

Faculdade de Engenharia da Universidade do Porto

Evaluating High-Level SystemC Formal Verification Compared to RTL Verification

Renato Pedrosa de Jesus



Master in Electrical and Computer Engineering

Supervisor: Alexandra Sofia Ferreira Mendes

July 31, 2024

Abstract

The Electronic Design Automation (EDA) industry relies on rigorous testing metrics, utilizing extensive randomized simulation environments to functionally test a device and employing formal verification (FV) in the most critical areas. However, FV using traditional approaches poses a significant challenge, especially when dealing with complex designs.

The advent of High-Level Synthesis (HLS), which has led to the development of circuit description languages such as SystemC, has enabled the representation of devices more abstractly. Several tools, including STATE, SPIN, and VerCors, have been created to fulfill this purpose.

This thesis aims to investigate whether formal verification at the SystemC level can verify larger designs faster than at the lowest level, namely in the SystemVerilog (SV) language. To this end, one tool was chosen for each verification level.

The work concludes that the high-level verification is still in its early stages of investigation, with the performance of the tool being slower in comparison to the low-level tool. All designs verified operated based on clock cycles, thus the expected advantages of high-level abstraction were not fully realized. Furthermore, the high-level tool lacks the optimization present in the low-level tool, contributing to its reduced efficiency.

Keywords: Formal Verification (FV), SystemC, High-Level Synthesis (HLS), Deductive Verification, VerCors, SymbiYosys (SBY)

Resumo

A indústria de Software de Projeto de Circuitos Integrados (EDA) baseia-se em métricas de teste rigorosas, utilizando ambientes de simulação aleatórios extensivos para testar funcionalmente um dispositivo e empregando a verificação formal (FV) nas áreas mais críticas. No entanto, a FV utilizando abordagens tradicionais representa um desafio significativo, especialmente para verificar projetos complexos.

A síntese de alto nível (HLS), que levou ao desenvolvimento de linguagens de descrição de circuitos como o SystemC, permitiu a representação de dispositivos de forma mais abstrata. Várias ferramentas, incluindo STATE, SPIN e VerCors, foram criadas para cumprir este objetivo.

Esta tese tem como objetivo verificar se a FV ao nível do SystemC consegue verificar designs de maior dimensão do que ao nível mais baixo, nomeadamente na linguagem SystemVerilog (SV). Para o efeito, foi escolhida uma ferramenta de cada nível de verificação.

O trabalho conclui que a verificação de alto nível está ainda numa fase inicial de investigação, sendo o desempenho da ferramenta mais lento em comparação com a ferramenta de baixo nível. Todos os designs verificados eram baseados no relógio, pelo que as vantagens esperadas da abstração de alto nível não foram totalmente realizadas. Além disso, a ferramenta de alto nível carece de optimizações presente na ferramenta de baixo nível, contribuindo para a sua reduzida eficiência.

Palavras-chave: Verificação Formal, SystemC, Verificação Dedutiva, VerCors, SymbiYosys (SBY)

Sustainable Development Goals

Engineering is crucial for solving problems, addressing human needs, and driving sustainable socio-economic development. Engineers contribute to disaster resilience, public health, and technological innovation.

The 17 World Sustainable Development Goals, or SDGs, aim to address contemporary challenges and ethical considerations of new technologies and to raise awareness of the role of engineering in economic and sustainable development. The following table shows the goals that are more related to this thesis and can be achieved with it.

SDG	Goal	Contribution	Target and Indicators
7 - Affordable and Clean Energy	7.1 - By 2030, ensure universal access to affordable, reliable, and modern energy services	The improvement of the verification process will result in a reduction of the technology prices, thus contributing to the development of affordable energy.	7.1.1 - Proportion of population with access to electricity
9 - Industry, Innovation, and Infrastructure	9.5 - Enhance scientific research, and upgrade the technological capabilities of industrial sectors in all countries, in particular developing countries.	The enhancement of the verification process has the potential to facilitate the generation of new technologies and innovations.	9.5.1 - Research and development expenditure as a proportion of GDP
17 - Partnerships for the Goals	17.1 - Strengthen domestic resource mobilization, including through international support to developing countries, to improve domestic capacity for tax and other revenue collection	The collaborative development of verification methods and standards facilitates the formation of global partnerships that are conducive to sustainable development.	17.7.1 - Total government revenue as a proportion of GDP, by source

Table 1: Contribution of formal verification to SDGs

Acknowledgements

I want to express my gratitude to the Prof. Alexandra Mendes and Eng. Hugo Pereira for their invaluable guidance and support throughout this project.

