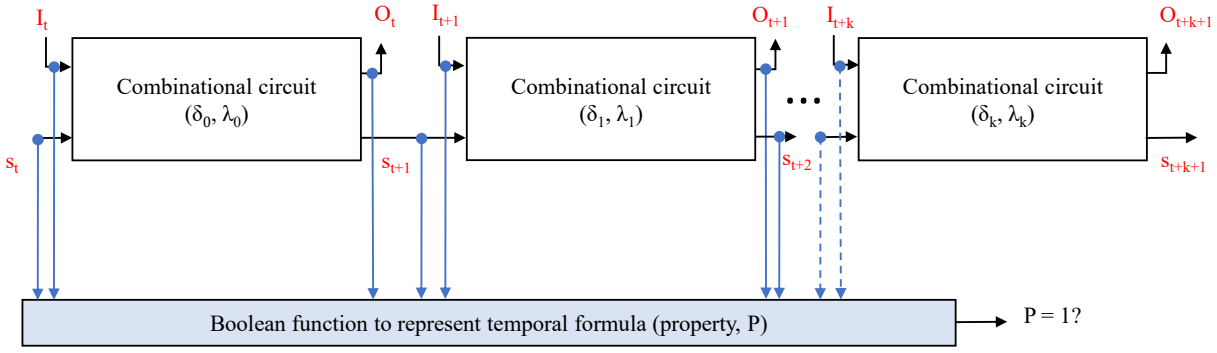


Figure 2.5: IPC iterative design model for a bound depth of k

Figure 2.5 shows the unrolling of a sequential design for a depth of k in interval property checking. The starting state in Figure 2.5, s_t, and the input variables I_t, I_{t+1} ... I_{t+k} are free variables unless constrained by the assumption of the property. The SAT solver checks if there

exists an assignment to variables such that the negation of the property ($\neg P$) exists. If $\neg P$ is unsatisfiable, then the property P holds on the iterated design model, globally.

Although the interval property checking provides globally valid proofs, since the proof starts from a symbolic initial state, it is necessary to validate the outcome of the proof. This is because, since the starting or the initial state is not constrained, the initial state could be an unreachable state. As a result, "false negatives" (or spurious counterexamples) may be encountered in which the counterexamples show behaviors that are not valid within the scope of a design's specification. Spurious counterexamples resulting from choosing unreachable states as the starting state can be excluded by manual analysis of the counterexamples. The Verification Engineer can add the reachability information in the form of assumptions or constraints. However, such an approach may also lead to valid states being excluded. An induction-based approach is proposed in [109], which uses automated methods to exclude the "false negatives" resulting from unreachable starting states. The proposed method uses mathematical induction, where a bounded proof for k clock cycles after reset is used as a *induction base* case to generate invariants. The *induction base* case in an induction-based property checking is a proof of $A \implies C$ over the first k periods after reset. The *induction step* is proven by adding an additional assumption that $A \implies C$ (i.e., induction base) holds on the design. With this approach the manual efforts required to exclude spurious counterexamples is reduced in some cases and in most cases, completely nullified.

Note that in IPC, "false positives" cannot occur, since the proof results are valid for any reachable starting state[2]. Therefore, the proof model of IPC can be termed as "pessimistic" or "conservative".

2.5 Complete Property Set

In the existing industrial verification flows, coverage is one of the key metrics used for the verification sign-off. As outlined in Chapter 1, in simulation-based verification setups, two types of coverage metrics are collected: functional coverage and structural (or code) coverage. However, "coverage" in formal verification setups takes a different perspective. This is due to the fundamental difference in the verification model of the techniques.

In formal verification, every property is evaluated *exhaustively* against the design implementation as described in the previous sections. Here, the term "exhaustive" expresses that the design is analyzed for all possible "input combinations" excluded only by cases specified by the requirements. For every property, the formal tool computes the "cone of influence" (COI) in the design implementation. The COI is the result of the dependency analysis of a property, which includes logic functions, primary inputs, and internal variables of the design [120, 99]. A coarse coverage collection mechanism includes all RTL code lines that are part of the COI of a property as "covered". For a given design implementation, a cumulative aggregation of the RTL code lines covered by a set of properties (that HOLD for the design implementation) are considered as "covered". Additionally, the formal tools implement different abstraction and mutation techniques to determine only the part of the COI of a property that is needed to prove or disprove the property [85, 19, 78].

Although the coverage collected as described above can give useful feedback on the part of

[2]False positives are positive (pass/hold) outcomes of the property proof even when the implemented design's behavior violates the specification.

the RTL code that is analyzed by a set of properties, it does not make a clear statement on the presence or the absence of design bugs. In other words, this type of approach is insufficient to ensure that there are no gaps in the property set. As a result, the responsibility of ensuring the "completeness" of a property set relies on the expertise of the Verification Engineer.

A formal approach for developing a complete property set and measuring the completeness with a set of completeness checks has been proposed [81, 14, 16, 111]. This methodology termed as *Complete Interval Property Checking* (C-IPC) provides a formal "completeness criterion" for a set of operation properties. The property generation flow presented in this thesis adapts the principles of C-IPC for generating a complete set of properties from design specifications. Further, the processor verification method proposed in this thesis benchmarks C-IPC to argue the completeness of the presented approach. We provide an overview of C-IPC in the following.

2.5.1 Complete Interval Property Checking

In Section 2.4.6, we outlined the principles of interval property checking. Interval property checking uses a specific type of properties called interval or operation properties and establishes an unbounded proof. A sequential design's behavior can be captured in terms of operations. An operation is defined as a set of finite sequences of state transitions between two "important"states such that only "unimportant"states are visited in between. Each operation performed by the design fulfills a specific requirement. Therefore, a set of operations in which each operation spans a finite time interval can be conceptualized to completely capture a design's behavior. For each operation, an interval or operation property is developed such that the collective sum of properties "completely" capture the output behavior of the design as a function of input signals.

A formal definition of an *operation* is given as follows [111]:

Definition 10 [Operation]:
Let P be an interval property with $P := G(A \implies C)$, where A is the assumption and C is the commitment of the property P. An operation O is a sequence predicate of finite length l in a Kripke structure $\mathcal{K}$ characterized by the pair (P,l) such that property P holds on the Kripke Structure $\mathcal{K}$, i.e., $P \models \mathcal{K}$. □

In the pair (P,l), l represents the time interval between the starting time point ($t_{P_{start}}$) of assumption A and the ending time point ($t_{P_{end}}$) of the commitment C, i.e., $l = t_{P_{end}} - t_{P_{start}}$. In other words, l is the length of the operation in terms of clock cycles.

For a given sequential design, a set of properties $P = (P_1, P_2, \dots)$ are created such that each property is mapped to its corresponding operation in a set of operations $OP = (OP_1, OP_2, \dots)$. Each operation starts and ends in certain important states in the design. These states are referred to as *conceptual states*. An important state corresponds to one or more concrete states and each concrete state can only belong to one important state. Operational properties are supposed to be chained in the sense that the end state of one operation is the start state of the succeeding operation. These operations (transitions between conceptual states) can be viewed to form a conceptual state machine (CSM). The CSM is a finite automaton describing the sequencing of operations and is close to the specification. These states may not be concrete states in the design implementation, instead can be captured as Boolean predicates from the variables (or signals) of the design. Let $t^1_{P_{start}}$ be the starting time point of the assumption and $t^1_{P_{end}}$ be the

ending time point of the commitment of the property P_1. Let $t^2_{P_{start}}$ be the starting time point of the assumption and $t^2_{P_{end}}$ be the ending time point of the commitment of the property P_2. At time point $t^1_{P_{start}}$, the design is assumed to be in a conceptual state from which the operation OP_1 starts. At time point $t^1_{P_{end}}$, the design is in a conceptual state in which the operation OP_1 is expected to end. The ending state of the operation OP_1 and the starting state of the operation OP_2 are the same, i.e., $t^1_{P_{end}} = t^2_{P_{start}}$. Here, P_2 is the successor property of P_1 as the operation OP_2 succeeds operation OP_1. A set of properties are developed as described such that they collectively capture all the operations performed by the design.

2.5.2 Completeness Criterion

For a property set to be complete, the properties must capture the values of output signals and other important states of the design at all time points. That is, the output sequences of a design need to be determined by a set of properties according to the determination requirements. The determination requirements are specifications of signals that describe which signals must be determined and at what time points. A signal is determined, if its value is specified by a set of properties at all time points. These signals should be determined as a function of input signals and other determined signals (internal and output signals) of the design [81, 22].

Definition 11 [Complete Property Set]:
A property set $V = \{P_1, P_2, \ldots, P_n\}$ is complete, if two arbitrary state machines satisfying all properties in the set are sequentially equivalent with respect to the determination requirements. □

The Verification Engineer makes a property set complete by ensuring that every possible operation is covered by an operational property and that all outputs and other signals referred to in the determination requirements are uniquely specified at every time point by the operational properties. Completeness of a set of properties can be checked automatically, and independent of a design. For a complete set of *operational IPC properties*, a global sequential equivalence check as in [22] which may be computationally complex is not needed. Instead, completeness can be established inductively by considering all pairs (P_i, P_j) of properties describing an operation OP_i and a direct successor operation OP_j and performing a set of *completeness checks* [14, 81].

2.5.3 Completeness Checks

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

A) **Case Split Test**: The case split test checks that all paths between the important states are covered by at least one property. In other words, it checks that at the ending important state of each operation, for every possible input combination, there exists at least one operational property which determines the next important state. This ensures that there is no input scenario missed in the property set.

B) **Successor Test**: The successor test checks for every operation, whether the successor operation is uniquely determined. For every pair of predecessor/successor operations in the property graph, the execution of successor operation must be uniquely determined by

the predecessor. Passing successor and case split tests ensures that there exists a unique chain of operations for every input trace.

C) **Determination Test**: The determination test checks whether a set of operation properties uniquely determine the outputs of a circuit (or other signals in determination requirements, e.g., general purpose registers in processors) at all time points.

D) **Reset Test**: The above three tests form an inductive proof that if an operation determines its ending important state and outputs, then there exists a successor operation that will uniquely determine the next important state and outputs. The validity of this inductive reasoning depends on the reset state (i.e., the induction base). The reset test checks whether the reset traverses the system deterministically to a unique important state and it fulfills its determination requirements.

Chapter 3

Automated Code Generation Techniques

One of the major productivity improvement methods widely applied in industrial practice is automatic code generation. Code generation bridges the semantic gap between an abstract description and target code, and helps to streamline the development flows. When the abstract description is a formalized specification, code generation also provides a means of traceability from the specification items to the implemented code. The automated property generation flow developed as part of this thesis adapts a software development methodology for code generation called **M**odel **D**riven **A**rchitecture (MDA) [110, 102, 84]. The adaption of MDA for property generation is implemented within an existing metamodel-based automation framework [39]. To facilitate the understanding of the property generation flow in Chapter 4, an overview on fundamental concepts of the metamodel-based automation and MDA principles are outlined in the subsequent sections.

3.1 Metamodeling

The term "*meta*" is derived from the Greek language, which means "after" or "beyond". A *model* is used to represent a system at a certain abstraction level. Similarly, a metamodel is used to define the structure of a model and the relation between the constituents of a model. Metamodels go beyond models and represent models of models.

Definition 12 [Model]:
A *model* is a mathematical or logical representation of a circuit, a system or an entity, according to the rules of a specific modeling language. A *model* $m \in \mathbb{M}$ is a tuple,

$$m = (n, \mathbb{O}, \mathbb{M}_{\mathbb{M}}) \tag{3.1}$$

where, $\mathbb{M}$ is a set of all models, n is the name of the model m and belongs to a set of all names $\mathbb{N}$, $\mathbb{O}$ is the set of objects in the model m, and $\mathbb{M}_{\mathbb{M}}$ is the definition of the model m. That is, $\mathbb{M}_{\mathbb{M}}$ is the metamodel which defines the structure, rules and constituents of the model m. □

The term meta also emphasizes a hierarchy between models and metamodels. That is, models are instances of metamodels as the circuit (or entity/system) are instances of models. Fig. 3.1 graphically depicts the hierarchical relation between a circuit, a model and a metamodel. In otherwords, metamodels define the set of rules model artifacts adhere to and allow to group and classify them with similar characteristics. *Metamodeling* is a term used to describe a methodology where abstract models and model instances are used to achieve a specific goal,

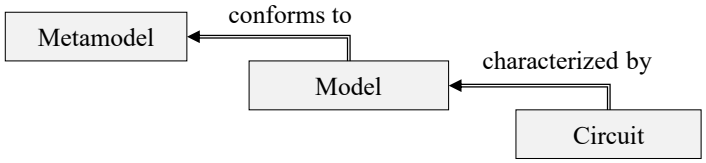


Figure 3.1: Hierarchy between a metamodel, a model and a circuit

for example, code automation. Metamodeling assists the automation as it provides a formalization of the models and a structure for code generation. A high degree of productivity can be achieved through automated transformations of a model into a target view such as code or documentation.

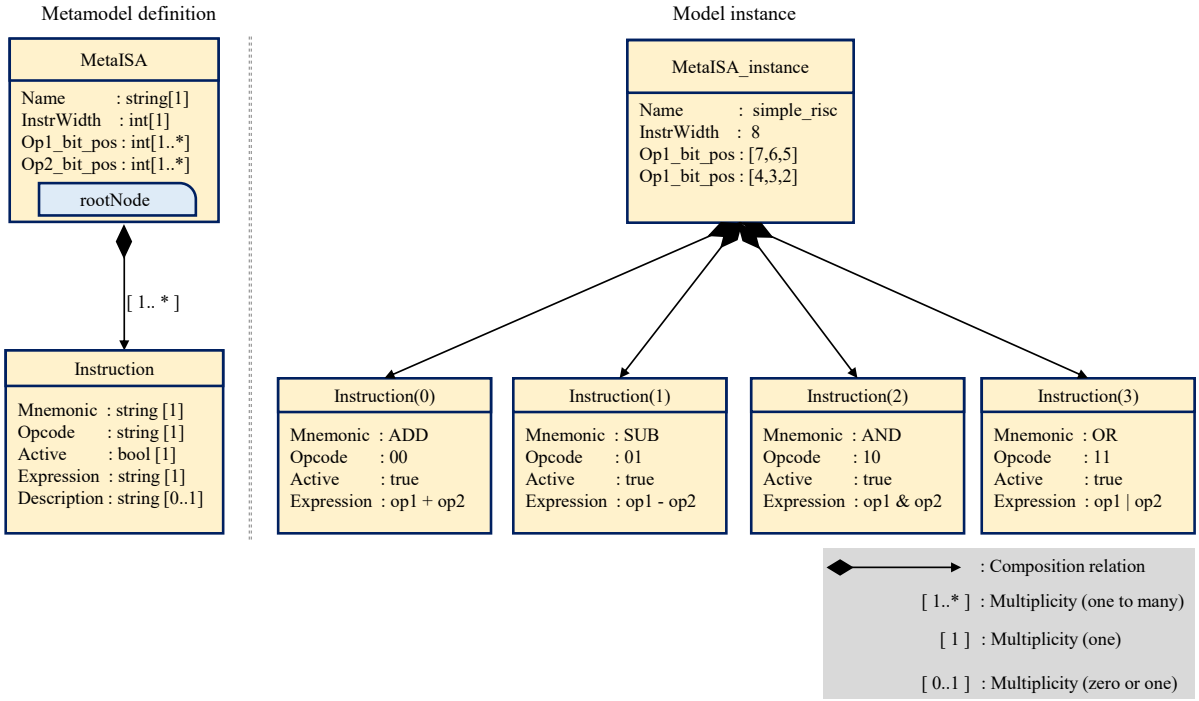


Figure 3.2: UML class diagram showing an example of metamodel and a model instance

In Figure 3.2, a metamodel definition (left) and a model instance (right) are shown. The metamodel definition is created to model a simple instruction set architecture (ISA) and is shown as a Unified Markup Language (UML) class diagram, created with a graphical software tool. The root node of the metamodel is called *MetaISA* and has 4 attributes. These attributes are *Name*, *InstrWidth*, *Op1_bit_pos* and *Op2_bit_pos*. *InstrWidth* denotes the width of the instruction word, whereas *Op1_bit_pos* and *Op2_bit_pos* denote the bit positions of the operands 1 and 2, respectively. Each attribute has a pre-defined type (e.g., string or int) and an associated *multiplicity* value, which determines the acceptable number of attribute instances. The class *MetaISA* has a composition relation to the class *Instruction* with a multiplicity of $1..\star$ (one to many).

The *Instruction* class has its own set of attributes, which define the high level properties of an instruction. From the metamodel definition one or more model instances can be created that adhere to the rules and relations set by the metamodel. An example of a model instance is shown which includes specific values to the attributes of the respective class definitions. The

example instance shown models a *simple_risc* ISA and has an instruction width of 8 bits. Bit positions 7-to-5 and 4-to-3 carry the operand values, and bit positions 1-to-0 denote the opcode. The corresponding attribute values for the individual *Instruction* classes are as shown.

The metamodel describes in general how to model an ISA, a model instance describes one instance of such an ISA. This instance meets the constraints imposed by the metamodel: It has exactly one root node and several instructions. Furthermore, the individual artifacts have valid values assigned to their attributes, which is also a requirement imposed by the metamodel.

3.2 Metamodel-based Code Automation

The metamodel-based methodology offers a huge potential for high productivity automation. The definition of a metamodel precisely describes how the model looks and determines how it can be accessed and processed with a metamodeling framework. The metamodeling framework can provide the ability to read and write models based on their metamodel definition. More importantly, the frameworks can be used to provide an interface between the data and models based on the metamodel definition.

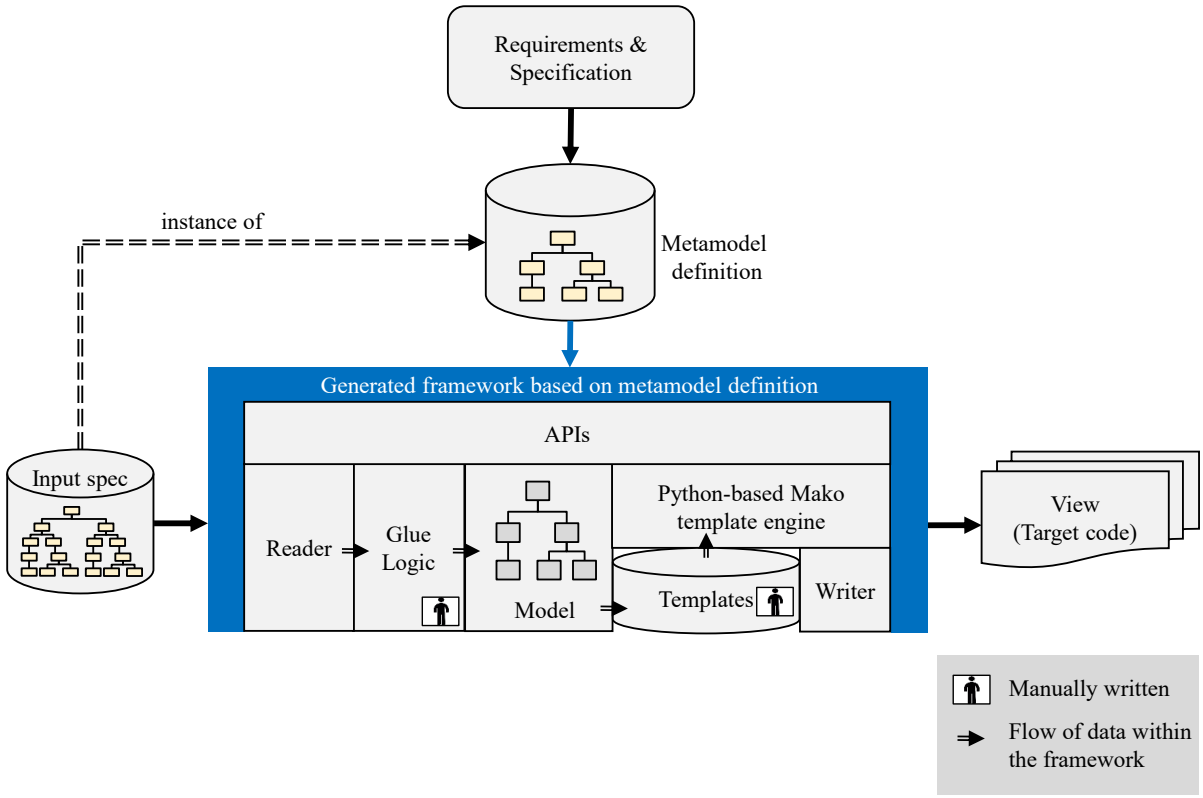


Figure 3.3: Metamodel-based automation framework

At Infineon, a metamodel-based code generation framework is heavily used to reduce the Non-Recurring Engineering (NRE) costs [38, 40, 39]. Figure 3.3 illustrates the typical generation flow within the framework and describes the role of metamodel in the flow. To utilize metamodeling for a certain task, a metamodel (shown as *Metamodel definition* in Figure 3.3) needs to be defined from a given set of requirements and specification. This is done with a

UML modeling tool, the output of which is the metamodel description. The metamodel definition plays a central role in the flow as a Python-based infrastructure is generated targeting the objects, their attributes and relations between the objects set by the metamodel definition. The infrastructure provides a graphical user interface (GUI) to allow the User to create model instances by filling in with the data. Further, input/output plugins and input/output extensions are generated to allow reading-in and writing-out the default and modified metamodel instances.

At run-time, the models are read into the metamodeling framework with a *Reader*. The reader is auto-generated for certain input formats such as SQL databases, XML or other structured markups. Once the models are read into the framework, *Glue Logic* (written in Python) may be used to modify the model. The framework provides in-built readers for known input formats such as XML, XLS, CSV. At this point the model is accessible through the Python Application Program Interfaces (APIs).

After creating a model object (*Model* in Figure 3.3) inside the framework, the control is passed to the template engine. Mako templates, the Python template library, are used to map the data captured in the models to produce the required target code. There are in-built templates in the framework, also called as "writers", that support dumping the model data in a pre-defined format such as XML, CSV or XLS. The main purpose of writers - whether automatically generated or manually written- is to automatically generate target code or documentation from models.

Figure 3.3 shows the most important and most commonly used output mechanism for this purpose: a template engine. Template engines are tools that generate target code from a template file and input data [6]. Templates are output documents that contain placeholders which embed certain content into the output document. Moreover, the model content can also be examined, altered and processed, generating new view-specific content for insertion into the templates. It is important to note here that it is possible and feasible to use the same Metamodel and reader, and develop several different sets of templates to generate different views (e.g., generated C-code as one view and HTML documentation as the other).

The metamodeling framework has been successfully employed for achieving higher productivity on several design tasks [38, 40, 39]. For overall chip design, productivity has been increased by up to 60% and by up to 95% for individual design tasks.

3.2.1 Challenges in Developing Code Generators

Although the described metamodeling framework has resulted in a high degree of automation for specific design tasks, there are shortcomings of this approach. For automating certain design tasks, a huge semantic gap must be bridged between the specification and the target code. In the framework described, the APIs, the reader and the writer are provided for a given metamodel.

When the created metamodel is close to the specification, the effort needed to develop the reader is minimum and complex manual annotations to the specification are not needed. This in turn shifts the work to the template development. As the templates need to cover any model instance of the metamodel, the development of the templates becomes increasingly difficult. This requires a large amount of complex code to examine the model and make decisions based on it. In particular the development of complex templates is not developer-friendly due to limited programming features and results in higher manual effort.

When the created metamodel is close to the view/target code, the development effort for template files reduces considerably. However the automation process is decoupled from the

specification, which requires more effort to develop the reader and glue logic. In order to realize a high degree of automation and to develop reusable code generators, the created framework needs to be close to the specification.

Due to the above challenges in developing the code generators, a new solution is desired with the following features:

- a framework shall be generated which is close to both the specification and the target code
- more than one metamodel is required to allow the generation of APIs, readers and writers customized to different goals
- the generated framework shall be reusable

3.3 Model Driven Architecture for Code Automation

The Object Management Group (OMG) proposed the idea of MDA as an approach for using models in software development to address the productivity gap that also burdens the software development. The MDA principle was proposed to help reducing complexity, lower costs and fostering re-usability. A "model" has well-defined semantics and is an abstracted version of the intended system [77, 110, 71, 70]. MDA emphasizes the use of modeling techniques to increase the level of abstraction and productivity during the implementation of software and hardware systems. There are three main models involved in MDA, namely ***C**omputation **I**ndependent **M**odel* (CIM), ***P**latform **I**ndependent **M**odel* (PIM) and ***P**latform **S**pecific **M**odel* (PSM). The idea of MDA captured as Y-chart [71] is illustrated in the Figure 3.4. These models are involved in model-to-model transformations in which the resulting model is a less abstracted model of the intended system.

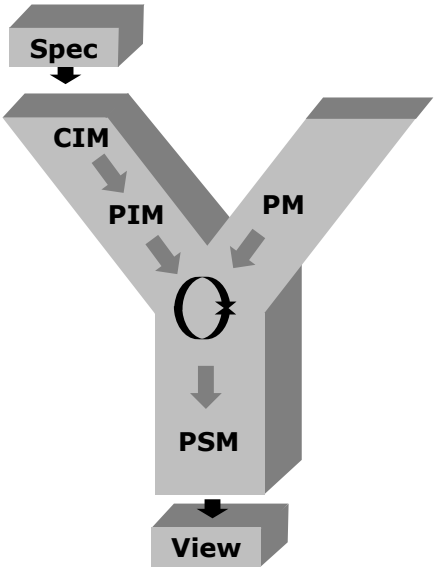


Figure 3.4: MDA pictured as Y-Chart from [71]

CIM The Computation Independent Model is the first model layer which is close to the business concept. The requirements and specifications of a specific business concept are captured in this model. As the name suggests, CIM focuses on the high-level details of the system without considering the structure and processing of the intended system. The

CIM is always closer to the specification and less dependent on actual implementation details than the PIM.

PIM A Platform Independent Model is the result of transformation of the CIM with additional details. PIM describes functionality and behavior of an implementation. The behavior is already described using the semantics of the target implementation at a high level of abstraction. This description is however independent of the targeted platform (e.g., a programming language, development and runtime environment).

PSM A Platform-specific Model adds the utilized platform to the PIM. PSM is closest to the target code and from this model, the view is generated. It therefore includes how the computation is performed on the given platform.

Figure 3.4 shows the dependency between the models in a Y-Chart. In addition to the aforementioned models, this figure also shows a Platform Model (PM), which contains the details of the target platform. The term "platform" is used to refer to the computing infrastructure on which generated target code can run. A platform for example defines libraries, APIs and the OS the generated view code compiles and executes on (e.g., FPGA). A sequence of model-to-model transformations are chained together to finally generate the view.

The MDA principle for code generation has been adapted for both RTL and property generation. The detailed description of the property generation is provided in the Chapter 4. The RTL generation following the MDA approach is not implemented as part of this thesis. A brief overview of the RTL generation is described in the next section.

3.4 MDA Applied to RTL Generation

The model-driven approach for hardware generation is explored by extending the metamodeling framework with MDA concepts [97, 36]. Figure 3.5 shows the MDA adaption for digital hardware generation. Similar to the MDA definition, the RTL generation flow uses multiple model layers as shown. The model layers are named as Model-of-Things (MoTs), Model-of-Design (MoD) and Model-of-View (MoV).

Typically the requirements of a hardware design are written in an informal language. The top most model layer, MoT layer, includes translating the informal specifications in formal specification models. This layer corresponds to the CIM layer in the OMG's MDA description. For a given set of specification items, one or more metamodels are constructed to capture the correct constituents or things of the intended design. That is, MoT layer defines things, their attributes and the relations to the intended functionality. For example, Figure 3.2 shows the metamodel of a SW/HW interface circuit. The instances of these metamodels are Model-of-Things, which carry the specific values and relationships for the attributes of the metamodel.

After collecting the requirements of an intended design in the formalized models, the MoTs are transformed into a design model called Model-of-Design. This layer corresponds to PIM layer of the OMG's MDA definition. The transformations are implemented in Python, in which the microarchitecture of the intended design is defined. These transformations are referred to as templates of design (ToD) since they describe the blue print of the circuitry. ToDs allow the designer to completely focus on effectively defining the architecture of the intended design since the MoD is independent of the simulation and synthesis artifacts. A metamodel called "MetaRTL" is created such that the metamodel is an abstract definition of any arbitrary digital design. In other words, MoD is an instance of the metamodel MetaRTL. Based on this

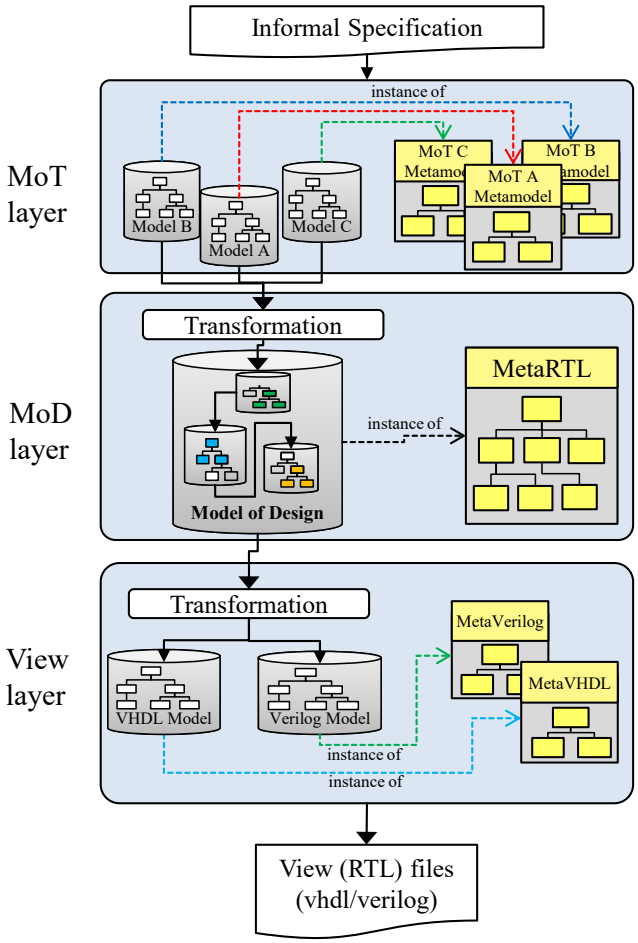


Figure 3.5: MDA applied to hardware generation

metamodel, the automation framework provides an extended set of APIs to simplify the ToD coding [50].

The view model or Model-of-View (MoV) corresponds to the PSM since it is the least abstract model with a straightforward mapping to the target view. Here, the MoD is transformed into a platform model where the platform-specific details are appended to the model. The availability of multiple target view models (for example, vhdl view model or verilog view model) allows the design to be generated in a required HDL.

Chapter 4

Model-driven Property Generation

This chapter describes leveraging the concepts of Model-Driven Architecture (MDA) introduced for software engineering [110, 71] for property generation in the hardware domain. For this purpose, the essential concepts of MDA are transferred to the domain of hardware design verification. We first discuss the challenges of building code generators for properties, and later, provide an overview of the property generation framework, which addresses the outlined challenges. A mechanism for annotating or binding the design implementation details in the generated properties such that the design and property generation flows obey the 4-eyes principle is also presented. Different approaches for modeling design specifications to facilitate a high degree of automation are elaborated and a relation of the work to existing approaches is discussed.

4.1 Challenges for Property Generation

Although the properties are an effective means of representing the design intent, property development poses several challenges. Interpreting the informal specification, manually developing properties for large designs, maintaining them and changing properties due to regular specification changes are error-prone, time tedious and not scalable. Automatic generation of properties addresses these challenges and effectively improves the overall productivity. However, the automatic generation of properties also faces several challenges. We outline the challenges in this section and detail how the implemented generation flow tackles these challenges in later sections.

The challenges for property generation are classified based on commonly agreed concepts in state-of-the-art digital verification. We classify them into the following two categories:

1. Ensuring the quality of generated properties.
2. Binding the design details and obeying 4-eyes principle in generation flows.

4.1.1 Ensuring the Quality of Generated Properties

When a property is evaluated in a formal verification (FV) tool to verify a specific behavior of the design, the outcome is either a pass or a fail. A failure is either due to a bug in the design implementation (true negative) or due to an incomplete property formulation (false negative). The FV tool provides a counterexample (CEX) to debug the failing property. For better debug and to exclude false negatives, the properties need to be concise, compact and correct.

Challenge 1 [Informal specifications]:
A common practice in the industry is to compose the requirements of an intended design in an informal/natural language (English, German, etc.). Informal specifications are often vague, ambiguous, badly structured, incomplete or incorrect. Developing properties from the informal specifications is tedious and error prone. For example, consider the following text extracted from an official RISC-V ISA release [117, page 14-15].

> **Example** "RV32I defines several arithmetic R-type operations. All operations read the $rs1$ and $rs2$ registers as source operands and write the result into register rd. The $funct7$ and $funct3$ fields select the type of operation. *ADD* and *SUB* instructions perform addition and subtraction respectively. Overflows are ignored and the low XLEN bits of results are written to the destination".

The example text describes the ADD (addition) and SUB (subtraction) instructions of *Integer Register-Register Operations*. The resultant of an addition remains unchanged (commutative law) irrespective of the order of operands (*rs1 + rs2 == rs2 + rs1*). However in case of subtraction, the resultant depends on the order of operands (*rs1 - rs2 != rs2 - rs1*). From the exemplified text, there is no clear semantics to define the order of operands, which can lead to mis-interpretations and possibly to logic bugs.

Further, the inductive nature of informal specifications bears the risk of missing important scenarios. In addition, the lack of formal semantics prevents from code automation. Therefore, a pre-requisite for the automatic generation of high quality properties is to transform the informal specifications to formal specifications.

Challenge 2 [Effort for creating formal specifications]:
Formal specifications have the following features:

1. Clearly and completely describe the expected behavior.
2. Avoid ambiguity, thus avoiding mis-interpretation.
3. Support code automation due to the availability of clear semantics.
4. Support for traceability i.e., the generated code can be traced back to the specification items.

Formal specifications need to provide an abstract view of the system. They shall not include any details that constrain the implementation choices. Although the formal specifications help to avoid ambiguity and enable automation, the effort required for building formal specifications can be significant. Also, the degree of automation depends on the expressiveness of the formal specification. Even for simple designs, building formal specification takes considerable effort and time [65]. Therefore, a supporting infrastructure is needed such that the creation of formal specifications requires significantly less manual efforts.

Challenge 3 [Completion criterion for a set of properties]:
A FV tool exhaustively verifies the design for each property considering all input combinations. However, the properties typically use an implication operator such that the property is evaluated only under certain trigger conditions. In addition, some properties can be proven only under specific input scenarios to avoid unrealistic behavior of the design implementation. A major problem in complex designs is to identify all corner case scenarios, which require prior anticipation before the respective properties are developed to validate such scenarios.

The factors outlined so far lead us to the following questions: How do we develop properties to verify every design function? Are there "*gaps*" in the property set? Is every output signal of the design determined for all input combinations and at all time points?

To answer the above questions, a comprehensive mechanism is needed such that the operations or functions of a design are captured in a control flow graph like format or a conceptual state machine [81]. In addition, a mapping should be created between the specified design operation and the generated properties to verify all documented features of a design implementation. This ensures that every design operation is verified by at-least one property.

4.1.2 Binding Design Details and Obeying the 4-Eyes Principle

Different verification approaches have been applied for validating the design implementations such as black-box, white-box or grey-box [120]. The selection of a suitable approach relies on many factors such as design type, design complexity or verification objective.

Challenge 4 [Binding design details]:
A significant benefit of the properties is that they can be used across all verification techniques such as simulation, formal verification and emulation.

In the black-box approach, the verification environment is unaware of the internal details of the design and typically deployed to verify the top-level behaviors. Properties for such an approach need to include only the top-level port details. However, this approach is not ideal for formal verification as the technique requires internal design details to avoid false negatives and also to speed-up the property proof. In the white-box approach, the verification environment has complete access to internal details of the design. This approach may dilute the verification quality since the verification environment is close to design implementation. Also, further changes in the design requires significant amount of rework to modify the properties.

The grey-box approach balances the requirements of internal details and provides a controlled access to the internal signals of the design. A challenge for property generation is: how to get the necessary internal signals details into the generation framework? This is because, the properties are to be generated from the formal specifications only and at this level the design internal signals are not available. As a result, a mechanism is required to encode the generated properties with the necessary design details.

Challenge 5 [Obeying 4-eyes principle (4EP)]:
A key objective of the functional verification is to ensure that the RTL implementation meets the specified behavior. To achieve this, the verification environment (testcases or properties) needs to ensure that the design implementation specifics are not copied directly in the testcases or properties. Otherwise, it may lead to false positives and there is a high risk of bug escapes. Therefore, the state-of-the-art functional verification flows need to follow strict guidelines guaranteeing the verification quality.

A fundamental requirement of the hardware design flows is to obey the *4-eyes principle*. 4EP, also referred to as *2-person rule*, is a well-known control mechanism followed widely in the decision making process for critical applications or operations. In hardware design flows, 4EP requires that the design and verification processes follow separate paths. Similarly, frameworks that automate both the design and verification tasks must follow separate flows in order to avoid common bugs due to the inherent flaws in the automation framework.

4.2 Model-driven Architecture for Property Generation

To verify the functional correctness of an RTL implementation, properties are developed from specifications of the intended design. These properties are written in a certain property specification language such as SVA or ITL, and are verified in a FV tool against the RTL implementation. The property languages have specific syntax and semantics associated with them. In order to generate the properties from the specifications, a large semantic gap must be bridged between the specifications and the properties. Building property generators directly from the specifications using template-based approach leads to complex development of code generators as outlined in the Section 3.2.

The Object Management Group®(OMG) proposed a software development methodology called Model-Driven Architecture®(MDA) for developing code generators [71, 102, 84]. In the proposed MDA approach, the software development follows a model-driven approach in which the data is captured in different model layers. These model layers represent different abstraction levels as outlined in Section 3.3. Adapting MDA for property generation within the metamodel-based framework is described in the following.

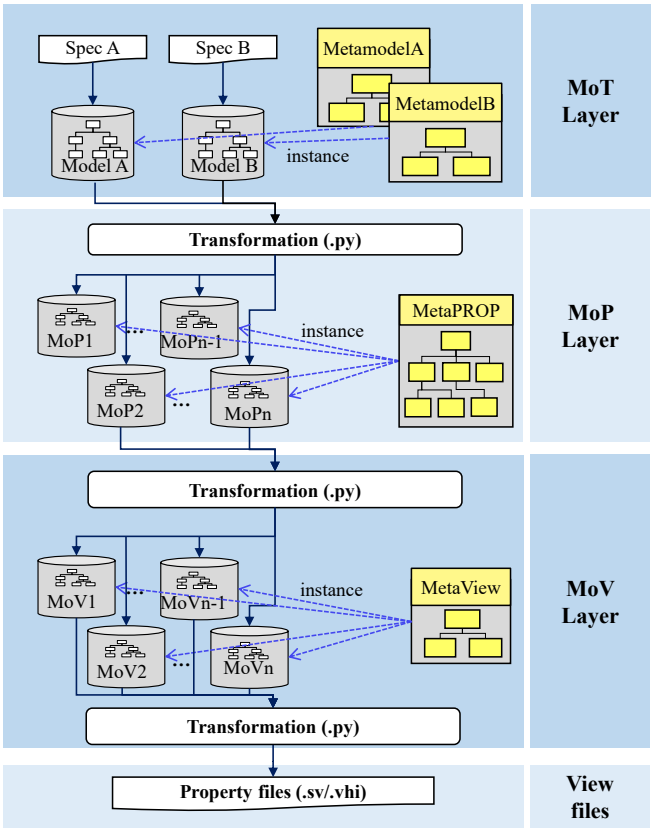


Figure 4.1: Model driven generation of properties

The adaption of model driven approach for automatic generation of properties is illustrated in Fig. 4.1. The flow conceptually follows the OMG'S MDA proposal but introduces new terms for various hardware-related models. The generation flow consists of three viewpoints of abstraction layers defined by MDA from top (high abstraction level) to bottom (low abstraction level). Each model-to-model transformation step adds additional details to a higher abstraction

model to build a lower abstraction model. The three model layers of the approach are:

1. MoT Layer: Corresponds to CIM in OMG's MDA description (cf. Section 3.3)
2. MoP Layer: Corresponds to PIM in OMG's MDA description (cf. Section 3.3)
3. MoV Layer: Corresponds to PSM in OMG's MDA description (cf. Section 3.3)

4.2.1 Model-of-Things Layer

The intention of the Model-of-Things (MoT) layer is to formally collect the data from informal specifications. These models are called "Model-of-Things" to indicate the purpose of specification models, which is to represent the abstract behavior and characteristics of specific "structures" or "components" or "things". As per the original MDA definition, MoTs are computation independent models, i.e., high-level details such as system configurations and intended behaviors are considered but not necessarily the implementation details. Here, the implementation details refer to the microarchitecture specifics of the design. Irrelevant details are ignored and the behavior is generalized to central features and characteristics. Therefore, the implementation choices are not constrained by the formalized specifications.

As shown in Fig. 4.1, there may be several specifications (*SpecA, SpecB*), e.g., for each interface and the design item itself. Metamodel definitions (*Metamodel A, Metamodel B*) describe objects (=things), object attributes and the relationships between the objects of respective specification items. These metamodels are used to capture the design domain. An example metamodel created to model a simple ISA is shown in Fig. 3.2. For every metamodel definition, an infrastructure is generated by the automation framework as described in Section 3.2.

Instances of metamodels (*Model A, Model B*) are then created, for example, by using a GUI tool provided by the underlying framework. This step is manual and the engineer needs to ensure that the correct values for all object attributes are populated in the model instances. Wherever structured information is found in the specification, it is parsed to create parts of the model and to reduce the manual effort. That is, when the specification items are available in a structured format, for example spreadsheets or CSV files, readers from the framework are used to extract the data and populate the models. The MoTs have a well defined structure within the automation framework, enabling the automatic model transformations to extract required data.

The translation of informal specifications to formal specification models addresses Challenge 1. Also, the effective utilization of the underlying automation framework reduces the overall manual efforts needed to create the formalized models, there by addressing Challenge 2. Different modeling paradigms introduced for formalizing the design specifications are discussed in Section 4.6.

4.2.2 Model-of-Property Layer

The MoTs are transformed into less abstract models called Model-of-Properties (MoPs). The model-to-model transformation involving MoTs and MoPs forms the central part of the generation flow. These transformations are coded in a Python-based Domain Specific Language (DSL) and are referred to as Templates-of-Properties (ToPs). They are called *templates*, since they reflect the structure of properties in a generic way. As illustrated in Fig. 4.1, ToPs extract the information from MoTs and define the MoP instances. The structure and semantics of the MoP instances are defined by the metamodel *MetaProp*.

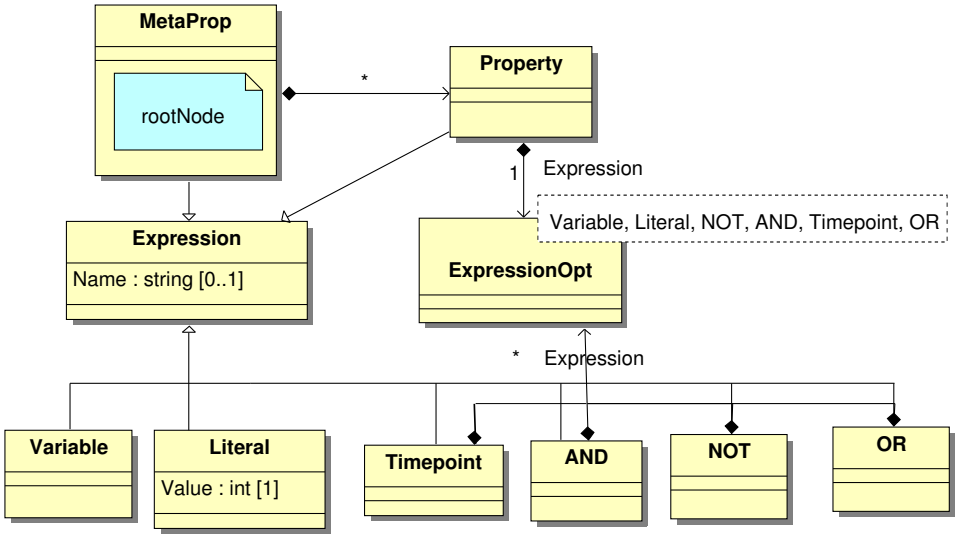


Figure 4.2: MetaProp – modeling the structure of an abstract temporal trace (simplified)

MetaProp Metamodel

Fig. 4.2 shows a simplified version of MetaProp, the metamodel definition of the Model-of-Property. It defines how the property structure is described in the MoP. The domain of the metamodel is the creation of an abstract temporal trace. A temporal trace describes the behavior of a sequential design over a time interval. Based on this notion, the metamodel definition is created.

The rootnode of the metamodel is *MetaProp* which has a composition relation to the class *Property*. The class *Property* composites of *ExpressionOpt*, which is an "option class" for all the operators. *Variable, Literal, AND, NOT* and *OR* represent a subset of all the primitive operators supported. An option class is a mechanism of enabling polymorphism within the automation framework. A list of all operators supported in the flow is tabulated in Appendix C. The *Timepoint* operator is a special operator used to insert time (or clock) delays in the expression trace. The option class *ExpressionOpt* takes any operator as its arguments. All primitive operators including the *Timepoint* operator can accept any expression as an argument. For the *MetaProp* metamodel, the framework provides an infrastructure (readers, writers, APIs). These API functions are utilized effectively to simplify the ToP coding.

Templates-of-Properties

The main purpose of ToPs is to extract information from the MoTs and to create MoP instances. The MoP instances represent a set of expected behaviors to be satisfied by the RTL implementation. The coding of ToPs is extensively aided by the APIs created for the *MetaProp* metamodel. Together with the underlying Python languages, these APIs form a highly flexible DSL for model transformations. Since the MoTs are created with an intention to allow various microarchitecture alternatives, ToPs are implemented to generate the property models for any supported microarchitecture. For various microarchitecture alternatives, the ToPs are easily adapted with minimal manual effort.

A half-adder circuit shown in Fig. 4.3 is considered for illustrating a ToP example. The results of an addition (sum and carry) are saved to register elements.

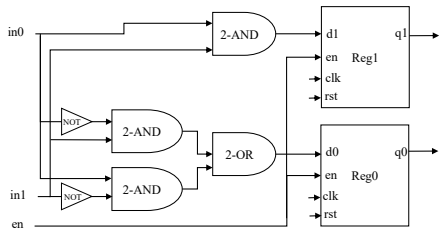


Figure 4.3: Half-adder circuit

```
1  def define_half_adder_mop(self, parent=metaprop):
2      self.metaprop = MetaProp()
3      sum_exp = XOR('in0','in1')
4      carry_exp = AND('in0','in1')
5      mop_sum = self.metaprop.addProperty(Name='ha_sum',
6                  Expression=IMPLY('en', DELAY(1, EQ('q0', PAST(sum_exp)))))
7      mop_carry = self.metaprop.addProperty(Name='ha_carry',
8                  Expression=IMPLY('en', DELAY(1, EQ('q1', PAST(carry_exp)))))
```

Figure 4.4: Code snippet of an example ToP

The ToP description for creating property models for a half-adder circuit is listed in Fig. 4.4. The Python function *define_half_adder_mop*() is executed to create the MoP instances. Variables *sum_exp* and *carry_exp* hold the expression traces for sum and carry outputs, respectively. These expressions are defined using the operators *XOR* and *AND*. In the next lines, the MoPs are defined by adding the instances of attributes *Name* and *Expression*. The variable *mop_sum* holds the MoP for *sum* in which the expression trace in *sum_exp* is extended by using the operators *IMPLY EQ*, *PAST* and *DELAY*. The operator *DELAY* is an alias for the *Timepoint* operator and is used to insert delays of one or more clock cycles in the expression tree. The *Delay* operator accepts two arguments in which the first argument specifies the number of clock cycle delays. The operator *PAST* is an extended temporal operator, which is used to sample the values of the argument in the previous clock cycle. In the end, *mop_sum* models the property that shall be satisfied by the output port *q1* in Fig. 4.3. Similarly, *mop_carry* models the property that shall be satisfied by the output port *q0* in Fig. 4.3.

Fig. 4.5 shows the property models created for the sum and carry outputs respectively. The operators are denoted by the nodes and the edges represent the arguments of the operators. The *IMPLY* operator is the root node of the expression tree, which takes exactly two arguments. For any operator, the arguments can either be an expression or a terminal node such as a *Variable* or a *Literal*. These MoP instances are platform independent, i.e., they do not contain artifacts of any specific property specification languages. Therefore, the MoPs can be mapped to any property specification language in the view layer.

Temporal Semantics for MoP

A definition of the MoP is provided by describing its temporal semantics. The temporal trace represented by the MoP can be interpreted in a discrete linear model of time. In temporal logic, the temporal trace or formula is validated against a formal model. The validation is performed over an infinite path of the formal model [65]. Temporal logic provides a convenient formalism to define properties for reactive systems. In order to define the semantics for MoP, we need a

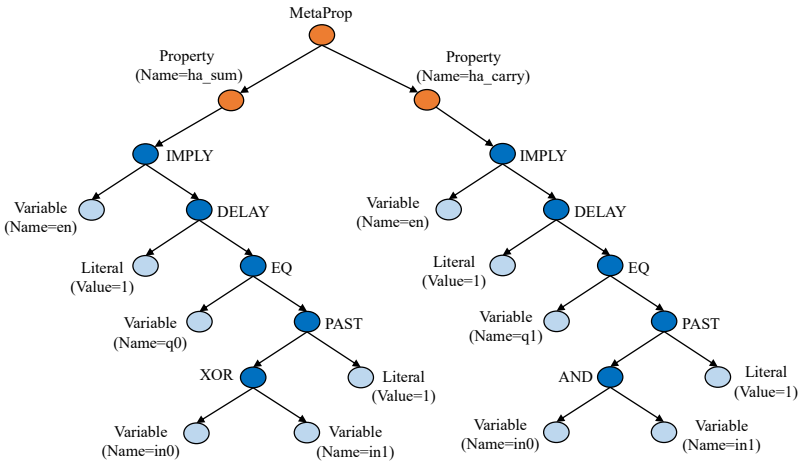


Figure 4.5: Expression trees of the MoP instances

notion of sequence of states with each point in time having an unique successor, based on a linear time model. Therefore the semantics of MoP are defined by mapping them to the linear temporal logic (cf. Section 2.4.2) as exemplified in the following.

For the circuit shown in Fig. 4.3, two MoP instances have been defined by the ToP. The temporal formula for the MoP defined for the *sum* output is given by:

$$G(\neg en \vee ((in_0 \oplus in_1) \rightarrow Xq_0)) \tag{4.1}$$

where, G and X are temporal operators *globally* and *next* respectively. Similarly, the temporal formula for the MoP defined for the *carry* output is given by:

$$G(\neg en \vee ((in_0 \wedge in_1) \rightarrow Xq_1)) \tag{4.2}$$

4.2.3 Model-of-View Layer

The Model-of-View (MoV) layer forms the final and least abstract model layer of our adaption of MDA for property generation. In the MoV layer, the MoPs defined in the previous layer are mapped to a specific property specification language to generate the properties. Since the MoPs are platform language independent, they can be mapped to any property specification language. The generation flow supports the SVA [2] and ITL [85] property specification languages.

The approach taken in the MoV layer allows the developer to think about the views that need to be generated instead of focusing on the formatting and indentation of the generated views. The formatting and indentation are specified independent of the MoV. In this approach, it is also possible to modify the formatting and indentation without modifying the mapping of MoPs to MoV.

The detailed view generation process is illustrated in Fig. 4.6. The main component of the view generation step is View Language Description (VLD). The VLD is created for each target language supported by the generation flow (e.g., VLD for SVA). The goal of the VLD is to define the structure and syntax of the target language. VLD is close to the Extended Back-Naur Form (EBNF) and describes the rules of the formal language [98]. The VLD is utilized to generate the majority of the necessary MoV layer components as shown in Fig. 4.6.

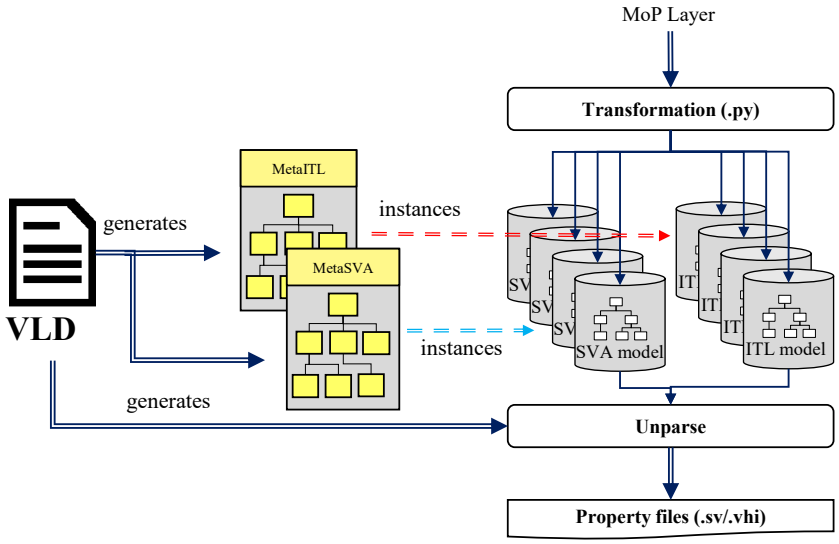


Figure 4.6: View generation based on abstract syntax tree of the property languages

The Metamodel (*MetaView* in Fig. 4.1) of the Model-of-View is built based on the VLD. The metamodeling framework provides the APIs which are used to read and write Model-of-View instances. Furthermore, the view generation step, i.e., building the target views from MoV instances is completely automated. Based on the VLD, a transformation code is generated which is based on the tree traversal. The steps are repeated for all supported property languages.

```
1 Poperty ::= `Property ` <PropertyName> `;\n`
2              [PropertyBody]
3             `endproperty`
4 PropertyBody ::= [TriggerCondition] |
5                  [DisableCondition] |
6                  <PropertyExpressionTrace>
7 TriggerCondition ::= $indent(`\t`)$`
8                      (@`<Edge> <ClockPort>`)\n`
9 DisableCondition ::= $indent(`\t`)$`
10                     (disable iff`<DisableExpressionTrace>`)\n`
```

Figure 4.7: Snippet of View language description of SVA property

View Language Description

Fig. 4.7 shows a simplified snippet of the VLD used for generating the properties in SystemVerilog Assertions language. The similarity between the VLD and the formal EBNF description of SVA is apparent. The VLD description consists of a list of production rules, where each of those rules in turn consist of a set of terminal or non-terminal symbols. As mentioned earlier, the EBNF of a certain language describes the formal grammar of the language and consists of a set of rules for distinguishing grammatically correct code from incorrect code. The VLD, in addition to describing the formal grammar with a set of rules, also provides other utility functions in the generation flow.

In order to facilitate the automatic generation of the metamodel of the MoV instances, several formalisms are introduced in the VLD. For every production rule in the VLD, a class is

added to the metamodel. In Fig. 4.7, *Property, PropertyBody, TriggerCondition* and *Disable-Condition* are the production rules. The rules inside '[...]' symbol are non-terminal rules and the rules inside '<...>' symbol are terminal rules. For every non-terminal symbol in a rule, an association relation is added to the class. For every terminal symbol in a rule, an attribute is added to the class. More details of the metamodel generation from the VLD description is outlined in [98].

The VLD is also used to define the indentation and for formatting the target code. A simple approach to formatting the target code is inserting terminal strings containing white spaces into the VLD. However, this approach cannot handle indentation correctly as many production rules can occur at different levels of indentation. For example, a line break may be required after the time point operator and the time point operator itself can occur at any level of the expression trace. To handle such scenarios, formatting directives are utilized. These directives are directly introduced into the view language description. Their handling is implemented as Python code and work by post-processing code that has been generated. A set of predefined directives are used for correct indentation of the target code, line breaks at certain line widths and correct alignment of neighboring lines. For example, $indent('\t')$ in Fig. 4.7 in line 7 ensures an additional indent for the clock trigger declaration for the property.

4.3 Binding Design Details by Obeying 4-eyes Principle

A critical aspect of assertion-based verification is that the properties need to be bound to the design implementation. Further as outlined in Challenge 4 (cf. Section 4.1), formal verification takes a grey-box verification approach and requires some of the design's internal signal details. The existing property specification languages use different mechanisms to accomplish the binding of properties to the design blocks. In ITL, the generated properties are automatically mapped to the design module, requiring the properties encoding correct RTL signals with correct hierarchy [85]. In SVA, there are 2 different ways to accomplish the binding [2]:

1. Declare the properties in a separate module, instantiate this module in the design module where the properties are required and pass the property variables into the module via ports. However, this method requires modification to the design module.
2. Using the "bind" construct: the module holding the properties can be directly bound to the top-level or any design module with the bind construct.

The encoding of design details for property generation ensuring that the 4-eyes principle is obeyed is described in the following.

The property generation flow complements the RTL generation flow outlined in Section 3.4. With the RTL and property generation flows, OMG's model-driven software development principles have been adapted for hardware design and verification domain. A practical methodology for automating the generation of RTL and property files from specifications is illustrated in Fig. 4.8. In Fig. 4.8, the RTL generation flow is shown on the left, while the property generation is shown on the right side. Both the RTL and property generation flows start with the same models, i.e., formal specification models compiled from the informal specifications. After the MoT layer, the generation flows take separate paths as required by the 4EP.

The immediate step in the RTL flow is to transform the MoTs — realized by coding ToDs — into a Model-of-Design (cf. Section 3.4). The Model-of-Design contains all the necessary details of an intended design in a language-independent representation. At this stage, the prop-

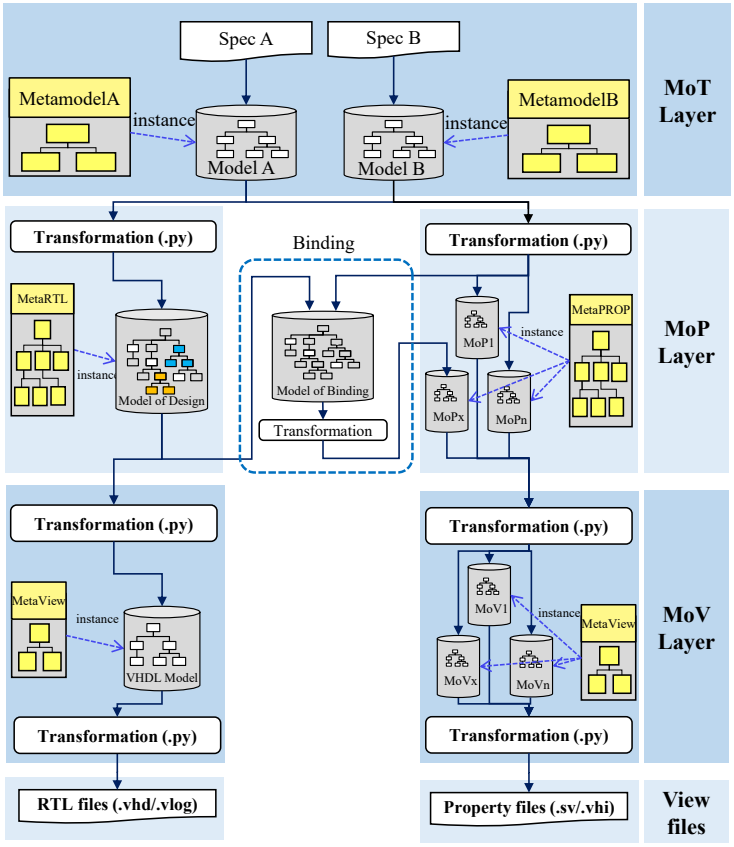


Figure 4.8: Generation of RTL and properties by obeying 4-eyes principle

erty models could also be derived from the same transformation step. Such a generation step is not ideal and disobeys the 4EP. This is because errors introduced during the ToD transformation step are propagated to both the RTL and properties (so called *common mode* errors). For example, an *AND* logic may be transformed as an *OR* logic due to a bug in the transformation code that is used for both, MoD and MoP. Further, in addition to the transformation errors, errors may also occur due to the bugs in generation flows. In such cases, the errors may not be identified by the verification step as both RTL and properties carry the incorrect logic. In order to exclude such errors, the RTL and property generation take separate flows as shown in Fig. 4.8.

Therefore, the overall generation framework for automating the digital design development has been split into two separate flows. In order to realize the binding of RTL signals with minimal manual efforts, an intermediate Model-of-Binding (MoB) is utilized. Similar to other models, the structure and constituents of the Model-of-Binding are defined by the metamodel *Metabind*.

The *Metabind* metamodel definition is shown as an UML class diagram in Fig. 4.9. The metamodel mainly consists of two parts: component details (*Class Component*) and property module details (*Class PropertyModule*). The rootnode has a composition relation to both the classes. A component (RTL block/module) can be a top module and is composed of zero-to-many *sub-components* and multiple *ports*. A property module has multiple variables (*Class Variable*) encoded in properties. A property module can be bound to the top RTL module or to a sub-module in the hierarchy. Additionally, each variable has a reference to a port signal of a component (*association relation RTLPort*) as shown in Fig. 4.9.

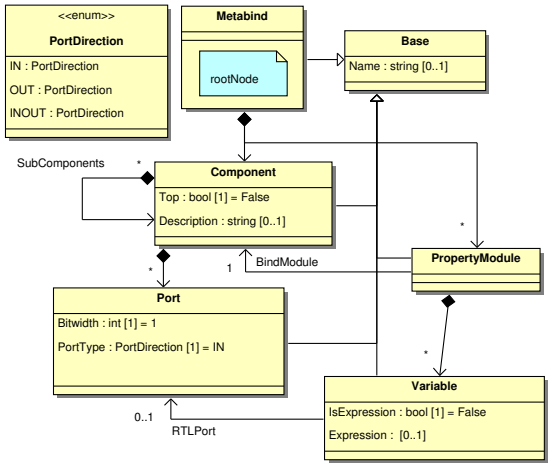


Figure 4.9: UML class diagram representing metamodel of binding

As shown Fig. 4.8, MoD is a component tree that contains the microarchitecture implementation of an intended design (cf. Section 3.4). Hence, MoB is automatically populated by iterating through the MoD components. Further, the generated MoB is appended with a list of variables, which are used to define the property models in ToP. The above steps are automatically accomplished by utilizing the infrastructure generated for *MetaRTL* and *Metabind* metamodels. After this step, for each variable used in the ToP, a port signal of a component is referred. For cases where a specific RTL port signal is not available, a Boolean expression is defined using the RTL signals as symbols. The Boolean expression is constructed with port signals as the symbols.

For every variable defined in the ToP, a macro definition is created such that a macro call returns the corresponding RTL signal. This improves the readability of the view files and additionally addresses any change in the RTL description (i.e., changes in the MoD due to changes in the transformation step). Additionally, the properties are bound to the RTL module which will be specified in the Model-of-Binding.

4.4 Modeling Design Specifications for Property Generation

In the previous sections, a model-driven property generation flow has been introduced. A major goal of the proposed flow is to establish an automatic flow to target code from design specifications. Typically, the benefits of a generation flow are quantified in terms of productivity gains, i.e., the overall time saved when compared with the manual property development. The approach proposes to model the design specifications in formalized models with clear semantics. A Python-based DSL for properties is utilized, which forms the substantial part of the flow. It is used to transform the formalized specification models to property models. The DSL is further optimized for describing property traces with extended APIs generated from metamodel definitions [50].

Modeling of design specifications becomes significant as it defines the amount of effort that is spent on developing ToPs. To this end, effective modeling of design specification takes a central role such that the generation framework is efficiently utilized and the efforts for developing ToPs are reduced. In the subsequent sections, we discuss the modeling of combinational and sequential design specifications for property generation.

4.5 Generation of Properties for Combinational Designs

Hardware designs in which the behavior of output signals depend only on the input signal values are referred to as combinational designs. At any given time point t, the output signals depend only on the values of input signals. A generic block diagram of a combinational design is shown in Fig. 4.10.

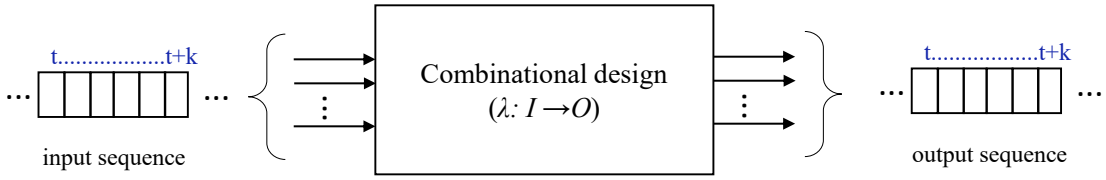


Figure 4.10: Block diagram of a generic combinational design

Definition 13 [Combinational function]:
Consider a combinational design $\mathcal{M}$. Let $\{i_1, i_2, \cdots i_n\} \in I$ and $\{o_1, o_2, \cdots o_n\} \in O$ be the input and output symbols, respectively. Let λ be the output function of the combinational logic. The output symbol O is a function of the input symbol I, i.e., $\lambda : I \mapsto O$. □

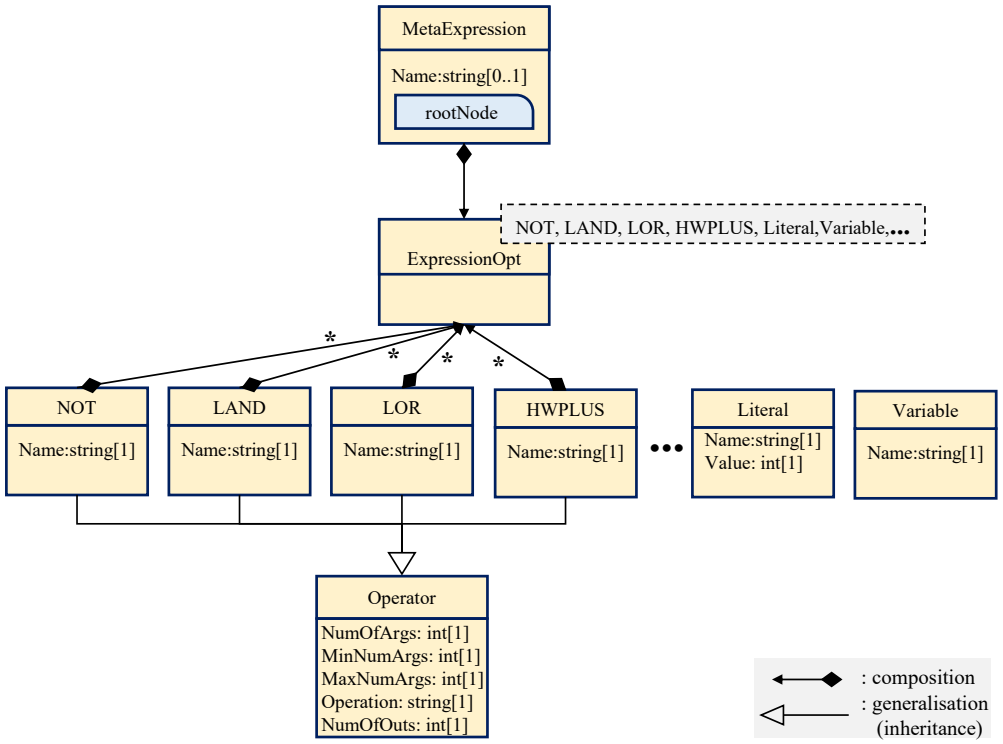


Figure 4.11: MetaExpression: metamodel definition to model combinational logic function

Combinational designs are composed of logic gates (*AND, OR, NOT,*$\cdots$) which are assembled in a required layout to produce a specific (often complex) logic function. The function of a combinational design can be specified using Boolean expressions, truth-tables or logic diagrams. Logic diagrams in turn can be represented as Boolean expressions. The specifications

that are available in structured formats (e.g., truth-tables) are automatically parsed by the readers provided by the automation framework (cf. Section 3.2). ToPs are implemented to extract the data from these tables and to generate the properties in a required pattern.

4.5.1 MetaExpression

For specifications available in informal formats, it is necessary to capture the data in models for enabling property generation. In order to model the combinational logic or Boolean Expressions, a metamodel called *MetaExpression* is developed as shown in Fig. 4.11. The shown metamodel defines the structure of a combinational logic expression in a generic way. The root node of the metamodel is *MetaExpression*, which holds a composition relation to the class *ExpressionOpt*. *ExpressionOpt* is an option class which takes the form of its argument operator. All primitive operators such as *LAND, LOR, NOT, etc.* (all operators are not shown in Fig. 4.11 for reasons of simplicity) are supported by the *MetaExpression* metamodel. The class *Operator* defines the generic attributes for all operators. All operators except *Literal* and *Variable* hold a generalization (or inheritance) relation with the class *Operator*. In other words, all operators except *Literal* and *Variable* inherit the attributes of the class *Operator*. Further, the MetaExpression metamodel supports compositional operators such as *HWPLUS* or *MUX* to simplify the logic function definitions. Each primitive operator — except *Literal* and *Variable* which are terminal operators — accept other operators as arguments. That is, each operator has a composition relation to the class *ExpressionOpt*. This allows to capture the specification items in the form an expression tree.

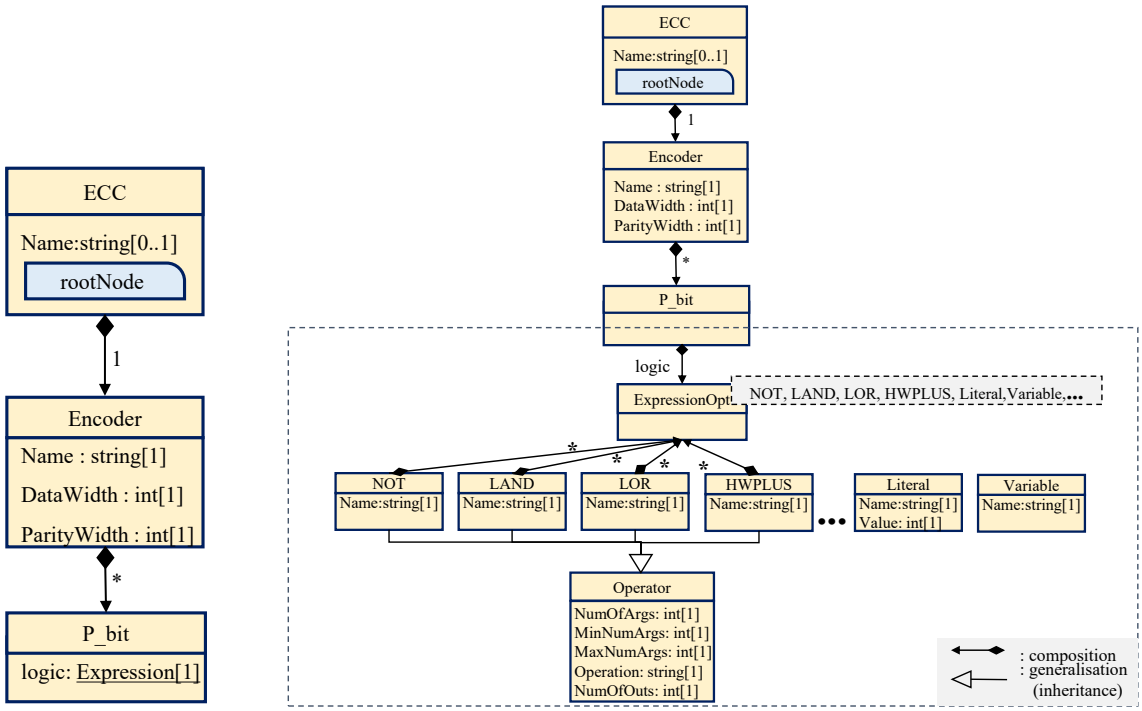
The expression metamodel is used as an external reference to define the type of attributes in other metamodel definitions. This is possible within the underlying automation framework, which allows advanced features such as model merging, model difference and model modification (cf. Section 3.2).

4.5.2 Illustrative Example: ECC Encoder

We consider an Error Correction Code (ECC) encoder as an example to illustrate the specification modeling of combinational designs. ECCs are utilized for detecting and correcting bit errors in safety-critical designs [31]. The ECC encoder is used to generate redundant bits for the input data vector.

A metamodel definition created to model the abstract characteristics of an ECC encoder is shown in Fig. 4.12. In Fig. 4.12 (a), the class *Encoder* defines attributes to capture the width of data and parity bits of the encoder. This class has a composition relation to the class *P_bit*, which includes an attribute to define the logic for parity computation. The attribute *logic* is of type *Expression* and the structure of the *Expression* type is defined by the metamodel definition shown in Fig. 4.11. Utilizing the features of the automation framework, the two metamodel definitions are merged. The resulting metamodel definition used to describe an ECC encoder model is shown in Fig. 4.12 (b).

The model instances are created with specific values for data and parity widths. Further, the logic expression for parity bit computation is captured in the form of a Boolean expression following a specific encoding mechanism. The ToPs are then implemented such that the properties are generated for any model instance of the ECC encoder. First, the ToPs extract the



(a) Metamodel definition of ECC encoder

(b) ECC encoder metamodel merged with expression metamodel

Figure 4.12: Modeling ECC encoder specifications

characteristics of the ECC encoder. Then the logic expression for parity computation is mapped to a property model, such that a property is generated for each output parity bit of the encoder.

4.6 Generation of Properties for Sequential Designs

Combinational designs do not contain state elements to store the design state. The output function of such designs ($\lambda : I \mapsto O$) depends only on the values of input signals at an arbitrary time point t. In contrast, sequential designs use state elements (registers/flip-flops) to store the design state. Sequential designs contain both combinational logic and state elements to realize the required functionality as shown in Fig. 4.13. The state elements introduce the temporal relationship between the input and output signals. The behavior of a sequential design can be described by discrete and deterministic Finite State Machines (FSMs) [65, 111]. The usage of state charts for modeling system behavior at transaction level has been explored in [46].

Definition 14 [Finite State Machine]:
A finite state machine $\mathcal{M}$ is a 6-tuple $\mathcal{M} := (\mathcal{S}, \mathcal{S}_i, I, O, \delta, \lambda)$.

- $\mathcal{S}$ is a finite set of states,
- $\mathcal{S}_i \in \mathcal{S}$ is a non-empty set of initial states,
- I is a finite set of input symbols,
- O is a finite set of output symbols,
- δ is a state transition function with $\delta : \mathcal{S} \times I \mapsto \mathcal{S}$, and
- λ is a output function with $\lambda : \mathcal{S} \times I \mapsto O$. □

The described FSM models the sequential behavior of a design as a Mealy state machine.

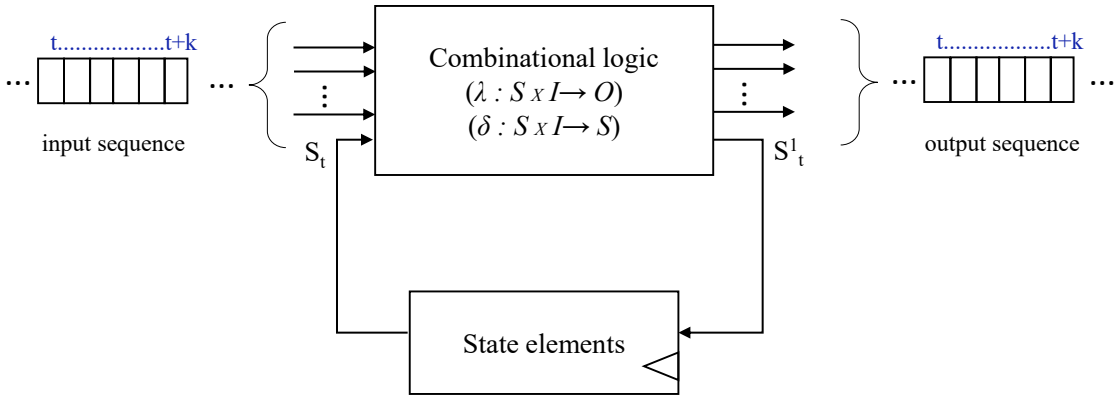
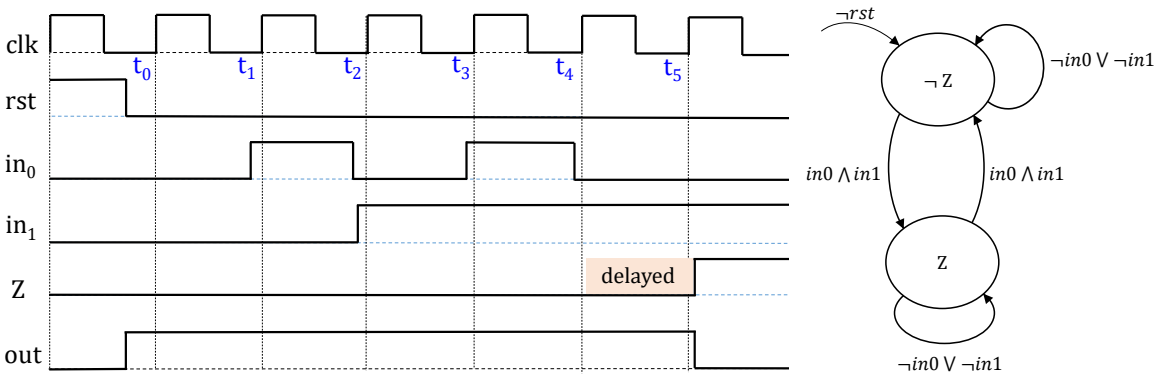


Figure 4.13: Block diagram of a generic sequential design

In a Mealy machine, the outputs are a function of both inputs and present state of the design. In case of a Moore state machine, the outputs are a function of present state of the design only, given by $\lambda : \mathcal{S} \mapsto O$.

State machines can also be used to model the specifications of sequential designs from which the properties can be automated. However, the formalism lacks the time annotation required for property generation. Let us consider an example for illustration.



(a) Trace (signal behavior) (b) State diagram

Figure 4.14: Signal behavior and state transition diagram of a sequential design example

Example Let us consider a sequential design with two input signals in_0 and in_1, one state bit z, encoding two states $s_0 := \neg z$ and $s_1 := z$, and one output signal *out*. The input and output signal behaviors are illustrated in the waveform diagram shown in Fig. 4.14(a). The design uses an asynchronous active-high reset signal *rst* and is sensitive to the positive (or rising) edge of the clock signal *clk*. After releasing the reset signal, the design enters the initial state s_0. The output signal *out* is high only when the design is in the state $s = s_0$. The design transitions between states s_0 and s_1 when both the input signals are high ($in_0 \wedge in_1$), delayed by one clock cycle. In other cases, the design remains in the same state.

For the example sequential design, the state machine is defined as $\mathcal{M} := (\mathcal{S}, \mathcal{S}_i, I, O, \delta, \lambda)$, where $\mathcal{S} := \{s_0, s_1\}$ is the set of states with the initial state $\mathcal{S}_i = \{s_0\}$, $I := 2^{\{in_0, in_1\}}$ and $O := 2^{\{out\}}$ are the input and output alphabets, respectively, δ is the transition function with $\delta := \mathcal{S} \times I \mapsto \mathcal{S}$, and λ is the output function with $\lambda := \mathcal{S} \mapsto O$. The state transition graph of the FSM is shown in Fig. 4.14(b). The state transitions from the state diagram are given by:

$$\begin{array}{lll} \delta_{rst} & := rst & \rightarrow \neg z \\ \delta_{s_0 s_0} & := \neg rst \wedge (\neg in_0 \vee \neg in_1) \wedge \neg z & \rightarrow \neg z \\ \delta_{s_0 s_1} & := \neg rst \wedge (in_0 \wedge in_1) \wedge \neg z & \rightarrow z \\ \delta_{s_1 s_0} & := \neg rst \wedge (in_0 \wedge in_1) \wedge z & \rightarrow \neg z \\ \delta_{s_1 s_1} & := \neg rst \wedge (\neg in_0 \vee \neg in_1) \wedge z & \rightarrow z \end{array} \tag{4.3}$$

After modeling the behavior of an example design in a state machine, the properties can be automatically generated from the behavior model. Let us consider a property generated for the state transition $\delta_{s_0 s_1}$. Its LTL equivalent is shown in Eqn. 4.4. The LTL formula fails on the example design when evaluated in a FV tool due to the missing timing information. This is because the design changes to the next state s_1 from the current state s_0 only on the second rising clock edge (due to delayed transition). The timing information is not explicitly included in the state machine model of the example design.

$$G(\neg rst \wedge \neg z \wedge in_0 \wedge in_1 \rightarrow z) \tag{4.4}$$

A valid LTL formula for the state transitions should also include the timing information as shown in Eqn. 4.5. In the shown LTL formula, X^2 represents the delay of two clock cycles before the expected state transition can be asserted on the state variable z.

$$G(\neg rst \wedge \neg z \wedge in_0 \wedge in_1 \rightarrow X^2 z) \tag{4.5}$$

4.6.1 Trace-based Approach using Finite State Machine Notations

For a given design, a set of properties are developed such that each property captures a specific behavior over a finite time interval. The behavior of signals over a finite time interval is referred to as a *"trace"*. A *trace* defines specific values for signals at every time point within the specified time interval. We can say that a trace starts in an arbitrary important state and, after traversing through a finite number of unimportant states, reaches another important state. A property can be built to model a specific *trace* and a set of properties can be built to model all traces of a design. In the context of C-IPC (cf. Section 2.5), a trace or a set of traces can be equated to an operation of the design.

For the purpose of property generation with precise timing information, we utilize the notion of traces to model the sequential behavior. Further, FSM-like notation is used to define the *trace structure*.

Definition 15 [Trace Structure]:
A trace structure is a state transition system $\mathcal{T} := (\mathcal{S}, \mathcal{S}_i, I, O, \delta^l, \lambda^l, \theta)$, where

- $\mathcal{S}$ is a finite set of states,
- $\mathcal{S}_i \in \mathcal{S}$ is a non-empty initial state set,
- I is an input alphabet (finite set of input symbols),
- O is an output alphabet (finite set of output symbols),

- δ^l is a state transition function with $\delta^l : \mathcal{S} \times I \mapsto \mathcal{S}$,
- λ^l is the output function with $\lambda^l : \mathcal{S} \times I \mapsto O$, and
- θ is the trace function. □

In the state transition function $\delta^l : \mathcal{S} \times I \mapsto \mathcal{S}$, the superscript l (δ^l) denotes that it requires l clock cycles for the system to change from the current state to the next state. Similarly, in the output function $\lambda^l : \mathcal{S} \times I \rightarrow O$, the superscript l (λ^l) denotes that it requires l clock cycles for the system for computing the output signals.

Definition 16 [Trace function]:
A trace function θ of a trace structure $\mathcal{T}$ is a quadruple θ:= (s_s, s_e, Δ, Λ).

- $s_s \in \mathcal{S}$ is the state from which the trace begins at an arbitrary time point t,
- $s_e \in \mathcal{S}$ is the ending state of the trace,
- Δ is the sequence of state transitions with $\Delta = \langle s_s, \ldots, s_e \rangle$,
- Λ is the sequence of output evaluations with $\Lambda = \langle o_s, \ldots, o_e \rangle$, where o_s and o_e are the set of output symbols values when the design is in state s_s and s_e, respectively. □

A sequence of state transitions Δ of a trace function θ begins from the starting state of the trace (s_s) and ends in an ending state s_e. Similarly, the sequence of output evaluations Λ of a trace function θ starts in s_s and ends in s_e.

With the extended state transition function δ^l, the state transitions for the example design shown in Fig. 4.14(b) are given by:

$$\begin{array}{lcll} \delta^1_{rst} & := & rst & \rightarrow X^1\ \neg z \\ \delta^1_{s_0s_0} & := & \neg rst \wedge (\neg in_0 \vee \neg in_1) \wedge \neg z & \rightarrow X^1\ \neg z \\ \delta^1_{s_0s_1} & := & \neg rst \wedge (in_0 \wedge in_1) \wedge \neg z & \rightarrow X^2\ z \\ \delta^1_{s_1s_0} & := & \neg rst \wedge (in_0 \wedge in_1) \wedge z & \rightarrow X^2\ \neg z \\ \delta^1_{s_1s_1} & := & \neg rst \wedge (\neg in_0 \vee \neg in_1) \wedge z & \rightarrow X^1\ z \end{array} \tag{4.6}$$

The traces for the sample design are formulated as follows:

$$\begin{array}{lcll} \theta_{rst} & := & (-, s_0, \delta^1_{rst}, \lambda^1_{rst}) & := rst \rightarrow X^1\ (\neg z \wedge out) \\ \theta_{s_0s_0} & := & (s_0, s_0, \delta^1_{s_0s_0}, \lambda^1_{s_0s_0}) & := \neg rst \wedge (\neg in_0 \vee \neg in_1) \wedge \neg z \rightarrow X^1\ (\neg z \wedge out) \\ \theta_{s_0z_1} & := & (s_0, s_1, \delta^1_{s_0s_1}, \lambda^1_{s_0s_1}) & := \neg rst \wedge (in_0 \wedge in_1) \wedge \neg z \rightarrow X^2\ (z \wedge \neg out) \\ \theta_{s_1z_0} & := & (s_1, s_0, \delta^1_{s_1s_0}, \lambda^1_{s_1s_0}) & := \neg rst \wedge (in_0 \wedge in_1) \wedge z \rightarrow X^2\ (\neg z \wedge out) \\ \theta_{s_1z_1} & := & (s_1, s_1, \delta^1_{s_1s_1}, \lambda^1_{s_1s_1}) & := \neg rst \wedge (\neg in_0 \vee \neg in_1) \wedge z \rightarrow X^1\ (z \wedge \neg out) \end{array} \tag{4.7}$$

The reformulated LTL formula from the trace function $\theta_{z_0z_1}$ is given by Eqn. 4.8. The LTL formula correctly captures the signal values of the sample design and holds on the design implementation when evaluated in the FV tool.

$$G(\neg rst \wedge \neg z \wedge in_0 \wedge in_1 \rightarrow X^2\ (z \wedge \neg out)) \tag{4.8}$$

With the trace-based approach modeling the sequential designs using FSM-like notations, we bridge the missing time annotation between the specification items and the generated properties. Due to the presence of accurate time information in the formalized specification items, developing ToPs (described in Section 4.2.2) becomes straightforward. The focus is shifted to annotating temporal behavior of design alternatives. This in turn reduces the manual effort

for property generation. In the following sections, we introduce the metamodel definition and illustrate the trace-based approach for pipelined processors implementing RISC based ISAs.

Metamodel for trace-based approach using FSM notations

A metamodel definition is developed to model the specifications of a sequential design based on trace-based approach using FSM like notations. The metamodel definition is shown in Fig. 4.15 as a UML class diagram with *MetaSTS* as the root node.

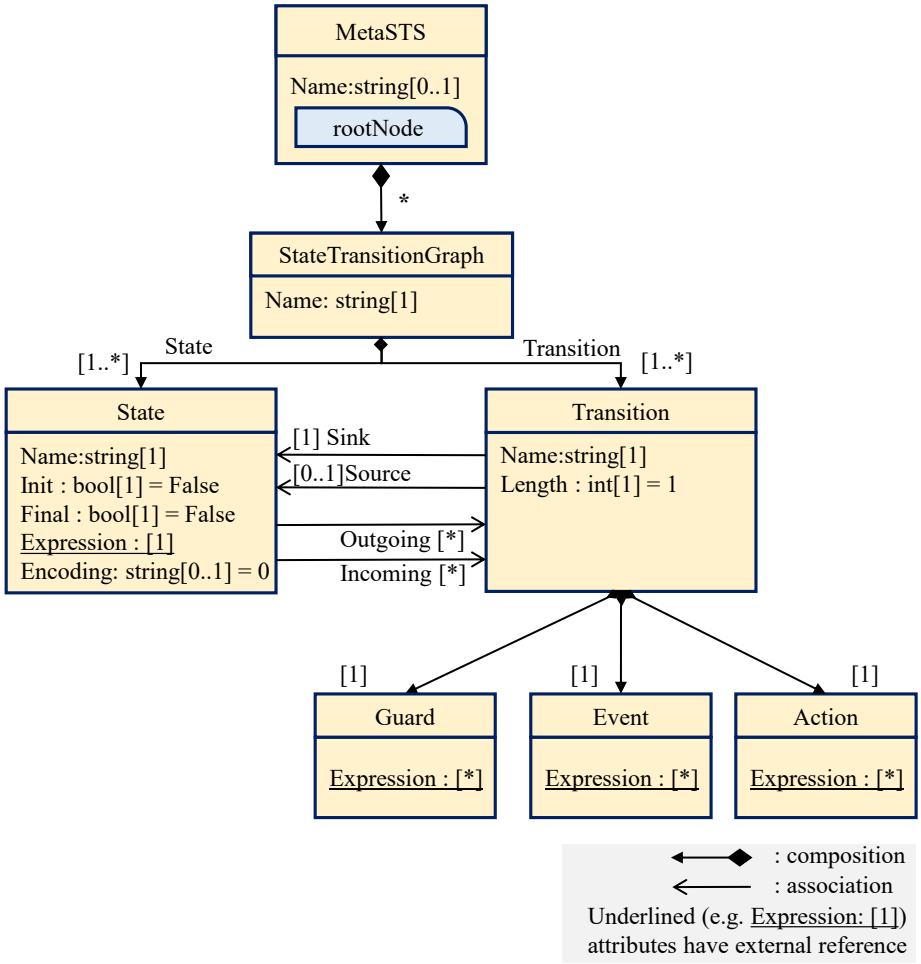


Figure 4.15: Metamodel definition for trace-based approach using FSM notations

The root node *MetaSTS*, which stands for meta state transition system, has a composition relation to the class *StateTransitionGraph* with multiplicity $*$, i.e., 0, 1 or more. This allows to create more than one transition graph definition for a given design. For example, in case of processor cores, multiple state transition graphs are needed to model the behavior of different instruction classes such as register-arithmetic, load-store, control-flow.

Each *StateTransitionGraph* has arbitrary number of states (class *State*) and state transitions (class *Transition*). A state can be an initial state (attribute *Init*) or a final (attribute *Final*) state of the transition graph. A state definition may not be an explicit hardware state, but can be captured as a sequence predicate. That is, a CNF clause of signals (including state bits) can be captured as a state of the design. For this purpose, the class *state* has an attribute called *Expression* to model

the state as a Boolean predicate. The type of attribute *Expression* is defined by the metamodel definition shown in Fig. 4.11. By utilizing the features of the automation framework, i.e., by utilizing the external referencing mechanism, a composition relation is added from the class *State* to the class *ExpressionOpt* in the *MetaExpression* metamodel definition. The attribute *Encoding* is used when an explicit state encoding is available in the design. Each state may have one or more incoming and outgoing transitions and these are denoted by the association relations *Incoming* and *Outgoing*, respectively. It should be noted that only important states are given a state definition and the unimportant states that a trace visits before reaching another important state are captured as part of the *transition*, as explained below.

Each *transition* has a name, a source state (association relation *Source [0..1]*) from which the transition originates, and a sink state (association relation *Sink* [1]) in which the transition terminates. In case of an initial transition (e.g., reset), it is possible not to have a source state and for this reason the *Source* association has a multiplicity of 0..1 (zero or one). The *transition* class defines the attribute *Length* to specify the number of clock cycles required for the transition.

The *Transition* class has composition relation to classes *Guard*, *Event* and *Action*. The *Expression* attribute of the class *Guard* models the Boolean expression under which a specific transition can take place. In other words, the transition takes place only when the guard expression evaluates to *"true"*. Similarly, the *Expression* attribute of the class *Event* models the events which trigger the transition. Finally, the *Expression* attribute of the class *Action* models the actions performed by the design after the transition has been triggered. The type of the *Expression* attribute is defined by the metamodel shown in Fig. 4.11, similar to the *Expression* attribute of the class *State*.

The expression metamodel shown in Fig. 4.11 models the Boolean expressions including sequence predicates (cf. Section 2.5). However, the guard conditions and the actions of a transition are required to model the (unimportant) state transitions over a finite time frame. In order to model a trace that visits unimportant states, the next operator (X^l) is added to the metamodel definition of *MetaExpression* shown in Fig. 4.11. In particular, the *next* operator is added similar to other primitive operators. The *next* operator is a pair $next(E,l)$ which adds a temporal shift of the Boolean expression E by l clock cycles.

The metamodel definition shown in Fig. 4.15 enables to completely model the behavior of sequential designs such that the transitions, guard conditions, events and actions are embedded with the precise time information. Model instances of the metamodel are built to specify the sequential behavior in terms of traces. A property model (Model-of-Property) is created for each trace and a set of properties are generated covering all the trace definitions in the model instance.

4.6.2 Modeling ISAs and Processor Microarchitecture Behavior

The execution semantics of a processor core or central processing unit (CPU) are described by an instruction set architecture (ISA) manual [53, 117, 5, 100]. An ISA describes the programmer's model of a CPU and serves as the interface between hardware design and software program. The ISA defines the size of an instruction word, types of instructions and different instruction formats. Instruction set architectures can be broadly classified into CISC and RISC. CISC is an acronym for *complex instruction set computer* and emphasizes on the hardware design. The main idea of CISC is to define a set of complex instructions such that each instruction performs a series of operations and requires several clock cycles for completion.

RISC, which stands for *reduced instruction set computer*, emphasizes on the software and defines a set of simple instructions such that each instruction performs a specific operation and requires one clock cycle for completion. Since CISC uses complex instructions that perform multiple operations, the size of the program is small, but involves complex decoding logic in the processor implementation. On the other hand, RISC involves a simple and uniform hardware implementation, and requires a comparably larger program size. Further, RISC operations are straightforward to be mapped from high-level languages such as C or C++, where many CISC operations are only partially supported by compilers. Both CISC and RISC architectures have their own set of advantages and disadvantages [12]. Due to the simple and uniform hardware implementation and sophisticated compiler technology, RISC ISAs have gained a wider adoption.

The microarchitectural implementation of RISC processors typically follows a pipelined architecture. The pipelined architecture improves the program throughput and enables to apply higher clock frequencies. Depending on various requirements and trade-offs (e.g., area and speed), a specific number of pipeline stages are considered for a processor implementation. As elaborated in Section 4.2.1, the central idea of the MoT layer is to capture informal specifications in a formal model such that the implementation choices are not constrained. A combined formal model of ISA and pipelined behavior of a processor restricts the MoT layer to a specific architecture. Therefore, it is essential that the formal model of a processor does not constrain the microarchitectural alternatives (e.g., 3-stage pipeline or 5-stage pipeline). However, for efficient property generation, the microarchitectural behavior of a processor with precise timing information needs to be captured in a formal model. To address this, modeling of processor cores is done in 2 stages.