I would like to extend my gratitude to Eng. Álvaro Silva for his invaluable assistance in the development of this thesis, always with solutions where I did not see any.

Furthermore, I would like to express my gratitude to my family for their unwavering support and encouragement. Finally, I would like to express my gratitude to my colleagues and friends who provided me with invaluable encouragement and support throughout the bachelor's and master's courses.

Thank you.

Renato Pedrosa de Jesus

*“The beautiful thing about learning
is that no one can take it away from you.”*

B.B. King

Contents

1	Introduction	1
1.1	Context	1
1.2	Objective	2
1.3	Methodology	2
1.4	Structure	3
2	Background	5
2.1	Design	6
2.2	Verification	6
2.2.1	Formal Verification	8
2.3	The use of High-Level Synthesis	10
2.4	The use of SystemC	11
2.5	Tools and Frameworks	13
2.5.1	HL tools	13
2.5.2	Converter	17
2.5.3	LL tools	17
3	Programming Languages	19
3.1	LL Language	20
3.2	HL Language	20
3.3	Assertion Based	21
3.4	SystemC	21
3.4.1	Transaction Level Modeling (TLM)	23
4	Implementation	25
4.1	Methodology	25
4.2	Case Studies	25
4.3	Development	30
4.3.1	SystemC Verification: HL tool	30
4.3.2	SystemC to SV conversion	33
4.3.3	SV Verification: LL tool	36
4.4	Systematic Verification	37
5	Results Analysis	39
5.1	Test Environment	39
5.2	Experimental Results	39
5.3	Discussion	41
5.4	Performance	42

5.5	Efficiency	43
6	Conclusions	45
6.1	Results	45
6.2	Limitations	46
6.3	Further Work	47
	References	49
A	Guide Installation	55
A.1	WSL	55
A.2	VerCors	55
A.3	ICSC	56
A.4	SBY	56
B	Source Code	59
B.1	SystemC	59
B.1.1	Robot	59
B.1.2	Half Adder	59
B.1.3	Full Adder	60
B.1.4	Bit 2 Adder	60
B.1.5	Bit 4 Adder	61
B.1.6	ALU	62
B.1.7	ALU 62	63
C	Terminal Output PVL	65
C.1	Robot	65
C.2	Bit 4 Adder	66

List of Figures

1.1	Implementation Data Flow	3
2.1	UVM diagram flow. Adapted from [56].	7
2.2	Vercors verification flow from [53].	15
3.1	Language Comparation, adapted from [4]	19
4.1	Final Implementation Data Flow	26
4.2	Comparison between Full Adder, Bit 2 Adder and Bit 4 Adder	28
4.3	Diagram Block of the ALU for N bits	28
5.1	Diagram of the Difference Average Experimental Results at HL and LL . . .	41
5.2	Analysis of Terminal Output from the Robot and Bit 4 Adder Assertion HL Verification Test	42

List of Tables

1	Contribution of formal verification to SDGs	v
2.1	Input languages used by 17 tools and the number of tools that use each language, analyzed in the study [35]	12
2.2	Input languages used by the 46 papers analyzed in the study [27].	12
4.1	Designs used for the tests	29
5.1	Characteristics Designs Used	39
5.2	Verification Time Experimental Results - HL Verification	40
5.3	Verification Time Experimental Results - LL Verification	40

Abbreviations

ABV	Assertion-Based Verification
ACL2	A Computational Logic for Applicative Common Lisp
ASIC	Application-Specific Integrated Circuit
AST	Abstract Syntax Tree
DUT	Design Under Test
EDA	Electronic Design Automation
ESST	Explicit-Scheduler/Symbolic-Threads
FEV	Formal Equivalence Verification
FPGA	Field-Programmable Gate Arrays
FPV	Formal Property Verification
FV	Formal Verification
HDL	Hardware Description Language
HL	High-Level
HLS	High-Level Synthesis
HVL	Hardware Verification Language
IVL	Intermediate Verification Language
LL	Low-Level
LTL	Linear Temporal Logic
OOP	Object Oriented Programming
OVN	Open Verification Methodology
POR	Partial Order Reduction
PSL	Property Specification Language
PVL	Prototypal Verification Language
RASI	Reachable Abstract States Invariant