1. *Untimed model*: Model an ISA with all the necessary information for a processor implementation such that the implementation choices are not constrained.
2. *Timed model*: Extend the untimed model with precise timing information introduced by the pipelined architecture of processor cores. Further, the timed model is flexible and enables to model different microarchitecture alternatives of a processor.

MetaRISC: Metamodel for RISC-based ISAs

A metamodel definition for modeling RISC-based ISAs[1] is introduced in the following. The metamodel definition is shown in Fig. 4.16 as a UML class diagram. Typically, an ISA describes the size of an instruction word and a set of instructions. Each instruction has a predefined encoding which identifies the instruction class such as control flow, register arithmetic, load-store. Additionally, each instruction performs a specific operation and updates one or more architectural states of the processor upon completion. Execution of an instruction sequence (a program) can be interrupted by certain events called exceptions, which occur either internal or external to the processor core. To represent these characteristics the metamodel consists of four main components: 1) *architectural states*, 2) *encoding tree*, 3) *instructions* and 4) *exceptions*. The metamodel captures these 4 components with a composition relation from the root node *MetaRISC* to each component.

Architectural States: A processor implementation consists of several architectural states such as general purpose register file, program counter, control and status registers. Further, a

[1]The introduced metamodel definition is also applicable to model CISC-based ISAs. However, it may require more manual effort as the instruction encoding and behavior description are not uniformly describable.

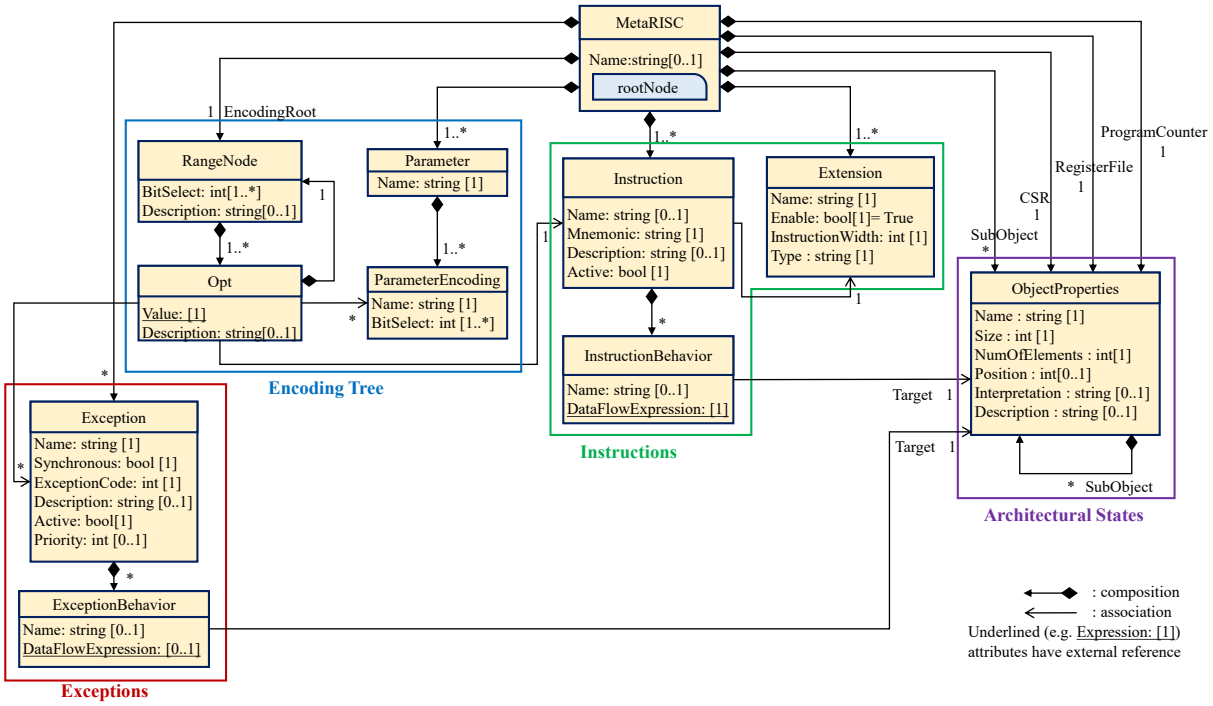


Figure 4.16: MetaRISC: metamodel definition for RISC-based ISAs

processor core interacts with the data memory through load-store instructions. Therefore, data memory is added as an external architectural state element (not shown in Fig. 4.16). To represent these important state elements, a composition relation is added from the root node to the class *ObjectProperties*. The class *ObjectProperties* has its own set of attributes to specify various properties of the state elements. An *ObjectProperties* instance is defined for each architectural state element with the corresponding attribute values.

Encoding Tree: As the name suggests, the encoding tree builds a tree structure to determine the format and parameters of an instruction word. The root node *MetaRISC* has a composition relation called *EncodingRoot* to the class *RangeNode*. The *RangeNode* class has an attribute called *BitSelect*, which is used to select specific bit positions of an instruction word. The *RangeNode* class has a composition relation to the class *Opt*, which has an attribute (*Value*) to specify the corresponding values for the bit positions. The *Opt* and *RangeNode* classes have recursive composition relations to each other, which helps to build a tree structure to capture the required format of an instruction word. Further, the root node has a set of parameters (class *Parameter*) and parameter encoding (class *ParameterEncoding*) which define the operand bit positions of the instruction word.

The encoding tree provides a high level of flexibility both in the way instruction sets can be described and in the types of instruction sets that can be captured. It is possible to describe any irregular instruction set (e.g., CISC type) with variable instruction width. The encoding tree also provides an option to build one single tree for all aspects of instruction decoding. That is, the tree provides sufficient information to decode instructions and to identify instruction parameters required for the property generation.

Instructions: An ISA has a set of instructions and instruction extensions. For example, the fifth iteration of RISC ISA [117] proposes several extensions to the 32-bit base-integer in-

struction set (RV32I) such as 16-bit compressed instruction extension, multiplier and division extension, 64-bit instruction extension and more. To capture this, the root node has a composition relation to the classes *Instruction* and *Extension* with multiplicity [1..*]. Each instruction belongs to a specific extension in the ISA. This is captured via an association relation between the classes *Instruction* and *Extension*. Each instruction has a *Name*, a *Mnemonic* and an attribute (*Active*) to specify whether the instruction is supported in a specific microarchitecture.

The behavior of an instruction is described as a set of changes the instruction triggers in the architectural state of the CPU. For every piece of state that is influenced, the *Instruction* instance has one *InstructionBehavior* child. For example, an *ADD* instruction performs an addition of the operands and stores the result in the general purpose register file. The program counter value is also incremented to the next instruction address. To capture the behaviors in the form of an expression tree, the class *InstructionBehavior* has an attribute called *DataFlowExpression*, whose type is defined by the metamodel definition shown in Fig. 4.11. Each operation of the instruction influences an architectural state of the design. To specify this, an association relation is added to the *ObjectProperties* class.

Exceptions: Exceptions are special events that may occur during the course of a program's execution. Exceptions can be termed as unexpected events that interrupt the program execution flow. For example, when the decoder encounters an instruction word that has not been defined by the ISA, the processor rectifies this anomaly by executing a predefined exception service routine. Exception events follow a similar execution semantics as instructions. When an exception is triggered, a set of predefined operations are performed and respective architectural states are updated.

The *MetaRISC* metamodel enables to define a set of possible exception events. The class *Exception* defines a set of attributes to describe various properties of an exception. The attribute *Synchronous* is used to indicate the triggering mechanism of an exception. The attributes *ExceptionCode* and *Priority* are used to assign a specific code and a priority value for each exception. The *Active* attribute is used to indicate the support for an exception event in the microarchitecture. Some exceptions are always associated with specific instruction execution. For example, the divide-by-zero exception can only occur with operations that utilize the divider circuit. Similarly, data memory access fault exceptions can occur during the execution of load-store instructions. Therefore, to indicate this type of relation, an association relation is added between the class *Exception* and class *Opt* which also holds an association relation to the specific instruction.

The model instances of the metamodel definition shown in Fig. 4.16 are created by filling the details of a specific ISA. The model instances of the *MetaRISC* metamodel are models of the processor cores without any timing annotation or constraints and do not restrict the microarchitectural choices. For the RTL generation, the flow outlined in Section 3.4 is used to define various architecture alternatives, in which the Templates of Design (ToD) describe the microarchitecture blue-prints [98].

Trace-based Approach for Modeling Pipelined Architectures of Processors

A model instance created as described in the previous section includes all details of an ISA required for a processor implementation. However, the details are not sufficient for property generation as the microarchitectural implementation of a processor core may utilize different pipelined architecture alternatives to improve the program throughput [87]. The pipelined ar-

chitecture introduces a deeper sequential behavior for each instruction execution. For example in a 5 stage pipeline architecture, instead of one clock cycle latency, it may take 5 or more clock cycles for an instruction to complete. Therefore, for the purpose of property generation, a timed model of the processor core is required to precisely model the behavior of an instruction execution.

The notion of "traces" introduced in Section 4.6.1 is utilized to model the behavior of an instruction execution in a pipelined processor. In Fig. 4.15, a metamodel definition for modeling a state transition system based on the notion of traces has been introduced. A processor core is a state transition system, in which a group of instructions belonging to a certain instruction class exhibit similar behaviors except that they perform a specific dataflow operation depending on the opcode value (e.g., addition or subtraction). For example, *ADD* and *SUB* instructions belong to the same instruction class in RISC-V ISA [117] and only differ in the dataflow operation they perform, i.e., addition and subtraction of operands. A set of state transition graphs are created such that the operations of all the instructions supported by the processor core are modeled.

The model instances of the metamodel shown in Fig. 4.15 can be created using the GUI provided by the automation framework (cf. Section 3.2). Alternatively, the model instances can be defined in a program utilizing the APIs and plugins provided by the automation framework. For processor cores, the latter approach is more suited as it enables flexibility to define model instances based on the number of pipeline stages and other configurable parameters (e.g., support for different instruction extensions).

A pseudocode snippet is shown in Fig. 4.17 for extracting the ISA details and to define state transition graphs for each instruction class[2] for a processor core implementing a RISC-based ISA. A model instance of the *MetaRISC* metamodel (Fig. 4.16) is passed as an argument to the procedure $STG\_DEFINITIONS$. The procedure definition accepts the default length of state transitions as another argument, and returns a list of state transition graphs.

A graphical representation of the combined state transition graphs defined for a 5-stage processor is shown in Fig. 4.18. In a 5-stage pipeline, each instruction traverses different pipeline stages modifying the corresponding signals based on its instruction encoding and specified operation. The graphical representation captures the behavior of different instructions classes of a typical RISC-based ISA.

For each instruction class, a state transition graph is initialized as shown in line 4. An initialized state transition graph models the behavior of all instructions belonging to a certain instruction class. A state object if (instruction-fetch) is added to the state transition graph. The state object if is defined by a Boolean expression $expression\_fetch$ (line 7), which represents a state of the processor when a new instruction is fetched. Lines 8 and 9 show the addition of two state transitions if_reset and if_if to the state transition graph with the corresponding attributes. For example, the state transition if_if has state object if as both source and sink states, and a transition length of one clock cycle (def_len).

Similarly, lines 11-14 and 16-19 show the addition of state objects (*id*, *ex*) and state transitions (if_id, id_ex) relevant to the processor pipelines instruction-decode and instruction-execute, respectively. During state transitions, the processor performs one or more operations and the expected signal behaviors are captured as action expressions ($action\_expression\_if\_id$, $action\_expression\_id\_ex$). The state objects if, *id* and *ex* are common to all instruction classes. However, state objects *mem* (memory-access) and *wb* (write-back) are specific to certain in-

[2]It should be noted that a state transition graph is used to define the traces. And this does not necessarily have the behavior of an FSM.

```
1:  procedure STG_DEFINITIONS(isa_model_ins, def_len)
2:      transition_graphs ← {∅}
3:      for each inst_type ∈ isa_model_ins.get_types() do
4:          stg = isa_model_ins.addStateTransitionGraph(inst_type.getName())
5:
6:          if = stg.addState()   //instruction fetch
7:          if.insert(expression_fetch)
8:          reset_if = stg.addTransition('reset', ∅, if, def_len)
9:          if_if = stg.addTransition('wait', if, if, def_len)
10:
11:         id = stg.addState()   //instruction decode
12:         id.insert(expression_decode)
13:         if_id = stg.addTransition('if2id', if, id, def_len)
14:         if_id.insert(action_expression_if_id)
15:
16:         ex = stg.addState()   //instruction execute
17:         ex.insert(expression_execute)
18:         id_ex = stg.addTransition('id2ex', id, ex, def_len)
19:         id_ex.insert(action_expression_id_ex)
20:
21:         if inst_type = (branch||jump) then
22:             ex_if = stg.addTransition('jump', ex, if, def_len)
23:         else
24:             if inst_type = (load||store) then
25:                 mem = stg.addState()   //memory-access
26:                 mem.insert(expression_mem_access)
27:                 ex_mem = stg.addTransition('ex2mem', ex, mem, def_len)
28:                 mem_mem = stg.addTransition('wait', mem, mem, def_len)
29:             if inst_type = store then
30:                 mem_if = stg.addTransition('mem2if', mem, if, def_len)
31:             else
32:                 wb = stg.addState()   //write back
33:                 wb.insert(expression_write_back)
34:                 if inst_type = load then
35:                     mem_wb = stg.addTransition('mem2wb', mem, wb, def_len)
36:                 else
37:                     ex_wb = stg.addTransition('ex2wb', ex, wb, def_len)
38:                     ex_wb.insert(action_expression_ex_wb)
39:                 wb_if = stg.addTransition('wb2if', wb, if, def_len)
40:         transition_graphs.insert(stg)
41:     return transition_graphs
```

Figure 4.17: Pseudocode for defining state transition graphs

struction classes. For example, instructions that belong to branch or jump instruction class are not expected to perform any operations in memory-access and write-back pipeline stages. Accordingly, lines 21-39 show the conditional definition of state and state transition objects based on the instruction class.

An important aspect of modeling the design specifications is to allow architecture alternatives. Towards this end, the transitions are created such that the number of clock cycles required to transition from one state to another is configurable. This is achieved by setting the *Length* attribute of the class *Transition* in Fig. 4.15. A state transition graph for a 3-stage pipeline is shown in Fig. 4.19. Here, the state definitions from 5-stage pipeline are preserved and com-

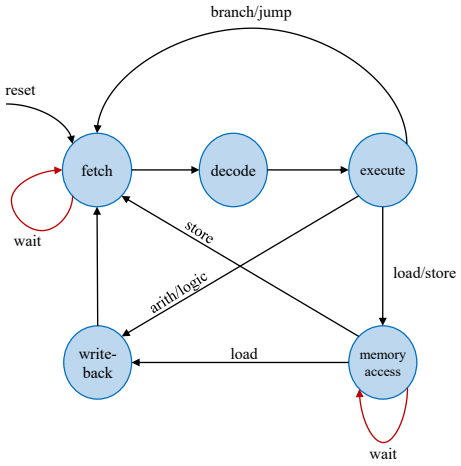


Figure 4.18: State transition graph for a 5-stage pipeline processor

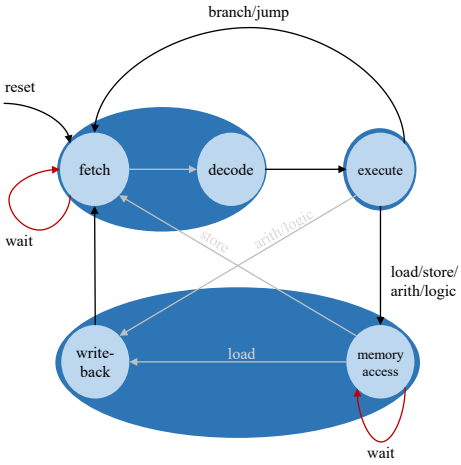


Figure 4.19: State transition graph for a 3-stage pipeline processor

bined to form composite state definitions. For example, the length of the transition *trn_if_id* is set to zero and the states *fetch* and *decode* are combined to form a composite state *fetch_decode*. Similarly, other state definitions are merged based on the length of the transitions. The required combination of state and transition definitions are extracted automatically by transforming the existing expression definitions based on the target architecture. The correct behavior is preserved due to the dataflow semantics of the expression.

4.6.3 Coverage Perspective

A mechanism is needed to validate that all specification items are considered by a set of properties and that all functionalities of a design under verification (DUV) are verified by a set of properties. In Section 2.5, a completeness criterion for a set of properties is elaborated. Such a completeness mechanism is not applicable for purely combinational designs due to the absence of state elements (cf. Section 4.5). A notion of operations and transition from one important state to the next important state cannot be conceptualized for combinatorial designs. However, the case-split test from completeness checks (cf. Section 2.5) can be used to validate that all

input combinations are considered by a set of properties. For ensuring that the DUV is fully verified, a property is generated for each specification item. In other words, a set of properties are generated targeting all output signals. Additionally, tracing of API calls while generating properties ensures that every specification item is covered at-least once.

For sequential designs, the completeness criterion described in Section 2.5 is followed. First the sequential behavior of a circuit is modeled as a trace structure. A trace function of a trace structure can be equated to an "operation" of the C-IPC methodology defined in Section 2.5 (cd. 10). On the same notion, a conceptual state machine (CSM) can be derived from a trace structure. After modeling the design specifications as a trace structure, ToPs are setup to generate the properties following the C-IPC methodology. A set of operation properties are generated for a trace structure such that each property captures a specific trace of the design. By using the C-IPC methodology it can be proven that no verification holes has been left. Otherwise, the specification items are incomplete and must be extended.

Additionally, to ensure that all specification items are considered for property generation, function calls to retrieve specification items are traced. So it is ensured that every specification item is covered by at-least one property model. The completeness strategy for processor cores differs from those of typical sequential designs (e.g., serial-peripheral interface bus). This is because in processor cores, multiple operations may be simultaneously active at a certain time point, whereas most sequential designs perform only one operation at a specific time point. A completeness strategy for processor cores following C-IPC methodology is outlined in Section 5.3. We introduce a new processor verification methodology by property generation and S^2QED, and formulate a completeness strategy in Sections 5.5.2 and 5.5.5. For typical sequential designs, we provide a full overview of the property generation flow including specification modeling, generation of properties and subsequent completeness analysis and validate it in a real life design in Section 6.2.1.

4.7 Related work: Property Generation

Property generation approaches for functional verification of hardware designs have been explored before. It is interesting to compare the existing works to the property generation flow proposed in this work. The related works for property generation can be classified to three categories as described in the following.

4.7.1 Automated Formal Apps

Commercial formal verification tools provide pre-packaged solutions commonly referred to as "formal apps" [85, 19, 78, 108]. Some well known examples are connectivity verification app, control and status register verification app, security path verification app, fault injection and detection app, etc. The formal apps take the metadata information as input and produce properties for a given use case. For example in connectivity verification, the connections at an SoC level or at block level are captured in a metadata format such as comma separated values (CSV), IP-XACT or extensible markup language (XML). The properties are generated from the metadata information and are used to verify that the blocks or IPs are connected according to the specifications. Similarly in other use cases, the DUV is verified with the generated properties to ensure that the DUV conforms to the metadata description.

The formal apps offered by the commercial tools address specific use cases where the verification intent is limited and the specification details are available in a metadata format. These formal apps do not address the main use case, where the properties need to be developed from informal specifications. In many cases, the significant (manual) effort is needed to develop the metadata information required for formal apps.

4.7.2 Generation from Simulation Data and Static RTL Analysis

A number of approaches exist that generate/extract properties from the data collected during simulation runs [115, 101, 51, 92, 25, 122]. Although these approaches differ in their implementation mechanism, they use simulation test cases to generate and dump data required for automatic extraction of assertions. The proposed techniques use data mining, static analysis of the RTL and machine learning algorithms to generate candidate assertions. The candidate assertions are derived from the temporal traces created during simulation runs. These candidate assertions may also include incorrect assertions and hence, need revision in a formal or simulation tool to eliminate the spurious assertions.

Assertions generated from such mechanisms may not be well suited for comprehensive design verification. Building extensive/exhaustive set of test cases for simulation runs is not realistic (due to large and complex designs) and hence, the available simulation database may not cover the entire design space. Thus, the generated assertions are incomplete, insufficient and can only be used to supplement the simulation based verification. The use cases include coverage closure, documentation and bug hunting.

A static approach is proposed in [122], in which the assertions are generated by adopting an RTL automatic test pattern generation (ATPG) algorithm. A set of assertions is generated for each output signal and only a minimal set required for covering the complete design space are considered. Even though the approach can be used to obtain 100% design space coverage, potential bug escapes are possible as the assertions are not generated from specifications.

In [123], a technique is proposed in which the assertions are generated from natural specifications by performing semantic analysis of the sentences. The approach parses sentences and examines the syntactic parse tree to locate subtrees that are associated with important specification clauses. Although this approach creates more interest, the applicability and scalability to real life designs is questionable.

4.7.3 Generation Following the Property-Driven Development Paradigm

In [113, 73], a design methodology called “Property-Driven Design” (PDD) is proposed. In PDD, the system-level specifications are captured in an abstract system model and this system model is used as the golden reference for further design steps. In other words, PDD attempts to shift the design focus from RTL to the system level by establishing a sound abstraction between RTL and system-level descriptions.

In PDD, a set of operation properties and RTL templates are generated from an abstract system model. In the next step (refinement step), the design engineer adds microarchitectural details to the generated RTL templates. During this refinement activity, the generated operation properties are appended with cycle- and bit-accurate details. After the refinement step, an RTL design and a set of operation properties that are formally proven to hold on the RTL design are realized.

The PDD method builds upon *path predicate abstraction* to establish a sound abstraction between the RTL and the system-level description. The advantage of the PDD paradigm is to obtain a formally verified design at lower costs when compared to conventional design flows with property checking.

In contrast to the generation flow presented in this thesis, PDD is a design paradigm that aims to shift the design focus from RTL to the system level. Similar to the work presented in this thesis, the PDD method starts from abstract system specifications. Considering only the property generation part of PDD, both methods aim to improve the overall verification productivity by generating a complete set of properties. The main difference is the stage at which cycle- and bit- accurate details are added. In the generation flow of this thesis, these details are captured as part of the formal specification models. In case of PDD, these details are added as part of the refinement activity.

Chapter 5

Formal Verification of Processor Cores

Property generation by following the principles of model-driven software development is presented in Chapter 4. The generation flow partially addresses the growing productivity gap in design verification domain due to the constant increase in complexity of designs. The generated properties are used to verify design implementations in a formal verification (FV) tool.

Although the combination of automatic property generation and FV provides an efficient methodology to design verification, they do not address other major concerns. 1) FV (with brute-force approach) often runs into complexity issues due to the complex nature of designs i.e., large or sequentially deep designs. 2) FV requires high expertise and in-depth design knowledge to ensure that all design functionalities are effectively analyzed. 3) The specification of a design may not cover the complete behavior of the intended system. Even though the specification items of individual design components may be completely and correctly specified, the interaction between the components may not be completely specified, leaving room for undocumented design behaviors. These undocumented behaviors lead to ambiguity when dealing with counterexamples. While some counterexamples show design behaviors that can be ignored as acceptable behaviors within the scope of the design application, others reveal behaviors that must be avoided in the implementation. To address these concerns, effective strategies are required to realize the benefits of property generation and formal verification.

In the subsequent sections of this chapter, formal verification of processor cores is explored. In particular, formal verification of processors including both simple and superscalar pipelined processor cores is discussed[1].

The specifications of processor cores are described by instruction set architecture (ISA) manuals. An ISA describes the programmer's model of a processor by specifying a set of instructions to be supported by the processor core and by defining the semantics of an instruction execution [53, 106, 117]. For example, an ISA describes the size of an instruction word and assigns specific meaning to different bit positions such that, upon encountering a specific instruction word, the processor core performs a set of pre-defined operations. While some ISAs specify the microarchitecture of the processor cores, most ISAs enable design engineers to implement custom microarchitectures. For example, RISC-V ISA is the 5^{th} iteration in the RISC based ISAs [117] and allows design engineers to customize the microarchitecture implementations. This paves the way for design engineers to customize the implementation to realize non-functional requirements such as safety, throughput, timing, chip area, power consumption

[1]Simple pipelined processors contain only one pipeline, whereas superscalar processors include two or more pipelines.

and more.

Design engineers typically employ techniques such as pipelining, branch prediction or speculative execution, parallel processing, out-of-order execution, scoreboard, caching, etc. to customize the processor core implementation for different applications [87]. As a result, ensuring the correct implementation of an instruction set in a given microarchitecture poses an enormous verification challenge. Adding only a subset of the optimization techniques increases the verification complexity by many folds.

A number of approaches for formal verification of processor cores have been proposed [18, 10, 15, 59, 91, 7]. To briefly summarize, these techniques require high manual effort and high verification expertise in applying formal methods. Additionally, the methods in [18, 10, 59, 91] do not present a complete verification strategy and as a result, can only be used as bug hunting techniques. Recently, two processor verification methods named ***S***ymbolic ***Q***uick ***E***rror ***D***etection (SQED) [104] and ***S***ymbolic initial state ***S***ymbolic ***Q***uick ***E***rror ***D***etection (S^2QED) [45] have been proposed. Both SQED and S^2QED are highly effective in detecting difficult to find bugs, but do not provide a complete verification solution for processor cores. In the following these methods are elaborated and, subsequently, a complete processor verification solution based on S^2QED is proposed.

The chapter is structured as follows: a motivating example is discussed in Section 5.1 to illustrate the complexity posed by processor verification. Different types of logic bugs in processor implementations are discussed in Section 5.2, with examples. A well established criterion for complete processor verification based on C-IPC is elaborated in Section 5.3. A brief background on SQED and an extension to the technique with symbolic initial states with IPC is described in Section 5.4. In Section 5.5, a complete processor verification method by extending the principles of S^2QED is proposed. The extensions for covering exception handlers and superscalar processor verification are elaborated in Sections 5.5.3 and 5.5.4, respectively. Completeness of the proposed approach for processor verification is discussed in Section 5.5.5. A summary of the related processor verification methods and their comparison to the method proposed in this thesis is outlined in Section 5.6.

5.1 Motivating Example

Consider a 5-stage pipelined processor implementation shown in Fig. 5.1. Let us assume that the processor core supports arithmetic, logical, branch/jump and load-store instructions. The processor core follows a Harvard architecture by implementing separate instruction and data memory interfaces.

An instruction that is fetched from the instruction memory is decoded by the *Decoder* to determine the operation to be performed by the core. Depending on the instruction word's encoding and pre-defined operation, one or more program-visible states of the processor core are modified. For an *add* instruction, the *ALU* is utilized in the *execute* stage to compute the sum of the operands. Similarly for a *branch* instruction, the *Branch Control* unit is utilized in the *execute* stage to determine the fetch address of the next instruction. For load-store instructions, the memory address is computed during the *execute* stage and the value from memory is read in the *memory-access* stage. The *Increment* unit in the *execute* stage is used to increment the contents of source register for load-store instructions. The increment operation is performed after computing the effective memory address. The *Register File* is updated in the *write-back*

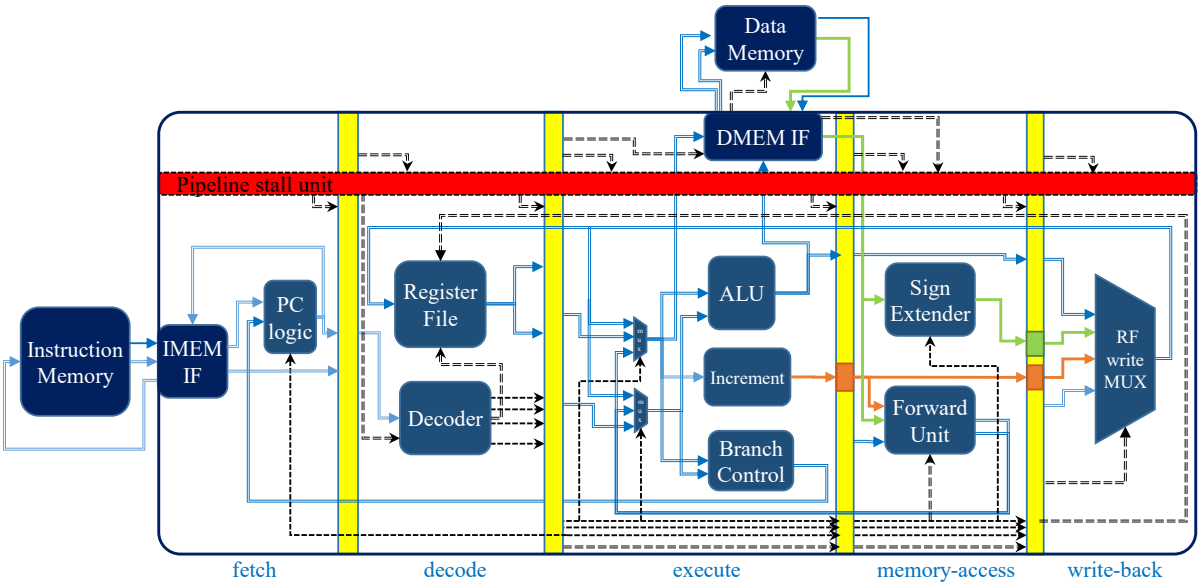


Figure 5.1: Block diagram of a 5-stage pipelined processor ([87])

stage with the results of an instruction execution.

Due to overlapping execution of instructions in a pipelined architecture, dependency scenarios such as read-after-write (RAW), write-after-write (WAW) or write-after-read (WAR) hazards may occur. A pipelined processor core implements units such as *Pipeline stall unit* and *Forward unit* to handle such dependencies. During load-store instructions, it is possible that the data memory requests the core to insert wait states when the data is not ready to be delivered or accepted.

```
1: mv R4, #0024          // [R4] = #0024
2: mv R3, #0004          // [R3] = #0004
3: ld R4, [R4+], #0100   // [R4] = DMEM[#0124] and [R4] = [R4] + #0004
4: add R2, R4, R3        // [R2] = [R4] + [R3]
```

Figure 5.2: Instruction sequence executed by the processor (shown in Fig. 5.1)

Let us consider an instruction sequence shown in Fig. 5.2. The instructions in lines 1-2 move the constant values #0024 and #0004 to registers *R*4 and *R*3, respectively. The instruction in line 3 loads a value from data memory to the destination register (*R*4) and post-increments the contents of the source register by #0004. The addresss of the memory location is computed by adding the contents of source register (*R*4) and an immediate value (#0124). The instruction in line 4 is an *add* instruction that adds the contents of source registers (*R*4, *R*3) and stores the result in a destination register (*R*2). When the *add* instruction is in the *decode* stage, the *ld* instruction is in the *execute* stage. Due to a RAW hazard between the *add* and *ld* instructions (*R*4 is destination of *ld* and source of *add*), decoding of the *add* instruction is stalled for as many clock cycles as the data is not available from memory.

It should be noted that both destination and source registers of the instruction in line 3 is *R*4. In such a scenario, since it is architecturally not possible to update the same register with two different values, the specification gives priority to the value from data memory. Therefore, the incremented value of *R*4 shall be ignored and the value from memory shall be written to the *Register File*. However due to a bug in the processor, although the read value from memory is written to the *Register File*, a wrong value (from post-increment) is put into the *Forward Unit*

to be forwarded to the *add* instruction.

The bug occurs due to an incorrect implementation of hazard detection and forwarding units. The bug results in a wrong value being written to the *Register File* and may corrupt the execution of an entire program. These types of bugs occur only when a specific sequence of instructions is executed. In order to detect such bugs, properties shall be written such that the formal tool is free to exercise every possible instruction execution scenario. To come up with a set of properties that activate all bugs in a processor core requires high manual effort and high verification expertise with traditional approaches.

5.2 Design Errors in Processor Cores

The microarchitecture of a processor core implementing an ISA is typically verified at the Register-Transfer Level (RTL). The RTL implementation of the processor needs to conform to the ISA and the microarchitecture specifications must be free of design errors. *Design errors* in hardware, often called "bugs" or "logic bugs", lead to incorrect behavior of the implementation in certain scenarios. The design errors may occur during the instruction execution, where the implemented behavior deviates from the specified behavior.

Definition 17 [Error Scenario]:
In the context of processor designs, an *error* is a deviation of the implementation's behavior from its specification, in a certain *error scenario*. An error scenario consists of (1) an instruction in which the error becomes observable in an architectural state, (2) the instruction's operands, and, (3) its *program context*, i.e., the sequence of previously executed instructions. □

An error scenario *activates* a logic bug in the processor core and results in a deviation from the specified behavior. In a processor core, logic bugs can be categorized into *single-instruction bugs* or *multiple-instruction bugs*. Our categorization is based on the assumption that the processor implementation consists of at-least 2 or more pipeline stages, a valid assumption for modern processor cores. In case of single-cycle implementation of processor cores, only *single-instruction bugs* can occur since *multiple-instruction bugs* require overlapping of instruction executions.

Definition 18 [Single-instruction bug]:
A bug is a *single-instruction bug* if there exists (1) an instruction opcode and (2) a set of operands such that the execution of the instruction leads to an error *in all program contexts*, i.e., independently of all previously executed instructions. □

> **Example** An instruction sequence is shown in Fig. 5.3 to illustrate a single-instruction bug. The instructions in the first four lines move the constant values to respective registers. The instructions in line 5 and line 6 perform the addition (*ADD*) operation. Both operands of the *ADD* instruction in line 6 are zeros. The result of this instruction execution is *#FFFF*, while the expected result is *#0000* (zero). Error scenario: a logic bug is activated when both operands of an *ADD* instruction are zero. The error is observable in the register file after the result has been committed. The bug is observable at every instance when both the operands of the *ADD* instruction are zero. From this illustration it is clear that the program context is not relevant for single-instruction bugs.

Definition 19 [Multiple-instruction bug]:
A bug is a *multiple-instruction bug* if it is not a single-instruction bug and if there exists (1) an

```
1: MOV R1, #4       // -> REGFILE[R1] = 4
2: MOV R2, #2       // -> REGFILE[R2] = 2
3: MOV R3, #0       // -> REGFILE[R3] = 0
4: MOV R4, #0       // -> REGFILE[R4] = 0
5: ADD R5, R1, R2   // -> REGFILE[R5] = 6
6: ADD R6, R3, R4   // -> REGFILE[R6] = #FFFF
```

Figure 5.3: Example of a single-instruction bugs

instruction opcode, (2) a set of operands, and, (3) a program context such that the execution of the instruction leads to an error. □

A multiple-instruction bug requires error scenarios consisting of specific instruction sequences that set up the microarchitecture of the processor such that the bug is activated. In contrast to a single-instruction bug, there are program contexts in which a multiple-instruction bug is not activated. In case of a processor core with only one pipeline, multiple-instruction bugs occur within the context of one pipeline. We refer to these bugs as *intra-pipeline multiple-instruction bugs*. An example for an intra-pipeline multiple-instruction bug is presented in the following.

Example Consider the instruction sequence shown in Fig. 5.4. A logic bug is activated when an *ADD* instruction (line 2) is executed immediately after a *MAC* (multiply-accumulate) instruction such that there is a Read-After-Write (RAW) data hazard between the instructions. The *MAC* instruction has a latency of 3 clock cycles and, for forwarding the correct value, the pipeline has to stall for 2 clock cycles before executing the *ADD* instruction. However, due to wrong hazard detection logic, the stall signal is high for only one clock cycle and a wrong value is forwarded to the *ADD* instruction. The bug leads to an error in an observable register (e.g., R_2) after the result of the *ADD* instruction has been committed. However, the same bug is not activated when the *ADD* instruction (line 5) is preceded by any instruction (e.g., NOP) such that there is no RAW data hazard. The implementation's behavior only deviates from the specification when a specific sequence of instructions is executed. Therefore, the program context is relevant for intra-pipeline multiple-instruction bugs.

```
1: MAC R4, R3, R6, #123 // bug activation - step 1
2: ADD R2, R3, R4       // bug occurs:  pipeline stalls for only one cycle,
                        // wrong forwarding of R4
3: NOP                  // bug not activated
4: NOP                  // bug not activated
5: ADD R2, R3, R4       // ADD executed without error
```

Figure 5.4: Example of an intra-pipeline multiple-instruction bug

In superscalar processors, instructions are executed simultaneously in two or more pipelines. Different pipelines are configured to execute specific classes of instructions [105, 21]. For example, instructions that perform integer operations are executed by the *integer pipeline* and all memory (load-store) instructions are executed by the *load-store* pipeline. The instruction stream is pre-decoded before the instructions are issued to specific pipelines. Intra-pipeline multiple-instruction bugs discussed above for simple pipelines occur in superscalar processors in the context of one isolated pipeline. Additionally, in superscalar processors with 2 or more parallel pipelines, logic bugs can also occur due to the incorrect implementation of the logic that

controls the data dependency between instructions executed in different pipelines. We refer to these bugs as *inter-pipeline multiple-instruction bugs* to imply that the error scenario is caused between different pipelines.

Example Consider an instruction sequence shown in Fig. 5.5. The load-word (*LW*) instruction precedes the *ADD* instruction in the program. The *LW* instruction is executed by the load-store pipeline, whereas the *ADD* instruction is executed by the integer pipeline. A logic bug is activated when there is a RAW data hazard between the instructions. The bug leads to an error in an observable register (e.g., R_2) after the result of the *ADD* instruction has been committed. However, the same bug is not activated when the *ADD* instruction (line 4) is preceded by any instruction (e.g., *AND*) that is executed by the integer pipeline. The correct value (e.g., R_3) is forwarded to the *ADD* instruction from the results of *AND* instruction within the integer pipeline.

```
1: LW R4, R3, #1   // bug activation - step 1
2: ADD R2, R3, R4  // bug occurs:  wrong forwarding of R4
3: AND R3, R3, R1  // bug not activated
4: ADD R2, R3, R4  // ADD executed without error
```

Figure 5.5: Example of an inter-pipeline multiple-instruction bug

A multiple-instruction bug may not necessarily lead to an incorrect architectural state after the program has completed executing. Instead, it may affect the overall performance of the program by unnecessarily inserting stalls in the pipeline. These type of bugs are referred to as *performance bugs* in the context of this thesis. They occur due to the incorrect logic that wrongly computes the data dependency between successive instructions and incurs stalls in the pipeline. Although such error scenarios cause delays and impact the overall performance of the processor, they do not result in an incorrect architectural state of the processor.

Example Consider the instruction sequence shown in Fig. 5.6. A logic bug is activated when an *ADD* instruction (line 2) is executed immediately after a *LW* instruction such that there is a RAW data hazard (register R_0) between the instructions. Here, the *LW* instruction loads a value from the data memory to the register R_0. Typically, the register R_0 is hardwired to zero, and any instruction issuing a write to R_0 is either dropped or ignored. In such a scenario, the *ADD* instruction does need to wait until the data value is available from the memory. However, the incorrect hazard detection unit detects a dependency between the two instructions and issues a stall. The same bug is not activated when the *ADD* instruction (line 4) is preceded by any instruction when there is no RAW data hazard.

```
1: LW R0, R3, #1   // bug activation - step 1
2: ADD R2, R0, R4  // bug occurs:  unnecessary stalling of pipeline
3: LW R5, R0, #1   // bug not activated
4: ADD R4, R0, R3  // bug does not occur:  no stalling of pipeline
```

Figure 5.6: Example of a multiple-instruction (performance) bug

5.3 Completeness Criteria for Processor Verification

A major goal in formal processor verification is to fully analyze the microarchitectural implementation with a complete set of properties. A completeness criterion for a set of properties is outlined in Section 2.5. Processor verification can also follow the notion of Complete Interval Property Checking (C-IPC). In contrast to controller-based designs, processors are typically implemented following a pipelined architecture. In controller-based designs, at an arbitrary timepoint t, one specific operation is active depending on the architectural state of the design. In case of pipelined processors, several operations are performed simultaneously such that different operations overlap by several clock cycles. In the example shown in Fig. 5.6, when the *ADD* instruction is in the decode stage, the *LW* instruction may be in the execute stage depending on the pipeline architecture. In a 5-stage pipeline, these two instructions are simultaneously active for 4 clock cycles. In pipelined processor implementations, each instruction execution can be considered an operation, which is preceded and succeeded by other instructions. Thus, an operational property in a pipeline captures the execution of a single instruction independent of preceding and succeeding instructions.

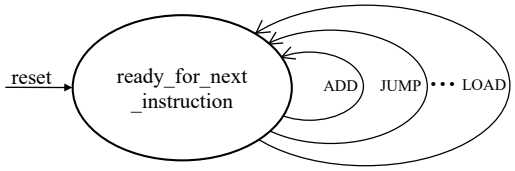


Figure 5.7: Conceptual State Machine of a pipelined processor

A conceptual state machine (CSM) can be constructed as shown in Fig. 5.7 for illustrating the operations in a processor core. Each operation starts and ends in a state in which the processor is ready for a new instruction. The state machine is trivial with the only state *ready_for_next_instruction*, with all transitions starting and ending in this state and reset leading to this state.

For determining the completeness of a set of properties, several tests are used in C-IPC which are also applicable to processor verification.

1. Successor test: In case of pipelined processors, the successor operation overlaps with the predecessor operation. This is due to interleaving of instructions in the pipeline. Consequently, there is no possibility of reaching a non-unique important state after the execution of an operation. As a result, proving the successor test is trivial which checks every operation is succeeded by another operation and that the sequence of operations are correctly captured [81].
2. Determination test: For the determination test, a property set needs to determine the values of program-visible states and outputs signals at every clock cycle. The determination test ensures that all important architectural states and outputs of the processor core are uniquely determined for every instruction execution. Formulating the determination requirements and proving the determination test for a processor core is similar to other designs.
3. Case-split test: In a processor core every instruction is encoded in an unique opcode. The case-split test for processor verification ensures that every instruction is uniquely covered by at least one property. The case-split test also ensures that for each instruction, the

execution of the next instruction is covered by at least one property. Since the case-split test considers every possible instruction word, it also considers illegal opcodes. Typically, all illegal opcode words are handled similarly in a processor implementation. A property must be implemented to cover the behavior of the processor when an illegal opcode is encountered.

4. Reset test: The processor core starts executing instructions immediately after the reset. In other words, the processor core enters the state *ready_for_next_instruction* after the release of reset signal. Since there is only one unique state, proving the reset test simply checks that the reset moves the processor to a unique important state.

These checks inductively prove that a complete property set uniquely describes the behavior of a processor for every instruction, in every possible program context (since every sequence of instructions is covered by a chain of operations). As a result, the set of properties verifies the correct execution of every instruction for every possible program context.

5.4 Symbolic Quick Error Detection

Symbolic Quick Error Detection originated from the principles of Quick Error Detection (QED) tests. QED is a post-silicon validation mechanism which detects and localizes errors with a quick turn-around time [72]. QED performs a set of systematic transformations of existing post-silicon validation tests into a new family of QED tests. The main advantage of QED over other post-silicon validation methods is that QED reduces the error detection latency, i.e., the time elapsed between the activation of a bug and the detection of an observable failure, by several orders of magnitude. Symbolic Quick Error Detection (SQED) is a formal verification technique originally proposed to find short error traces during post-silicon validation. Among several transformations, Error Detection using Duplicated Instructions for Validation (EDDI-V) [104] is used to target bugs in the processor core.

5.4.1 Background

Quick Error Detection: EDDI-V targets bugs in a processor by frequently checking the results of original instructions against the results of duplicated instructions created by EDDI-V. EDDI-V partitions the register file into two halves, with a unique one-to-one mapping between the registers in the two halves (e.g., R0 and R16, R1 and R17, etc.). For every load, store, arithmetic, logical, shift, or move instruction in the original test, EDDI-V creates a duplicate instruction using the duplicate registers. The execution starts from a QED-consistent registers state.

Definition 20 [QED-consistent registers]:
For a given processor with N registers, a QED-consistent register state is defined by the following equation:

$$qed\_consistent\_registers = \bigwedge_{a \in \{0..(N/2)-1\}} R_a = R_{a'} \tag{5.1}$$

where, R_a and $R_{a'}$ are the original and (corresponding) duplicate registers, respectively. □

The results of original instruction execution are committed to the first half of the register file, while the results of duplicate instruction execution are committed to the second register file half. Both instruction streams execute in the same order but are intertwined. To compare

intermediate results from both instruction streams, the transformation inserts frequent QED-consistency checks. A mismatch in any consistency check indicates an error (i.e., the QED test fails).

```
1: MOV R0, #0          // initialize R0 = 0
2: MOV R1, #1          // initialize R1 = 1
3: MOV R2, #2          // initialize R2 = 2
4: MOV R3, #3          // initialize R3 = 3
5: MOV R4, #4          // initialize R4 = 4
6: MOV R5, #5          // initialize R5 = 5
7: ADD R1, R2, R3      // R1 <- R2 + R3
8: SUB R4, R5, R1      // R4 <- R5 - R1
9: ADD R4, R2, R4      // R4 <- R2 + R4
```

Figure 5.8: Quick Error Detection: original instruction sequence

```
 1: MOV R0, #0          // initialize R0 = 0
 2: MOV R1, #1          // initialize R1 = 1
 3: MOV R2, #2          // initialize R2 = 2
 4: MOV R3, #3          // initialize R3 = 3
 5: MOV R4, #4          // initialize R4 = 4
 6: MOV R5, #5          // initialize R5 = 5
 7: MOV R16, #0         // initialize R16 = 0
 8: MOV R17, #1         // initialize R17 = 1
 9: MOV R18, #2         // initialize R18 = 2
10: MOV R19, #3         // initialize R19 = 3
11: MOV R20, #4         // initialize R20 = 4
12: MOV R21, #5         // initialize R21 = 5
13: ADD R1, R2, R3      // R1 <- R2 + R3
14: SUB R4, R5, R1      // R4 <- R5 - R1
15: ADD R4, R2, R4      // R4 <- R2 + R4
16: ADD R17, R18, R19   // R1 <- R2 + R3
17: SUB R20, R21, R17   // R4 <- R5 - R6
18: ADD R20, R18, R20   // R20 <- R18 + R20
19: CMP R4, 20          // compare corresponding register values
20: BNE qed_test_fail   // branch taken ⟹ bug detected
```

Figure 5.9: Quick Error Detection: instruction sequence with EDDI-V transformation

An example (original) instruction sequence is shown in Fig. 5.8. The original instruction sequence is converted to a QED test using EDDI-V transformations as shown in Fig. 5.9. Initially the duplicate registers (second half) are also initialized with the same values as the first half of the register file. The initialization is important to ensure that the sequence starts from a "QED-consistent register state". After executing the EDDI-V instruction stream, the corresponding register values are compared according to Eq. 5.1. When the execution of the branch-if-not-equal *BNE* instruction returns true, an error is detected.

Symbolic Quick Error Detection: In order to adapt QED for pre-silicon verification, SQED uses a BMC proof method for logic bug detection and localization. The BMC tool searches the space of all possible QED tests (within its bound).

The computation model of SQED for BMC is illustrated in Fig. 5.10. The computation model includes two modes, *original mode* and *duplicate mode*,that are used in series to form a QED test from an instruction sequence fetched by the processor. The processor core (shown as CPU in Fig. 5.10) is instrumented with a *QED module*, which is conceptually placed between the fetch and decode stages of the pipeline. The role of the QED module is to buffer the instructions considered by the BMC tool during the original mode, and to direct the BMC tool to execute the same but transformed instructions in the duplicate mode, in the same order. In

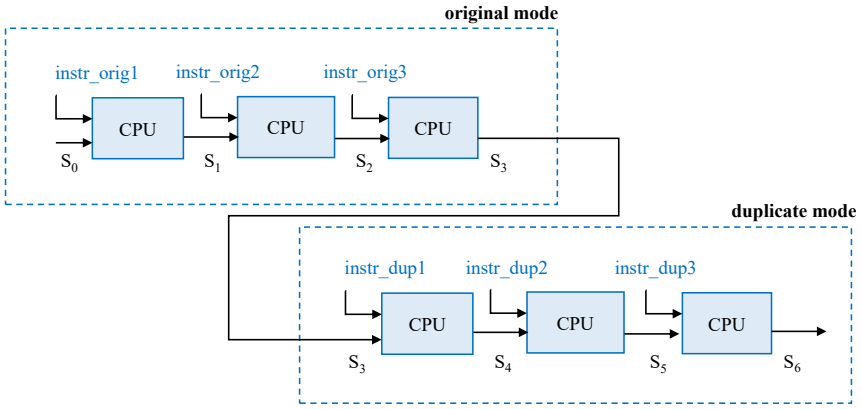


Figure 5.10: Computation model of SQED for Bounded Model Checking

Fig. 5.10, the processor core is unrolled for three iterations each in both original and duplicate mode. That is, the model is unrolled for a QED test of length three. While the BMC tool can consider any sequence of instructions in the original mode, the initial state S_0 is a fixed state from which the design unrolling starts (cf. Section 2.4.5).

The property provided to BMC is derived from the check that would detect the error during a QED test, for example using the EDDI-V transform. At the end of duplicated sequence, i.e., after executing the EDDI-V instruction sequence, the QED module raises a flag (*qed_ready*) to indicate that the BMC tool can compare the two register halves for consistency. The flag *qed_ready* is high, when the design state is expected to be QED-consistent in a bug free implementation. As a result, the BMC tool attempts to find a counterexample to the following property:

$$qed\_ready \implies qed\_consistent\_registers \tag{5.2}$$

where *qed_consistent_registers* represents the state given by the Eq. 5.1. The BMC tool checks this property for all EDDI-V QED tests within its bound, but only after a complete original and duplicate sequence of instructions have executed.

5.4.2 Extending SQED with Interval Property Checking

In Section 2.4.5, the computation model of BMC and the proof method is outlined. BMC starts unrolling the design from a fixed starting state and searches through the design state space to find a design state that violates the property formulation. When a counterexample could not be found, BMC has proven that the design fulfills the property for a given depth of k.

Experiments on real designs have shown that SQED with a BMC tool can identify hard-to-find bugs [104, 72]. However, since SQED uses BMC to search for failing QED tests, it cannot make a clear (unbounded) statement on the absence of QED tests that could fail. Consequently, failing QED tests that can be detected only if started from a specific starting state — which the BMC tool didn't consider — cannot be found.

To make a concrete statement about the absence of a failing QED test, SQED is extended with the interval property checking (IPC) proof method. IPC is a SAT-based model checking technique that provides unbounded proofs (cf. Section 2.4.6). The computation model of SQED with interval property checking is shown in Fig. 5.11. The main difference to the BMC-based approach is the state from which the unrolling of the design starts. In case of BMC, the starting

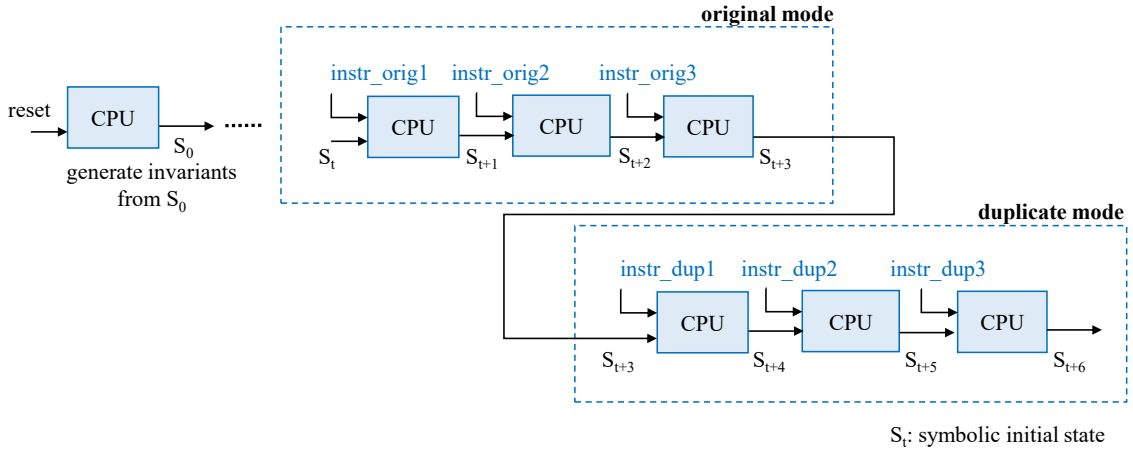


Figure 5.11: Computation model of SQED for Interval Property Checking

state (S_0 in Fig. 5.10) is the initial state (e.g., reset state) of the design. In IPC, the solver considers *all* states (S_t in Fig. 5.11) such that the assumption part of the property holds for the state. In other words, the starting state S_t is *symbolic*. In order to avoid spurious counterexamples (or false negatives), IPC uses induction-based techniques to generate invariants from the reset state.

```
assume:
    at t:  qed_ready;
prove:
    at t:  qed_consistent_registers();
```

Figure 5.12: SQED property in ITL style

At an arbitrary time point t, the IPC solver chooses a QED test to execute on the processor with a symbolic starting state S_t. When the QED module raises the flag *qed_ready*, the IPC solver attempts to find a counterexample to the property shown in Eq. 5.2. The property is written as an interval property in Fig. 5.12. Without further constraints on the starting state S_t, the property fails for the following reasons: (1) the starting state may not be a QED-consistent state; (2) the starting state may include active instructions in the pipeline that were issued before the timepoint t. Therefore, without additional constraints on starting state the solver returns with spurious counterexamples for the property shown in Fig. 5.12.

```
assume:
    at t:      qed_consistent_registers();
    at t:      flushed_pipeline();
    at t+k:    qed_ready;
prove:
    at t+k:    qed_consistent_registers();
```

Figure 5.13: SQED property in ITL style with additional constraints

To meaningfully extend SQED with IPC, additional constraints are needed on the starting state S_t. First, the state S_t needs to be a QED-consistent state, and second, at t there should not be any active instruction in the pipeline. The extended SQED property with additional constraints is shown in Fig. 5.13. The constraint *qed_consistent_registers*() at t restricts the starting state

to a QED-consistent state, while the constraint *flushed_pipeline*() ensures that the pipeline is in a clean state and there are no active instructions in the pipeline. The solver then attempts to check the QED-consistency when the QED module raises the *qed_ready* flag at an arbitrary time point $t+k$, where k is the length of the QED test in terms of number of instructions.

5.4.3 Observations on SQED

The IPC-based extension to SQED ensures that any QED test that can fail in the state space of a processor core is detected by the SQED property shown in Fig. 5.13. However, for making a clear statement on the absence of a failing QED test, the SQED property should pass on the computation model shown in Fig. 5.11. The IPC-based extension to SQED was applied on an industrial automotive micro-controller [103] and a RISC-V processor core with 5 pipeline stages. In the case of the RISC-V core, the solver was able to reach a depth of $k = 12$ after 60 hours of proof runtime, where k is the number of sequential unrollings of the design. Similarly on the micro-controller core, the solver was able to reach a depth of $k = 12$ after 24 hours of proof runtime. Although SQED presents an effective method for detecting hard-to-find bugs, there are shortcomings of the SQED method.

1. *SQED cannot detect all design errors in processor cores*: SQED finds a certain class of multiple-instruction bugs that result in a QED-inconsistent state (cf. Section 5.2), but fails to prove their absence. Further, SQED fails to detect single-instruction bugs as these bugs affect both original and duplicate modes identically and do not result in a QED-inconsistent state.
2. *Computation model of SQED increases the complexity*: The computation models of SQED for BMC and IPC-based proof methods are depicted in Fig. 5.10 and Fig. 5.11, respectively. The design unrolling is shown for a QED test consisting of three instructions. For such a a QED test of three instructions, the design is unrolled for six iterations (original and duplicate mode). In general, the sequential depth of the design is doubled for finding failing QED tests. High sequential depth coupled with the complexity of processor cores hits the state space explosion problem for formal tools. As a consequence, it is not possible to prove the absence of logic bugs with SQED in this case as well.

5.5 Complete-S^{2}QED for Processor Verification

In order to provide a valid statement on the absence of failing QED tests, Symbolic initial state Symbolic Quick Error Detection (S^{2}QED) is proposed [45]. S^{2}QED is different to the IPC based extension to SQED presented in Section 5.4.2. S^{2}QED proposes a new computation model in order to minimize the complexity for the solvers by reducing the number of design unrollings. In the subsequent sections, a complete processor verification methodology with S^{2}QED and property generation is presented.

5.5.1 Background: SQED with Symbolic Initial States

S^{2}QED attempts to address the complexity issue of SQED by modifying the computation model. The main source of complexity in SQED is the verification model in which the processor design is unrolled in a serial manner. In S^{2}QED, the processor under verification is duplicated

into two identical instances that are unrolled in parallel. That is, the computational model of S^{2}QED consists of two identical and independent instances of the processor design which execute instructions in parallel. At an arbitrary time point t, both instances are constrained to execute the same instruction. S^{2}QED proves that every instruction executes independently of previous pending instructions in the pipeline, i.e., independently of its program context.

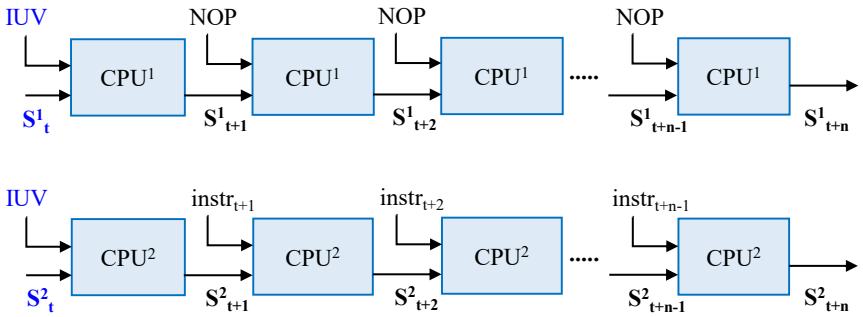


Figure 5.14: S^{2}QED Computational model

Fig. 5.14 shows the computational model of S^{2}QED in which the two CPU instances (CPU^1 and CPU^2) of the same processor are unrolled for a time window as large as the upper bound of the execution time of an instruction in the pipeline. In case of SQED, the QED-consistent state is defined for two halves of the same register file (cf. Section 5.3). In contrast, the QED-consistent state in the S^{2}QED computation model is defined with respect to the register files of two independent but identical CPU instances. For a processor with N registers, a QED-consistent register state is characterized by the named logic expression:

$$qed\_consistent\_registers := \bigwedge_{i=0}^{N-1} \left(R^i_{cpu1} = R^i_{cpu2}\right) \tag{5.3}$$

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

Definition 21 [QED consistency]:
In the S^{2}QED computational model, the two CPU instances are *QED-consistent* at a time point t, if the corresponding architectural state elements of both instances at time point t hold the same values. □

```
assume:
    at t_IF:                 cpu2_fetched_instr() = cpu1_fetched_instr();
    during [t_IF+1, t_WB]:   cpu1_fetched_instr() = NOP;
    at t_IF:                 cpu1_state() = S^1_t;
    at t_WB:                 qed_consistent_registers();
prove:
    at t_WB+1:               qed_consistent_registers();
```

Figure 5.15: S^{2}QED property (in ITL style)

Consider an S^{2}QED computational model, in which R_{cpu1} and R_{cpu2} represent the general purpose register files of CPU^1 and CPU^2, respectively. Fig. 5.15 shows the S^{2}QED property

that is to be proven on this model. The property specifies that if two independent CPU instances fetch the same instruction and the register files are consistent with each other before the write-back (the macro *qed_consistent_registers*() specifies this consistency), then the two CPU instances must be QED-consistent also after the write-back, independently of the pipeline context. The CPU^1 instance is constrained to start from a flushed-pipeline state S_t^1 and fetches only NOPs in the time frames for $t > 1$. A flushed-pipeline state S_t^1 is forced on the CPU^1 instance by letting it execute only NOPs for as many time frames before time point t as there are pipeline stages. This results in a significant reduction of proof complexity and excludes any false counterexample to the property that can result from an inconsistent pipeline register. The CPU^2 instance is left unconstrained to start from a symbolic initial state S_t^2 and is allowed to execute an arbitrary sequence of instructions for the time frames $t > 1$. In this computational model, the SAT solver compares the scenario 1, where the *Instruction Under Verification* (IUV) is executed in a flushed-pipeline context, with all scenarios 2 where the IUV is executed in an arbitrary context including the ones where bugs are activated and propagated.

5.5.2 Extending S^{2}QED for Completeness

The S^{2}QED property shown in Fig. 5.15 can detect all functional bugs resulting in a QED-inconsistent state [45]. In other words, the instruction execution is verified to be independent of the program context (or previously fetched and executed instructions). As a result, the property covers multiple-instruction bugs that result in a QED-inconsistent state. However, the method does not prove the absence of logic bugs in a processor core. For example, single-instruction bugs can be masked due to the fact that "common-mode" bugs like a bug in the data path of the ALU have the same effect on both CPU instances and may not lead to a QED-inconsistent state. Further, certain kinds of multiple-instruction bugs such as performance bugs and other bugs that do not result in QED inconsistency are not detected. As a result, the S^{2}QED method can be used to detect a certain class of processor bugs, but S^{2}QED alone cannot guarantee the functional correctness of a processor core.

The S^{2}QED approach is extended such that it detects all logic bugs in a processor including single-instruction bugs. We define the completeness of the approach with respect to the C-IPC-based complete processor verification criterion described in Sec. 5.3. With the extended approach, a complete processor verification can be achieved with substantially less manual effort when compared to traditional C-IPC. This is possible since S^{2}QED "automatically" explores all possible program contexts so that only a much simpler property set is needed to cover all logic bugs in the processor.

For detecting all logic bugs in a processor, a set of S^{2}QED properties are generated such that each property represents the correct execution of instructions of a certain instruction class. For instance, an extended S^{2}QED property is developed for each instruction class such as "register-type", "memory", "control flow". The extended S^{2}QED property is shown in Fig. 5.16 for register-type instructions considering a 5-stage RISC processor. This is denoted by *instr_register_type*() in the assumption part. At time point t_{IF}, the same instruction under verification (IUV) is fetched by both the CPU instances, CPU^1 and CPU^2 (Fig. 5.14). Similar to the original S^{2}QED property we assume that (i) CPU^1 starts from a flushed pipeline state S_t^1, (ii) it fetches only NOPs after the time point t_{IF} and (iii) the previous instruction execution has resulted in a QED-consistent state. The macro *ready_for_next_instruction*() describes the state (cf. Fig. 5.16) of the CPU^1 and CPU^2 pipelines, when they are ready for the next instruction.

assume:
at t_{IF}: *cpu2_fetched_instr*() = *cpu1_fetched_instr*();
at t_{IF}: *cpu1_state*() = S_t^1;
during [t_{IF}+1, t_{WB}]: *cpu1_fetched_instr*() = *NOP*;
at t_{ID}: *ready_for_next_instruction*();
at t_{ID}: *instr_register_type*();
at t_{WB}: *qed_consistent_registers*();
prove:
at t_{EX}: *ready_for_next_instruction*();
at $t_{WB} + 1$: *qed_consistent_registers*();
at $t_{WB} + 1$: *cpu1_reg_value*(*reg_addr* @ t_{ID}) = *expected_value*(*funct_type* @ t_{ID});

Figure 5.16: Extended S^{2}QED property for Register-type instructions (in ITL style)

At time point t_{ID}, both pipelines begin processing the IUV. Due to the pipelined architecture of processors, the next instruction may be decoded already in the next clock cycle when there are no data hazards with the previous instruction. That is, at time point t_{EX} the next instruction is considered for execution. In terms of executing the IUV, the conceptual starting and ending states of the operation are assumed at t_{ID} and t_{EX}, even though the processing of the IUV continues for several more clock cycles (until t_{WB}).

At time point $t_{WB} + 1$, one clock cycle after the results of an instruction execution are committed, the QED consistency (macro *qed_consistent_registers*()) and the expected values of the instruction execution (macro *expected_value*(*funct_type* @ t_{ID})) are checked. The check for QED consistency ensures that any logic bug resulting from dependencies between instructions are detected. This in turn proves that each instruction executes independently of its program context in a bug-free pipeline. The macro *expected_value*(*funct_type* @ t_{ID}) ensures that each instruction of the register-type instruction class executes as described by the ISA. These checks ensure that any logic bug in a processor is found by the new extended S^{2}QED property.

The macro *expected_value* (*funct_type* @ t_{ID}) checks the correctness of results of the instruction execution in the *CPU* ^{1} instance, for the following reason: Since the *CPU* ^{1} instance fetches NOPs before and after the time point t_{IF}, the complex instruction interleaving scenarios such as forwarding or control transfers do not need to be considered. These scenarios are, instead, covered by checking QED consistency between the CPU instances. This leads to a simplification of the property and its generation from an ISA model (cf. Section 6.1). Also, checking QED consistency ensures that the *CPU* ^{2} instance completes with the same, expected results. The property shown detects all logic bugs with respect to the execution of register type instructions. Similarly, an S^{2}QED property is developed for each instruction class of the ISA.

5.5.3 Extending S^{2}QED for Exception Handling Verification

In the context of processor designs, the execution of a program can be interrupted by unexpected events called *exceptions*. We call those exceptions that are triggered by events external to the processor core (e.g., reset power, timer request, etc.) as *asynchronous exceptions*. Asynchronous exceptions are also called *interrupts* and are caused when an external device requests a service from the processor core. *Synchronous exceptions* are those that are caused due to un-

expected events within the processor core. For example, a synchronous exception is triggered when the processor core encounters an undefined[2] instruction word or when an arithmetic overflow occurs in the ALU. In order to resolve an exception event, the current program execution is suspended and a pre-defined exception service routine is executed. After completing the execution of a service routine the suspended program execution is resumed.

In addition to the general purpose register file, processor cores also implement one or more control and status registers (CSRs) to store the status information of the exception events. In addition to exception events, CSRs may be used to store the status of peripheral components. Similar to the general purpose register file, we define the consistency for CSRs. For a processor with N CSRs, a QED-consistent CSR state is characterized by the named logic expression:

$$qed\_consistent\_csrs := \bigwedge_{i=0}^{N-1} \left(CSR^i_{cpu1} = CSR^i_{cpu2} \right) \tag{5.4}$$

This expression is a Boolean predicate that can be captured as a *macro* in the property language of the verification tool. It represents architectural states in which the corresponding CSRs of both CPU instances have identical contents.

When an exception occurs, following actions are performed by the processor core in a pre-defined order specified by the architectural specifications:

- Store the cause of the exception in a dedicated CSR.
- Complete the execution of all or certain active instructions (e.g., instructions in execute, memory and write-back stages) in the pipeline at the time point when the exception occurs.
- Store the return program counter (PC) value in a dedicated CSR, i.e., address of the instruction word from where the program resumes execution after servicing the exception. Additionally, the architectural state of the processor, for example, contents of the general purpose register file are stored on the stack.
- Move the PC to the first instruction address of the exception service routine, which may be also stored in a CSR.

The extended S^{2}QED property shown in Fig. 5.16 for register-type instructions is proven assuming that the execution of IUV is not interrupted by an exception. Therefore, behavior of the processor core when the IUV is interrupted by an exception requires to be proven separately. Due to pipelining, exceptions can occur simultaneously at different pipeline stages. For instance, at an arbitrary time point t, two or more exceptions can be activated by different instructions at different pipeline stages of their execution. For example, at an arbitrary time point t, a divide (*DIV*) instruction in the execute-stage causes a divide-by-zero exception, while the load-word (*LW*) instruction causes a memory access fault[3] exception in the memory-access stage. Consequently, for exception handling verification, the S^{2}QED computation model of Fig. 5.14 is used without modifications.

For verifying the correct implementation of the processor pipelining to execute instructions as specified by the ISA, an S^{2}QED property set is developed (cf. Fig. 5.16). Similarly, for detecting logic bugs associated with the exception handling, a set of S^{2}QED properties are developed such that each property represents the correct response of the processor core for an

[2]Undefined instruction words are also referred to as illegal instructions.

[3]A memory access fault occurs when the processor core attempts to access an invalid or a prohibited memory address.

assume:
- at t_{IF}: *cpu2_fetched_instr*() = *cpu1_fetched_instr*();
- at t_{IF}: *cpu2_pc*() = *cpu1_pc*();
- at t_{IF}: *cpu1_state*() = S_t^1;
- during $[t_{IF}+1, t_{WB}]$: *cpu1_fetched_instr*() = *NOP*;
- at t_{ID}: *ready_for_next_instruction*();
- at t_{EX}: *cpu2_exception_type*() = *cpu1_exception_type*();
- at t_{WB}: *qed_consistent_registers*();
- at t_{WB}: *qed_consistent_csrs*();

prove:
- at t_{EX}: *ready_for_next_instruction*();
- at $t_{WB}+1$: *qed_consistent_registers*();
- at $t_{WB}+1$: *qed_consistent_csrs*();
- at $t_{WB}+1$: *cpu1_rpc_value*() = *cpu1_pc* @ t_{IF};
- at $t_{WB}+1$: *cpu1_epc_value*() = *exception_pc*(*exception_type* @ t_{EX});
- at $t_{WB}+1$: *cpu1_csr_cause*() = *exception_cause*(*exception_type* @ t_{EX});

Figure 5.17: S^{2}QED property for exception events at execute stage (in ITL style)

exception event that occurs in a certain pipeline stage. In other words, an S^{2}QED property is developed for exception events at each pipeline stage such as fetch, decode, execute, memory-access. An S^{2}QED property is shown in Fig. 5.17, which represents the correct response of the processor core for exception events at "execute" stage. At time point t_{IF}, both CPU instances fetch the same instruction that causes an exception at the execute stage and have the same PC value (*cpu2_pc*() and *cpu1_pc*()). The *CPU* ^{1} instance is in a flushed pipeline state S_t^1 at t_{IF} and fetches only NOPs for time points $t > t_{IF}$. At t_{EX}, it is assumed that both CPU instances cause the same type of exception that are represented by the macros *cpu2_exception_type*() and *cpu1_exception_type*(). Similar to the QED-consistency of register files, it is also assumed that the previous instructions resulted in a QED-consistent CSR state (*qed_consistent_csrs*()) at t_{WB}.

At time point t_{WB}, the IUV that caused the exception at execute stage reaches write-back stage and the CSRs are updated with status information. At time point t_{WB+1}, one clock cycle after the CSRs have been updated, the QED consistency of register files and CSRs, and the expected values of the exception event are checked. The check for the QED consistency ensures that any logic bug associated with the sequence of instructions is detected. The checks for the expected values of the exception event (macros *cpu1_rpc_value*(), *cpu1_epc_value*() and *cpu1_csr_cause*()) ensures that the processor implements the exception handling as specified by the architectural specification. *cpu1_rpc_value*() captures the CSR value that stores the return PC value and *cpu1_epc_value*() captures the CSR value that stores the exception PC of the current exception event. Similarly, *cpu1_csr_cause*() captures the CSR that stores the *exception id* of the current exception. The combined checks for QED consistency and expected values ensure that all logic bugs associated with the exception handling are detected by a set of S^{2}QED properties.

5.5.4 Extending S^{2}QED for Superscalar Processor Verification

In the previous two sections, we outlined the extended S^{2}QED property for simple pipelined processors targeting both instruction and exception behaviors. A set of S^{2}QED properties com-

pletely verify a processor implementation targeting both single- and multiple-instruction bugs. As described in Section. 5.2, only intra-pipeline multiple-instruction bugs can occur in simple processor implementations. Inter-pipeline multiple-instruction logic bugs occur in processors with multiple pipelines. In the following, an extended S^{2}QED property for superscalar processors is described.

Superscalar processors implement multiple pipelines, and, as a result, execute more than one instruction per clock cycle. Each pipeline in a superscalar processor is optimized to execute a specific class of instructions. During the execution of a program, the instruction stream is pre-decoded and the instructions are assigned/issued to specific pipelines according to their instruction encoding. At an arbitrary time point t, each pipeline is issued an appropriate instruction and the instructions are executed in parallel by different pipelines [105, 21]. Due to the concurrent nature of instruction execution across pipelines, various scenarios such as data hazards (e.g., RAW, WAW), resource conflicts, branches and exceptions become relevant. Depending on the microarchitectural specifications, each pipeline updates the register file or other target architectural states (e.g., PC, CSR) in a specified order. As a result, in addition to logic bugs within the same pipeline, the bugs can also occur due to the incorrect logic that controls the dependency between different pipelines. We refer to the latter type of bugs as inter-pipeline multiple-instruction bugs (cf. Section 5.2).

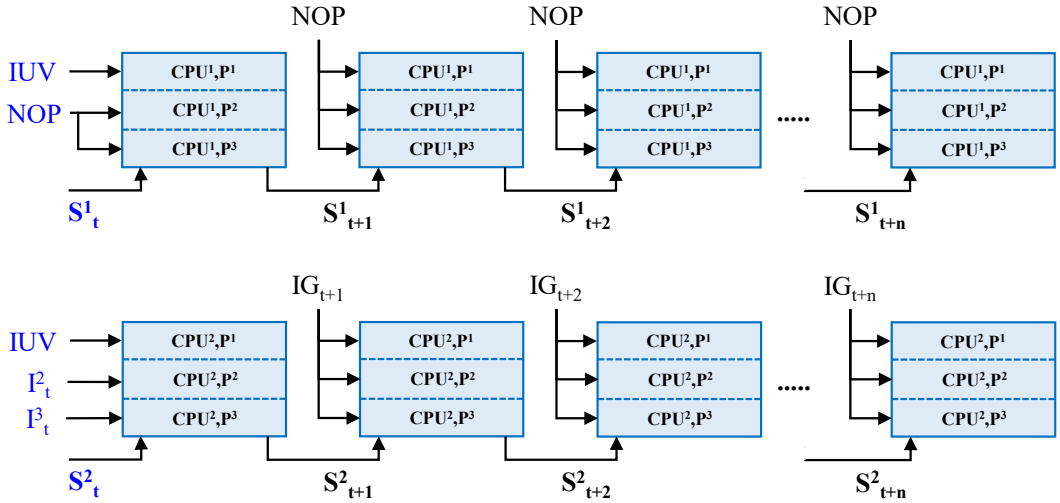


Figure 5.18: S^{2}QED Computational model for superscalar processors

The basic idea of the computation model of S^{2}QED shown in Fig. 5.14 is also applicable to superscalar processors. Fig. 5.18 shows the computation model of S^{2}QED with modifications required for superscalar processors. We consider a superscalar processor with three pipelines P^1, P^2 and P^3 for illustrative purposes. However, the method presented here is applicable to superscalar processors with arbitrary number of pipelines. Two identical instances of the same superscalar processor are unrolled for a time window as large as the upper bound of the execution time of an instruction in the pipelines. Let I_t^1, I_t^2 and I_t^3 be the instructions issued at an arbitrary time point t to the pipelines P_1, P_2 and P_3, respectively. An instruction group IG_t is the combination of instructions issued to different pipelines at time point t, i.e., $IG_t = (I_t^1 \& I_t^2 \& I_t^3)$, where '&' is the concatenation operator. Typically, the superscalar processors include two types of general-purpose register files, data register file and address register file. While the data register file is used to store the results of arithmetic operations, the address register file is used to

store the addresses of the load-store operations. Let DR_{cpu1} and DR_{cpu2} be the data register files of CPU^1 and CPU^2 instances, respectively. Similarly, AR_{cpu1} and AR_{cpu2} be the address register files of CPU^1 and CPU^2 instances, respectively. For a processor with N data registers and M address registers, the QED-consistent register states are characterized by the logic expressions Eq. 5.5 and Eq. 5.6, respectively.

$$qed\_consistent\_data\_registers := \bigwedge_{i=0}^{N-1} \left(DR^i_{cpu1} = DR^i_{cpu2}\right) \tag{5.5}$$

$$qed\_consistent\_address\_registers := \bigwedge_{i=0}^{M-1} \left(AR^i_{cpu1} = AR^i_{cpu2}\right) \tag{5.6}$$

For detecting all logic bugs in a superscalar processor, a set of S^{2}QED properties are developed such that each property captures the correct execution of instructions of a specific instruction class. In case of superscalar processors, a set of instructions belonging to a certain instruction class are always executed by a specific pipeline. For example, data-register arithmetic type instructions are always executed by pipeline P^1, load-store instructions are executed by pipeline P_2 and branch instructions are executed by pipeline P^3. The S^{2}QED property for data-register arithmetic type instruction class is shown in Fig. 5.19.

assume:
- at t_{IF}: *cpu2_p1_fetched_instr*() = *cpu1_p1_fetched_instr*();
- at t_{IF}: *cpu1_p2_instr*() = *NOP*;
- at t_{IF}: *cpu1_p3_instr*() = *NOP*;
- at t_{IF}: *cpu1_state*() = S^1_t;
- during [t_{IF}+1, t_{WB}]: *cpu1_ig*() = *IG_NOP*;
- at t_{ID}: *ready_for_next_instruction*();
- at t_{ID}: *instr_data_register_arith_type*();
- at t_{WB}: *qed_consistent_data_registers*();
- at t_{WB}: *qed_consistent_address_registers*();

prove:
- at t_{EX}: *ready_for_next_instruction*();
- at $t_{WB}+1$: *qed_consistent_data_registers*();
- at $t_{WB}+1$: *qed_consistent_address_registers*();
- at $t_{WB}+1$: *cpu1_dreg_value*(*reg_addr*@t_{ID}) = *expected_value*(*funct_type*@t_{ID});

Figure 5.19: S^{2}QED property for data-register arithmetic type instructions (in ITL style)

Assume clauses: At time point t_{IF}, both CPU instances fetch the same IUV on pipeline P^1. In a buggy pipeline implementation, the IUV is the instruction that propagates the logic bug into an observable architectural state (e.g., data register). At t_{IF}, when both CPU instances fetch the same instruction group, bugs that result from the incorrect handling of dependency between different pipelines are masked. That is, inter-pipeline multiple-instruction bugs that result from the dependency between instructions of the same instruction group will be masked. Therefore, CPU^1 fetches NOPs on pipelines P^2 and P^3, and CPU^2 fetches arbitrary instructions I^2_t and I^3_t on pipelines P^2 and P^3, respectively. At t_{IF}, CPU^1 instance is

in a flushed pipeline state S_t^1, i.e., there are no active instructions in any of the pipelines[4]. CPU^2 starts from a symbolic state S_t^2, which indicates that prior to the time point t_{IF}, CPU^2 instance executes an instruction sequence that activates a logic bug. Between time points t_{IF} and t_{WB}, CPU^1 fetches only NOPs on all pipelines (*IG_NOP*). For time points $t > t_{IF}$, CPU^2 instance fetches arbitrary instructions ($IG_{t+1}, IG_{t+2}, \cdots$). Further, the previous instruction is assumed to have executed resulting in a QED-consistent state (*qed_consistent_data_registers*() and *qed_consistent_address_registers*()). The macro *instr_data_register_arith_type*() indicates that the IUV belongs to the data-register arithmetic type instruction class.

Prove clauses: The macro *ready_for_next_instruction*() defines the state of the CPU^1 and CPU^2 pipelines, when they are ready for the next instruction. Due to pipelining of processors, the next instruction is considered in the next clock cycle if there are no data hazards scenarios. That is, at time point t_{EX} next instruction is considered for execution (macro *ready_for_next_instruction*()). At time point $t_{WB}+1$, one clock cycle after the results of an instruction execution have been committed, the QED consistency for both data and address registers are checked (*qed_consistent_data_registers*() and *qed_consistent_address_registers*()). Multiple-instruction logic bugs are detected by the check for QED-consistency. The check for the expected values of the instruction execution (*expected_value*(*funct_type* @ t_{ID})) ensures that each instruction of the data-register arithmetic type instruction class executes as described by the ISA and all single-instruction logic bugs are caught. These checks ensure that any logic bug in a processor that is associated with the execution of data-register arithmetic type instructions is found by the S^{2}QED property.

Extensions for accommodating WAW Hazard or OoO Write-Back

In superscalar processors, write-after-write (WAW) hazards and out-of-order (OoO) write-backs are a common occurrence. The WAW hazard is a scenario in which the results of the later instruction (e.g., I_{t+1} issued at $t+1$) are committed before the results of the earlier instruction (e.g., I_t issued at t) such that both instructions update the same target register. Similarly, in case of out-of-order execution, it is possible that an instruction commits its results before the completion of its preceding instructions. To accommodate these scenarios, the S^{2}QED property shown in Fig. 5.20 extends the property shown in Fig. 5.19.

> **Example** In the computation model shown in Fig. 5.18, let us assume that the instruction issued to pipeline P^1 is the oldest instruction, followed by the instruction on P^2 and the instruction on P^3 which is the latest instruction. At time point t_{IF}, the pipelines P^1 and P^2 of CPU^2 instance fetch instructions $I^1_{t_{IF}}$ and $I^2_{t_{IF}}$ such that they have the same destination register ($rd\_p1 = rd\_p2$). Since the instruction on pipeline P^2 succeeds the instruction on P^1, the results of the instruction on pipeline P^1 are inhibited from being written to the register file at t_{WB}. Instead, the results of the instruction on P^2 are committed to the register file at $I^1_{t_{WB}}$. A WAW hazard does not occur on the CPU^1 instance as it fetches a NOP on P^2 at t_{IF}. In this scenario, the property shown in Fig. 5.19 fails since the results of the IUV execution are not committed on the CPU^2 instance, but the results of the IUV are committed to the register file on CPU^1 instance.

[4]Note that CPU^1 instance is free to execute any sequence of instructions before reaching a flushed pipeline state at t_{IF}

assume:
- at t_{IF}: *cpu2_p1_fetched_instr*() = *cpu1_p1_fetched_instr*();
- at t_{IF}: *cpu1_p2_instr*() = *NOP*;
- at t_{IF}: *cpu1_p3_instr*() = *NOP*;
- at t_{IF}: *cpu1_state*() = S_t^1;
- during [t_{IF}+1, t_{WB}]: *cpu1_ig*() = *IG_NOP*;
- at t_{ID}: *ready_for_next_instruction*();
- at t_{ID}: *instr_data_register_arith_type*();
- at t_{WB}: *qed_consistent_data_registers*();
- at t_{WB}: *qed_consistent_address_registers*();

prove:
- at t_{EX}: *ready_for_next_instruction*();
- at t_{WB}: if (*waw_valid* or *ooo_valid*) then
  (*cpu1_p1_dreg_wr_data* = *cpu2_p1_dreg_wr_data*) and
  (*cpu1_p1_dreg_wr_data* = *expected_value*(*funct_type*@t_{ID}))
  end if;
- at $t_{WB}+1$: if (not(*waw_valid* or *ooo_valid*) then
  qed_consistent_data_registers() and
  qed_consistent_address_registers() and
  (*cpu1_dreg_value*(*reg_addr*@t_{ID}) = *expected_value*(*funct_type*@t_{ID}))
  end if;

Figure 5.20: S^{2}QED property for data-register arithmetic type instructions considering *Write-after-Write* and *Out-of-Order* scenarios (in ITL style)

The S^{2}QED property extended for accommodating WAW hazard and OoO write-back scenarios is shown in Fig. 5.20. In particular, the "prove" part of the property needs to be modified. In scenarios where WAW hazards and OoO write-backs are applicable in the processor implementation (*waw_valid* and *ooo_valid*), the consistency of the write data on both CPU instances is checked at time point t_{WB} (*cpu1_p1_dreg_wr_data* and *cpu2_p1_dreg_wr_data*). In addition, the expected value of the instruction execution is checked on the *CPU* [1] instance (*expected_value*(*funct_type*@t_{ID})). When WAW hazards and OoO write-backs are not applicable, the QED consistency for both data and address registers are checked at time point $t_{WB}+1$, similar to the property shown in Fig. 5.19.

5.5.5 Completeness Analysis for a set of S^{2}QED properties

Typically, ISAs describe a set of instruction classes with each instruction class consisting of several instructions. In the previous section, it is shown how an extended S^{2}QED property can be used to ensure the correct implementation of every instruction belonging to a certain instruction class. A set of extended S^{2}QED properties cover the complete behavior of a processor implementation. The completeness of the approach according to the completeness criterion described in Sec. 2.5.1 is discussed in the following.

An operational property P is a property written in the form of an implication $A \implies C$, where the antecedent A is a set of *assumptions* and the consequent C is a set of *commitments*. An assumption or a commitment is an LTL formula, where the only temporal operator allowed is X. Such a formula can be mapped to a finite unrolling of a transition structure (cf. the

computational model for S^{2}QED, Fig. 5.14). In the following, we use the notation $next(A,l)$ to denote a "temporal shift" of the formula A by l clock cycles, i.e., $next(A,l) := X^{\,l}A$. This adds a temporal offset l to all time points referred to in the formula A.

Case-split test

Let a set of important states be given by the commitments $\{C_{P_1}, C_{P_2}, C_{P_3}, \ldots\}$ of the properties $\{P_1, P_2, P_3, \ldots\}$ for an arbitrary design. Then, for every important state reached in an operation or property P_1, it is checked whether the disjunction of the assumptions $\{A_{Q_1}, A_{Q_2}, A_{Q_3}, \ldots\}$ of all successor properties $\{Q_1, Q_2, Q_3, \ldots\}$ completely covers the commitment C_{P_1}, i.e., for every path starting in a substate of the important state C_{P_1} there exists an operation property Q_1 whose assumption A_{Q_1} describes the path. Fig. 5.21 shows a path in which the assumption A_{Q_j} of the property Q_j (successor) covers the commitment C_{P_i} of the property P_i (predecessor). Let $\{A_{Q_1}, A_{Q_2}, \ldots\}$ be the set of assumptions of the successor properties, then the case-split test checks whether

$$C_{P_1} \rightarrow next((A_{Q_1} \vee A_{Q_2} \vee A_{Q_3} \vee ....),\ l_{P_1}) \tag{5.7}$$

where l_{P_1} is the length of the property P_1, i.e., the number of clock cycles between the starting and the ending state of property P_1 (cf. Fig. 5.21). The *next* operator aligns the starting (important) state of property Q_1 (or Q_2 or $Q_3 \ldots$) with the ending state of property P_1.

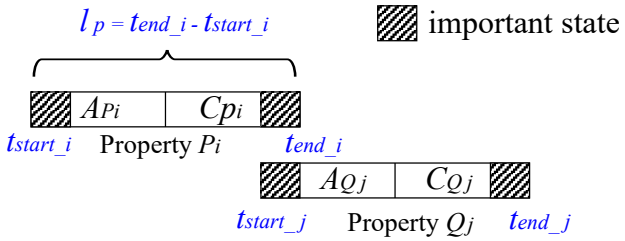


Figure 5.21: Interleaving of Property P_i and Q_j for case-split test

In a processor pipeline, all operations begin and end in the same state as depicted in Fig. 5.7. This is mainly due to the nature of the pipeline which, in principle, considers a new instruction at every clock cycle. With a set of extended S^{2}QED properties for a processor, instructions are grouped into properties according to the instruction classes, i.e., a total of n properties to consider if the processor core supports n instruction classes. Let us assume that the only important state in a processor execution is given by the commitments $\{C_{P_1}, C_{P_2}, C_{P_3}, \ldots\}$ of the S^{2}QED properties $\{P_1, P_2, P_3, \ldots\}$, where P_1 is an S^{2}QED property for a specific instruction class (e.g., register-type). For the only important state reached during an instruction execution, it is checked whether the disjunction of the assumptions $\{A_{Q_1}, A_{Q_2}, A_{Q_3}, \ldots\}$ of all the successor properties $\{Q_1, Q_2, Q_3, \ldots\}$ completely covers the commitment C_{P_1}. Let $\{A^2_{Q_1}, A^2_{Q_2}, A^2_{Q_3}, \cdots\}$ and $\{A^1_{Q_1}, A^1_{Q_2}, A^1_{Q_3}, \cdots\}$ be the assumptions of the successor properties on the $CPU^{\,2}$ and $CPU^{\,1}$ instances, respectively. In order to prove that all successor properties $\{Q_1, Q_2, Q_3, \ldots\}$ completely cover the commitment C_{P_1} of property P_1, it is necessary to prove:

$$C_{P_1} \rightarrow next(\ (\ (A^1_{Q_1} \wedge A^2_{Q_1}) \vee (A^1_{Q_2} \wedge A^2_{Q_2}) \vee (A^1_{Q_3} \wedge A^2_{Q_3}) \vee \cdots),\ l_{P_1}) \tag{5.8}$$

Assuming that the S^{2}QED property set holds on the computational model (cf. Fig. 5.14), it is proven that the result of every operation is consistent between the two CPU instances,

regardless of the predecessor operations. Therefore, it is sufficient to prove the case-split test on one of the CPU instances. In other words, in order to prove the case-split test, every possible instruction sequence can be considered on the CPU^2 instance. The case-split test for S^{2}QED is reduced to:

$$C_{P_1} \rightarrow next((A^2_{Q_1} \vee A^2_{Q_2} \vee A^2_{Q_3} \vee \cdots),\ l_{P_1}) \tag{5.9}$$

When the case split test passes, it ensures that for every possible instruction sequence of the processor there exists a chain of properties that is executed. If this test passes it means that every (cycle-accurate) execution trace of the processor can be partitioned into finite non-overlapping segments of behavior such that each segment is described by a property of the property set. In other words, every execution trace is "covered" by a sequence of operation properties.

The case-split test is easy to satisfy for a set of S^{2}QED properties. For a processor that implements n instruction classes, we have n S^{2}QED properties. By ensuring the correctness of the conditions that capture the opcode of the instruction classes (e.g., *instr_register_type*() in Fig. 5.16), the case-split can be easily proven for a set of S^{2}QED properties.

Successor Test

The successor test for a general property suite checks for every predecessor/successor pair (P_i, Q_j) of properties whether the assumption A_{Q_j} of property Q_j depends solely on inputs and on signals determined by the predecessor property P_i. This is checked in a SAT instance created in the following way: The set of signals mentioned in the properties P_i and Q_j are duplicated. The first set of signals is used to describe executions of an operation P_i followed by operation Q_j and the second set describes operation P_i followed by an operation not being Q_j, while both sets receive the same input. Let A'_{P_i}, C'_{P_i} and A'_{Q_j} be the assumption and commitment of property P_i and the assumption of property Q_j, respectively, expressed in the copied signals. Further, let D be the set of determination requirements specifying that for every signal mentioned in the determination requirement, the value is the same between the two sets. The successor test checks the following implication on the SAT instance (with the same input values in each time frame):

$$A_{P_i} \wedge C_{P_i} \wedge A'_{P_i} \wedge C'_{P_i} \wedge D \wedge next(A_{Q_j}) \rightarrow next(A'_{Q_j}) \tag{5.10}$$

If this implication does not hold, then there exists an input sequence such that operation P_i is executed and the assumption of property Q_j may hold or may not hold, depending on the other signals mentioned in the properties. This is the case if the assumption A_{Q_j} was written such that it depends on some undetermined variable, i.e., a variable that is not marked or computed as "determined" by P_i. For the case of processor verification, such a situation is hard to create, especially if the properties are generated automatically. All properties of an instruction class share the same set of assumptions, except for the assumption on the fetched instruction opcode that defines the operation performed by the instruction. Each property P_i must *determine* the architectural state that is evaluated by the assumption of successor property Q_j. The successor test can only fail if that is not the case, i.e., the predecessor property P_i does not fully describe the architectural state that is produced by the instruction specified in P_i. This implies that the successive behavior is ambiguous.

Determination Test

The determination tests are performed to check if a property set uniquely determines the values of all important states and output signals of a design at all time points. The determination test checks whether each property Q_j fulfills its determination requirements provided the predecessor operation P_i, in turn, fulfilled its determination requirements. The determination requirements are specifications of signals that describe which signals must be determined and at what time points (cf. Section 2.5). The determination test creates a SAT instance that is satisfied if a determination requirement is violated, i.e., if a signal or an important state required to be determined by the property Q_j is actually not a function of the variables determined by P_i and/or of inputs during the operation Q_j. Similar to the successor test, the set of signals mentioned in the properties P_i and Q_j is duplicated in order to describe two executions. In both executions, Q_j is followed by P_i and the same input sequences are applied. Again, also, the state variables that are assumed to be determined are given the same values in both executions in the time points specified by the guards of the determination requirements. Let D_{P_i} and D_{Q_j} be the determination requirements of property P_i and Q_j, respectively. The determination test checks the following implication on the SAT instance (with the same input sequences applied in both executions):

$$A_{P_i} \wedge C_{P_i} \wedge A'_{P_i} \wedge C'_{P_i} \wedge D_{P_i} \rightarrow next(D_{Q_j}, l_{P_i}) \tag{5.11}$$

If this implication does not hold, then there exists an instruction sequence and sequences of other signals mentioned in the properties such that Q is executed after P but signals that are supposed to be determined may have different values in different executions. An S^{2}QED property for a specific instruction class captures the unique values for all output signals on the CPU^1 instance through *expected_value(funct_type @ t_{ID})* (see Fig. 5.16). Further, since S^{2}QED properties are interleaved such that a new operation may occur in the pipeline at every clock cycle, the output signals and the architectural states are determined at every clock cycle. The QED consistency check (*qed_consistent_registers*()) on the CPU^1 and CPU^2 instances ensures that the output values of CPU^2 instance are consistent with the outputs of the CPU^1 instance.

Reset Test

Case-split, successor and determination tests described above form an inductive proof in which a set of operational properties P uniquely determine the states in which the design traverses and the sequence of output signals values. The starting state of this inductive proof is the state from which the design starts executing instructions. The reset test checks whether the reset can be applied deterministically and whether the reset state satisfies the determination requirements. In other words, after the reset, a processor comes to a state from which it starts executing instructions and at this time point all important states and output signals of the processor are determined.

In addition to a set of extended S^{2}QED properties set, a reset property is created which checks whether the reset brings the processor to an important state and all the output signals have specified values. After the reset, the pipelines are empty in a processor as the first instruction is yet to be fetched. This means that the decode stage is empty which implies that no instruction can start executing right after the reset. A clock cycle delay is added to the reset operation such that the release of reset brings the processor to an important state (cf. Fig. 5.7).

5.5.6 Complete - S^{2}QED

A set of extended S^{2}QED properties fulfilling the completeness criterion is referred to as *Complete-S^{2}QED* (C-S^{2}QED). Based on the successor test, determination test, reset test and S^{2}QED-adapted case-split test, following theorem is formulated.

Theorem 1 [C-S^{2}QED detects all logic bugs in a processor]:
Given a set of S^{2}QED properties $\mathbf{V} = \{P_1, P_2, \ldots, P_n\}$ in which P_i is the property for a specific instruction class, created for a given ISA, as described in Sec. 5.5.2. If the property set fulfills the completeness criterion for S^{2}QED, then it detects all logic bugs in the ISA implementation of a processor core.

Proof. Assume that there is a logic bug in the processor. The logic bug can be a single-instruction bug or a multiple-instruction bug.

- If the bug is a *single-instruction bug*, it means that there is an erroneous instruction that produces a wrong result in any program context. The wrong result is observable even if it is executed in a flushed pipeline. Because the property set passes the case-split test and the successor test, there exists an S^{2}QED property targeting the instruction. Since the property set passes the determination test, there exists a sub-predicate *expected_value*(*funct_type*) which is called by the commitment of the property and which verifies the computation result of the instruction. According to this sub-predicate, the property fails for the expected architectural state in the CPU^1 instance.
- If the bug is a *multiple-instruction bug*, two cases have to be distinguished: (1) The bug is activated in a flushed-pipeline context. Then, the bug will be detected in the same way as described above for the single-instruction bug. (2) The bug is not activated in a flushed-pipeline context. Then, there exists a specific program context that activates the bug. Because the property set passes the case-split test and the successor test, there exists a unique sequence of operation properties covering the program context, i.e., the sequence of instructions in the pipeline. Hence, there exists an S^{2}QED property that covers the program context in the CPU^2 instance of the model and has the instruction exposing the bug as the IUV. Because the property set passes the determination test, the sub-predicate *qed_consistent_registers*() determines the full architectural state including the state variables exposing the bug. Since the bug is not activated in the CPU^1 instance, it is detected as an inconsistency between CPU^1 and CPU^2 architectural states.

□

5.6 Related work: Processor Verification

Functional verification of processors has gathered a lot of interest both in academia and industry. This high-level of interest started with the discovery of a functional bug (famously known as *FDIV* bug) in the early Intel *Pentium* processors [90]. Thereafter, the processor verification has been well researched employing different verification techniques. Although simulation remains the primary verification technique employed in the industry, simulation-based methods cannot guarantee an exhaustive verification of processor implementations. Due to this shortcoming of simulation-based methods, formal methods have been accepted as the preferred techniques for processor verification.

Several formal verification approaches have been proposed to verify processor cores. A summary of the most related works [18, 59, 24, 10, 15, 66, 91, 7] is provided in the following:

Burch and Jones et.al proposed an approach for verifying control logic of pipelined processors and showed that the formal methods are scalable to industrial microprocessors [18, 59]. The approach abstracts the datapath logic to reduce the complexity for formal analysis. The main idea of the approach is to compare the pipelined implementation to an architectural description by using an efficient validity checker for a logic of uninterpreted functions with equality. The uninterpreted functions are used to represent arithmetic and logic operations, without detailing the functionality. This approach of abstracting the datapath logic to reduce the complexity inspired other works [24, 10], which used symbolic model checkers to compare the processor implementations to the architectural description. These approaches showed for the first time that the formal verification is applicable on industrial processor designs. However, the proposed methods targeted only a specific class logic bugs in the processor and did not offer a complete processor verification strategy. Moreover, developing the architectural description of the processor implementation and the refinement mapping between implementation and specification requires significant manual effort.

In [91], a formal processor verification approach is presented which employs generation of properties from an executable ISA specification. This approach uses bounded model checking as its proof method and compares the results of instruction execution in the processor implementation with the results obtained from executable specifications by considering the same instruction sequence. Being based on bounded proofs and not covering all corner cases of instruction execution, this work contributes to efficient bug hunting but cannot prove the absence of processor bugs.

The works proposed in [15, 66, 7] follow the complete processor verification strategy outlined in Section 5.3. Bormann et.al used an industrial microcontroller to demonstrate the processor verification with a complete set of properties developed following the C-IPC methodology [16]. The approach used C-IPC principles to determine the gaps in the property set, thus ensuring a high design quality. Developing a complete set of properties and passing the completeness checks, however, requires a large amount of manual work and high expertise.

In [66], formal verification of processors with a complete set of properties is proposed, in which the properties are generated from an architectural description of the processor. Similarly in [7], the authors generated a complete set of properties for processor verification from semi-formal specifications. The generation of properties is fixed to the specific microarchitecture and as a result, requires substantial rework when migrating to a new processor architecture. and also, when the processor is extended to support additional ISA extensions.

Chapter 6

Evaluation on Real Designs

Chapter 4 proposes modeling of design specifications in formal models and subsequent property generation hereof following the principles of model-driven architecture. Chapter 5 proposes a complete processor verification method called C-S^{2}QED. It requires low implementation effort and detects all logic bugs in processor core implementations. For efficient verification of processor cores, C-S^{2}QED properties are generated from a formal ISA model following the model-driven property generation flow. It can be concluded that for efficient formal verification of hardware designs, a combination of property generation coupled with a suitable verification strategy that fits the design under verification (DUV) is required.

In this chapter, different formal strategies applied on various real-life designs are elaborated. In Section 6.1, formal verification of a RISC-V processor core is considered. Here, the C-S^{2}QED method has been employed with generated properties to further improve the verification productivity. In Section 6.2, formal verification of several peripheral designs is considered. Specifications of peripheral designs such as AHB-to-APB bridge and programmable interrupt controller that are sequential in nature are modeled in a state-transition-like formalism. The details of specification modeling in this formalism and subsequent property generation are elaborated in Sections 6.2.1 and 6.2.2. Further, in Section 6.2.3, generation of properties for a bus matrix design which is combinatorial in nature is considered and discussed in detail. A summary of the experimental results in Section 6.2.4 concludes the chapter.

6.1 Formal Verification of a RISC-V processor core

For demonstrating the effectiveness and applicability of our property generation flow and the C-S^{2}QED method to real-life designs, the proposed approaches are applied to verify RISC-V [117]-based processor core variants. RISC-V is an open source instruction set architecture (ISA) that targets both embedded 32-bit devices and larger 64-bit or 128-bit devices. The RISC-V ISA is split into a user-level ISA and a privileged ISA. The privileged ISA specifies instructions and registers that are relevant when creating a system with an operating system. The user-level ISA is divided into a base ISA and several standardized extensions. The base ISA is denoted as *RV32I*. With addition of extensions, the acronym string is appended with specific characters denoting the corresponding extension. For example, *RV32IC* refers to the base ISA extended with the compressed instruction set. In the following, generation of C-S^{2}QED properties for user-level RISC-V processor core variants is summarized and experimental results are discussed.

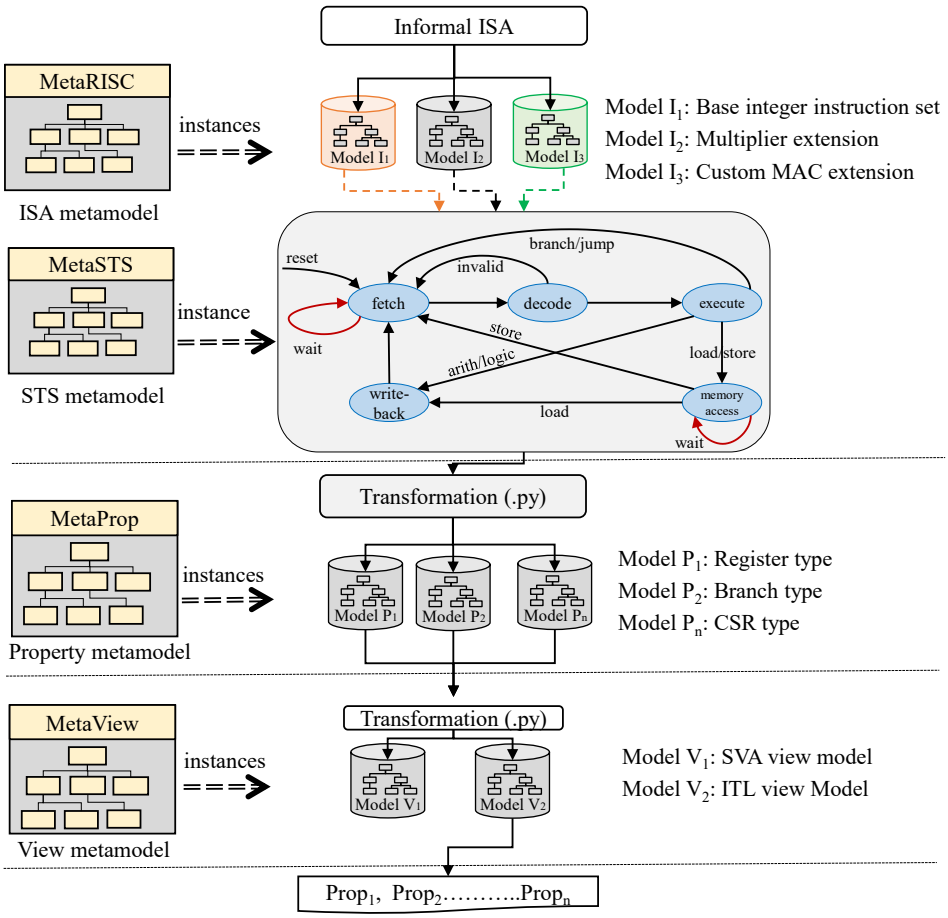


Figure 6.1: Property generation flow for processor cores

The property generation flow for processor cores starts with formalizing an informal ISA in untimed formal models as shown in Fig. 6.1. An untimed model of an ISA is a model instance of the *MetaRISC* metamodel previously introduced in Chapter 4 (cf. Fig. 4.6.2). In Fig. 6.1, *Model* I_1, *Model* I_2 and *Model* I_3 are model instances of the *MetaRISC* metamodel and represent different instruction set extensions from RISC-V ISA. In order to enable architectural alternatives, these model instances are microarchitecture-independent and capture the details of an ISA such as instruction encoding, instruction behavior and target objects of each instruction with clear semantics. For RTL development, the RTL generation flow introduced in Section 3.4 consumes these models and defines microarchitecture blueprints based on high-level configuration parameters.

Due to pipelined implementation of processor cores, the information available in model instances *Model* I_1, *Model* I_2 and *Model* I_3 are not sufficient for property generation. This is because these model instances do not include timing information of instruction execution in the pipeline, which is needed to prove the functional correctness of the processor pipeline implementation. Therefore, model instances of *MetaRISC* metamodel are extended with timing information by modeling the instruction execution in state transition graphs (STGs) or traces as elaborated in Section 4.6.2. A metamodel definition called *MetaSTS* is utilized to facilitate the creation and extension of STGs as shown in Fig. 6.1. A set of STGs are created such that each STG represents the behavior of all instructions belonging to a certain instruction class (e.g.,

load, store, branch, etc.). A code snippet for creating STG definitions is shown in Fig. 4.17.

After modeling the specifications of a processor implementation in formal models, the next step is to define property models. The main task of this layer is to transform the specification models to property models (cf. Section 4.2.2). The transformation script, which is written in Python, performs two major tasks. First, for each path in the state transition graph in the specification model, it creates a base property model. Next, the property model is appended with clauses required to generate a C-S^{2}QED property set. The definition of property models is facilitated by the APIs provided by the automation framework for the *MetaProp* metamodel. The transformation in the bottom layer, metamodel *MetaView* in Fig. 6.1, maps each property model to a corresponding view model and generates the property in the specified target language, such as SVA or ITL (cf. Section 4.2).

```
assume:
        at t_IF:                  cpu1_fetch_valid();
        at t_IF:                  cpu2_fetched_instr() = cpu1_fetched_instr();
        at t_IF:                  cpu1_state() = S_t^1;
        during [t_IF+1, t_WB]:    cpu1_fetched_instr() = NOP;
        at t_ID:                  instr_load_type();
        at t_ID:                  ready_for_next_instruction();
        at t_WB:                  qed_consistent_registers();
prove:
        at t_EX:                  ready_for_next_instruction();
        at t_EX:                  cpu1_mem_rd_addr = expected_mem_rd_addr();
        at t_EX:                  cpu1_mem_rd_en();
        at t_EX:                  cpu1_mem_rd_access_size() =
                                  expected_mem_rd_access_size( funct_type @ t_ID);
        at t_WB + 1:              cpu1_reg_value( reg_addr @ t_ID) =
                                  expected_mem_rd_value( funct_type @ t_ID);
        at t_WB + 1:              qed_consistent_registers();
```

Figure 6.2: Generated C-S^{2}QED property for Load-type instructions (in ITL style)

In the context of generating properties following the C-S^{2}QED method, the STGs are mapped to the CPU^1 instance and additional clauses are generated targeting both CPU^1 and CPU^2 instances. A C-S^{2}QED property generated for load-type instructions is shown in Fig. 6.2. At time point t_{IF}, an assumption is made that a new instruction is fetched by the fetch unit (clause *cpu1_fetch_valid*()). Further at time point t_{ID}, clauses *ready_for_next_instruction*() and *instr_load_type*() assume that the instruction word is valid and belongs to the load-type instruction class. The clauses shown in green color represent the assumptions needed for the C-S^{2}QED approach (cf. Section 5.5).

In the *prove* section, the clause *ready_for_next_instruction*() at time point t_{EX} specifies that the next instruction is decoded in the pipeline. This is because, as the instruction under verification (IUV) moves from the decode stage to the execute stage, the instruction fetched after the IUV has to move from the fetch stage to the decode stage. At time point t_{EX}, clauses *expected_mem_rd_addr*(), *cpu1_mem_rd_en*() and *expected_mem_rd_access_size*(*funct_type* @ t_{ID}) capture the expected values of data memory interface signals. These clauses are generated

from the instruction behavior that is captured in the model instances of the *MetaRISC* meta-model (e.g., $Model_1$). At time point $t_{WB}+1$, the clause *expected_mem_rd_value*(*funct_type* @ t_{ID}) captures the read value from data memory when it is expected to be visible in the register file. Additionally at $t_{WB}+1$, both CPU instances are expected to be consistent. This is captured by the clause *qed_consistent_registers*().

In a traditional IPC-based processor verification approach without C-S^{2}QED the clauses shown in green color are not needed. Then, however, when compared to C-S^{2}QED, the transformation script in the intermediate layer is more complex and requires more manual development effort. The reason is that the macro generation needs to consider different special scenarios such as forwarding, stalling, exception, etc., so that more microarchitectural details need to be considered in the transformation script. This requires deep understanding of the pipeline implementation and prior anticipation of instruction sequences that would lead to an error scenario (cf. Section 5.2). In contrast, in the C-S^{2}QED approach, the clauses or macros are generated targeting the *CPU* [1] instance which is in a flushed pipeline at time point t_{IF} and fetches only NOPs after time point t_{IF}. As a result, the macros that need to be generated do not need to consider microarchitecture details (e.g., forwarding logic). This simplifies the transformation step and reduces the effort required in developing the transformation code to create property models.

6.1.1 Experimental Results

A set of C-S^{2}QED properties are generated for verifying RISC-V processor core variants. The RTL generation flow introduced in Section 3.4 is used to generate various architectural alternatives of the processor core. The generation flow for RISC-V cores is built in a highly configurable manner such that different standardized ISA extensions, number of pipeline stages, memory interface protocols and support for exception handling are selected from high-level configuration parameters. The RISC-V core implements a Harvard architecture with separate program and data memory interfaces. The processor-memory interface can be configured to implement an Advanced High-performance Bus (AHB) protocol or a simple bus protocol. The core supports different standard extensions in addition to the base ISA and supports nested exception handling of both synchronous and asynchronous exceptions.

Table 6.1: Bug Detection Results

	SIC	**S^{2}QED**	**C-S^{2}QED**
Finds single-instruction bugs	yes	no	yes
Finds multiple-instruction bugs	yes	yes	yes
Effort for base instr. set (person days)	10+6	-	10+2
Runtime (with bugs)	< 30 s	< 60 s	< 30 s
Runtime (without bugs)	27 min	6 min	18 min
CEX length ([min, max] instructions)	[1, 5]	[2, 5]	[1, 5]

The processor core has been previously verified with a complete set of properties generated using the property generation flow described in Sec. 4.2. However, this verification approach shown as *Spec Implemented Correctly* (SIC) in Tab. 6.1, is based on conventional C-IPC [15,

113]. 18 logic bugs comprising both single-instruction bugs and multiple-instruction bugs were found during verification. All detected logic bugs have been injected to the RTL for verification with the C-S^{2}QED approach. For property checking, the commercial formal verification tool OneSpin 360 DV-Verify is used on an Intel® Xeon® E5-2690 v3 @2.6GHz with 32 GB RAM.

All 18 logic bugs that were previously detected by the SIC method are also detected by C-S^{2}QED properties within 30 s of computation time. Additionally, two error scenarios were detected by the C-S^{2}QED properties. The reported errors are activated in a flushed-pipeline context, in which the failing properties identified unnecessary stalling of the pipeline. This type of bugs is referred to as *performance bugs* within the scope of this thesis (cf. Section 5.2). As the *CPU* [1] instance is fetching NOPs before and after the time point t_{IF}, the results of the instruction (fetched at t_{IF}) execution are expected at specific time points. Because of this, any unnecessary stalls (which result in performance loss) associated with any instruction class are identified by the C-S^{2}QED property set.

Tab. 6.1 summarizes the C-S^{2}QED experiments (last column) and compares them with other approaches applied to verify the RISC-V core. It should be noted that a processor core supporting only the RV32I base ISA is considered for the comparison. Columns 1 and 2 correspond to SIC and the original S^{2}QED method [45], respectively. Rows 1 and 2 report on the categories of bugs the different approaches were able to detect. Row 3 shows the manual effort required for property generation and for formal verification setup. It required 10 person days for developing metamodel definition, creating a model instance for base ISA and for defining state transition graphs, which is applicable to both SIC and C-S^{2}QED methods. The SIC method requires 6 person days of effort for defining property traces, whereas it took 2 person days for creating property traces following the C-S^{2}QED method. "Runtime" refers to the computation time spent to detect a bug (row 4) or to prove its absence (row 5). In case of a bug being reported by the formal tool, the length of the counterexample (CEX) is reported for each method in row 6.

Table 6.2: Results: Property generation

ISA support	#LoC-MoT	#LoC-ToP	#LoC-ITL	#properties	effort
RV32I (base ISA)	700	370	2280	12	12 person days
RV32IM	+00	+00	2380	12	+0 person days
RV32IMC	+50	+100	5520	38	+5 person days
RV32IMCX	+00	+20	5800	39	+3 person days
RV32IMCXZicsr	+30	+20	6070	40	+1 person days

Tab. 6.2 shows a quantitative analysis of C-S^{2}QED property generation for RISC-V cores supporting different ISA extensions (column 1) in addition to the base ISA. Column 2 shows the lines of code (LoC) in Python for extracting the details of instructions from the model instances of the *MetaRISC* metamodel and for defining STGs for different instruction classes. The LoC in Python-based DSL required to define property models is shown in column 3 and the LoC of generated properties in ITL syntax is shown in column 4. The number of properties generated and the effort required for property generation is shown in columns 5 and 6, respectively.

Row 1 shows the initial effort required to setup the property generation for base ISA i.e., RV32I. The effort includes creation of the *MetaRISC* model instances, definition of STGs and definition of property models. From row 2 onwards, only additional LoC and effort required to

support added ISA extensions are shown. It should be noted that no additional effort is required for RV32IM (row 2), i.e., support for the multiply extension on top of the base ISA. This is because instructions in the *M* (Multiply and Division) extension belong to the *Integer Register-Register* instruction class from RV32I. 5 person days of additional effort are needed to generate the C-S^{2}QED properties supporting the compressed (*C*) instruction set extension (RV32IMC). Little effort is required to add the support for custom (X) and control and status register (*Zicsr*) instruction extensions (rows 4 and 5).

6.1.2 Observations

The following observations are made from the experimental results of verifying RISC-V processor core variants with the C-S^{2}QED method:

Observation 1

A complete set of C-S^{2}QED properties can detect all logic bugs in a processor, irrespective of their context in the program within a reasonable amount of time. The length of the counterexample is of significant importance to identify the root cause of a bug as it corresponds to the number of instructions that need to be executed to trigger a certain bug scenario. C-S^{2}QED significantly reduces the debugging time due to short counterexamples. The minimum counterexample length for a C-S^{2}QED property is 1 instruction, when a single-instruction bug is detected.

Observation 2

C-S^{2}QED requires no modification in the RTL code of the design and it has no restriction on the type of instructions it can consider. Developing the C-S^{2}QED property for each instruction class is straightforward and, by employing a generation flow, the manual effort for various extensions in an ISA is low.

Observation 3

The property generation flow significantly reduces the manual effort required to develop a complete set of properties. When the microarchitecture supports additional ISA extensions, the effort required for generating new properties to support the added extensions is minimal. The extensions are straightforward to implement. This, in turn, improves the overall verification productivity as well.

6.2 Peripheral Verification

In this section, modeling the behavior of peripheral design blocks using one or more state transition graphs is discussed. Due to the diverse nature of peripheral designs, different modeling approaches are required for an efficient property generation such that a set of properties completely captures the behavior of a design under verification (DUV). For example, the behavior of an AHB-to-APB bridge can be captured with a single state transition graph, whereas the behavior of a programmable interrupt controller that is connected to multiple interrupt sources

requires multiple state transition graphs to efficiently capture the complete behavior. For property generation, designs that are combinational in nature (e.g., AHB bus matrix) do not require state transition graphs to capture the design behavior[1]. Instead, structural models are more favorable to capture the specification details for effective property generation. In the following, several example designs are used to show different approaches employed for efficient formal verification with generated properties.

6.2.1 AHB-to-APB Bridge

The Advanced High-performance Bus (AHB) is a bus protocol introduced as part of the Advanced Micro-Controller Bus Architecture (AMBA) by ARM® [4]. AMBA consists of a set of standardized interconnect specifications that describe communication protocols between various functional blocks in an SoC. AHB is used for connecting functional blocks that require high communication bandwidth (e.g., processor core, on-chip RAM, direct memory access (DMA), external memory interface, etc.). These functional blocks are commonly referred to as *controllers* (previously referred to as "masters" blocks) and *responders* (previously referred to as "slave blocks"). An AHB may be connected to multiple controllers and responders. AHB uses a pipeline (address phase and data phase) architecture to facilitate high performance communication between interconnected functional blocks.

On the other hand, some peripheral components (e.g., UART, timer, SPI, IO devices, etc.) operate at lower communication bandwidth. The Advanced Peripheral Bus (APB) is used to establish communication between such functional blocks. The APB uses a simple non-pipelined architecture for communication and does not support some features of AHB (e.g., burst transfers). To establish communication between processing blocks such as a processor core or a DMA with the peripheral blocks, an AHB-to-APB bridge is needed to synchronize the time domains of the high-speed AHB bus with the low-speed peripheral blocks.

Modeling an AHB-to-APB Bridge in a State Transition Graph

The behavior of an AHB-to-APB bridge is typically specified as a control state machine [4]. Therefore, the approach proposed in Section 4.6 where the specified behavior of a sequential design is modeled in a state transition graph for property generation can be applied in a straightforward manner. Every operation performed by an AHB-to-APB bridge is captured as a state transition. Hereof, properties are generated targeting each state transition path such that a set of properties capture the complete behavior of the bridge.

A code snippet for modeling the specification of an AHB-to-APB bridge in a state transition graph is shown in Fig. 6.3. A model instance of the metamodel shown in Fig. 4.15 is created to facilitate the creation of a state transition graph. Line 4 in the code snippet creates an object instance of the class *MetaSTS* (cf. Fig. 4.15). Line 7 adds a state transition graph object to the model instance. Lines 10-15 show different state definitions created for an AHB-to-APB bridge: *idle, sel, read, write, read-wait, write-wait*. Lines 18-24 show the definition of expression variables that are used as *Action* expressions of corresponding state transitions. Line 28 shows the definition of the initial transition (*reset_idle* transition) with *idle* state as the sink state. It should be noted that the *reset_idle* transition has no source state. Line 29 shows the

[1]It should be noted that the designs that include sequential elements such as registers or flip-flops can be considered as highly combinatorial when the sequential depth of the design is a few clock cycles (e.g., <3)

```
1  class Ahb_2_Apb(object):
2    def __init__(self, api):
3      #-- Declare a model instance of the metamodel MetaSTS
4      self.sts_instance = MetaSTS(Name="Root_STS")
5
6      #-- Add a state transition graph child
7      self.ahb2apb_stg = self.sts_instance.addStateTransitionGraph("AHB2APB_Bridge_stg")
8
9      #-- States of the Ahb-2-Apb bridge
10     idle    = self.ahb2apb_stg.addState(Name='idle', Encoding=self.getEncoding(0,3), Init =True)
11     sel     = self.ahb2apb_stg.addState(Name='sel', Encoding=self.getEncoding(1,3))
12     write   = self.ahb2apb_stg.addState(Name='write', Encoding=self.getEncoding(2,3))
13     read    = self.ahb2apb_stg.addState(Name='read', Encoding=self.getEncoding(3,3))
14     wr_wait = self.ahb2apb_stg.addState(Name='write_wait', Encoding=self.getEncoding(4,3))
15     rd_wait = self.ahb2apb_stg.addState(Name='read_wait', Encoding=self.getEncoding(5,3))
16
17     #-- Expected signal description for different states
18     common_outs = EQ(clk,pclk), EQ(pslverr,hresp)
19     sate_idle_outputs    = LAND(common_outs, NOT(psel), NOT(penable), NOT(busy), NOT(pwrite), hready)
20     sate_sel_outputs     = LAND(common_outs, psel, NOT(penable), busy, NOT(pwrite), NOT(hready))
21     sate_write_outputs   = LAND(common_outs, psel, penable, busy, pwrite, hready)
22     sate_read_outputs    = LAND(common_outs, psel, penable, busy, NOT(pwrite), hready)
23     sate_wr_wait_outputs = LAND(common_outs, psel, penable, busy, pwrite, NOT(hready))
24     sate_rd_wait_outputs = LAND(common_outs, psel, penable, busy, NOT(pwrite), NOT(hready))
25
26     #-- Declare transitions - positional arguments : Name, SourceRef, SinkRef, Length
27     ## Transition reset-idle
28     reset_idle = self.ahb2apb_stg.addTransition('reset_idle', None, idle, 1)
29     reset_idle.createTrigger(Expression='reset_sequence')
30     reset_idle.createAction(Expression=state_idle_outputs)
31
32     #--Transition idle-idle
33     idle_idle = self.ahb2apb_stg.addTransition('idle_idle', idle, idle, 1)
34     idle_idle.createTrigger(Expression=LOR(LTEQ(htrans,1),NOT(hsel)))
35     idle_idle.createAction(Expression=state_idle_outputs)
36
37     #--Transition idle-sel
38     idle_sel = self.ahb2apb_stg.addTransition('idle_sel', idle, sel, 1)
39     idle_sel.createTrigger(Expression=LAND(GT(htrans,1), hsel))
40     idle_sel.createAction(Expression=state_sel_outputs)
41     ...remaining transitions are defined in similar manner...
```

Figure 6.3: Code snippet for defining a state transition graph for AHB-to-APB bridge

Trigger expression for reset transition, which is the reset sequence of the design. Line 30 shows the *Actions* of the reset transition i.e., expected signal behavior at state *idle*. State transition definitions for *idle_idle* and *idle_sel* that have *idle* state as their source state are shown in lines 32-40. Both state transitions are defined with *Length* of 1, which means that they require one clock cycle for transiting from source state to the sink state. Similarly, other state transition definitions are added to the state transition graph.

A graphical representation of the state transition graph of an AHB-to-APB bridge is shown in Fig. 6.4. All state transitions are shown with their respective trigger conditions and number of clock cycles required for the state transitions (*Length*). As mentioned in the previous paragraph, the design starts in the *idle* state and remains in *idle* until there is a transaction request ($htrans \leqslant 1 \vee hsel$) from the AHB master. When the AHB master initiates a transaction ($htrans \geqslant 1 \wedge hsel$) the design transitions to state *sel* and remains in that state until the slave device is ready ($\neg pready$). When the addressed slave device is ready ($pready$) to accept a transaction request, the design transitions to state *read* ($\neg hwrite$) or to state *write* ($hwrite$) depending on the type of transaction request. From state *read*, the design transitions to state *read-wait* when the slave device is busy ($\neg pready$). Otherwise, the design transitions to state

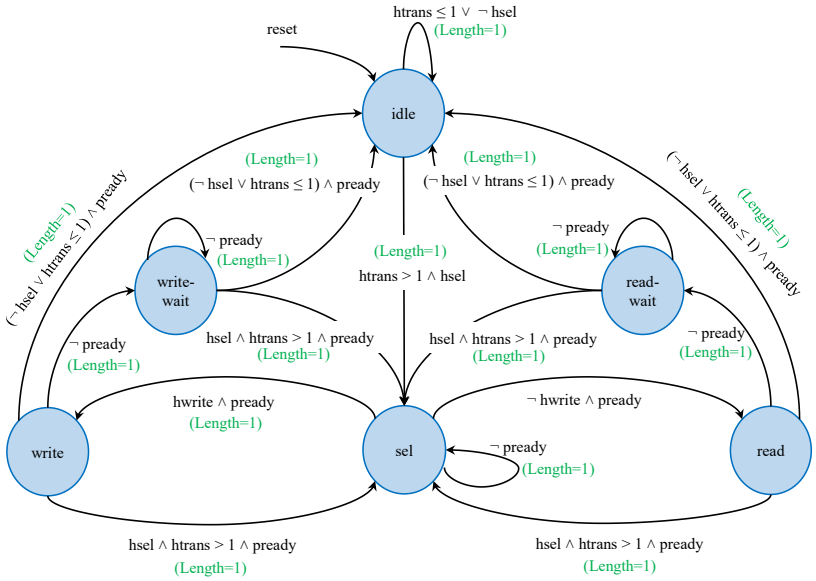


Figure 6.4: State transition graph of AHB-to-APB bridge (each state transition is shown with the *Trigger* (as a Boolean predicate) and *Length* attribute values)

sel when there is an incoming transaction request ($hsel \wedge htrans \geqslant 1 \wedge pready$) or to state *idle* if there is no request ($(\neg hsel \vee htrans \leqslant 1) \wedge pready$) from the AHB master. Similarly, when there is a write transaction request the design transitions through states *write*, *write-wait*, *sel* and *idle* depending on signal values at specific time points.

Property Generation and Experimental Results

The state transition graph created for capturing the behavior of an AHB-to-APB bridge consists of 6 state definitions and 17 state transitions. In order to cover the complete behavior of the design with a set of properties, an operational property is generated for each transition following the notion of Complete-Interval Property Checking (C-IPC) (cf. Section 2.5). As presented in Section 4.2, Template-of-Properties are used to extract the details from formal specification models and to define the property models.

A code snippet of the Template-of-Properties for the AHB-to-APB bridge is shown in Fig. 6.5. Line 4 shows the definition of a Python method that is used to define property models. In line 6, the name for the property module is set followed by a definition of clock and reset in lines 7-8, respectively. Line 9 defines an object instance of the class *Ahb_2_Apb* (cf. Fig. 6.3) and line 10 shows the extraction of the state transition graph of the AHB-to-APB bridge. Lines 13-28 show the creation of a property model for each state transition in the state transition graph. For each transition, the source state (line 14) of the transition is combined with the event expression (line 20) that triggered the transition to form the antecedent expression (line 22) of the property model. Similarly, the sink state (line 15) of the transition is combined with the action expression (line 21) to form the consequent expression (line 23) of the property model. The consequent expression is shifted by l clock cycles where l is length of the transition. This is achieved by the *DELAY* operator which is an alias for the *next (X)* operator (cf. Section 4.6). Lines 27-28 create a property model instance with corresponding attributes (name, expression, type, reset and clock) and the instance is added to the *metaProp* instance.

```
1  import MetaProp_api as mp
2  from ahb_2_apb_sts import *
3
4  def ahb2apb_mop(api, metaProp):
5      if api is not None:
6          metaProp.setName('ahb2apb_prop')
7          self.Clk = Clock(clk, 'Rise')
8          self.Rst = Reset(rst, 'Low')
9          ahb2apb_sts = Ahb_2_Apb()
10         stg = ahb2apb_sts.getSTG()
11
12         # Define a Model of Property (MoP)for each transition/operation
13         for transition in stg.getTransitions():
14             src_state = transition.getSourceRef()
15             snk_state = transition.getSinkRef()
16             if transition.getName() == 'reset_idle':
17                 self.Rst = None
18                 snk_state = None
19             # Extract the expressions from MoT
20             trn_event = transition.getEvent().getExpression()
21             trn_action =  transition.getAction().getExpression()
22             expr_antecedent = LAND(src_state, trn_event)
23             expr_consequent = LAND(snk_state, trn_action)
24             # Add a property instance to metaProp instance
25             expr_antecedent = expr_antecedent.mapExpression(mp)
26             expr_consequent = DELAY(transition.getLength(), expr_consequent.mapExpression(mp))
27             prop = metaProp.addProp(Name=transition.getName(), Type= 'assert',
28                         Expression=IMPLY(expr_antecedent,expr_consequent), Reset=self.Rst, Clock=self.Clk)
```

Figure 6.5: Code snippet of Templates-of-Property for AHB-to-APB bridge

From a C-IPC point of view, each transition relates to an operation and for each operation, source and sink states represent the conceptual or important states. Therefore, a state transition graph can be equated to a conceptual state machine proposed in C-IPC [81]. For proving the completeness of property set, a property graph is generated from the state transition graph definition. A property graph is a tree like structure that captures all possible sequence of transitions starting from the reset transition. Additionally, the property set must pass the determination requirements of the design. In other words, a complete property set must determine the values of all important states and output signals of the design at all time points. Towards this end, determination requirements are developed considering all important state elements and output signals of the design and it is checked if the generated property set satisfies these determination requirements.

Table 6.3: Results: Property generation and formal runs

#LoC-MoT	#LoC-ToP	#LoC-ITL	#Properties	#Bugs	Runtime	Effort
170	28	440	17	2	4 min	2 person days

Tab. 6.3 shows a quantitative analysis of the property generation and formal property run. For property checking, the commercial formal verification tool OneSpin 360 DV-Verify is used on an Intel® Xeon® E5-2690 v3 @2.6GHz with 32 GB RAM. Column 1 shows the Lines of Code (LoC) in Python required to capture the behavior of the AHB-to-APB bridge in a state transition graph. Column 2 shows the LoC in Python DSL needed to define the property models (shown in Fig. 6.5). A total of 17 properties (*#Properties*) have been generated with 440 LoC in ITL (*#LoC-ITL*). With the generated set of properties 2 logic bugs are found in the DUV.

A complete set of properties generated (in ITL) for the AHB-to-APB bridge is listed in Appendix D. The formal property run and completeness check took 4 minutes of runtime to reach convergence (*Runtime*). It required only 2 person days (*Effort*) to develop the state transition graph and for setting up the subsequent property generation.

6.2.2 Programmable Interrupt Controller

A Programmable Interrupt Controller (PIC) is an integral part of any CPU subsystem. An interrupt controller is mainly used to provide an interface between external Input/Output (I/O) devices and a processor unit. For example, when a timer device counts to its maximum value, it may initiate a service request to the interrupt controller. Service requests are referred to as interrupts, which are external events that occur asynchronously to the instruction execution of a processor core. The interrupt controller monitors such requests from one or more external devices and sets an interrupt request (IRQ) signal to the processor core. An interrupt controller may use a priority-based mechanism to resolve simultaneous service requests from multiple devices. The steps taken to formally verify a programmable interrupt controller by applying the techniques proposed in Chapter 4 are described in the following.

Modeling PICs is done in two steps. First, an untimed model is created as an instance of the metamodel definition capturing the high-level characteristics of a PIC. However, this model does not describe the time-accurate behavior of the design as it would restrict the implementation choices. Therefore, the untimed model of a PIC is extended with precise timing information by modeling the behavior of each interrupt source in a state transition graph.

Metamodel Definition of PIC

The metamodel definition of PICs is shown in Fig. 6.6. The *rootNode* of metamodel *MetaPIC* has three child nodes *Host*, *InterruptSource* and *InterruptController* with multiplicities 1, 1..∗ and 1, respectively. The *InterruptController* class has generic attributes to define the properties of an interrupt controller design. It defines the attributes that are applicable to all the interrupt sources. As an interrupt controller may support one-to-many (multiplicity 1..∗) interrupt sources the attributes of *InterruptSource* class are defined separately for each interrupt source.

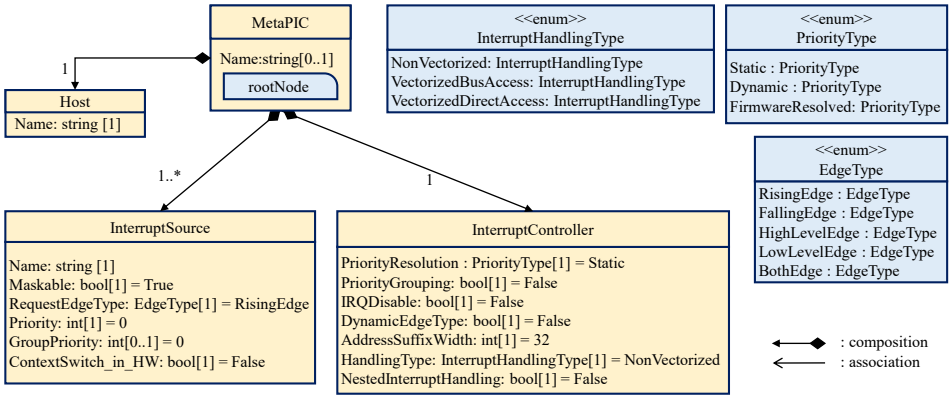


Figure 6.6: Metamodel definition of a programmable interrupt controller

Attributes of the *InterruptController* class are described as follows: The *PriorityResolution* attribute belongs to a type defined by an enum class *PriorityType* and indicates the priority resolution mechanism to be implemented by an interrupt controller design. The default value for

this attribute is set to *Static* which infers that the priorities of interrupt sources are pre-defined. *PriorityGrouping* is an attribute of Boolean type and indicates if a set of interrupt sources belong to a certain priority value. If this attribute is set to *False*, the *GroupPriority* attribute of the *InterruptSource* class becomes obsolete. The *IRQDisable* attribute is of Boolean type and indicates whether the interrupt controller is allowed to enable or disable interrupt sources upon a transaction from the processor core (by the software program). Attribute *DynamicEdgeType* indicates whether the edge (rising or falling) on which the interrupt request is notified can be set dynamically. If this attribute is set to *True*, a register bitfield is used to dynamically set the edge of an interrupt request to which the interrupt channel is sensitive to. *AddressSuffixWidth* is an integer value that indicates the address size of the interrupt device (e.g., timer). The *HandlingType* attribute determines the type of interrupt handling mechanism such as vectorized or non-vectorized. *NestedInterruptHandling* is a Boolean attribute that indicates whether the interrupt controller supports pre-emption of a low-priority interrupt when a high-priority interrupt request is received.

The attributes of the *InterruptSource* class are described as follows: Each interrupt source has a name defined by the *Name* attribute. *Maskable* is a Boolean attribute which indicates whether a request from an interrupt source is maskable. When this attribute is set to *True*, a register bitfield in the interrupt controller design is used to indicate whether a request is serviced or masked at a certain timepoint t. The attribute *RequestEdgeType* belongs to a type defined by an enum class *EdgeType* and has a default value of *RisingEdge*. It describes how an interrupt request from an external device is recognized by the interrupt controller. The *Priority* attribute is an integer values that indicates the priority of that interrupt source. *GroupPriority* is also an integer value that indicates the group priority of the interrupt source. Two or more interrupt sources may belong to the same group priority. The group priority value is considered before the individual priority value when resolving simultaneous interrupt requests from multiple sources. The *ContextSwitch* attribute is used to indicate whether the context switching logic is implemented by the hardware or by the software.

Hardware Implementation Overview

A model instance of the *MetaPIC* metamodel is created with specific values for each attribute and represents an abstract model of the features implemented by an interrupt controller. The RTL generation (cf. Section 3.4) of interrupt controller is built such that it takes any model instance and generates a corresponding hardware design.

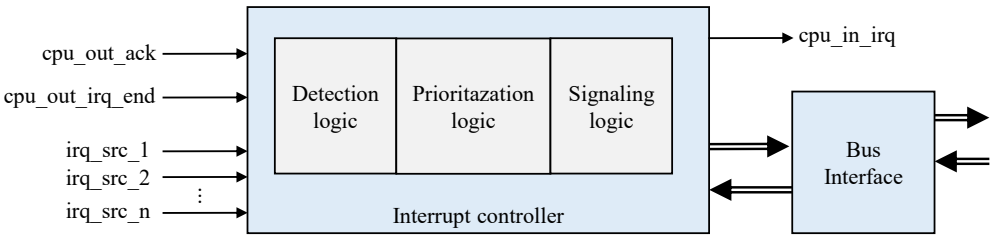


Figure 6.7: Block diagram of programmable interrupt controller

The top-level block diagram of a PIC is shown in Fig. 6.7. A programmable interrupt controller consists of three main logic blocks: detection, prioritization and signaling. As the name suggests, the *Detection logic* is responsible for identifying the interrupt requests from different sources (*irq_src_1*, *irq_src_1*, ... *irq_src_n*). The *Prioritization logic* handles simultaneous

interrupt requests and also computes the logic for preemption of an ongoing interrupt service routine. The *Signaling logic* communicates with the CPU to raise interrupt requests. The *Bus interface* circuit is used for the transactions with the CPU. For example, the CPU may initiate a transaction through the bus to enable or disable certain interrupt sources.

The state of an interrupt source is monitored by utilizing two register bitfields: *pending* and *active*. When an interrupt source raises a request, the *pending* bitfield is set and an interrupt request signal (*cpu_in_irq*) is sent to the CPU, provided no other requests are pending. When the CPU acknowledges (*cpu_out_ack*) the request, the *active* bitfield is set. After receiving an acknowledgment, the PIC passes the address of an interrupt service routine (ISR) to the CPU and the CPU jumps to the address and executes the specified ISR.

Modeling Interrupt Behavior in a State Transition Graph

A set of properties is generated such that they completely capture the behavior of an interrupt controller design. Towards this end, the behavior of an interrupt controller is modeled in a state transition graph. For each interrupt source request, a state transition graph is created by considering its *pending* and *active* register bitfields as the state bits. For each interrupt request, there are three possible reachable states:

- State *idle* := {pending-register = 0, active-register = 0}
- State *pending* := {pending-register = 1, active-register = 0}
- State *active* := {pending-register = 1, active-register = 1}

A state transition graph can be created for an entire interrupt controller by considering the *pending* and *active* bits of all sources. However, such a state transition graph leads to a high number of state definitions and transitions, and results in a complex formulation of the state transition graph. To illustrate, let us consider an interrupt controller that is connected to three interrupt sources ($n = 3$). This implies that there are 3 pending and 3 active register bitfields in the design. The state variable s for n interrupt sources can be realized by concatenating all pending and active registers as follows:

$$s = \{active_1, active_2, active_3, pending_1, pending_2, pending_3\} \tag{6.1}$$

In general, for an interrupt controller that is connected to n peripheral devices, there are $2^{(2*n)}$ (two register bits for each interrupt source) possible state definitions. However, for computing the number of reachable states, we have to consider the following conditions: 1) only one interrupt source can be in state *active* at any given time point and, 2) an interrupt source can be in state *active* only if it was in state *pending* before. Further, the priority of individual interrupt sources has to be considered for computing the reachable states. After removing all unreachable state definitions, the number of reachable states for an interrupt controller that is connected to n interrupt sources is given by $2^{(n+1)} - 1$.

In order to cover every possible operation performed by the design, a property shall be generated for each state transition. For an interrupt controller supporting three external devices, there are 30 possible transitions. When the number of interrupt sources increases, the number of state definitions and state transitions increase exponentially leading to a complex state transition graph. Properties generated from such a state transition graph are not user-friendly as they encode state information of all interrupt sources in every property.

To avoid a single but complex definition of the state transition graph, separate but identical state transition graphs are created for each interrupt source. This state transition graph is shown

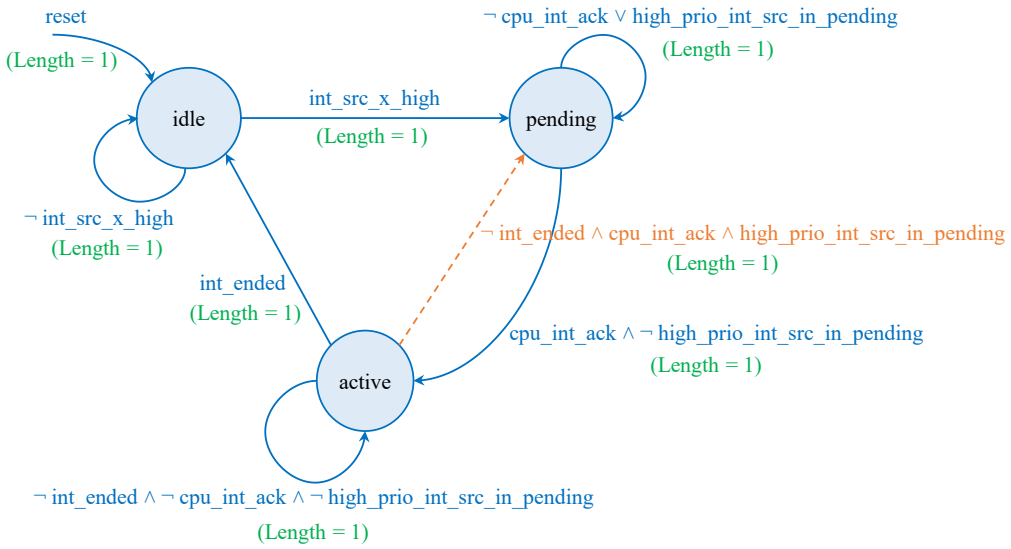


Figure 6.8: State transition graph of an interrupt source (each state transition is shown with the *Trigger* (as a Boolean predicate) and *Length* attribute values)

in Fig. 6.8. State encoding for each interrupt source is formed by combining the pending and active register fields. As mentioned earlier, each interrupt source request has three reachable states as shown in Fig. 6.8. The number of clock cycles taken for each state transition is also shown ($Length = 1$).

The state *idle* is an initial state reached initially through the reset transition. An interrupt source remains in state *idle* as long as there is no service request from an interrupt source ($int\_src\_x\_high = 0$). When there is an interrupt request from a source ($int\_src\_x\_high = 1$) the state of the interrupt request transitions to state *pending*. If the CPU is not serving any other interrupt requests and if no other higher-priority interrupt requests are pending, the state of the interrupt request transitions to state *active* when the interrupt controller receives an acknowledge (cpu_int_ack) from the CPU. The state of the interrupt request remains in state *active* until the completion of its ISR, unless it is preempted by a higher-priority interrupt which causes the state of an interrupt request to become *pending*. Note that the transition is marked in dotted lines to indicate that the transition from state *active* to state *pending* is not possible for the highest-priority interrupt source. The state of the interrupt request becomes *idle* when the corresponding ISR has been completed (int_ended) by the CPU.

Property Generation and Experimental Results

To model the behavior of each interrupt source, a state transition graph is created that consists of 3 state definitions and 8 state transitions as described above. For capturing the complete behavior of an interrupt controller, operational properties are generated following the C-IPC notion presented in Section 2.5. Similar to the AHB-to-APB bridge, a template-of-properties is developed to extract the architectural characteristics and behavioral details of each interrupt source and to define the property models.

Tab. 6.4 shows the results of property generation and formal verification of multiple instances of the interrupt controller connected to different number of interrupt sources. For property checking, the commercial formal verification tool OneSpin 360 DV-Verify is used on an Intel® Xeon® E5-2690 v3 @2.6GHz with 32 GB RAM. The state transition graphs have to be

Table 6.4: Results: Property generation and formal runs

#LoC-MoT	#LoC-ToP	Effort	#Sources	#LoC-SVA	#Properties	Runtime
450	250	12 per. days	3	500	25	3.5 min
-	-	-	4	612	32	4 min
-	-	-	5	745	39	6 min
-	-	-	6	890	46	6 min
-	-	-	7	1050	53	12 min
-	-	-	7*	3960	60	40 min

defined considering different architectural choices of an interrupt controller (e.g., static or dynamic priority). Column 1 (*#LoC-MoT*) indicates the LoC in Python needed to define the state transition graphs in a configurable manner. Column 2 shows the LoC in Python-based DSL for defining property traces. The effort required to define state transition graphs and property traces is only one time effort and is shown in column 3.

The number of interrupt sources connected to the interrupt controller is shown in column 4. LoC in SVA and the number of generated properties are shown in columns 5 and 6, respectively. Runtime of properties including the completeness checks is shown in column 7. Rows represent the corresponding numbers for interrupt controller connected to different number of interrupt sources. It is important to note that *#LoC-MoT*, *#LoC-ToP* and *Effort* are shown only for row 2 to indicate that the MoT and ToP are implemented once and re-used for all variants of the interrupt controller. As a result, additional effort is not needed for property generation of subsequent interrupt controller instances.

Rows 2-6 show the numbers for interrupt controllers that are configured to use a static (predefined) priority mechanism to resolve simultaneous service requests from different interrupt sources. Row 7 shows the numbers in which an interrupt controller is configured to use a dynamic priority mechanism for resolving simultaneous requests. Dynamic priority means that the priority of individual interrupt sources can be modified by the software program during runtime. As a result, additional properties are generated to verify the dynamic priority values of interrupt sources. As a consequence, the runtime of the properties also increases. Note that the ToP is configured to define the property models considering all possible MoT instances. Therefore, no extra effort is needed for generating additional properties required for verifying the dynamic priority feature. The shown results demonstrate the reusability and subsequent productivity gain of the property generation flow.

6.2.3 Bus Matrix

A bus matrix is a commonly found component in multi-master multi-slave designs and provides a platform to connect multiple masters to multiple slave devices. Based on the access request from master devices, the bus matrix determines the bus master that obtains access to a bus slave, and connects control and data signals between them. In the following, modeling specifications of bus matrices in a structural model and subsequent property generation is discussed.

Metamodel Definition of Bus Matrix

The metamodel definition of a bus matrix is shown as a UML class diagram in Fig. 6.9. The metamodel definition captures the high-level characteristics of the bus matrix. Root node of the metamodel is *MetaBusMatrix* which defines name, data width, address width and arbitration policy as its attributes. The type of *Arbitration* attribute is defined by an enum class *PriorityResolution* and has the default value *Static*. It specifies the type of arbitration mechanism implemented by the bus matrix to resolve simultaneous access requests to a specific bus slave from two or more bus masters.

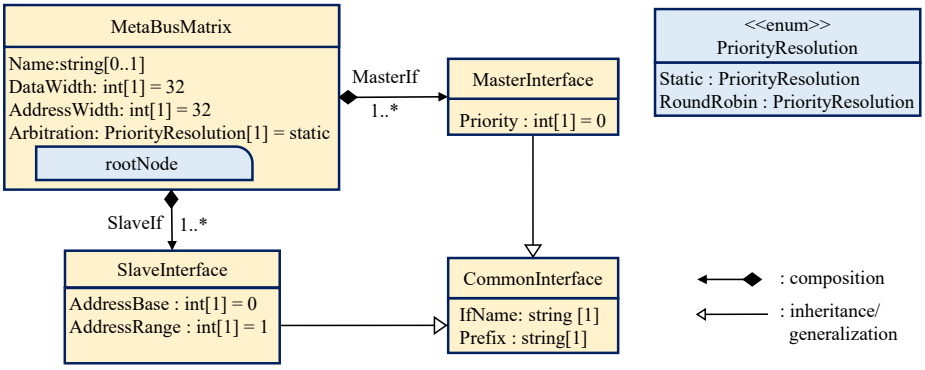


Figure 6.9: Metamodel definition of a bus matrix

Class *MetaBusMatrix* holds a composition relation to classes *MasterInterface* and *SlaveInterface*. The multiplicity of the composition relation is set to $1..*$, which indicates that there can be multiple bus masters and slave device connected to the bus matrix. Both *MasterInterface* and *SlaveInterface* classes inherit the attributes (*IfName, Prefix*) of parent class *CommonInterface*. The *MasterInterface* class defines an attribute *Priority* to attach a priority value to every bus master. The *SlaveInterface* class defines as its attributes base address (*AddressBase*) and range of address space (*AddressRange*).

Hardware Implementation Overview

Model instances of the metamodel definition are created by filling the specific values for various attributes shown in Fig. 6.9. The RTL generation flow outlined in Section 3.4 takes any model instance as the input and generates a corresponding RTL design.

A model instance is created for an AHB bus matrix with two master and seven slave interfaces. AHB matrix provides an interconnection between multiple AHB masters and multiple AHB slave devices. A simplified top-level block diagram of the AHB matrix is shown in Fig. 6.10. Ports of both the masters are prefixed with "*m_*" (e.g., *m_cpu_HADDR*) while the ports of slave devices are prefixed with "*s_*" (e.g., *s_dmem_HRDATA*). Both data and address width are set to 32 bits and a static arbitration policy is selected for resolving simultaneous accesses.

From a functional point of view, an AHB matrix has three stages: an input stage, a decode stage and an output stage. The input stage receives a transaction request from a master device and stores the incoming transaction when the addressed slave device is not free. The decode stage generates the select signal to the slave devices based on the transaction request from a master device during the address phase of the transaction. During the data phase, incoming

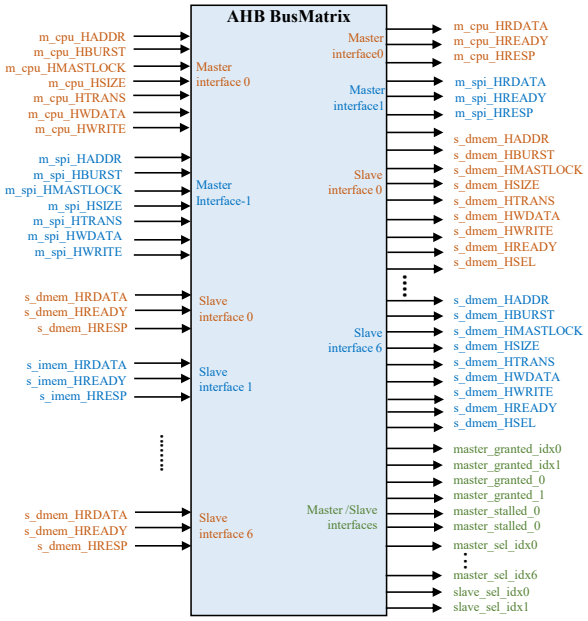


Figure 6.10: Block diagram of an AHB matrix with 2 master and 7 slave interfaces

response signals from slave devices are connected back to the appropriate master device interface. The output stage selects control, address and data signals from the input stage. It also determines when to switch between input ports in the input stage.

Property Generation and Experimental Results

The metamodel definition shown in Fig. 6.9 enables a high degree of configurability in building bus matrices connected to multiple master and slave devices. The function of a bus matrix is highly combinatorial in nature and the model instances of the shown metamodel capture the required specification details of a bus matrix needed for property generation. As a result, modeling the behavior of a bus matrix in a state transition graph is not beneficial.

Following the property generation flow (cf. Section 4) the specifications details of a bus matrix are extracted from the model instance of the metamodel *MetaBusMatrix* in Template-of-Properties. Property models are defined targeting each master and its connection to all slave devices.

A simplified ToP code snippet is shown in Fig. 6.11, where the property models for access requests are defined. Lines 1-21 show a Python function which returns expression variables *req_expression* and *ack_expression*. Transaction requests from a bus master to a bus slave are captured in *req_expression*, wheras the response or the expected behavior of a slave device is captured in *ack_expression*. Lines 24-39 show a Python function that defines access request property models from each bus master to every bus slave. Lines 24 and 25 extract the details of master and slave devices from the MoT. Line 27 shows a *for loop* that iterates over all the bus masters. For the highest-priority master the property model has to be different as it gains access to the bus slave over other masters that request the access simultaneously (shown in lines 28-33). Lines 34-39 show the definition of property models for low-priority masters in which an additional constraint is added such that the higher-priority masters do not issue a read or write request (*high_prio_master_idle*).

Tab. 6.5 shows the quantitative results of the property generation and formal verification

```
def req_expression(m,i,s,j):
    req_expression = LAND(LAND(GTEQ(m.getName()+'HADDR', s.getAddressBase()),
                        LT(m.getName()+'HADDR', s.getAddressBase()+s.getAddressRange())),
                    GTEQ(m.getName()+'HTRANS',1),
                    EQ(s.getName()+'HREADY',1),
                    s.getName()+str(s[j])+'is_idle',
                    DELAY([1,wait_window],EQ(s.getName()+'HREADY',1)))
    ack_expression = LAND(m.getName()+str(m[i])+'is_busy',
                    EQ(m.getName()+'granted', 1),
                    EQ(m.getName()+'HRDATA',  s.getName()+'HRDATA'),
                    EQ(m.getName()+'HREADY',  1),
                    EQ(m.getName()+'HRESP',   s.getName()+'HRESP'),
                    EQ(m.getName()+'HADDR',   s.getName()+'HADDR'),
                    EQ(m.getName()+'HBURST',  s.getName()+'HBURST'),
                    EQ(m.getName()+'HMASTLOCK',s.getName()+'HMASTLOCK'),
                    EQ(m.getName()+'HSEL',    1),
                    EQ(m.getName()+'HSIZE',   s.getName()+'HSIZE'),
                    EQ(m.getName()+'HTRANS',  s.getName()+'HTRANS'),
                    EQ(m.getName()+'HWDATA',  s.getName()+'HWDATA'),
                    EQ(m.getName()+'HWRITE',  s.getName()+'HWRITE'))
    return [req_expression,ack_expression]

def access_request(mot, metaprop):
    bus_masters = mot.getMasterIfs()
    bus_slaves = mot.getSlaveIfs()
    #define property models for all bus masters targeting transaction requests to each slave
    for i,m in enumerate(bus_masters):
        if m.getPriority() == 0:#highest priority master
            for j,s in enumerate(bus_slaves):
                antecedent = req_expression(m,i,s,j)[0]
                consequent = ack_expression(m,i,s,j)[1]
                metaprop.addProp(Name=m.getName()+'trans_req_prop', Expression=IMPLY(antecedent,consequent),
                        Type='assert', Clock=Clk, Reset=Rst)
        else:# low priority masters
            for j,s in enumerate(bus_slaves):
                antecedent = LAND(higr_prio_masters_idle(m,i),req_expression(m,i,s,j)[0])
                consequent = ack_expression(m,i,s,j)[1]
                metaprop.addProp(Name=m.getName()+'trans_req__prop', Expression=IMPLY(antecedent,consequent),
                        Type='assert', Clock=Clk, Reset=Rst)
```

Figure 6.11: Code snippet of Templates-of-Property for AHB matrix

Table 6.5: Results: Property generation and formal runs

#LoC-ToP	Effort	#Master	#Slave	#LoC-SVA	#Properties	Runtime	Coverage
200	4 days	2	4	560	22	2 min	100%
–	–	3	4	820	31	3.5 min	100%
–	–	3	6	1190	45	5 min	100%
–	–	4	7	1815	67	7 min	100%

runs. For property checking, the commercial formal verification tool OneSpin 360 DV-Verify is used on an Intel® Xeon® E5-2690 v3 @2.6GHz with 32 GB RAM. Column 1 shows the LoC in Python DSL needed to define property models. It required 4 person days (*Effort*) to develop the metamodel definition, model instances, Template-of-Properties and to obtain a convergence of the properties. Column 3 and 4 show the number of master and slave devices, respectively. In the table, the rows show the results for different numbers of master and slave instances. It should be noted that the ToP is developed only once such that it is applicable to any number of master and slave instances. For an AHB matrix with 2 master and 4 slave devices, 22 prop-

erties (column 6) with 560 LoC in SVA (column 5) are generated to completely verify the bus matrix. Columns 7 shows the property runtime of the generated properties. In contrast to the designs that are sequential in nature (e.g., PIC or AHB-to-APB), the coverage strategy taken for combinatorial designs such as the AHB matrix is different. It is necessary to ensure that all expected connections between the master and slave interfaces exist and that a set of properties verifies the required conditions under which the data is transmitted through these connections. An useful metric available after the property runs is code coverage. Code coverage is a measure of which RTL code lines are analyzed during the property runs. A cumulative collection of code coverage of all the properties can be used to ensure that the entire design implementation is evaluated. Towards this end, for AHB matrix 100% (column 8) of the RTL code coverage is achieved indicating that a set of properties fully evaluated the design implementation.

6.2.4 Summary and Observations

In addition to the designs discussed in this chapter, the proposed specification modeling and property generation approaches have been applied to several other design blocks. These design blocks include a timer with multiple timer channels, HW/SW interface registers, communication peripherals such as UART (Universal Asynchronous Receiver Transmitter), I^2C (Inter-Integrated Circuit) and safety-relevant components such as CRC (Cyclic Redundancy Check), and ECC (Error Correction Code) circuits. The following observations are made from the application of property generation techniques to several real-life design blocks.

Observation: Productivity and Reusability

The proposed modeling and generation approaches simplify the verification of digital design blocks by automating the property development. For measuring the effectiveness of the generation flow, LoC gain can be used as an useful metric. For the case of RISC-V processor cores, a total of 1290 LoC in Python were required to generate 6070 LoC in ITL, resulting in a LoC gain of 4.6 (cf. Tab. 6.2). LoC in Python include both specification modeling using state transition notation and property model definitions. Moreover, the generation flow can be extended to support any added instruction extension with a small manual effort.

The PIC design can be configured to enable different combinations of features (cf. Section 6.2.2). This allows for a high degree of flexibility and the property generation is set up considering all combinations of parameters. The property generation for a PIC design with dynamic priority results in a LoC gain of 5 (cf. Tab. 6.4, row 7). For the case of the AHB-to-APB bridge, the configuration space is low and the LoC gain is 2 (cf. Tab. 6.3). Similar to the PIC design, the AHB bus matrix can be configured to interconnect multiple master and multiple slave devices. For a bus matrix with 4 master and 7 slave interfaces, a LoC gain of 9 is achieved. Moreover, the property generation is configurable such that any instance of a bus matrix connected to any numbers of master and slave devices can be described.

As the generation is set up considering all combination of configuration space (e.g., microarchitectural choices and different values for object attributes in the metamodel definitions), the LoC gain factor and the reusability factors are high. Overall, a high productivity gain is achieved with the proposed specification modeling and property generation approaches.

Observation: Effectiveness and Quality

The proposed property generation approach considers different specification modeling techniques based on the nature of design. A state transition notation is used to capture the sequential behavior (e.g., PIC and AHB-to-APB bridge), whereas the combinatorial behavior (e.g., AHB matrix) is captured with structural models. This helps to simplify the definition of property models and allows to generate the properties that are effective on the respective designs.

The quality of a design implementation is often measured by the type of bugs detected and the coverage achieved during the verification. For designs of sequential nature, a set of generated properties satisfy the completeness criterion according to C-IPC (cf. Section 2.5) and detected several difficult-to-find bugs. For the combinational type of designs 100% code coverage is achieved through a set of generated properties.

Chapter 7

Summary of Contributions

The novel solutions proposed and developed in this thesis along with the key findings have been pre-published in several scientific conferences [27, 33, 28, 35, 103, 29, 50, 30, 32]. The introduced concepts and methods have been applied to several industrial designs with the courtesy of Infineon Technologies AG. A summary of the contributions is provided in the following.

In this thesis, the software development principles of Model-Driven Architecture have been adopted for developing a generation flow for properties. The taken approach for property generation introduces three models, namely the Model-of-Things, the Model-of-Property, and the Model-of-View. Each model belongs to a distinct model layer in the generation flow and each model layer addresses a specific concern of the property generation. The separation of concerns through model layers ensures modular flow development, and enables uncomplicated enhancements and feature extensions. The properties are generated through a series of model-to-model transformations between these model layers [27, 33, 28]. Python is used as the domain-specific language for describing the intermediate transformations. A metamodel-based automation framework is utilized to generate an infrastructure that facilitates the description of transformations. The APIs that form the central part of the infrastructure are generated from the metamodel definitions of the models mentioned before. The generated APIs are further extended with domain-specific APIs to significantly reduce the effort required for developing the transformations [35, 50]. The property generation solution developed in this thesis is termed as "*MetaProp*". The various aspects of introduced models and model transformations are discussed in Chapter 4.

A key aspect of the property generation flow is the translation of informal specifications to formal specification models. Due to the diverse nature of hardware designs, the methodology includes different modeling paradigms to formalize the specifications. The metamodel *MetaExpression* provides features to describe the behavior of combinational designs in the form of expression trees and dataflow expressions. The *MetaExpression* metamodel is modular in nature and can be integrated into other metamodel definitions that capture the specification level configurations of the design. For modeling the behavior of sequential designs, a formalism using finite state machine-like notations for traces is introduced. The metamodel *MetaSTS* defines this formalism. The *MetaSTS* metamodel enables to define the behavior of sequential designs with annotated timing information for transitions between important states. Annotation is also used to map abstract states in the Model-of-Things to the Model-of-Property and, finally, to the design implementation. Such an annotation or binding mechanism enables Model-of-Properties to be applicable on a variety of design implementations [28, 29, 32].

Another important contribution of this thesis is a complete processor verification methodology, which is based on the aforementioned generation approach. The introduced methods for specification modeling are employed to formalize the ISA and the behavior of instructions within the processor pipelines. However, it requires substantial manual efforts and in-depth knowledge of the microarchitectural details of the processor implementation to describe the transformations that define the Model-of-Properties. The prime reason for this requirement is the overlapped execution of instructions within the pipelined architectures of processors and the numerous internal and external pipeline stall scenarios. For a complete processor verification, a set of generated properties must consider all combinations of instruction overlapping coupled with all scenarios of pipeline stalls. In retrospect, the Model-of-Properties — from which the properties are generated — are required to consider all combinations of the aforementioned scenarios. To address these aspects, the C-S^{2}QED method — an extension of the S^{2}QED method — has been developed to completely verify a processor. The C-S^{2}QED method is also applicable to exceptions within the processor pipelines and superscalar pipeline architectures. The C-S^{2}QED method detects all functional bugs in a processor implementation and requires significantly less manual efforts compared to state-of-the-art processor verification methods [103, 30]. The different aspects of processor verification with the C-S^{2}QED method are discussed in Chapter 5. The completeness hypothesis of the C-S^{2}QED method based on the completeness criterion defined by C-IPC [81] and a completeness proof are also part of this thesis (cf. section 5.5.5). The property generation flow has been leveraged to generate a set of C-S^{2}QED properties to further enhance the effectiveness of the methodology.

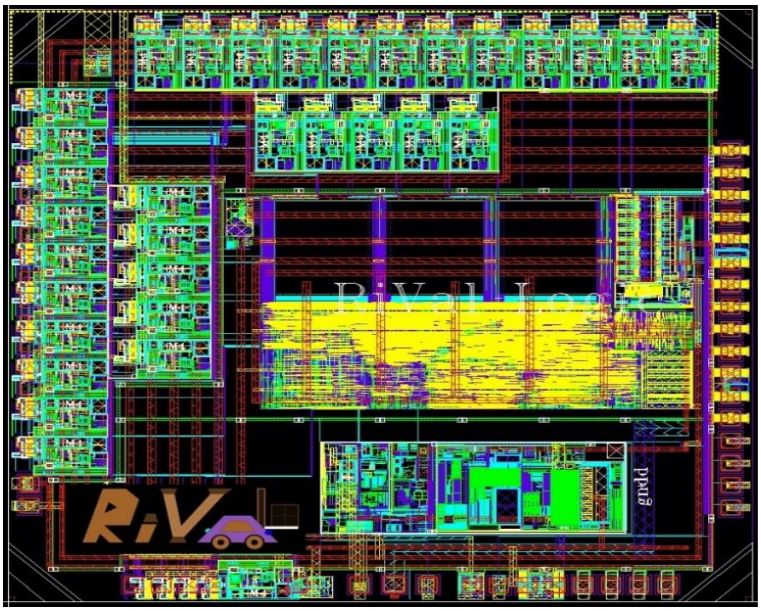


Figure 7.1: Layout of the “Rival” chip

The applicability and effectiveness of the introduced modeling paradigms and developed methods have been demonstrated with the formal verification of several industry strength designs. Numerous logic bugs including the bugs that are typically regarded as difficult to find have been detected during the formal verification with generated properties. Fig. 7.1 shows the layout of the “Rival” SoC, where most of the IPs including the RISC-V core and excluding the legacy IPs have been formally verified only with the proposed methods in this thesis. The

Rival SoC is used in the powertrain and safety automotive applications. The manufactured chip works "first time right" and no logic bug has been detected during the post-manufacturing tests.

Various architectural alternatives of the RISC-V based processor designs are verified with the generated C-S^{2}QED properties. The property generation is built in a configurable manner such that any changes in microarchitecture of the processor — that may be caused by the changes in specifications — are implicitly covered by the generation flow. Thus, additional manual efforts are not required and the functional flaws due to the changes in specifications are neutralized. Furthermore, the proposed methods have also been applied to communication protocol IPs, bus bridges, interrupt controllers and safety-relevant designs. The various aspects of formally verifying the mentioned designs together with the results are discussed in Chapter 6.

In summary, the scientific contributions of this thesis are enumerated as follows:

- Principles of Model-Driven Architecture have been applied to achieve a quantum leap in the verification productivity of hardware designs.
- Novel modeling paradigms are introduced to formalize the hardware design specifications. Especially, a new FSM-based notation for traces formalizes the implicit design know-how.
- An automatic annotation mechanism is introduced to generate properties for various architecture alternatives of a design without additional overhead.
- A complete processor verification methodology called C-S^{2}QED is introduced, which is also applicable to exceptions and superscalar architectures. The methodology has been proven to detect all functional bugs in a processor.
- The property generation is also applicable on communication and house-keeping IPs.
- The introduced methods are applicable on wide variety of designs including safety-critical IPs such as error correction and cyclic redundancy designs (cf. appendix E).
- The applicability of the introduced modeling paradigms and the property generation flow is industry-strength.

In future, the work shall be extended to support formal verification of designs realized with automatic design transformations [9]. Design transformations are a set of rules that are applied on an intermediate design model to achieve specific functional and/or non-functional design changes. The property generation solution developed in this thesis shall be extended to formally verify the expected changes in the design behavior. Further, applicability of the C-S^{2}QED method for non-core designs with pipeline architectures — for example, bus protocols that use pipelined transaction phases — shall be explored.

Kapitel 8

Deutsche Zusammenfassung

Die exponentielle Zunahme der Komplexität moderner System-on-Chips (SoCs) trägt zu einem äußerst komplexen Entwurfsprozess bei. Das Feststellen der funktionalen Korrektheit von Hardware-Entwürfen (Designs) ist essentiell, um sicherzustellen, dass alle Designfehler gefunden werden, bevor der Chip gefertigt wird. Alle während des Verifikationsprozesses nicht entdeckten Designfehler können gravierende Folgen haben, wenn sie erst während der Anwendung entdeckt werden. Das gilt insbesondere in sicherheitskritischen Bereichen, wie im Automobilbereich oder der Luftfahrt. Trotz bedeutsamer Fortschritte bei der Verifikation, insbesondere im Bereich von Entwurfsverifikation, hinken die Fähigkeiten und Kapazitäten existierender Verifikationsmethoden den Bedürfnissen moderner SoCs hinterher. Um mit der steigenden Komplexität der Designs schritthalten zu können, sind neue Verifikationmethoden nötig, die die Designfehler wirksam entdecken können. Formale Verfahren sind hierbei hinsichtlich ihrer Gründlichkeit überlegen und garantieren eine hohe Designqualität. Die formalen Verifikationsmethoden haben sich, trotz der anfänglichen Probleme mit der Skalierbarkeit bei größeren Designs, über die letzten Jahre weiterentwickelt und sind dadurch auch auf größere Entwürfe anwendbar geworden. Nichtsdestotrotz werden formale Methoden noch selten in der Industrie genutzt, hier werden häufig Simulationsmethoden bevorzugt. Die größten Schwierigkeiten bereiten hierbei das Entwickeln von "guten Properties" für die Designspezifikationen, fehlende Expertise für die Anwendung von formalen Verfahren und die Verfügbarkeit effektiver Verifikationsprozesse.

Die Motivation dieser Arbeit ist begründet in der Beobachtung, dass Produktivität und Qualität der Verifikation von Designs durch die Automatisierung von Properties für den formalen Verifikationsprozess gesteigert werden. Darüber hinaus schlägt diese Arbeit neue Verifikationsstrategien für die effektive Verifikation von Hardware mit generierten Properties vor, speziell mit einem Fokus auf die formale Verifikation von Prozessorkernen.

8.1 Modellgetriebene Generierung von Eigenschaften

Formale Eigenschaften (Properties) werden von der Spezifikation in einer Weise abgeleitet, welche die Intention des Designs erfasst, und werden anschließend von einem formalen Verifikationstool benutzt, um die in einer bestimmen Hardwarebeschreibungssprache erstellte Implementierung zu verifizieren. Eine automatische Generierung von Eigenschaften muss den folgenden Anforderungen an einen "guten Satz von Eigenschaften" (Property-Set) genügen:

- Properties sollen aus einer formalen Spezifikationen generiert werden.

- Generierte Properties sollen korrekt und vollständig sein im Hinblick auf die Spezifikationsobjekte.
- Ein Set von Properties sollte alle Spezifikationsobjekte abbilden.
- Ein Set von Properties sollte zu seinen Spezifikationsobjekten rückverfolgbar sein.
- Der Generierungsprozess von Properties sollte das 4-Augen-Prinzip befolgen.
- Ein generiertes Set von Properties sollte verschiedene Verifikationsverfahren unterstützen.
- Ein Set von Properties sollte in einer für Menschen lesbaren Form formuliert sein.

Ein möglicher Ansatz ist das vorlagen-basierte Verfahren. Dabei kann die Übersetzung von der Spezifikation zu Properties anhand einer Vorlagen-Bibliothek realisiert werden, wie es zum Beispiel die Mako Vorlage-Bibliothek für Python ist. Nichtsdestotrotz ist solch ein vorlagenbasiertes Verfahren nur auf eine kleine Untermenge von Designs anwendbar. Um die Nachteile des vorlagen-basierten Verfahrens auszugleichen und die oben gennanten Anforderungen zu erfüllen wurde die modellgetriebene Generierung von Properties entwickelt. Dieses Verfahren adaptiert die Model-Getriebene Architektur (MGA) für Codegenerierung aus der Object Management Group (OMG). Der Prozess folgt konzeptionell OMG's MGA Vorschlag, führt aber zusätzlich neue Begriffe für verschiedene hardwarebezogene Modelle ein. Die eingeführten Modelle sind Model-der-Dinge, Model-der-Eigenschaften und Model-der-Sicht. Jedes Model gehört zu einer bestimmten Model-Ebene im Generierungssprozess. Jede Model-Ebene bezieht sich auf eine spezifische Aufgabe in der Propertygenerierung. Die Aufgabentrennung durch die Model-Ebenen garantiert eine modulare Prozessentwicklung, und gestattet eine unkomplizierte Weiterentwicklung und Erweiterung der Funktionalitäten. Die Properties werden durch eine Abfolge von Model-zu-Model Transformationen zwischen diesen Model-Ebenen generiert. Python wird als domainspezifische Sprache für die Beschreibung der Zwischentransformationen genutzt.

Für die Generierung der Infrastruktur, die das Beschreibung von Transformationen vereinfacht, wird ein Metamodel-basiertes Automatisierungsframework angewendet. Die Programmierschnittstelle, die den zentralen Teil der Infrastruktur bildet, wird aus den Metamodel-Definitionen der obengenannten Modellen generiert. Die generierte Programmierschnittstelle wird erweitert durch eine domain-spezifische Programmierschnittstelle, um den Aufwand für die Entwicklung der Transformationen zu reduzieren. Die in dieser Thesis entwickelte Generierungslösung von Properties wird "MetaProp" genannt.

Ein wesentlicher Aspekt des Generierungsprozesses von Properties ist die Übersetzung von informellen Spezifikationen in formale Spezifikationsmodelle. Aufgrund der Verschiedenartigkeit von Hardwareentwürfen stellt das Verfahren unterschiedliche Modellierungsparadigmen für das Formalisieren der Spezifikationenbereit. Das sogenannte "MetaExpression" Metamodell bietet Unterstützung zur Beschreibung des Verhaltens von kombinatorischen Designs in Form von Expression-bäumen und Dataflow-Ausdrücken. Das MetaExpression Metamodel ist modularer Natur und kann in anderen Metamodelsbeschreibungen integriert werden, die die Konfiguration des Designs auf Spezifikationsebene beschreiben. Für die Modellierung des Verhaltens von sequentiellen Designs wird ein Formalismus für Traces eingeführt, dessen Notation verwandt mit der endlicher Automaten ist. Das Metamodel gennant "MetaSTS" beschreibt diesen Formalismus. Das MetaSTS Metamodel erlaubt die Definition des Verhaltens sequentieller Designs, mithilfe annotierter Timing-Informationen für die jeweiligen Übergänge zwischen relevanten Zuständen. Diese Annotationen werden auch benutzt, um die abstrahierten Zustände im Model-der-Dinge auf das Model-der-Eigenschaften und schließlich auf die Entwurfsimple-

mentierung abzubilden. Solche Annotationen oder Verbindungsmechanismen erlauben es, das Model-der-Eigenschaften auf unterschiedlichen Entwurfsimplementierungen anzuwenden.

8.2 Prozessorverifizierung durch C-S^{2}QED

Die Propertygenerierung für formale Verifikation von Hardware-Entwürfen liefert ein effizientes und effektives Verfahren für Entwurfverifikation. Nichtsdestotrotz ist formale Verifikation von Prozessorkernen eine enorme Herausforderung. Die Mikroarchitektur von Prozessorkernen ist optimiert für Leistung, Fläche, Timing, Energieverbrauch, Exceptionbehandlung, etc. Aufgrund dieser Optimierungen ist sowohl ein erheblicher manueller Aufwand zu treiben, als sind auch tiefgehende Kenntnisse über mikroarchitekturelle Details vonnöten, um die Transformationen, die das Model-der-Eigenschaften definieren, zu beschreiben. Der Hauptgrund für diese Problematik ist die überschneidende Ausführung von Instruktionen innerhalb der gepipelineten Architekturen von Prozessoren und zahlreiche interne und externe Pipeline Stall Szenarien.

Um formale Verifikation auch von Prozessorkernen zu ermöglichen, schlägt diese Arbeit ein vollständiges Verfahren für die Prozessorverifikation vor, das auf der obengenannten Generierung basiert. Das eingeführte Verfahren für die Modellierung der Spezifikation wird für die Formalisierung der Instruktion-Set-Architektur (ISA) und das Verhalten von Instruktionen innerhalb der Prozessorpipeline angewendet. Für die vollständige Verifikation des Prozessors muss ein Set von generierten Properties alle Kombinationen von Instruktionen und alle Szenarien von Pipeline Stalls betrachten. Um das zu ermöglichen, muss das Model-der-Eigenschaften, aus dem die Properties generiert werden und die das erwartete zeitliche Verhalten des Designs, alle oben genannten Kombinationen von Instruktionen in allen möglichen Pipeline-Szenarien modellieren. Hierbei ist Verifikationsexpertise vonnöten, damit sichergestellt wrid, dass durch die generierten Properties keine ungewollten Einschränkungen des Entwurfsraums entstehen. Um diese Herausforderungen zu bewältigen, wurde das C-S^{2}QED Verfahren, eine Erweiterung des S^{2}QED Verfahrens, zur vollständigen Verifikation eines Prozessors entwickelt. Das C-S^{2}QED Verfahren ist sowohl für Ausnahmebehandlung (Exception Handling) innerhalb der Prozessorpipeline als auch für Superskalare Pipeline Architekturen geeignet. Es findet alle funktionalen Fehler in einer Prozessorimplementierung und erfordert deutlich weniger Aufwand im Vergleich zu bestehenden Prozessorverifikationsverfahren. Die Garantie der vollständigen Verifikation für das C-S^{2}QED-Verfahren basiert auf dem Vollständigkeitkriterium von C-IPC, einer bereits industriell eingesetzten vollständigen formalen Verifikationsmethodik. Sowohl das C-S^{2}QED-Verfahren als auch der Beweis seiner Vollständigkeit sind wesentliche Beiträge der vorliegenden Arbeit. Die Effektivität des Verfahrens und die Produktivität bei der Verifikation wird durch die automatische Generierung von C-S^{2}QED-Properties erheblich gesteigert.

8.3 Anwendung auf Industriedesigns

Die Anwendbarkeit und Effektivität der eingeführten Modellierungsparadigmen und entwickelten Verfahren wurden anhand der formalen Verifikation mehrerer industrieller Designs bewiesen. Zahlreiche logische Fehler, einschließlich vieler Fehler, die bekanntermaßen schwer zu entdecken sind, konnten während des formalen Verifikationprozesses mit generierten Properties gefunden werden. Abbildung 8.1 zeigt das Layout des “Rival” SoC. Die Mehrzahl der gezeigten IPs einschließlich des RISC-V-Kerns, ohne die legacy IPs, wurden ausschließlich mit

dem in dieser Arbeit vorgeschlagenen Verfahren verifiziert. Der Rival SoC wird in Powertrain und Safety Automotive Anwendungen eingesetzt. Der gefertigte Chip funktionierte “first time right”, ohne dass ein logischer Fehler bei post-Fertigungs-Tests gefunden wurde.

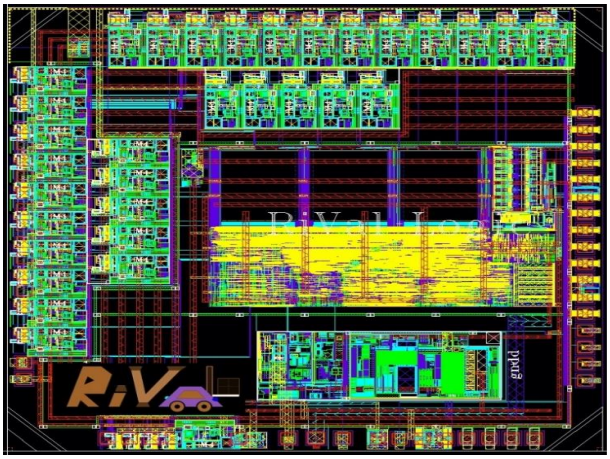


Abbildung 8.1: Layout des Rival chip

Unterschiedliche architekturelle Alternativen von RISC-V-basierten Prozessorentwürfen wurden mit den generierten C-S^{2}QED Properties verifiziert. Die Propertygenerierung ist konfigurierbar, so dass Änderungen in der Mikroarchitektur des Prozessors, verursacht von Änderungen der Spezifikation, automatisch vom Generierungsprozess abgedeckt sind. Dadurch sind keine weiteren manuellen Eingriffe notwendig. Neben der Verifikation von Prozessorkernen wurden die vorgeschlagenen Verfahren auch auf Communication Protocol IPs, Bus Bridges, Interrupt Controller und sicherheitsrelevante Designs angewendet.

8.4 Fazit

Die wissenschaftlichen Beiträge dieser Arbeit lassen sich folgendermaßen zusammenfassen: Die Prinzipien der Model-getriebenen Architektur wurden angewendet, um die Verifikationproduktivität von Hardware Designs zu steigern. Gleichzeitig erreicht man durch die neuen Verfahren eine hohe Qualität der verifizierten Designs, aufgrund der Vollständigkeit der formalen Verifikation. Neue Modellierungsparadigmen wurden eingeführt, um die Hardware Design Spezifikationen zu formalisieren. Hervorzuheben ist eine neue FSM-basierte Notation für Traces, die das implizite Design Know-How formalisiert. Ein automatischer Annotationsmechanismus wurde eingeführt, um Properties für unterschiedliche Designarchitektur-Alternativen ohne zusätzlichen Aufwand zu generieren. Ein vollständiges Verifikationsverfahren, nämlich C-S^{2}QED, wurde eingeführt. Dieses behandelt sowohl Exceptions als auch Superskalare Architekturen. Für das C-S^{2}QED Verfahren wurde gezeigt, dass es alle funktionalen Fehler in einem Prozessor findet. Die Propertygenerierung ist zusätzlich auch auf Communication und House-Keeping IPs anwendbar. Die eingeführten Verfahren können auf unterschiedlichste Designs angewendet werden, einschließlich sicherheitsrelevante IPs, sowie Fehlerkorrektur- und CRC-Designs. Die Anwendbarkeit der eingeführten Modellierungparadigmen und die Propertygenerierung sind auf industriellem Niveau.

Bibliography

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

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

[3] ALTERA. Hardware/Software Co-Verification Using FPGA Platforms. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01070-hardware-software-coverification-fpga.pdf`. [Online: Accessed on 20.September.2020].

[4] ARM. AHB Example AMBA SYstem - Technical Reference Manual. `https://static.docs.arm.com/ddi0170/a/DDI0170.pdf`. [Online: Accessed on 15.August.2020].

[5] ARM. ARM®A64 Instruction Set Architecture, 2018. `https://static.docs.arm.com/ddi0596/a/DDI_0596_ARM_a64_instruction_set_architecture.pdf`.

[6] ARNOULDUS, J., BRAND, M., SEREBRENIK, A., AND BRUNEKREEF, J. *Code Generation with Templates*, 1st ed. Springer Science and Business Media, Secaucus, NJ, USA, 2012.

[7] BARANOWSKI, R., AND TRUNZER, M. Complete formal verification of a family of automotive dsps. In *Design & Verification Conference & Exhibition (DVCon) Europe* (2016).

[8] BARTH, P. *A Davis-Putnam Based Enumeration Algorithm for Linear Pseudo-Boolean Optimization*. Research report: Max-Planck-Institut für Informatik. Max-Planck-Inst. für Informatik, Bibliothek & Dokumentation, 1995.

[9] BAVACHE, V. B., ZHAO, H., HARTLIEHE, H., KAJA, E., DEVARAJEGOWDA, K., AND ECKER, W. Automated soc hardening with model transformation. In *2020 Baltic Electronics Conference)* (2020).

[10] BEREZIN, S., BIERE, A., CLARKE, E. M., AND ZHU, Y. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design* (London, UK, 1998), FMCAD '98, Springer-Verlag, pp. 369–386.

[11] BHADRA, J., ABADIR, M. S., WANG, L., AND RAY, S. A survey of hybrid techniques for functional verification. *IEEE Design Test of Computers 24* (2007), pp. 112–122.

[12] BLEM, E., MENON, J., AND SANKARALINGAM, K. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)* (2013), pp. 1–12.

[13] BOFILL, M., NIEUWENHUIS, R., OLIVERAS, A., RODRIGUEZ-CARBONELL, E., AND RUBIO, A. A write-based solver for sat modulo the theory of arrays. In *2008 Formal Methods in Computer-Aided Design* (2008), pp. 1–8.

[14] BORMANN, J. *Vollständige Verifikation (Complete Functional Verification)*. PhD thesis, University of Kaiserslautern, June 2009. `https://kluedo.ub.uni-kl.de/frontdoor/deliver/index/docId/2108/file/diss_joerg_bormann.pdf`.

[15] BORMANN, J., BEYER, S., MAGGIORE, A., SIEGEL, M., SKALBERG, S., BLACKMORE, T., AND BRUNO, F. Complete formal verification of tricore2 and other processors. In *Design and Verification Conference and Exhibition* (2007), DVCon.

[16] BORMANN, J., AND BUSCH, H. Verfahren zur bestimmung der güte einer menge von eigenschaften (method for determining the quality of a set of properties). european patent application, publication number ep1764715, Sept. 2005.

[17] BRAND, D. Verification of large synthesized designs. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)* (1993), pp. 534–537.

[18] BURCH, J. R., AND DILL, D. L. Automatic verification of pipelined microprocessor control. In *Proceedings of the 6th International Conference on Computer Aided Verification* (London, UK, UK, 1994), CAV '94, Springer-Verlag, pp. 68–80.

[19] CADENCE DESIGN SYSTEMS, INC. JasperGold Apps User Guide. `https://www.cadence.com/`. [Online: Accessed 05.December.2019].

[20] CHEN, W., RAY, S., BHADRA, J., ABADIR, M., AND WANG, L. Challenges and trends in modern soc design verification. *IEEE Design Test 34*, 5 (2017), pp. 7–22.

[21] CHIU, P., CELIO, C., ASANOVIĆ, K., PATTERSON, D., AND NIKOLIĆ, B. An out-of-order risc-v processor with resilient low-voltage operation in 28nm cmos. In *2018 IEEE Symposium on VLSI Circuits* (2018), pp. 61–62.

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

[23] CLARKE, E., BIERE, A., RAIMI, R., AND ZHU, Y. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design 19*, 1 (July 2001).

[24] DAMM, W., PNUELI, A., AND RUAH, S. Herbrand automata for hardware verification. In *Proceedings of the 9th International Conference on Concurrency Theory* (Berlin, Heidelberg, 1998), CONCUR '98, Springer-Verlag, pp. 67–83.

[25] DANESE, A., GHASEMPOURI, T., AND PRAVADELLI, G. Automatic extraction of assertions from execution traces of behavioural models. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)* (2015), pp. 67–72.

[26] DAVIS, M., AND PUTNAM, H. A Computing Procedure for Quantification Theory. *J. ACM 7*, 3 (July 1960), pp. 201–215.

[27] DEVARAJEGOWDA, K., AND ECKER, W. On generation of properties from specification. In *2017 IEEE International High Level Design Validation and Test Workshop (HLDVT)* (Oct 2017), pp. 95–98.

[28] 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)* (Oct 2018), pp. 231–236.

[29] DEVARAJEGOWDA, K., ECKER, W., AND KUNZ, W. How to keep 4-eyes principle in a design and property generation flow. In *MBMV 2019; 22nd Workshop - Methods and Description Languages for Modelling and Verification of Circuits and Systems* (April 2019), pp. 1–6.

[30] DEVARAJEGOWDA, K., FADIHEH, M. R., SINGH, E., BARRETT, C., MITRA, S., ECKER, W., STOFFEL, D., AND KUNZ, W. Gap-free processor verification by s2qed and property generation. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)* (2020), pp. 526–531.

[31] DEVARAJEGOWDA, K., HILTL, V., RABENALT, T., STOFFEL, D., KUNZ, W., AND ECKER, W. Formal verification by the book: Error detection and correction codes. In *Design and Verification Conference and Exhibition* (2020), DVCon.

[32] DEVARAJEGOWDA, K., KAJA, E., PREBECK, S., AND ECKER, W. Isa modeling with trace notation for context free property generation. In *2021 Design Automation Conference (DAC)* (2021).

[33] DEVARAJEGOWDA, K., SCHREINER, J., FINDENIG, R., AND ECKER, W. Python based framework for hdsls with an underlying formal semantics: (invited paper). In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (Nov 2017), pp. 1019–1025.

[34] DEVARAJEGOWDA, K., SERVADEI, L., HAN, Z., WERNER, M., AND ECKER, W. Formal verification methodology in an industrial setup. In *2019 22nd Euromicro Conference on Digital System Design (DSD)* (Aug 2019), pp. 610–614.

[35] ECKER, W., DEVARAJEGOWDA, K., WERNER, M., HAN, Z., AND SERVADEI, L. Embedded systems' automation following omg's model driven architecture vision. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)* (March 2019), pp. 1301–1306.

[36] ECKER, W., AND SCHREINER, J. Introducing model-of-things (mot) and model-of-design (mod) for simpler and more efficient hardware generators. In *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)* (Sept 2016), pp. 1–6.

[37] ECKER, W., AND SCHREINER, J. Metamodeling. In *Handbook of Hardware/Software Codesign*, S. Ha and J. Teich, Eds. Springer, 2017, ch. 10, pp. 266–290.

[38] ECKER, W., VELTEN, M., ZAFARI, L., AND GOYAL, A. Metamodeling and Code Generation -The Infineon Approach. In *MeCoES - Metamodelling and Code Generation for Embedded Systems: Workshop with ESWEEK* (2012), W. Mueller and W. Ecker, Eds.

[39] ECKER, W., VELTEN, M., ZAFARI, L., AND GOYAL, A. Metasynthesis for Designing Automotive SoCs. In *2014 51st ACM/EDAC/IEEE Design Automation Conference* (June 2014).

[40] ECKER, W., VELTEN, M., ZAFARI, L., AND GOYAL, A. The metamodeling approach to system level synthesis. In *DATE* (2014), G. Fettweis and W. Nebel, Eds., European Design and Automation Association, pp. 1–2.

[41] EEN, N., MISHCHENKO, A., AND BRAYTON, R. Efficient implementation of property directed reachability. In *2011 Formal Methods in Computer-Aided Design (FMCAD)* (2011), pp. 125–134.

[42] EMERSON, E., AND TREFLER, R. Parametric quantitative temporal reasoning. In *Proceedings 14th Symposium on Logic in Computer Science* (1999), pp. 336 –343.

[43] EMERSON, E. A. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics* (1990).

[44] F. SCHIRRMEISTER. Improving Emulation Throughput for Multi-Project SoC Designs. `https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/system-design-verification/emulation-throughput-wp.pdf`. [Online: Accessed on 20.September.2020].

[45] FADIHEH, M. R., URDAHL, J., NUTHAKKI, S. S., MITRA, S., BARRETT, C., STOFFEL, D., AND KUNZ, W. Symbolic quick error detection using symbolic initial state for pre-silicon verification. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)* (March 2018), pp. 55–60.

[46] FINDENIG, R., LEITNER, T., AND ECKER, W. Single-source hardware modeling of different abstraction levels with State Charts. In *2012 IEEE International High Level Design Validation and Test Workshop (HLDVT)* (2012), pp. 41–48.

[47] FOSTER, H. D. Trends in functional verification: A 2014 industry study. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)* (2015), pp. 1–6.

[48] GLASSER, M. *Open Verification Methodology Cookbook*. Springer Science & Business Media, 2009.

[49] GONZALEZ, D. L. *Error Detection and Correction Codes*. Springer Netherlands, Dordrecht, 2008.

[50] HAN, Z., DEVARAJEGOWDA, K., WERNER, M., AND ECKER, W. Towards a python-based one language ecosystem for embedded systems automation. In *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)* (Oct 2019), pp. 1–7.

[51] HANGAL, S., NARAYANAN, S., CHANDRA, N., AND CHAKRAVORTY, S. IODINE: a tool to automatically infer dynamic invariants for hardware designs. In *Proceedings 42nd Design Automation Conference, 2005.* (June 2005), pp. 775–778.

[52] HANNA, Z. Challenging problems in industrial formal verification. In *2014 Formal Methods in Computer-Aided Design (FMCAD)* (2014), pp. 1–1.

[53] HENNESSY, J., JOUPPI, N., PRZYBYLSKI, S., ROWEN, C., GROSS, T., BASKETT, F., AND GILL, J. Mips: A microprocessor architecture. In *15th Annual Workshop on Microprogramming, MICRO 15, Palo Alto, California, USA: IEEE Press* (1982), pp. 17–22.

[54] HO, Y., MISHCHENKO, A., AND BRAYTON, R. Property directed reachability with word-level abstraction. In *2017 Formal Methods in Computer Aided Design (FMCAD)* (2017), pp. 132–139.

[55] HOLZMANN, G. J. *Explicit-State Model Checking*. Springer International Publishing, Cham, 2018.

[56] HYMANS, C. Verification of an error correcting code by abstract interpretation. In *Verification, Model Checking, and Abstract Interpretation* (Berlin, Heidelberg, 2005), R. Cousot, Ed., Springer Berlin Heidelberg, pp. 330–345.

[57] IMAN, S., AND JOSHI, S. *e Reuse Methodology*. Springer US, Boston, MA, 2004, pp. 267–278.

[58] JEROSLOW, R. G., AND WANG, J. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence 1* (1990), pp. 167–187.

[59] JONES, R. B., DILL, D. L., AND BURCH, J. R. Efficient validity checking for processor verification. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-aided Design* (Washington, DC, USA, 1995), ICCAD '95, pp. 2–6.

[60] KAMKIN, A. S., AND CHUPILKO, M. M. Survey of modern technologies of simulation-based verification of hardware. *Program. Comput. Softw. 37*, 3 (May 2011), pp. 147–152.

[61] KANEKAWA, N., IBE, E. H., SUGA, T., AND UEMATSU, Y. *Dependability in Electronic Systems: Mitigation of Hardware Failures, Soft Errors, and Electro-Magnetic Disturbances*, 1st ed. Springer-Verlag New York, 2011.

[62] KAUFMANN, M., MOORE, J. S., AND BOYER, O. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering 23* (1997), pp. 203–213.

[63] KINNIMENT, D. J., AND KOELMANS, A. Modelling and Verification of Timing Conditions with the Boyer Moore Prover. In *Theorem Provers in Circuit Design, Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, Nijmegen, The Netherlands, 22-24 June 1992, Proceedings* (1992), V. Stavridou, T. F. Melham, and R. T. Boute, Eds., vol. A-10 of *IFIP Transactions*, North-Holland, pp. ßß. 111–127.

[64] KRIPKE, S. A. Semantical considerations for modal logics. Proceedings of a Colloquium on Modal and Many-valued Logics, Helsinki, 23-26 August, 1962, Acta Philosophica Fennica1963. *Journal of Symbolic Logic 34*, 3 (1969).

[65] KROPF, T. *Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems*, 1st ed. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[66] KÜHNE, U., BEYER, S., BORMANN, J., AND BARSTOW, J. Automated formal verification of processors based on architectural models. In *Proc. of Conference on Formal Methods in Computer-Aided Design (FMCAD)* (2010), pp. 129–136.

[67] KUNZ, W. Hannibal: An efficient tool for logic verification based on recursive learning. In *Computer-Aided Design, 1993. ICCAD-93. Digest of Technical Papers., 1993 IEEE/ACM International Conference on* (Nov 1993), pp. 538–543.

[68] KUNZ, W. Verification Of Digital Systems. University lecture, 2014. [Lecture notes - VDS, from Prof. Wolfgang Kunz at TU Kaiserslautern].

[69] KUNZ, W., STOFFEL, D., AND MENON, P. R. Logic optimization and equivalence checking by implication analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 16*, 3 (1997), pp. 266–281.

[70] LECOMTEA, S., GUILLOUARD, S., MOYB, C., LERAY, P., AND SOULARD, P. A co-design methodology based on model driven architecture for real time embedded systems. *Mathematical and Computer Modeling 53*, 3-4 (2011), pp. 471–484.

[71] LIANGORA RESEARCH LAB. What is MDA? Why considering BNPM. `https://research.linagora.com/pages/viewpage.action?pageId=3639295`.

[72] LIN, D., HONG, T., LI, Y., S, E., KUMAR, S., FALLAH, F., HAKIM, N., GARDNER, D. S., AND MITRA, S. Effective post-silicon validation of system-on-chips using quick error detection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 33*, 10 (Oct 2014), pp. 1573–1590.

[73] LUDWIG, T., URDAHL, J., STOFFEL, D., AND KUNZ, W. Properties first - correct-by-construction rtl design in system-level design flows. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019).

[74] LVOV, A., LASTRAS-MONTAÑO, L. A., PARUTHI, V., SHADOWEN, R., AND EL-ZEIN, A. Formal Verification of Error Correcting Circuits Using Computational Algebraic Geometry. In *2012 Formal Methods in Computer-Aided Design (FMCAD)* (Oct 2012), pp. 141–148.

[75] MARX, O., WEDLER, M., STOFFEL, D., KUNZ, W., AND DREYER, A. Proof logging for computer algebra based smt solving. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2013), pp. 677–684.

[76] MCMILLAN, K. L. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

[77] MELLOR, S. J., SCOTT, K., UHL, A., AND WEISE, D. Model-driven architecture. In *Advances in Object-Oriented Information Systems*. Springer, 2002, pp. 290–297.

[78] MENTOR GRAPHICS. Questa Property Generation. `https://www.mentor.com/`. [Online: Accessed 05.December.2019].

[79] MENTOR GRAPHICS. The 2020 Wilson Research Group Functional Verification Study. `https://blogs.sw.siemens.com/verificationhorizons/`. [Online: Accessed 14.October.2020].

[80] MICHAEL J. C. GORDON. The HOL-4 Proof Tool. `http://hol.sf.net`. [Online: Accessed on 05.December.2019].

[81] NGUYEN, M. D., THALMAIER, M., WEDLER, M., BORMANN, J., STOFFEL, D., AND KUNZ, W. Unbounded Protocol Compliance Verification Using Interval Property Checking With Invariants. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27*, 11 (Nov 2008), pp. 2068–2082.

[82] NICOLAIDIS, M. *Soft Errors in Modern Electronic Systems*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[83] NIEUWENHUIS, R., OLIVERAS, A., AND TINELLI, C. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM 53* (2006), pp. 937–977.

[84] "OMG". MDA - The Architecture of Choice for a Changing World, 2016. `http://www.omg.org/mda/`.

[85] ONESPIN SOLUTIONS GMBH. OneSpin 360MV. `https://www.onespin.com/`. [Online: Accessed 05.December.2019].

[86] OWRE, S., RUSHBY, J. M., , AND SHANKAR, N. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)* (Saratoga, NY, jun 1992), D. Kapur, Ed., vol. 607 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 748–752.

[87] PATTERSON, D. A., AND HENNESSY, J. L. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[88] PAULSON, L. C. Isabelle: The next seven hundred theorem provers. *In Ewing L. Lusk and Ross A. Overbeek, editors, 9th International Conference on Automated Deduction, volume 310 of LNCS 23* (1988), pp. 772–773.

[89] PIERRE, L. Describing and verifying synchronous circuits with the boyer-moore theorem prover. In *Correct Hardware Design and Verification Methods* (Berlin, Heidelberg, 1995), P. E. Camurati and H. Eveking, Eds., Springer Berlin Heidelberg, pp. 35–55.

[90] PRICE, D. Pentium fdiv flaw-lessons learned. *IEEE Micro 15*, 2 (1995), pp. 86–88.

[91] REID, A., CHEN, R., DELIGIANNIS, A., GILDAY, D., HOYES, D., KEEN, W., PATHIRANE, A., SHEPHERD, O., VRABEL, P., AND ZAIDI, A. End-to-End Verification of ARM ® Processors with ISA-Formal. In *Proceedings of 28th International Conference on Computer Aided Verification* (2016).

[92] ROGIN, F., KLOTZ, T., FEY, G., DRECHSLER, R., AND RÜLKE, S. Automatic Generation of Complex Properties for Hardware Designs. In *Proceedings of the Conference on Design, Automation and Test in Europe* (New York, NY, USA, 2008), DATE '08, ACM, pp. 545–548.

[93] ROSS, I. M. The invention of the transistor. *Proceedings of the IEEE 86*, 1 (1998), pp. 7–28.

[94] ROSSI, D., TIMONCINI, N., SPICA, M., AND METRA, C. Error Correcting Code Analysis for Cache Memory High Reliability and Performance. *2011 Design, Automation and Test in Europe* (2011), pp. 1–6.

[95] ROY, S. K., AND RAMESH, S. Functional verification of system on chips - practices, issues and challenges. In *Proceedings of ASP-DAC/VLSI Design 2002. 7th Asia and South Pacific Design Automation Conference and 15h International Conference on VLSI Design* (2002), pp. 11–13.

[96] RUF, J., HOFFMANN, D. W., KROPF, T., AND ROSENSTIEL, W. Simulation-guided property checking based on multi-valued ar-automata. In *Proc. International Conference on Design, Automation and Test in Europe (DATE)* (2001), pp. 742 –748.

[97] SCHREINER, J., FINDENIG, R., AND ECKER, W. Design Centric Modeling of Digital Hardware. In *IEEE International High Level Design Validation and Test Workshop, HLDVT 2016* (2016), pp. 46–52.

[98] SCHREINER, J., WILLGERODT, F., AND ECKER, W. A new approach for generating view generators. In *Design and Verification Conference - US* (Feb 2017). `http://events.dvcon.org/2017/proceedings/papers/04P_18.pdf`.

[99] SELIGMAN, E., SCHUBERT, T., AND KUMAR, M. V. A. K. *Formal Verification, An Essential Toolkit For Modern VLSI Design*. Morgan Kaufmann Publishers, 2015.

[100] SHANLEY, T. *X86 Instruction Set Architecture*. Mindshare Press, 2010.

[101] SHERIDAN, D., LIU, L., KIM, H., AND VASUDEVAN, S. A Coverage Guided Mining Approach for Automatic Generation of Succinct Assertions. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems* (Jan 2014), pp. 68–73.

[102] SIEGEL, J. M. Model Driven Architecture ® (MDA): The MDA Guide Rev 2.0, june 2014. https://www.omg.org/mda/presentations.htm.

[103] SINGH, E., DEVARAJEGOWDA, K., SIMON, S., SCHNIEDER, R., GANESAN, K., FADIHEH, M., STOFFEL, D., KUNZ, W., BARRETT, C., ECKER, W., AND MITRA, S. Symbolic qed pre-silicon verification for automotive microcontroller cores: Industrial case study. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)* (March 2019), pp. 1000–1005.

[104] SINGH, E., LIN, D., BARRETT, C., AND MITRA, S. Logic bug detection and localization using symbolic quick error detection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018).

[105] SMITH, J. E., AND SOHI, G. S. The microarchitecture of superscalar processors. *Proceedings of the IEEE 83*, 12 (1995), pp. 1609–1624.

[106] SPARC INTERNATIONAL INC. The SPARC Architecture Manual, Version 8, 1994.

[107] STOFFEL, D., AND W.KUNZ. Record and play: a structural fixed point iteration for sequential circuit verification. In *1997 Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)* (1997), pp. 394–399.

[108] SYNOPSYS. VC Formal Apps. https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html. [Online: Accessed on 05.December.2019].

[109] THALMAIER, M., NGUYEN, M. D., WEDLER, M., STOFFEL, D., BORMANN, J., AND KUNZ, W. Analyzing k-step induction to compute invariants for sat-based property checking. In *Proceedings of the 47th Design Automation Conference* (New York, NY, USA, 2010), DAC'10, Association for Computing Machinery, pp. 176–181.

[110] TRUYEN, F. The fast Guide to Model Driven Architecture. http://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf, 2006.

[111] URDAHL, J. Path predicate abstraction for sound system-level modeling of digital circuits, June 2016. http://www.odbms.org/wp-content/uploads/2017/08/2015-Urdahl-1.pdf.

[112] URDAHL, J., STOFFEL, D., BORMANN, J., WEDLER, M., AND KUNZ, W. Path predicate abstraction by complete interval property checking. In *Formal Methods in Computer Aided Design* (Oct 2010), pp. 207–215.

[113] URDAHL, J., UDUPI, S., LUDWIG, T., STOFFEL, D., AND KUNZ, W. Properties first? a new design methodology for hardware, and its perspectives in safety analysis. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (Nov 2016), pp. 1–8.

[114] URIBE, T. E. Combinations of model checking and theorem proving. In *Frontiers of Combining Systems* (Berlin, Heidelberg, 2000), H. Kirchner and C. Ringeissen, Eds., Springer Berlin Heidelberg, pp. 151–170.

[115] VASUDEVAN, S., SHERIDAN, D., PATEL, S., TCHENG, D., TUOHY, B., AND JOHNSON, D. GoldMine: Automatic assertion generation using data mining and static analysis. In *DATE 2010* (March 2010), pp. 626–629.

[116] VISALLI, G. UVM-based verification of ECC module for flash memories. In *2017 European Conference on Circuit Theory and Design (ECCTD)* (Sep. 2017), pp. 1–4.

[117] WATERMAN, A., AND ASANOVIC, K. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, May 2017. `https://riscv.org/specifications`.

[118] WEDLER, M., STOFFEL, D., BRINKMANN, R., AND KUNZ, W. A normalization method for arithmetic data-path verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 26*, 11 (2007), pp. 1909–1922.

[119] WEDLER, M., STOFFEL, D., AND KUNZ, W. Arithmetic Reasoning in DPLL-based SAT Solving. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition* (Feb 2004), vol. 1, pp. 30–35 Vol.1.

[120] WILLE, B., GROSS, J. C., AND ROESNER, W. *Comprehensive Functional Verification*. Morgan Kaufmann Publishers, 2005.

[121] XILINX. Xilinx FPGA Platforms. `https://www.xilinx.com/`. [Online: Accessed on 05.December.2019].

[122] ZHANG, T., SAAB, D., AND ABRAHAM, J. A. Automatic Assertion Generation for Simulation, Formal Verification and Emulation. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (2017), pp. 471–476.

[123] ZHAO, J., AND HARRIS, I. G. Automatic assertion generation from natural language specifications using subtree analysis. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)* (2019), pp. 598–601.

Appendix A

List of symbols

Symbol	Description
$\mathbb{M}$	logical model of a system/design
$\mathbb{M}_{\mathbb{M}}$	metamodel or definition of a model
$\mathbb{O}$	finite set of objects in the model
$\mathcal{M}$	mathematical model of a system/design in a model checker
$\mathcal{S}$	finite set of states
$\mathcal{S}_i$	finite set of initial states
I	finite set of input symbols
O	finite set of output symbols
δ	transition function
λ	output function
$\mathcal{K}$	a Kripke structure
$\mathcal{A}$	finite set of atomic formulas
$\mathcal{R}$	transition relation
$\mathcal{L}$	evaluation function
$\mathcal{T}$	a trace structure
Δ	sequence of state transitions
Λ	sequence of output evaluations
θ	trace function
A	modal operator *always*
E	modal operator *exists*

G	temporal operator *globally*
F	temporal operator *finally*
U	temporal operator *until*
W	temporal operator *weak until*
X	temporal operator *next*
R	temporal operator *release*
ϕ	a temporal formula
π	path in a linear time model

Appendix B

List of Notations

Notation	Example	Description
$\models$	$\mathcal{M} \models \phi$	$\mathcal{M}$ models ϕ
$\not\models$	$\mathcal{M} \not\models \phi$	$\mathcal{M}$ does not model ϕ (or) $\mathcal{M}$ models $\neg\phi$
$\mapsto$	$\mathcal{S} \mapsto O$	$\mathcal{S}$ maps to O
$:=$	$\lambda := \mathcal{S} \mapsto O$	λ is defined by $\mathcal{S} \mapsto O$
$\equiv$	$\phi_1 \equiv \phi_2$	ϕ_1 equivalent to ϕ_2
$>$	$x > y$	x greater than y
$\geq$	$x \geq y$	x is greater than or equal to y
$<$	$x < y$	x is less than y
$\leq$	$x \leq y$	x is less than or equal to y
$\wedge$	$a \wedge b$	logical conjunction, $a \wedge b$ is true, iff a = true and b = true
$\vee$	$a \vee b$	logical disjunction, $a \vee b$ is true, iff a = true or b = true
$\neg$	$\neg a$	negation, if a is true then $\neg a$ is false
$\implies$	$p \implies q$	implies, if p is true then q is true
$\subseteq$	$\mathcal{S}_i \subseteq \mathcal{S}$	subset or equal to
$\iff$	$s \models \neg\phi \iff s \not\models \phi$	s models $\neg\,\phi$ if and only if s does not model ϕ
$\in$	$s \in \mathcal{S}$	s exists in set $\mathcal{S}$

Appendix C

Supported Operators in Model-of-Property

Operator	Type
Out = NOT(arg1=I)	I = expression : bit, vector Out = expression : bit, vector
Out = CABS(arg1=I)	I = expression : bit, vector Out = expression : bit, vector
Out = CUMINUS(arg1=I)	I = expression : bit, vector Out = expression : bit, vector
Out = SLICE(arg1=I_s, arg2=I_a, arg3=I_b)	I_s = expression : bit, vector I_a = variable : number I_b = variable : number Out = expression : bit, vector
Out = INDEX(arg1=I_s, arg2=I_{idx})	I_s = expression : bit, vector I_{idx} = variable : number Out = literal : {0,1}
Out = REVERSE(arg1=I)	I = expression : bit, vector Out = expression : bit, vector
Out = HEAD(arg1=I_s, arg2=I_{val})	I_s = expression : bit, vector I_{val} = variable : number Out = expression : bit, vector
Out = TAIL(arg1=I_s, arg2=I_{val})	I_s = expression : bit, vector I_{val} = variable : number Out = expression : bit, vector
Out = ROR(arg1=I)	I = expression : bit, vector Out = literal : {0,1}
Out = RAND(arg1=I)	I = expression : bit, vector

	Out = literal : {0,1}
Out = RNAND(arg1=*I*)	*I* = expression : bit, vector Out = literal : {0,1}
Out = RNOR(arg1=*I*)	*I* = expression : bit, vector Out = literal : {0,1}
Out = RXOR(arg1=*I*)	*I* = expression : bit, vector Out = literal : {0,1}
Out = RXNOR(arg1=*I*)	*I* = expression : bit, vector Out = literal : {0,1}
Out = BAND(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = expression : bit, vector
Out = BNAND(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = expression : bit, vector
Out = BOR(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = expression : bit, vector
Out = BNOR(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = expression : bit, vector
Out = BXOR(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = expression : bit, vector
Out = BXNOR(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = expression : bit, vector
Out = LAND(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = literal : {0,1}
Out = LNAND(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = literal : {0,1}
Out = LOR(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = literal : {0,1}
Out = LNOR(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = literal : {0,1}

Out = LXOR(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = literal : {0,1}
Out = LXNOR(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = literal : {0,1}
Out = CPLUS(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = expression : bit, vector
Out = CMINUS(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = expression : bit, vector
Out = CMULT(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = expression : bit, vector
Out = HWMUL(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = expression : bit, vector
Out = CDIV(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = expression : bit, vector
Out = CMOD(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = expression : bit, vector
Out = CREM(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = expression : bit, vector
Out = SIGNEDCAST(arg1=I)	I = expression : bit, vector Out = expression : bit, vector
Out = UNSIGNEDCAST(arg1=I)	I = expression : bit, vector Out = expression : bit, vector
Out = UNSIGNEDCONV(arg1=I)	I = expression : bit, vector Out = expression : bit, vector
Out = SIGNEDCONV(arg1=I)	I = expression : bit, vector Out = expression : bit, vector
Out = HWPLUS(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = expression : bit, vector
Out = RSL(arg1=I_s, arg2=I_{nr})	I_s = expression : bit, vector I_{nr} = variable : number

	Out = expression : bit, vector
Out = RSA(arg1=I_s, arg2=I_{nr})	I_s = expression : bit, vector I_{nr} = variable : number Out = expression : bit, vector
Out = LS(arg1=I_s, arg2=I_{nr})	I_s = expression : bit, vector I_{nr} = variable : number Out = expression : bit, vector
Out = CONCAT(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = expression : bit, vector
Out = MUX(arg1=I_{sel}, arg2=I_a, arg3=I_b)	I_{sel} = literal : {0,1} I_a = expression : bit, vector I_b = expression : bit, vector Out = expression : bit, vector
Out = ISNEG(arg1=I)	I = expression : bit, vector Out = literal : {0,1}
Out = ISPOS(arg1=I)	I = expression : bit, vector Out = literal : {0,1}
Out = LT(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = literal : {0,1}
Out = LTEQ(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = literal : {0,1}
Out = GT(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = literal : {0,1}
Out = GTEQ(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = literal : {0,1}
Out = EQ(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = literal : {0,1}
Out = NEQ(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = literal : {0,1}
Out = HWMINUS(arg1=I_a, arg2=I_b)	I_a = expression : bit, vector I_b = expression : bit, vector Out = expression : bit, vector

Appendix D

Generated Properties for AHB-to-APB bridge

```
1  ================================================================================
2   PROPERTIES START HERE
3  ================================================================================
4  property transition_reset is
5  local trigger:rose(clk);
6
7  assume:
8    reset_sequence;
9  prove:
10    at t + 0: (comp_ahb2apb_fsm/state = "000") and
11              (psel = 0) and
12              (penable = 0) and
13              (comp_ahb2apb_fsm/busy = 0) and
14              (pwrite = 0) and
15              (comp_ahb2apb_fsm/hready = 1) and
16              (clk = pclk) and
17              (pslverr = HRESP) and
18              (pwdata = 0) and
19              (paddr = 0) and
20              (comp_ahb2apb_fsm/hwrite = 0) and
21              (HRDATA = ((0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) & prdata));
22    right_hook: t + 0;
23 end property;
24 --================================================================================
25
26 property transition_idle_idle is
27 local trigger:rose(clk);
28 disable iff:(rst);
29
30 assume:
31    at t: (comp_ahb2apb_fsm/state = "000") and
32       ((HTRANS <= 1) or (HSEL = 0));
33 prove:
34    at t + 1: (comp_ahb2apb_fsm/state = "000") and
35              (psel = 0) and
36              (penable = 0) and
37              (comp_ahb2apb_fsm/busy = 0) and
38              (pwrite = 0) and
39              (comp_ahb2apb_fsm/hready = 1) and
40              (clk = pclk) and
41              (pslverr = HRESP) and
42              (paddr = prev(paddr)) and
43              (pwdata = prev(pwdata)) and
44              (HRDATA = ((0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) & prdata));
45    right_hook: t + 1;
46 end property;
47 --================================================================================
```

```
48
49  property transition_idle_sel is
50  local trigger:rose(clk);
51  disable iff:(rst);
52
53  assume:
54     at t: (comp_ahb2apb_fsm/state = "000") and
55              (HTRANS > 1) and
56              (HSEL = 1);
57  prove:
58     at t + 1: (comp_ahb2apb_fsm/state = "001") and
59                  (clk = pclk) and
60                  (pslverr = HRESP) and
61                  (psel = 1) and
62                  (penable = 0) and
63                  (comp_ahb2apb_fsm/busy = 1) and
64                  (pwrite = 0) and
65                  (comp_ahb2apb_fsm/hready = 0) and
66                  (paddr = prev(paddr)) and
67                  ((comp_ahb2apb_fsm/hwrite = 1) or (comp_ahb2apb_fsm/hwrite = 0)) and
68                  (pwdata = prev(pwdata)) and
69                  (HRDATA = ((0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) & prdata));
70     right_hook: t + 1;
71  end property;
72  --============================================================================================
73
74  property transition_sel_read is
75  local trigger:rose(clk);
76  disable iff:(rst);
77
78  assume:
79     at t: (comp_ahb2apb_fsm/state = "001") and
80              (comp_ahb2apb_fsm/hwrite = 0) and
81              (pready = 1);
82  prove:
83     at t + 1: (comp_ahb2apb_fsm/state = "011") and
84                  (clk = pclk) and
85                  (pslverr = HRESP) and
86                  (psel = 1) and
87                  (penable = 1) and
88                  (comp_ahb2apb_fsm/busy = 1) and
89                  (pwrite = 0) and
90                  (comp_ahb2apb_fsm/hready = 1) and
91                  (paddr = (0 & prev((HADDR(15 downto 1))))) and
92                  (comp_ahb2apb_fsm/hwrite = prev(comp_ahb2apb_fsm/hwrite)) and
93                  (pwdata = prev(pwdata)) and
94                  (HRDATA = ((0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) & prdata));
95     right_hook: t + 1;
96  end property;
97  --============================================================================================
98
99  property transition_sel_write is
100 local trigger:rose(clk);
101 disable iff:(rst);
102
103 assume:
104    at t: (comp_ahb2apb_fsm/state = "001") and
105             (comp_ahb2apb_fsm/hwrite = 1) and
106             (pready = 1);
107 prove:
108    at t + 1: (comp_ahb2apb_fsm/state = "010") and
109                 (clk = pclk) and
110                 (pslverr = HRESP) and
111                 (psel = 1) and
112                 (penable = 1) and
113                 (comp_ahb2apb_fsm/busy = 1) and
114                 (pwrite = 1) and
115                 (comp_ahb2apb_fsm/hready = 1) and
116                 (paddr = (0 & prev((HADDR(15 downto 1))))) and
117                 (comp_ahb2apb_fsm/hwrite = prev(comp_ahb2apb_fsm/hwrite)) and
```

```
118                 (pwdata = prev(pwdata)) and
119                 (HRDATA = ((0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) & prdata));
120     right_hook: t + 1;
121 end property;
122 --=============================================================================================
123
124 property transition_sel_sel is
125 local trigger:rose(clk);
126 disable iff:(rst);
127
128 assume:
129     at t: (comp_ahb2apb_fsm/state = "001") and
130             (pready = 0) and
131             ((comp_ahb2apb_fsm/hwrite = 1) or (comp_ahb2apb_fsm/hwrite = 0));
132 prove:
133     at t + 1: (comp_ahb2apb_fsm/state = "001") and
134                 (clk = pclk) and
135                 (pslverr = HRESP) and
136                 (psel = 1) and
137                 (penable = 0) and
138                 (comp_ahb2apb_fsm/busy = 1) and
139                 (pwrite = 0) and
140                 (comp_ahb2apb_fsm/hready = 0) and
141                 (paddr = (0 & prev((HADDR(15 downto 1))))) and
142                 (pwdata = prev(pwdata)) and
143                 (HRDATA = ((0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) & prdata));
144     right_hook: t + 1;
145 end property;
146 --=============================================================================================
147
148 property transition_read_idle is
149 local trigger:rose(clk);
150 disable iff:(rst);
151
152 assume:
153     at t: (comp_ahb2apb_fsm/state = "011") and
154             (pready = 1) and
155             ((HSEL = 0) or (HTRANS <= 1));
156 prove:
157     at t + 1: (comp_ahb2apb_fsm/state = "000") and
158                 (clk = pclk) and
159                 (pslverr = HRESP) and
160                 (psel = 0) and
161                 (penable = 0) and
162                 (comp_ahb2apb_fsm/busy = 0) and
163                 (pwrite = 0) and
164                 (comp_ahb2apb_fsm/hready = 1) and
165                 (paddr = prev(paddr)) and
166                 (HRDATA = ((0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) & prdata)) a
167                 (pwdata = prev(pwdata));
168     right_hook: t + 1;
169 end property;
170 --=============================================================================================
171
172 property transition_write_idle is
173 local trigger:rose(clk);
174 disable iff:(rst);
175
176 assume:
177     at t: (comp_ahb2apb_fsm/state = "010") and
178             (pready = 1) and
179             ((HTRANS <= 1) or (HSEL = 0));
180 prove:
181     at t + 1: (comp_ahb2apb_fsm/state = "000") and
182                 (clk = pclk) and
183                 (pslverr = HRESP) and
184                 (psel = 0) and
185                 (penable = 0) and
186                 (comp_ahb2apb_fsm/busy = 0) and
187                 (pwrite = 0) and
```

```
188                 (comp_ahb2apb_fsm/hready = 1) and
189                 (paddr = prev(paddr)) and
190                 (pwdata = prev((HWDATA(15 downto 0)))) and
191                 (HRDATA = ((0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) & prdata));
192     right_hook: t + 1;
193   end property;
194   --==========================================================================================
195
196   property transition_read_sel is
197   local trigger:rose(clk);
198   disable iff:(rst);
199
200   assume:
201     at t: (comp_ahb2apb_fsm/state = "011") and
202               (pready = 1) and
203               (HTRANS > 1) and
204               (HSEL = 1);
205   prove:
206     at t + 1: (comp_ahb2apb_fsm/state = "001") and
207                 (clk = pclk) and
208                 (pslverr = HRESP) and
209                 (psel = 1) and
210                 (penable = 0) and
211                 (comp_ahb2apb_fsm/busy = 1) and
212                 (pwrite = 0) and
213                 (comp_ahb2apb_fsm/hready = 0) and
214                 ((comp_ahb2apb_fsm/hwrite = 1) or (comp_ahb2apb_fsm/hwrite = 0)) and
215                 (HRDATA = ((0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) & prdata)) a
216                 (paddr = prev(paddr)) and
217                 (pwdata = prev(pwdata));
218     right_hook: t + 1;
219   end property;
220   --==========================================================================================
221
222   property transition_write_sel is
223   local trigger:rose(clk);
224   disable iff:(rst);
225
226   assume:
227     at t: (comp_ahb2apb_fsm/state = "010") and
228               (HTRANS > 1) and
229               (HSEL = 1) and
230               (pready = 1);
231   prove:
232     at t + 1: (comp_ahb2apb_fsm/state = "001") and
233                 (clk = pclk) and
234                 (pslverr = HRESP) and
235                 (psel = 1) and
236                 (penable = 0) and
237                 (comp_ahb2apb_fsm/busy = 1) and
238                 (pwrite = 0) and
239                 (comp_ahb2apb_fsm/hready = 0) and
240                 (paddr = prev(paddr)) and
241                 (HRDATA = ((0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) & prdata)) a
242                 (pwdata = prev((HWDATA(15 downto 0)))) and
243                 ((comp_ahb2apb_fsm/hwrite = 1) or (comp_ahb2apb_fsm/hwrite = 0));
244     right_hook: t + 1;
245   end property;
246   --==========================================================================================
247
248   property transition_write_wait is
249   local trigger:rose(clk);
250   disable iff:(rst);
251
252   assume:
253     at t: (comp_ahb2apb_fsm/state = "010") and
254               (pready = 0);
255   prove:
256     at t + 1: (comp_ahb2apb_fsm/state = "100") and
257                 (clk = pclk) and
```

```
258                 (pslverr = HRESP) and
259                 (psel = 1) and
260                 (penable = 1) and
261                 (comp_ahb2apb_fsm/busy = 1) and
262                 (pwrite = 1) and
263                 (comp_ahb2apb_fsm/hready = 0) and
264                 (HRDATA = ((0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) & prdata)) a
265                 (paddr = prev(paddr)) and
266                 (pwdata = prev((HWDATA(15 downto 0))));
267     right_hook: t + 1;
268 end property;
269 --=============================================================================================
270
271 property transition_wr_wait_wr_wait is
272 local trigger:rose(clk);
273 disable iff:(rst);
274
275 assume:
276     at t: (comp_ahb2apb_fsm/state = "100") and
277             (pready = 0);
278 prove:
279     at t + 1: (comp_ahb2apb_fsm/state = "100") and
280                 (clk = pclk) and
281                 (pslverr = HRESP) and
282                 (psel = 1) and
283                 (penable = 1) and
284                 (comp_ahb2apb_fsm/busy = 1) and
285                 (pwrite = 1) and
286                 (comp_ahb2apb_fsm/hready = 0) and
287                 (HRDATA = ((0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) & prdata)) a
288                 (paddr = prev(paddr)) and
289                 (pwdata = prev(pwdata));
290     right_hook: t + 1;
291 end property;
292 --=============================================================================================
293
294 property transition_wr_wait_idle is
295 local trigger:rose(clk);
296 disable iff:(rst);
297
298 assume:
299     at t: (comp_ahb2apb_fsm/state = "100") and
300             (pready = 1) and
301             ((HTRANS <= 1) or (HSEL = 0));
302 prove:
303     at t + 1: (comp_ahb2apb_fsm/state = "000") and
304                 (clk = pclk) and
305                 (pslverr = HRESP) and
306                 (psel = 0) and
307                 (penable = 0) and
308                 (comp_ahb2apb_fsm/busy = 0) and
309                 (pwrite = 0) and
310                 (comp_ahb2apb_fsm/hready = 1) and
311                 (HRDATA = ((0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) & prdata)) a
312                 (paddr = prev(paddr)) and
313                 (pwdata = prev(pwdata));
314     right_hook: t + 1;
315 end property;
316 --=============================================================================================
317
318 property transition_wr_wait_sel is
319 local trigger:rose(clk);
320 disable iff:(rst);
321
322 assume:
323     at t: (comp_ahb2apb_fsm/state = "100") and
324             (HTRANS > 1) and
325             (HSEL = 1) and
326             (pready = 1);
327 prove:
```

```
328     at t + 1: (comp_ahb2apb_fsm/state = "001") and
329             (clk = pclk) and
330             (pslverr = HRESP) and
331             (psel = 1) and
332             (penable = 0) and
333             (comp_ahb2apb_fsm/busy = 1) and
334             (pwrite = 0) and
335             (comp_ahb2apb_fsm/hready = 0) and
336             (paddr = prev(paddr)) and
337             ((comp_ahb2apb_fsm/hwrite = 1) or (comp_ahb2apb_fsm/hwrite = 0)) and
338             (pwdata = prev(pwdata)) and
339             (HRDATA = ((0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) & prdata));
340     right_hook: t + 1;
341 end property;
342 --============================================================================================
343
344 property transition_read_wait is
345 local trigger:rose(clk);
346 disable iff:(rst);
347
348 assume:
349     at t: (comp_ahb2apb_fsm/state = "011") and
350             (pready = 0);
351 prove:
352     at t + 1: (comp_ahb2apb_fsm/state = "101") and
353             (clk = pclk) and
354             (pslverr = HRESP) and
355             (psel = 1) and
356             (penable = 1) and
357             (comp_ahb2apb_fsm/busy = 1) and
358             (pwrite = 0) and
359             (comp_ahb2apb_fsm/hready = 0) and
360             (HRDATA = ((0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) & prdata)) a
361             (paddr = prev(paddr)) and
362             (pwdata = prev(pwdata));
363     right_hook: t + 1;
364 end property;
365 --============================================================================================
366
367 property transition_rd_wait_rd_wait is
368 local trigger:rose(clk);
369 disable iff:(rst);
370
371 assume:
372     at t: (comp_ahb2apb_fsm/state = "101") and
373             (pready = 0);
374 prove:
375     at t + 1: (comp_ahb2apb_fsm/state = "101") and
376             (clk = pclk) and
377             (pslverr = HRESP) and
378             (psel = 1) and
379             (penable = 1) and
380             (comp_ahb2apb_fsm/busy = 1) and
381             (pwrite = 0) and
382             (comp_ahb2apb_fsm/hready = 0) and
383             (paddr = prev(paddr)) and
384             (HRDATA = ((0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) & prdata)) a
385             (pwdata = prev(pwdata));
386     right_hook: t + 1;
387 end property;
388 --============================================================================================
389
390 property transition_rd_wait_idle is
391 local trigger:rose(clk);
392 disable iff:(rst);
393
394 assume:
395     at t: (comp_ahb2apb_fsm/state = "101") and
396             (pready = 1) and
397             ((HTRANS <= 1) or (HSEL = 0));
```

```
prove:
   at t + 1: (comp_ahb2apb_fsm/state = "000") and
             (clk = pclk) and
             (pslverr = HRESP) and
             (psel = 0) and
             (penable = 0) and
             (comp_ahb2apb_fsm/busy = 0) and
             (pwrite = 0) and
             (comp_ahb2apb_fsm/hready = 1) and
             (paddr = prev(paddr)) and
             (pwdata = prev(pwdata)) and
             (HRDATA = ((0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) & prdata));
   right_hook: t + 1;
end property;
--=============================================================================================

property transition_rd_wait_sel is
local trigger:rose(clk);
disable iff:(rst);

assume:
   at t: (comp_ahb2apb_fsm/state = "101") and
         (HTRANS > 1) and
         (HSEL = 1) and
         (pready = 1);
prove:
   at t + 1: (comp_ahb2apb_fsm/state = "001") and
             (clk = pclk) and
             (pslverr = HRESP) and
             (psel = 1) and
             (penable = 0) and
             (comp_ahb2apb_fsm/busy = 1) and
             (pwrite = 0) and
             (comp_ahb2apb_fsm/hready = 0) and
             (paddr = prev(paddr)) and
             ((comp_ahb2apb_fsm/hwrite = 1) or (comp_ahb2apb_fsm/hwrite = 0)) and
             (pwdata = prev(pwdata)) and
             (HRDATA = ((0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0) & prdata));
   right_hook: t + 1;
end property;
--=============================================================================================
-- PROPERTIES END HERE
--=============================================================================================
```

Appendix E

Papers in the Scope of the Thesis

During the course of this doctoral work, a novel approach for formally verifying error detection and correction codes has been proposed. The approach was successfully applied to several real life designs that are used in safety critical applications. Further, an approach to extract control signals of a processor decoder using formal verification tools has been proposed. For both approaches, the property generation flow developed during this doctoral work is applied to improve the overall productivity and quality. The published papers of the proposed approaches are listed in the following.

E.1 Formal Verification by The Book: Error Detection and Correction Codes

Abstract: We present an approach to exhaustively verify the correctness of Error Correction Code designs using formal methods. The proposed approach exploits the linearity of *Syndrome Generators* to prove that the detection and correction of bit errors depends only on the erroneous bit positions, and not on the original data vector. By proving the linearity property, the input analysis space for error detection and correction properties is significantly reduced by a factor of 2^n, where n is the width of data vector. As a result, the proof runtimes of the properties are also reduced. For example, the proof runtime of a 3 bit error detection/correction property requires less than 2 hours for passing on a 256 bit Error Correction Code design in a commercial formal verification tool. The approach has been successfully applied to multiple instances of Error Correction Code designs implemented across several safety-critical automotive system-on-chips. An in-house property generation tool has been employed to foster re-usability and verification productivity. Results show that the proof runtime of the properties leveraging the linearity feature decreases by a factor of 50, and scales proportionally with the width of the data vector and detection and correction capability of the codes.

E.1.1 Introduction

Safety-critical automotive applications have stringent requirements for functional safety and reliability. The devices used in safety-critical applications need to ensure that the basic functionalities of the device are unaffected, even during the deviation from normal working conditions (e.g., magnetic interference, radiation errors, temperature fluctuations, electrical transients, heat

dissipation, etc.). To fulfill these requirements, devices used in safety-critical applications are equipped with additional hardware and software functions, and redundant memory elements. Error Detection and Correction Codes (ECCs) are one such safety mechanism that is widely implemented in memory and register file interfaces to safeguard memory elements against soft errors. Soft errors are errors in logic or data that are caused by radiation, electrical glitches or electro-magnetic interference during the lifespan of a device [82, 61].

ISO 26262, the functional safety standard for automotive products, requires that all features pertaining to safety be rigorously verified. However, the functional verification of ECC circuits poses a formidable challenge. To illustrate, consider the ECC circuit implemented in the program flash interface of an automotive SoC shown in Fig. E.1.

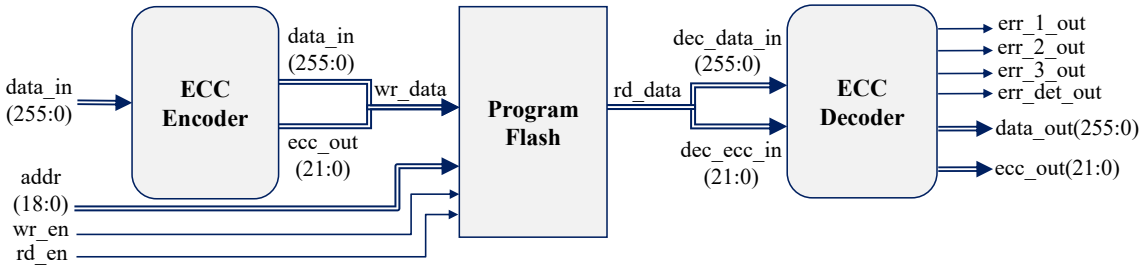


Figure E.1: ECC block in program flash interface

The ECC block shown in Fig. E.1 is a Double-Error Correct and Triple-Error Detect - DECTED module. During a write transfer (`wr_en=1`), the input data bits (`data_in`) are encoded in additional ECC bits (`ecc_out`) following a specific encoding algorithm (e.g., Reed Solomon encoding) [49]. The resulting concatenated codeword `wr_data` is written to the program flash memory. Soft errors may occur during the operation of a device, resulting in a corrupted codeword (non-codeword). When a read transfer (`rd_en=1`) takes place, the non-codeword is passed through the ECC decoder. The decoder computes the error syndrome of the non-codeword and sets the respective error flags depending on the number of bit errors. If there are no bit errors, or if the number of bit errors is within the correctable range, the bit errors are corrected and the `data_out` carries the original data value.

Exhaustive verification of ECC circuits requires to consider every possible input and bit error combinations. The ECC circuit shown in Fig. E.1 has an astronomically high number of input combinations (2^{256}) and an exhaustive analysis covering the entire input space is impossible. In addition, the input stimuli require to consider an exhaustive set of error injections depending on the detection and correction capability of the circuit. For example, the number of bit error combinations for the ECC circuit shown is given by the Eqn. E.1.Because of the huge analysis space (input and error combinations), simulation-based methods including post-silicon debug are inapplicable for the problem of ECC circuits.

$$\binom{278}{3} + \binom{278}{2} + \binom{278}{1} \approx 3.58 * 10^6 \tag{E.1}$$

On the other hand, formal verification (FV) is inherently exhaustive and checks the design behavior against all possible legal input combinations [23, 81]. The requirements of an ECC circuit can be captured in a set of properties. However, the formal engines which implement systematic SAT solving or graph-based canonical representations of the logic with Binary Decision Diagrams (BDD) often run into complexity issues due to the huge analysis space. As a

result, for the type of ECC circuits shown in Fig. E.1 the full proof within a reasonable amount of time cannot be expected.

Although FV tools run into complexity issues when the analysis space is huge, they are well suited to prove the underlying mathematical properties implemented by the RTL designs [119, 65]. ECC decoders implement *Syndrome Generators* to identify the bit positions with an error. The output of a syndrome generator is used to set the corresponding error flags and to correct the data vector. Syndrome generators exhibit the characteristics of a linear algebraic function. The input to a syndrome generator, possibly a non-codeword, can be expressed as the sum of a codeword and an error word. The output of the generator is the superposition of the syndromes of both components, i.e., the syndrome of the codeword (which is necessarily zero) and the syndrome of the error word. Hence, proving the linearity of syndrome generators proves that the syndrome output depends only on the erroneous bit positions and not on the input data vector. Once the linearity of the syndrome generator is proven, the properties capturing the detection and correction capabilities of the circuit are proven by assuming a fixed, arbitrarily chosen data vector.

The approach has been successfully applied to multiple ECC designs implemented across several safety-critical automotive SoCs. The proposed approach together with the automated generation of properties yielded high verification quality and productivity. The property runtimes are at least 50 times faster on big ECC designs (width of data vector > 64 bits). Further, following the approach we are able to achieve convergence (full proof) of properties on a 256 bit ECC design (Fig. E.1) within 2 hours of computation time using a commercial FV tool. Previously, these properties failed to converge after 100 hours of computation time.

The remaining part of the paper is structured as follows: Section E.1.2 provides a brief overview of the existing approaches for ECC verification. Section E.1.3 describes the working of ECC designs and the linearity of syndrome generators. Section E.1.5 describes how the linearity property of syndrome generators can be used to prove the correctness of ECC designs using formal methods. Section E.1.6 describes the automatic generation of properties using an in-house property automation tool. Application of the presented approach on several ECC designs and the property runtimes are detailed in Section E.1.7. A brief summary of the work in Section E.1.8 completes the paper.

E.1.2 Related Work

ECC designs are typically verified by following the current industry standards for pre-silicon verification, simulation and formal methods. A UVM-based approach for verifying ECC circuits is proposed in [116]. The approach uses assertions to validate the detection and correction behaviors in a coverage-driven verification methodology. For small ECC designs of low data width ($<$ 10 bits), simulation methods can provide the required coverage. However for ECC designs of moderate to large size, simulation-based methods, including both software simulation and hardware-assisted simulation, are inapplicable.

Techniques that use formal methods to validate the correct design of ECC circuits have been proposed. For ECC designs of moderate size (data width $<$ 32 bits), FV tools provide exhaustive analysis and require a permissible amount of computation resources. In [34], an approach is proposed to formally verify large ECC designs with automatically generated properties. The properties are constructed in such a way that only a small window of the data vector (e.g., 16 of 256 bits) is left unconstrained while the rest of the data bits are constrained to a fixed value and

assumed to be bit error free. Following the approach, when the size of unconstrained data vector window is increased (data width > 32 bits), the FV tools run into known complexity issues of memory and runtime. Although such approaches provide better coverage than simulation-based methods, they do not provide exhaustive analysis covering all input and bit error combinations.

In [74], a verification technique that is encapsulated in a reasoning tool called 'BLUEVERI' is proposed to verify large ECC designs. The tool is specifically applicable for the logic implementations defined over Galois fields to establish the correctness against mathematical specifications. The design to be verified and a check file containing a set of legal input values and the expected values for certain signals in the design are provided as inputs to the tool. The tool then employs custom-built algorithms to prove the correctness of the logic implementation. A similar approach is proposed in [56] which uses abstract interpretation theory to validate the ECC codes. Although these approaches are applicable to much larger ECC designs, they require extensive knowledge of the design implementation, require more manual effort and are limited to the respective toolchains.

The approach presented in this paper is independent of any specific toolchain and can be realized using any FV tool. The approach makes use of a succinct quality of the formal engines to prove the underlying mathematical reasoning of ECC circuits. The properties are simple and straightforward to develop and can be automated from high-level requirements.

E.1.3 Error Correction Codes

Error Correction Codes were primarily created to ensure reliable data transmission over unreliable noisy communication channels. In recent years, in order to safeguard against soft errors and to guarantee a tolerable level of risk, ECC circuits found their usage in safety-critical automotive chips to monitor the data corruption in memory systems. ECC designs consist of two stages: ECC encoding and ECC decoding (as shown in Fig.E.1).

ECC Encoder and Decoder

The function of an ECC encoder is to encode the data bits with additional check bits. The resulting codeword which is formed by combining the data bits and check bits (or parity bits) is written to the memory. Several encoding schemes are available for computing the check bits and a specific encoding scheme is selected based on the width of data bits, the bit error rate (BER), the minimum *Hamming distance* of the code and the error detection efficiency. The Hamming distance is the number of bit changes between two valid codewords. The minimum Hamming distance of the encoding scheme decides the number of bit errors that can be detected and corrected. Hamming codes, Hsiao Codes, Reed-Solomon Codes and Bose-Chaudhuri-Hocquenghem Codes are commonly used encoding schemes for memory elements [49, 94].

The function of an ECC decoder is to decode the codeword read from memory and to take required actions depending on the number of bit errors. The block diagram of a typical ECC decoder is shown in Fig. E.2. The diagram shows an extended DECTED decoder from Fig. E.1 (DECTED = double-error correction / triple-error detection). The *Syndrome Generator* computes the error syndrome (`syn_out`) for the incoming data vector (`rd_data`). The syndrome generation is based on the encoding scheme used to generate the check bits. The error syndrome output identifies the bit positions with an error. The *Error Detection* unit sets the corresponding error flags (`err_1_out`, `err_2_out`, `err_3_out`, `err_det_out`) and the *Error Correction* unit

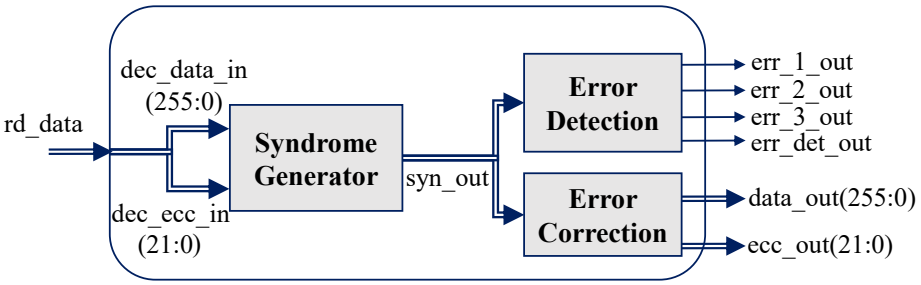


Figure E.2: Block diagram of a DECTED decoder

corrects the corrupted bits based on the syndrome output.

E.1.4 Linearity of Syndrome Generators

The output of syndrome generators is independent of the codeword and depends only on the bit error positions. In mathematical terms, the syndrome generators implement a *Linear Function*. A linear function preserves additivity, i.e., it satisfies the following rule:

$$f(x)+f(y)=f(x+y) \qquad \ldots\ldots\ldots (E.2)$$

where x,y are arbitrary vector spaces in the field M. A linear function preserves the vector addition and scalar multiplication irrespective of the vector spaces and the scalar value.

We consider Hamming codes for illustrating the linear property of the syndrome generators. Let c be a codeword of width n, formed by combining k data bits and $(n-k)$ parity bits. We call the set C of all codewords an $[n,k]$ Hamming code over the field M. Let us consider a $[7,4]$ Hamming code with an additional parity bit for the detection of double-bit errors such that its Hamming distance is 3. Besides detecting double-bit errors, such a code is capable of *correcting* single-bit errors. Let d be the word being decoded. The syndrome of the word d is computed according to Eq. E.3.

$$\mathbf{syn(d)} = \mathbf{H}\cdot\mathbf{d^T} \qquad \ldots\ldots\ldots (E.3)$$

where H is the parity check matrix of order $(n-k)\times k$. The parity check matrix is chosen in accordance with the ECC encoder. Vector addition and matrix multiplication in the field M are linear operations based on modulo-2 arithmetic. Let the parity check matrix for the $[7,4]$ Hamming code be given by:

$$\mathbf{H}=\begin{pmatrix}1&0&1&0&1&0&1\\0&1&1&0&0&1&1\\0&0&0&1&1&1&1\end{pmatrix} \qquad \ldots\ldots\ldots (E.4)$$

Let us assume that $c=0101010$ is the original codeword written to the memory. When there is no bit error, the data read is equal to the original codeword i.e., $d=c$. The syndrome of the

codeword c (no error) is given by:

$$\mathbf{syn(c)} = \mathbf{H} \cdot \mathbf{c^T} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \implies \textit{a null vector} \quad \ldots\ldots\ldots (E.5)$$

Let $d = 0101110$ be a corrupted word (non-codeword), namely codeword c with *bit* 5 flipped. The syndrome of this non-codeword is given by:

$$\mathbf{syn(d)} = \mathbf{H} \cdot \mathbf{d^T} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \implies \textit{error in } 5^{\text{th}} \textit{ bit}$$

$\ldots\ldots\ldots (E.6)$

Since vector addition is a linear operation in modulo-2 arithmetic, the non-codeword can be written as the sum of the error-free codeword and the error vector (e) i.e., $d = c + e$. For $d = 0101110$, the error vector $e = 0000100$. The syndrome of e can be computed as:

$$\mathbf{syn(e)} = \mathbf{H} \cdot \mathbf{e^T} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \quad \ldots\ldots\ldots (E.7)$$

Because of linearity, $syn(d) = syn(c+e) = syn(c) + syn(e)$. By construction of the parity check matrix, the syndrome of a codeword c is always zero: $syn(c) = 0$. Hence, the syndrome of an erroneous non-codeword d depends only on the error word:

$$syn(d) = syn(c+e) = syn(c) + syn(e) = syn(e) \quad \ldots\ldots\ldots (E.8)$$

E.1.5 Exploiting the Linearity of Syndrome Generator for FV

For ECC designs of moderate size (data width < 32 bits), modern FV tools provide full proof within a reasonable amount of runtime. However, FV tools still suffer from complexity issues for large ECC designs due to reasons outlined in Section E.1.1. In Section E.1.3, we presented the linearity property of the syndrome generators. Since FV tools are best suited to prove the underlying mathematical reasoning implemented by the RTL designs, the characteristics of a syndrome generator can be captured in properties and proven in a FV tool. Once these properties are proven, the properties capturing the detection and correction ability of the ECC designs

can be proven by assuming a fixed, arbitrarily chosen data vector. Exploiting the specifics of syndrome generator characteristics drastically reduces the analysis space for FV tools as demonstrated in the following.

FV environment setup for ECC

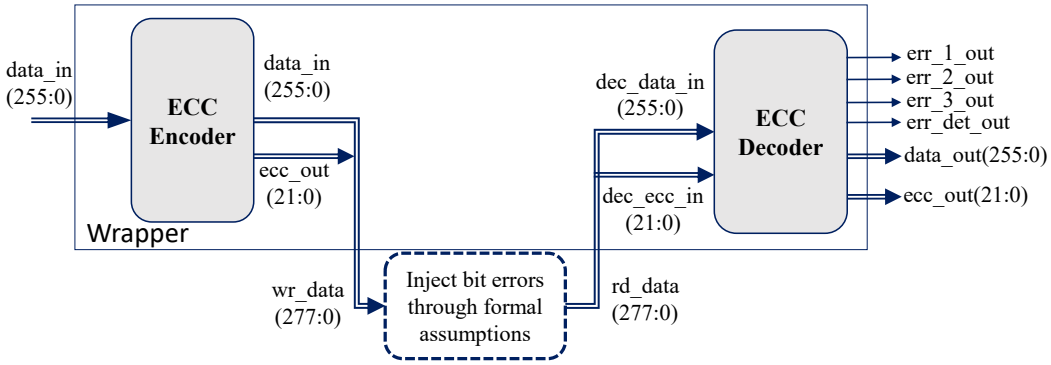


Figure E.3: Formal Verification setup for DECTED ECC design

A general setup for ECC circuit verification with formal methods is depicted in Fig. E.3. As the functionality of the program flash interface (from Fig. E.1) is irrelevant for the correct functioning of the combinatorial ECC logic, a verification wrapper is used that instantiates only the ECC encoder and decoder components. The encoder's output (`wr_data`) and the decoder's input (`rd_data`) are wired through to the wrapper's interface as shown. The bit errors are assumed referring to the encoder output and the decoder input to specify all possible error scenarios to be handled by the ECC logic.

ECC properties without considering the linearity

"Brute-force" properties capturing the double-bit error detection and correction behavior are shown in Fig. E.4a and Fig. E.4b, respectively. The properties shown do not consider the linearity characteristics of the syndrome generator. Since FV tools are inherently exhaustive in nature, these properties encompass $2^{256} \times \binom{278}{2}$ (input and double-bit error) combinations. Due to the huge analysis space, the full proof for these properties cannot be expected within a reasonable amount of time. During our experiments, the commercial FV tool ran out of available computation memory after 100 hours of computation time.

```
1 property double_biterr_detect;
2 $countones(wr_data ^ rd_data)==2
3 |->
4 err_2_out && err_det_out &&
5 !err_1_out && !err_3_out;
6 endproperty
```

(a) Property for double-bit error detection

```
1 property double_biterr_correct;
2 $countones(wr_data ^ rd_data)==2
3 |->
4 data_out == data_in;
5 endproperty
```

(b) Property for double-bit error correction

Figure E.4: Properties to detect and correct double-bit errors (in SVA) (without considering the linearity characteristic)

ECC properties with linearity

As described earlier, the characteristics of the syndrome generator can be captured as properties and proven in a FV tool. The property shown in Fig. E.5a captures the value of syndrome output when there is no error in the codewords. Here `syn_out`is the output of the syndrome generator as shown in Fig. E.2. The property shown in Fig. E.5b captures the linear characteristic of the syndrome generator. $Syn(x)$ is the RTL function that returns the syndrome vector (`syn_out`) for an arbitrary word x. The *XOR* ($\wedge$) operator computes the bitwise modulo-2 addition. Once these properties are validated in a FV tool, the following inferences are established:

- The syndrome output for an error-free codeword is always a null vector.
- The syndrome generator implements a linear algebraic function. As a result, the syndrome generator preserves the modulo-2 addition operation irrespective of the value of the data vector.

```
1 property syndrome_err_free_code;
2 wr_data == rd_data
3 |->
4 syn_out == 0;
5 endproperty
```

(a) Property for syndrome output of an error-free codeword

```
1 property linearity_syndrome_gen;
2 Syn(x) ^ Syn(y) == Syn(x ^ y);
3 endproperty
4 //x, y are arbitrary words of same width
5 //'^' operator performs bitwise addition
```

(b) Property for linearity of syndrome generator

Figure E.5: Properties capturing the characteristics of a syndrome generator (in SVA)

Once the properties shown in Fig. E.5 are proven, the remaining properties are verified under the assumption of an arbitrarily chosen, fixed data vector as shown in Fig. E.6. The property shown in Fig. E.6a captures double-bit error detect behavior with a fixed data vector (previously shown in Fig. E.4a). Because of assuming a fixed data vector, the input combinations are effectively reduced from 2^{256} to 1. The number of double-bit error combinations that a FV tool has to consider remains unchanged at $\binom{278}{2} = 38503$. Similarly, the double-bit error correct property is also proven by assuming an arbitrarily chosen data vector as shown in Fig. E.6b.

```
1 property double_bit_error_detect;
2 data_in == 256'h234FDBE76B23ABF &&
3 $countones(wr_data ^ rd_data) == 2
4 |->
5 err_2_out && err_det_out &&
6 !err_1_out && !err_3_out;
7 endproperty
```

(a) Property for double-bit error detection

```
1 property double_biterror_correct;
2 data_in ==256'h5346BEAC72643BFD &&
3 $countones(wr_data ^ rd_data) ==2
4 |->
5 data_out == data_in;
6 endproperty
```

(b) Property for double-bit error correction

Figure E.6: Properties to detect and correct double-bit errors (in SVA)

Coverage perspective

The presented verification approach is compositional and fully validates the correctness of the ECC encoder, the syndrome generator and the error detection and correction logic. For the syndrome generator as a core component of ECC circuitry, we prove the linearity property as shown in Fig. E.5b. This property exercises the syndrome generator with its full input space as x, y are arbitrary words. The property in Fig. E.5a verifies the encoder and the syndrome

generator in a chain. The properties Fig. E.5a and Fig. E.5b together prove that the output of the syndrome generator is a function of the XOR difference between `rd_data` and `wr_data`, because `wr_data` is a codeword for which `syn_out=0`, and `rd_data` may be a non-codeword with some bits flipped from the original codeword.

After proving the properties in Fig. E.5, the correctness of error correction/detection logic is proven based on the Fig. E.3 design. We have to consider the full input space for which the design behavior is specified. For Fig. E.3 this space does not contain all the combinations of `rd_data` and `wr_data`, because the design specification cares only about the situations when errors can be corrected/detected (here, up to 2/3 bit errors, respectively). Hence, the total input space is given by all situations when the Hamming distance (HD) between `rd_data` and `wr_data` is 0, 1, 2 or 3, i.e., the XOR difference between them has 0..3 ones.

Fig. E.6 shows the properties for the case HD = 2, for a specific data input `data_in` of the encoder. Since these properties do not specify/constrain which bit positions are erroneous, the FV tool enumerates all possible bit error combinations $\binom{278}{2}$ such that HD = 2. These properties prove that the syndrome generator drives the error detection and correction blocks such that they produce correct output, for the specific data input `data_in` of the encoder. When both properties have been proven for all four HD cases then all possible `syn_out` values have been generated and possible error detection/correction cases have been covered. This is independent of the specific data input `data_in` of the encoder.

Implicitly, this experiment also completes verification of the encoder that generates `wr_data`. We prove that decoding the codewords produced by the encoder recover the original data and/or detect/correct any corruption within the specified detection/correction range. As a result there is no need to separately prove the correctness of the ECC encoder.

E.1.6 Automated Property Generation

Property development is one of the major steps in the application of formal methods for design verification. The quality of properties is of great importance to ensure the complete coverage of design functionalities. The proposed approach of proving the specifics of a syndrome generator makes the properties for the class of ECC circuits straightforward to develop. Although different ECC designs implement different encoding/decoding schemes for error detection and correction, the properties required to verify the features of ECC designs remain similar. Therefore, an in-house property generation tool has been employed to foster re-usability and verification productivity.

At Infineon, an automation framework based on metamodeling has been widely used for code generation. The framework has been deployed for more than 100 applications and is the source of a high productivity benefit. The framework uses Unified Markup Language (UML) class diagrams for model definition and Python as the generation language. The framework is not only used for RTL generation but also for the automatic generation of properties [28]. The generation flow, which follows the idea of separating the generation steps into multiple layers, is depicted in Fig. E.7.

The first layer is the *specification layer*, which focuses on capturing the informal specifications in formal specification models. High-level details such as system configurations and intended behaviors of the design are considered. A model definition (metamodel) developed for ECC designs is shown in Fig. E.8a. The metamodel shown (as UML class diagram) captures the requirements of an ECC design. The *ECCBlock* contains an *Encode* unit which captures the

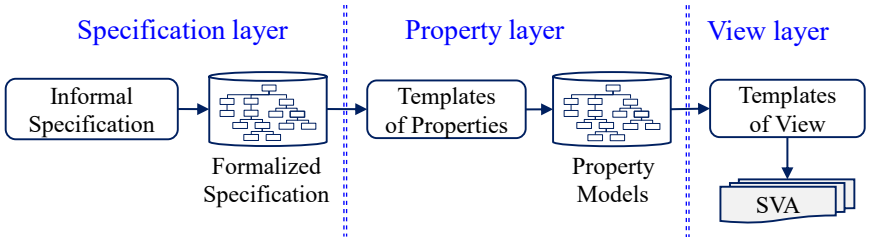
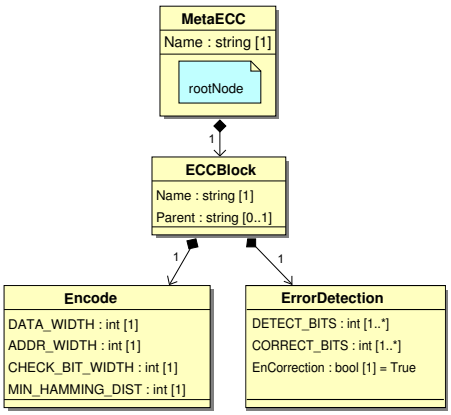


Figure E.7: Property generation flow

specifics of an encoder. The *ErrorDetection* unit captures the detection and correction capability of the ECC implementation. From this metamodel different instances can be created for different ECC implementations. An instance created for the ECC block (Fig. E.1) implemented in the program flash interface is shown in Fig. E.8b.



(a) Specification model for ECC designs

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <MetaECC>
 3 <Name>Flash-Bus-Interface</Name>
 4 <ECCBlock>
 5 <Name>ECC_block</Name>
 6 <Parent>Program-Flash</Parent>
 7 <Encode>
 8 <DATA_WIDTH>256</DATA_WIDTH>
 9 <ADDR_WIDTH>19</ADDR_WIDTH>
10 <CHECK_BIT_WIDTH>22</CHECK_BIT_WIDTH>
11 <MIN_HAMMING_DIST>6</MIN_HAMMING_DIST>
12 </Encode>
13 <ErrorDetection>
14 <DETECT_BITS>1</DETECT_BITS>
15 <DETECT_BITS>2</DETECT_BITS>
16 <DETECT_BITS>3</DETECT_BITS>
17 <CORRECT_BITS>1</CORRECT_BITS>
18 <CORRECT_BITS>2</CORRECT_BITS>
19 <EnCorrection>True</EnCorrection>
20 </ErrorDetection>
21 </ECCBlock>
22 </MetaECC>
23
```

(b) Specification instance for ECC block in PFlash interface

Figure E.8: Capturing informal specifications in formal specification model

In the intermediate *property layer*, the *Templates of Properties* (ToP) extract the expected behavior of the intended design from specification models and define a property model for each specification item [28]. A property model is a temporal expression trace. The ToP are written in Python, which extract specification details of an encoder (data width, check-bit width, minimum Hamming distance) and decoder (detect-bits, correct-bits) blocks. ToP are implemented such that the property models are defined based on the values of various attributes. For example, the correction property models are defined only if the attribute *EnCorrection* is *True*. Finally in the *view layer*, the property models are mapped to a specific property description language (e.g., SVA) to generate the properties. The properties shown in Fig. E.4, Fig. E.5 and Fig. E.6 are generated by the presented generation flow.

E.1.7 Application on Real Designs

The presented approach has been applied to multiple instances of the ECC circuits implemented in several automotive designs. Our experiments are carried out on 3 different SoCs that are used

in safety-critical automotive applications. SoC-1 is the central part of an MCU platform which is used in powertrain applications (including electric and hybrid vehicles) as well as safety applications (such as steering, braking, airbags and advanced driver assistance systems). SoC-2 is also a part of the MCU platform, which enables the electrification of powertrain functions in electric vehicles. SoC-3 is a Lidar sensor used in highly automated driver assistance systems.

Design information

Design information about exemplary ECC circuits implemented in the different SoCs is tabulated in Table E.1. Column 2 shows the type of interface where the ECC module is employed. Columns 3 and 4 show the width of data bits and check bits, respectively. Columns 5 and 6 depict the error detection and correction requirements of the respective ECC modules. The number of logic gates of each ECC implementation is shown in column 7. Since the ECC circuits are combinatorial in nature, the logic gates count represents the complexity for formal analysis.

Table E.1: ECC instances in different SoCs: Design information

ECC instance	Data bits	Check bits	Error detection	Error correction	#Logic Gates
PF-ECC	256	22	1,2,3 bit errors	1,2 bit errors	247k
DF-ECC	64	22	1,2,3,4 bit errors	1,2,3 bit errors	40.2k
DM-ECC	26	6	1,2 bit errors	1 bit error	1.95k
RF-ECC	16	6	1,2 bit errors	1 bit error	1.2k

FV setup and property runtimes

The FV setup for the PF-ECC circuit is shown in Fig. E.3. A similar FV setup is used for the remaining ECC modules. For the property development, we used the property generation tool outlined in Section E.1.6. Templates of Properties are set up once for the ECC metamodel and reused for all the ECC instances. To setup the initial property generation flow for PF-ECC design, it required 3 person-days of effort. For a new ECC design instance, creating the metamodel instance (Fig. E.8b), generating the properties and setting up the regression flow requires one person-day of effort.

The PF-ECC had been previously verified with manually developed properties. Since the linearity characteristics of the syndrome generator had not been considered, complex properties were developed, in which only a small window of the data vector was allowed to be symbolic. The remaining data bits were constrained to a fixed value and assumed to be error-free. This approach is similar to the approach followed in [34]. Overall, the initial property development, setting up the FV setup and refining the properties to attain a full proof required 4 person-months of effort.

The proof runtimes of the ECC properties for 4 different designs are tabulated in Table E.2. For the property evaluation, we used the commercial FV tool OneSpin 360® DV-Verify on an Intel® Xeon® E5-2690 v3 @2.6GHz with 32GB RAM. Column 2 shows the proof runtimes of the ECC properties where the data vector is allowed to be symbolic (e.g., Fig. E.4). With

Table E.2: Proof runtimes for FV of ECC circuits in different SoCs

ECC instance	Proof runtime without linearity	Proof runtime with linearity	Proof runtime with linearity, induction-based
PF-ECC	Given up after 100 hours	08:54:55	3:03:46
DF-ECC	Given up after 100 hours	06:55:17	–
DM-ECC	00:01:40	00:00:10	–
RF-ECC	00:04:40	00:01:10	–

The proof runtimes shown are for the complete property suite (in the format hh:mm:ss)

this approach, for PF-ECC and DF-ECC, the FV tool ran out of computation resources after 100 hours of runtime. Column 3 shows the proof runtimes of the complete property suite following the linearity approach. The property suite also includes the properties (shown in Fig. E.5) that validate the characteristics of the syndrome generator.

Further runtime improvements

In induction-based property checking, a certain operation of the design is separated into 2 or more small operations (sub-operation) and is proven as separate properties (e.g., induction base, induction step). These properties are compiled such that a set of properties capture the complete behavior of an operation [119, 81]. The consequent of the preceding property (e.g., ending state of the sub-operation-1) is part of the antecedent of the succeeding property (e.g., starting state of sub-operation-2). For PF-ECC design, we implemented a similar technique. Since the linearity of the syndrome generator is already proven with a separate property (Fig. E.5b), the expression $Syn(x) \wedge Syn(y) = Syn(x \wedge y)$ is added to the antecedent of the remaining detection and correction properties. With this enhancement, the proof runtimes are further reduced. The proof runtime for PF-ECC properties with further improvements are shown in column 4 of Table E.2. Overall, the property runtimes for large ECC designs are significantly reduced using the linearity approach. The FV tool requires 1:43:20 (h:mm:ss) to provide a full proof for the triple-bit error detect property on PF-ECC design.

Observations

The following observations are drawn from the results obtained by the application of the proposed method on real designs.

Observation 1: The proposed approach of first proving the linearity of the syndrome generator and then proving the remaining properties by assuming a fixed, arbitrarily chosen data vector significantly reduces the property runtimes. On large ECC designs the properties that were previously unable to converge after 100 hours of computation time, converge within 2 hours of computation time, thus providing a significant runtime improvement (at-least 50X on large ECC designs). The proposed approach provides a complete coverage for bit error detection and correction behavior of the ECC circuits and provides a high verification quality.

Observation 2: The proposed approach simplifies a set of properties needed to verify an ECC design. Due to the similarity of bit error detection and correction properties, the prop-

erties can be automated for different instances of ECC designs. The property automation tool simplified the property development, enabled the reuse of generation flow for multiple ECC instances and improved the overall verification productivity.

Observation 3: The proposed approach in the paper makes a well-defined contribution to the formal verification of data transformation blocks. Designs that perform data transformation are generally considered as hard to verify with formal methods. The vast input analysis space of data transformation blocks increases the complexity for FV tools. However, a major class of data transformation designs implement specific polynomials (e.g., noise generators, MPEG encoders, MPEG decoders, etc.) to perform the data transformation. When the underlying transformation algorithm (e.g., a polynomial multiplication) has characteristics that are independent of the input data, the properties of the transformation algorithm can be separately proven, as demonstrated in this paper for syndrome generators. This is possible since the formal engines can better abstract the properties capturing the mathematical reasoning of the designs. Afterwards, the remaining properties of the design are proven by assuming a fixed, arbitrarily chosen data word. This approach decomposes the design structurally based on mathematical properties and runs the restricted formal proofs on the design blocks without compromising full coverage. The restrictions on the formal proofs exploit the mathematical property and reduce the search complexity, but still exercise the full design space so that there is no verification gap.

E.1.8 Summary

We presented an approach to exploit the specifics of the underlying algebra implemented by the ECC designs using formal methods. The approach exploits the linearity of the *Syndrome Generator* to establish that the detection and correction of bit errors depends only on the erroneous bit positions and not on the input data word. Proving the linearity feature enables a significant reduction of the analysis space for formal tools. Results show a significant improvement in the property proof runtimes. The ECC properties that were previously unable to converge after long proof runtimes, now converge within few hours of computation time. A high verification productivity has been achieved by combining the approach with automatic property generation. The effort was reduced from months to days. The proposed approach is applicable to all design blocks that implement data transformation such that the transformation is independent of the input data.

E.2 Synthesis of Decoder Tables using Formal Verification Tools

Abstract— This paper discusses a use case of formal verification tools in automating the implementation of RTL modules. In detail, classical hardware design tasks involve constructing a micro-architecture and a control unit that makes the design functional. Traditionally, these control units are built by analyzing the micro-architecture implementation and the intended function. In this work, we introduce a novel approach for automatic synthesis of control signals for a given micro-architecture. Our approach uses formal methods to automatically derive a working control unit and/or decoder for a certain micro-architecture. For this purpose, the properties (SystemVerilog Assertions) developed for design verification are re-used with minor refinement. To show the applicability of our approach for real-life designs, we develop a RISC five-stage pipeline micro-architecture and automatically derive the instruction decoder. Our method saves a significant amount of work (i.e. manually tailoring units), allows to focus on the micro-architecture design and makes it easy to analyze various ISA alternatives, accelerators and architecture alternatives.

E.2.1 Introduction

Even though the effectiveness of formal verification tools for design analysis is well-known, to the best of our knowledge, formal verification tools are not widely applied for the same. In addition to design verification tasks (full-proof, bug-hunting, coverage analysis), formal verification tools are leveraged for several design related tasks such as automatic linting, reachability analysis, etc. In addition, the tools can be utilized for examining the design behavior. In this paper, we discuss the usage of formal tools for generation of control unit.

The first step in a hardware implementation is the definition of a micro-architecture as an implementation template (e.g. pipelines for CPUs, FIR filters, (programmable) FSMs). These templates are, in a second implementation step, equipped with necessary control signals so that the overall design provides the required functionality (i.e. the correct behavior of specific signals at specific points in time). Since the method involves trial and error approach, this is an elaborate and error prone task. Hence, we developed an approach to automatically derive decoder tables by utilizing formal verification tools.

For simple control-data-path architectures, the generation of both data-path and control unit can be automated by using high-level synthesis tools. This approach however maps to simple data-path control architecture and does not support other architectures. The technique proposed in this paper can be used for synthesizing control signals for all types of control-data-path micro-architectures.

For explaining and proving our approach, we synthesize a control unit, i.e. an instruction decoder, of a processor implementing the RiscV [117] open source Instruction Set Architecture (ISA). RiscV is becoming widely visible and is being labeled as the next standard open architecture for industry implementations. Hence, we consider a pipeline implementation of RiscV ISA as the test vehicle. RiscV ISA specifies various extensions spanning from RV32I base integer instruction set to standard extension for decimal floating-point and many other. The part that must be adopted for different extensions of the ISAs is mainly the look-up-table[1] in the

[1]Look-up-tables are generally used to replace run-time computation with simpler array indexing operations and

instruction decoder, which drives the control signals. These control signals are different for different instruction encoding and provide the required functionality of each instruction. The interface of the decoder look-up-table is shown in Fig. E.9 illustrating the complexity of the table.

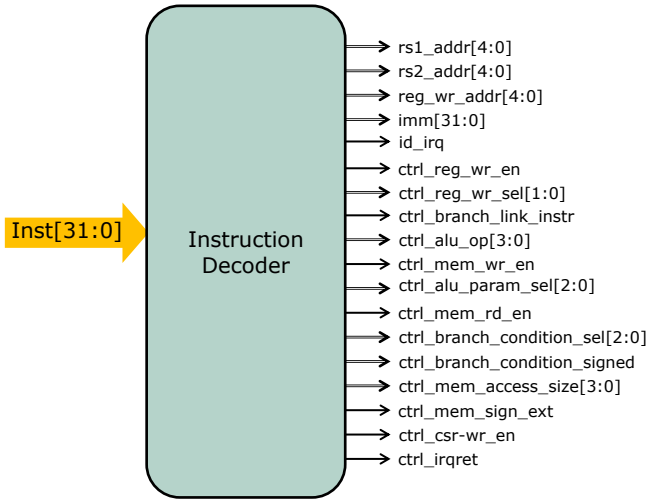


Figure E.9: Block view of Instruction Decoder

We first generated the RTL micro architecture design of the 5-stage pipeline CPU of RiscV ISA with an empty look-up-table. Next, we generate a set of properties for each instruction of the selected ISA that capture required micro-architecture behavior. The RTL and properties are generated using a meta-modeling framework following Object Management Group's (OMG) Model-Driven-Architecture (MDA) idea for code generation [70]. These generated properties are then used with cover directive in a formal verification tool to generate an execution trace and to extract the control signals hereof.

One of the major advantage our technique is reduction in turn-around time for late changes in the decoders due to extension and changes in the specification. The properties generated for extracting control signals can be re-used for design verification with minimal changes. As a result, our approach reduces considerable efforts for design verification. In addition, our approach provides maximum benefit when used in a design flow methodology proposed in [113]. The methodology follows the principle of correct-by-construction and proposes parallel development of both verification IP and the RTL design.

The rest of the paper is organized as follows. Section E.2.2 provides a discussion on deriving control signals using simulation techniques and manual analysis. Section E.2.3 elaborates our approach of automatic derivation of control signals. In section E.2.4, we provide an overview of the generation framework, property generation and their suitability to the proposed approach. A section on brief summary of the work completes the paper.

provide significant performance benefits.

E.2.2 State of the Art

Manual Coding

For typical digital design problems, a set of micro-architectural patterns is available. Usually, one of these patterns is implemented and some control signals are provided to program or make the micro-architecture configurable. Functional verification then automatically checks that the micro-architecture meets the specified requirements.

In order to derive the control or configuration signals, the design engineer manually combines his understanding of the requirements and his knowledge of the implemented micro-architecture. For example, given the understanding of a certain CPU instruction and the knowledge of the micro-architecture, the control signals are manually set. However, this step is often repetitive and requires deep knowledge of the micro-architecture.

Synthesis using Simulation Technique

While generating a 5-stage implementation of the RiscV ([37]), a simulation-based method was developed to generate the look-up-table. This method identifies the necessary control signals for each instruction using a trial-and-error approach. For this purpose, we generated a test-bench to check the required functional behavior from a formal ISA definition. For each instruction, the set of possible control signals was exhaustively simulated against the test-bench. When the test-bench did not report an error, a control vector was found so that our micro-architecture behaves correctly. Even for the most minimalist micro-architectures and simple instruction sets, repeating this process for all instructions takes many hours to complete. This can be accelerated by parallel simulations (e.g. one for each instruction). The effort required however still increases exponentially with the number of control bits required for a data-path architecture. A simulation-based technique for complex, real-world micro-architectures is thus not feasible.

E.2.3 Automated Generation of Decoder tables

Approach

The proposed approach for utilizing formal verification tools to derive control signals in an automated manner is depicted in Fig. E.10. Starting point is the micro-architecture template of the intended hardware system and the set of properties that are drawn from the specification (ex: ISA). A property is written/generated to verify the intended behavior of a particular operation (ex: ADD instruction) such that necessary signals of the micro-architecture template are captured. The properties are encoded in SVA following a coding style that should be supported by almost all commercial formal verification tools.

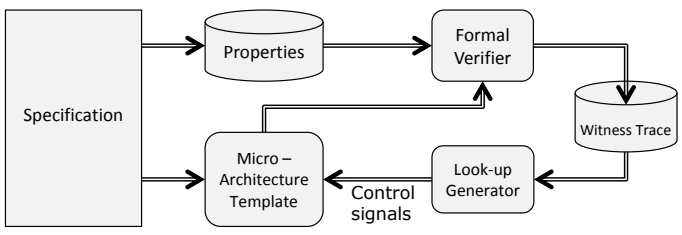


Figure E.10: Block diagram of automatic look-up-table generation

The set of properties and the microarchitecture with the empty look-up-table (microarchitecture template) are then applied to the formal tool. Due to the nature of formal verification[2], the tool is expected to provide a counter-example (CEX) if the properties are evaluated with *ASSERT* directive. This is due to the empty look-up-table in the micro-architecture. However, formal tools also provide a witness trace that demonstrates a scenario in which all signals attain the required values. Hence, the properties are evaluated with the *COVER* directive[3]. *COVER* witness traces are an optimistic approximation of all possible scenarios satisfying the property sequence. Due to this, the generated property for a specific operation must be encoded to instruct the formal tool to prohibit other operations which partly have the same results. A detailed example is provided in section E.2.4. The witness traces are investigated to extract necessary control signals and fed into the look-up-table generator. This task is automated by setting up scripts that parse the log files and generate control signal values required by the look-up generator. The look-up-generator then fills the micro-architecture template with necessary control signals.

The mechanism of encoding properties to select and exclude operations requires more effort if the properties are manually implemented. However, the complexity increased in coding properties to select and exclude operations can be mitigated by adopting automatic property generation approach. In addition, automatic code generation flow is simplified by transforming informal specifications into formal specifications. Generating properties from formal specification avoids ambiguity and enables simpler code generators. The approach taken for automating property generation is briefly outlined in section E.2.4. Nevertheless, the proposed method for automatic derivation of control signals is independent of any property generation flow and the method is applicable to nearly any control decoder design.

E.2.4 Application

We applied the technique described in section E.2.3 to the automated generation of instruction decoder look-up-tables of various micro-architectures supporting the RiscV RV32I Base Integer Instruction Set [117, page 27]. Fig. E.11 shows a well known template of a 5-stage CPU with the below part depicting our approach for generating control signals. As already mentioned in the previous section, in order to implement the properties with mechanism to exclude operations, we employed an automated code generation approach. The following subsections describe our code generation framework, property generation and derivation of control signals for an instruction decoder using cover properties.

Meta-modeling framework for code generation

At Infineon, a meta-modeling framework based on Python and Mako templates is widely used for code generation. This meta-modeling framework for efficient code generation is deployed for more than 100 applications and is the source of high productivity benefit [38, 37]. The framework is currently being enhanced by splitting the generation flow into different layers following the vision of OMG'S Model Driven Architecture approach for code generation [36].

[2]Formal methods provide exhaustive proof spanning all (legal) input combinations i.e., a property is verified against the design for all possible input combinations.

[3]The *COVER* directive instructs the verification tool to search for an example trace obeying the property sequence during analysis.

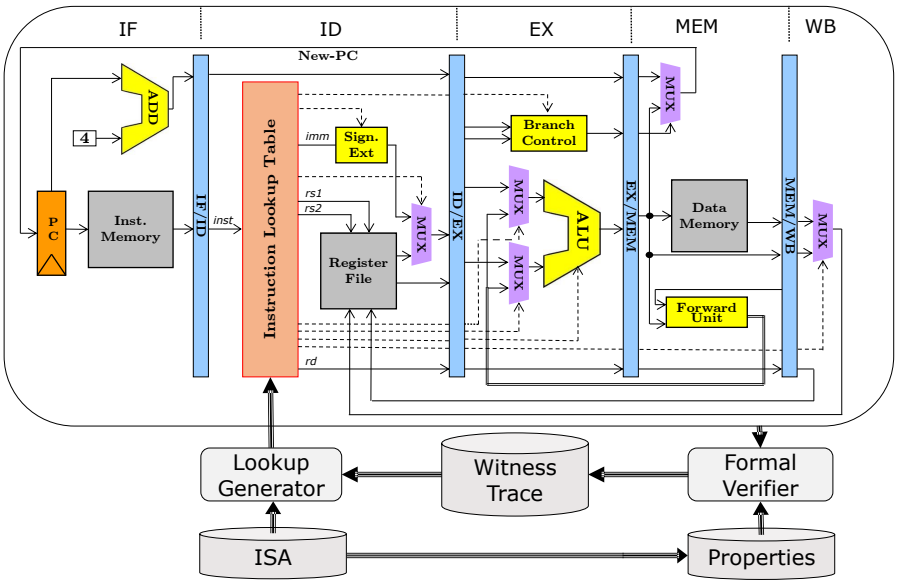


Figure E.11: 5-stage pipeline template according to [87] and look-up-table generation

The flow involves creation of a series of models in which the preceding model is an abstracted version of the current model. The code generation flow is split into following steps:

- In the first step, the specifications and requirements of the intended hardware design are transformed into an abstract model called Model-of-Things (MoTs). The MoT corresponds to computational-independent model (CIM) in the original MDA definition [70]. The implementation details are intentionally left-out from these abstract models to allow architectural alternative exploration. That is, the MoTs contain information of "what shall be implemented?" and abstracts away the information of "how shall be implemented?". For example, an abstract model is built to represent the RiscV ISA.
- Next, the abstract descriptions are transformed into an intermediate model that contains hardware implementation (micro-architecture) details. This layer corresponds to the platform-independent model (PIM) in the original MDA definition [70]. For property generation, these concrete models hold the intended property trace.
- Finally, these intermediate models are mapped onto the corresponding target code using Mako templates or using an additional intermediate model, which we call model-of-view (MoV). This MoV corresponds to the platform-specific model (PSM) in the original MDA definition [70].

The aforementioned meta-modeling framework is used for the generation of both, the CPU implementation and the properties. We first built the 5-stage pipeline RiscV processor using a model-driven flow mentioned above and elaborated in [36].

Generation of Properties

The generation of properties follows the MDA approach and is outlined in [33]. To facilitate the understanding of this paper, we provide a brief summary in the following.

Fig.E.12 shows the property generation flow following the MDA approach. As mentioned earlier, the first step is to convert specifications into abstract models which are called as Model-of-Things. These abstract models are a formalized version of informal specifications and do not

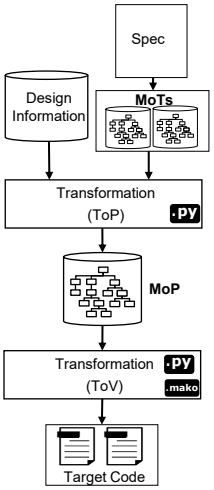


Figure E.12: Property generation flow

include implementation details. For RiscV ISA, we create an abstract model called 'MetaRisc' that contains instruction encoding, abstract instruction behavior, and several objects such as register files, program counter, memory, etc..

The translation from abstract models (MoTs) into a more concrete model is performed in Templates-of-Property (ToP). These Templates-of-Properties are used to aggregate the information from MoTs and define the trace for the properties. The property trace information is captured in a model called Model-of-Property (MoP). The definition of the property trace is aided by the underlying automation framework. In addition, ToPs also need information of the design implementation (hierarchical and input/output port names of blocks/sub-blocks). This information is either derived directly from the RTL files or from the Model-of-Design (MoD) if the design is generated following the MDA approach outlined in [36].

Model-of-Property is a platform independent way of specifying property traces and is the main model of our property generation flow. Finally, the MoP is mapped to Templates-of-View (ToV) to provide the properties in a property language of available tools. The generation framework is built to address multiple languages such as SVA, ITL or PSL.

Cover Properties

For the 5-stage pipeline CPU of RiscV ISA, we generate the properties using the MDA flow described in previous paragraphs. ToPs are used to unparse the abstract RiscV ISA model (MetaRisc) and construct the property trace for all instruction encoding types (R-Type, I-Type...). Hence for each instruction encoding type, a property trace is defined and re-used for all instructions of the particular encoding type. A property is generated for each instruction, such that it captures the required values[4] for several signals along the pipeline over a period of 4 - 5 clock cycles.

The property suite and the CPU design with an empty look-up-table in the instruction decoder are applied to the formal verification tool. It is important to make sure that while searching for control signals of a specific instruction, other operations are excluded. That is, the property generated to capture the control signals for one instruction performing a 'specific operation'

[4]We cover only those values needed to verify correct behavior

must exclude all 'other operations'. This is because, when a formal verification tool is asked to provide a witness trace for a property sequence, the witness trace provided by the tool is one of several possible traces satisfying the property sequence. For example, consider the below *ADD* instruction:

$$\text{ADD R3, R1, R2} \tag{E.9}$$

The *ADD* instruction performs the addition of contents of $R1$ and $R2$ source registers and stores the result in the destination register $R3$ [117]. Now, consider the *SLL* instruction:

$$\text{SLL R3, R1, R2} \tag{E.10}$$

The *SLL* instruction performs the logical left shift operation on the contents of the source register $R1$ by the shift amount stored in the lower 5 bits of source register $R2$ and stores the result in destination register $R3$ [117]. For both instructions, following trace is true:

$$\text{if R1 = 1 \& R2 = 1} \implies \text{R3 = 2} \tag{E.11}$$

Hence, while searching for witness trace for *ADD* instruction, the formal tool may provide the witness trace for an *SLL* instruction. As a consequence, the property must instruct the formal verification tool to provide a witness trace valid for the specific instruction only. As mentioned earlier, our property generation framework allows to specify the property trace in Python language. Selecting and excluding operations for each instruction is hence a simple and straightforward task.

A generated property that captures the required signal behavior in the pipeline for 'R-type, *ADD* instruction' ([117, page 15]) is shown in Fig. E.13. Lines 3 and 4 define the trigger and reset sequence respectively. Lines 5-11 (antecedent/enabling sequence) assume values for signals required for the ADD instruction in decode phase. Lines 12-36 (consequent/satisfying sequence) capture the expected values for various signals in decode (ID), execute (EX), memory-access (MEM) and write-back (WB) phases of the pipeline. Lines 14-16 capture correct decoding of source/destination register addresses in the decode phase. Lines 18-20 capture the *ADD* instruction behavior during execute phase. Lines 21-30 exclude the operations as already explained in previous paragraphs. Lines 32-36 reflect the pipeline behavior during mem-access and write-back phases.

If no witness trace can be generated, the given architecture is not capable to fully support the ISA[5] and must be enhanced. If a witness trace is generated by the tool, the expected control signals for each instruction are extracted and the instruction decoder is generated.

Results and Discussion

We used for our work the formal verification tool OneSpin 360 (Version: 2017_06(38)). The witness computation time for each property is around one second. The tool (generally a feature of all commercial formal tools) allows to dump the witness trace information into a text file. From this text file specific signal values at specific time-points can be extracted. We set-up a "Lookup Generator" to parse the text log file and extract the control signals in a dictionary format. The overall witness generation time for all instructions (RISC-V RV32I instruction set) is less than a minute. Table E.3 shows the generated control signals for R-type instructions.

[5] Assuming the verification environment is free of over-constraining

```
1   //Property for R-type, 'ADD' instruction with forwarding disabled and with exclusion mechanism
2   property _ADD;
3   @(posedge clk)
4   disable iff(reset)
5   $changed(instr)                                  &&
6   program_counter  == ($past(program_counter) + 4) &&
7   instr[6:0]       == R_TYPE                       &&
8   instr[14:12]     == FUCNT3                       &&
9   instr[31:25]     == FUNCT7                       &&
10  !forwarding_en                                   &&
11  !branch_en                                       &&
12  |->
13  ##0
14  rs1_addr         == instr[19:15]                            &&
15  rs2_addr         == instr[24:20]                            &&
16  reg_wr_addr      == instr[11:7]                             &&
17  ##1
18  alu_in1             == $past(rs1_data)                      &&
19  alu_in2             == $past(rs2_data)                      &&
20  alu_result          == alu_in_1 + alu_in2                   &&
21  alu_result          != 0                                    &&
22  alu_result          != (alu_in1 >> alu_in2[4:0])            &&
23  alu_result          != (alu_in1 & alu_in2))                 &&
24  alu_result          != (alu_in1 << (alu_in2[4:0])))         &&
25  alu_result          != (alu_in1 - alu_in2))                 &&
26  alu_result          != (alu_in1 < alu_in2))                 &&
27  $signed(alu_result) != ($signed(alu_in1) >>> (alu_in2[4:0]))&&
28  alu_result          != (alu_in1 | alu_in2))                 &&
29  alu_result          != (alu_in1 ^ alu_in2))                 &&
30  $signed(alu_result) != ($signed(alu_in1) < $signed(alu_in2))
31  ##1
32  !data_mem_wr_en
33  ##1
34  reg_wr_data ==  $past(alu_result,2)                         &&
35  reg_wr_en                                                   &&
36  reg_wr_addr ==  $past(instr[11:7],3);
37  endproperty
38
39  //Assertion Directive
40  COVER_add_instruction: cover property(_ADD);
```

Figure E.13: Property (in SVA) to generate control signals for R-Type, ADD instruction

In addition to the automatic derivation of control signals, one major advantage of our approach is that the same kind of properties can be used to synthesize the control signals for the instruction decoder and also to validate the micro-architecture. The properties for extracting control signals are modified in order to exclude operations not specified in the instruction. The aforementioned generator framework allows flexibility to easily exclude unintended behavior in the property trace. Existing languages such as SVA and existing tools can be used to provide the required signals and in turn synthesize the decoder. We used the same generated properties but without excluding mechanism for verifying the functional behavior of the pipeline for all instructions in RISC-V RV32I instruction set.

Verification is also done with constrained random simulation which does not cause big overhead, since such a simulation must be setup for use case verification anyhow. Later is needed since some behavior can be verified in conjunction with software - e.g. interrupt behavior - only. Nevertheless, we do not expect to find any bug in relation to ISA definition, since the formal verification tool guarantees that the architecture provides the expected behavior if the computed control signals are applied. In comparison to an exhaustive approach based on simulation, the method scales and is applicable in real-life designs.

Table E.3: Generated control signals using the approach depicted in Fig. E.10 (Control signals for R-Type instructions are shown here)

Inst	rf-wr-en	rf-wr-sel	br-link	alu-op	dm-wr-en	alu-par-sel	dm-rd-en	br-sel	br-sign	dm-acc-size	dm-sign-ext	csr-wr	irq-ret
AND	1	00	0	0110	0	000	0	000	0	0000	0	0	0
OR	1	00	0	0111	0	000	0	011	0	0000	0	0	0
SLL	1	00	0	0010	0	000	0	011	0	0000	0	0	0
SUB	1	00	0	0001	0	000	0	010	0	0000	0	0	0
ADD	1	00	0	0000	0	000	0	000	0	0000	0	0	0
SRL	1	00	0	0100	0	000	0	011	0	0000	0	0	0
SLTU	1	00	0	1010	0	000	0	011	0	0000	0	0	0
XOR	1	00	0	1000	0	000	0	000	0	0000	0	0	0
SRA	1	00	0	0101	0	000	0	000	0	0000	0	0	0
SLT	1	00	0	1001	0	000	0	011	0	0000	0	0	0

Also, as in traditional design flows the properties are developed once the RTL is ready for verification. However in [113], Urdahl et al. show that considerable verification efforts can be reduced by following a "Properties first" approach. The design flow proposes systematic development of a verification IP concurrently with the design process. The properties are first designed according to system-level specifications and are refined later in the design process. Our approach of automatic generation of control signals suits a similar design flow, where verification and design tasks are carried out concurrently.

E.2.5 Summary

In this paper, we have introduced a novel approach for generating control signals for decoder tables using formal verification tools. We also showed how the generated properties for control signals synthesis can be re-used for functional verification of the design with minimal modifications. The technique provides quick turn-around time for changes that relate to extensions or changes in the decoding structures. To show the applicability of our approach for real-life designs, we successfully synthesized the control unit for 3-stage, 5-stage pipelined RiscV CPU. In addition to the instruction decoder table synthesis, the technique can be used to derive the control signals for micro-architectures that involve a decoding operation to select various computations/operations based on certain input signals. Further, the technique unfolds a new direction of application for formal verification tools.

Curriculum Vitae

Keerthikumara Devarajegowda, born in Varadalapura, Karnataka, India

1994 - 2001	Primary education, GPS, Gorur, Hassan, Karnataka, India
2001 - 2004	Secondary education, GBHS, Hassan, Karnataka, India
2004	Graduated from secondary school (equivalent to Abitur)
2004 - 2006	Pre-university education, SVPUC, Hassan, Karnataka, India
2007 - 2010	Bachelor of Engineering in Electronics and Communication Engineering, Malnad College of Engineering, Hassan, Karnataka, India affiliated to Visweswaraya Technological University, Belguam, Karnataka, India
11/2010 - 09/2012	Employment with Tata Consultancy Services Pvt. Ltd. Bengaluru, Karnataka, India
10/2012 - 02/2014	Employment with Department of Technical Education, Government of Karnataka, India
04/2014 - 11/2016	Master of Science in Embedded Computing Systems, Electrical and Computer Engineering, Technische Universität Kaiserslautern, Germany
08/2015 - 02/2016	Internship, Intel Corporation, Neubiberg, Germany
03/2016 - 11/2016	Master Thesis, Infineon Technologies AG, Neubiberg, Germany. Thesis topic: Development of a Verification Methodology to Reduce Simulation Metrics by Using Formal Verification
since 02/2017	PhD Candidate under the guidance of Prof. Dr. Wolfgang Kunz, Prof. Dr. Wolfang Ecker and Prof. Dr. Dominik Stoffel. Thesis topic: Model-based Generation of Assertions for Pre-silicon Verification