RTL	Register-Transfer Level
SoC	Systems-on-Chip
SV	SystemVerilog
SVA	System Verilog Assertion
TB	Test Bench
TLM	Transaction Level Modeling
UVM	Universal Verification Methodology
VHDL	VHSIC Hardware Description Language
VLSI	Very Large Scale Integration
VM	Virtual Machine
WSL	Windows Subsystem for Linux

Chapter 1

Introduction

This section provides the context and motivation for this thesis. It outlines the primary objectives by presenting our research questions and concludes with an overview of the document's structure.

1.1 Context

The accelerated progress of technology has highlighted the necessity for implementing enhanced verification techniques in the field of hardware design. Moreover, there has been a notable advancement in technology, culminating in the creation of more sophisticated and innovative devices. This progress is largely driven by companies specializing in Electronic Design Automation (**EDA**).

The field of **EDA** encompasses software, hardware, and services that facilitate the definition, planning, design, implementation, verification, and manufacturing of semiconductor devices. The primary focus is on three key functions: design, simulation, and verification. Design tools facilitate the logical and physical assembly and connection of circuit elements, thereby enabling the creation of manufacturable geometric shapes. Simulation tools predict the behavior of circuits before their construction, while verification tools guarantee the logical and physical accuracy and performance of the design [52].

There are two principal approaches to verification: Functional verification and Formal Verification (**FV**). Functional Verification is a widely used technique in industry, where input data is applied, and the correctness of the outputs is confirmed. In contrast, **FV**, which has been used less frequently in the industry due to its limitations, relies on mathematical methods to rigorously analyze and verify the correctness of system outputs.

The significance of **FV** in the industry became evident in the mid-1990s, particularly following the discovery of a bug in the Intel Pentium Processor in 1994, known as the Pentium FDIV bug [15], which resulted in multi-million dollar losses. After an extensive

investigation, the issue was successfully found and solved by applying model-checking techniques, marking a pivotal advance in **FV**. This breakthrough prompted a notable increase in research activity in the field of **FV** [12].

Since then, numerous **FV** platforms and tools have been developed. Furthermore, recent advancements in High-Level Synthesis (**HLS**) research have introduced a novel and distinct level of abstraction. At this new level, design and verification processes are initiated and executed in languages such as C++ or SystemC, diverging from the conventional Low-Level (**LL**) techniques that utilize languages like Verilog and SystemVerilog (**SV**). Although the **HLS** approach does not yet fully replace **LL** synthesis and verification, it has the potential to significantly facilitate faster verification processes, making it a valuable addition to the verification toolkit.

1.2 Objective

The overarching goal of this research is to determine whether **FV** at the SystemC level can facilitate the verification of larger and more complex designs that are otherwise intractable using traditional **LL** formal verification techniques. This thesis aims to enhance the understanding of how **FV** can be scaled and applied more efficiently at higher levels of abstraction, potentially transforming verification practices for complex systems and accelerating the verification process.

To achieve this, a comprehensive investigation of the existing languages and tools currently being developed and employed at High-Level (**HL**) will be conducted. The adoption of **HL** techniques for design and verification is gradually being embraced by various companies and represents a nascent trend in the industry, though it is still in the early stages of implementation.

1.3 Methodology

The methodology employed in this research encompasses a comprehensive set of steps to ensure an evaluation between **HL** and **LL**. First, the existing tools were examined and the most appropriate ones were selected, one for **HL** and one for **LL**, taking into account their compatibility, maintenance, and relevance to the verification requirements. Another important aspect was the selection of designs to apply **FV**, this process consisted of finding open-source designs and evaluating them.

The subsequent phase involved the implementation of the designs within the framework of the chosen tools. This stage was critical in establishing a practical basis for evaluating the performance and efficacy of the **HL** and **LL** verification approaches. The implemented designs were then rigorously tested and verified using the respective **FV** tools, generating a comprehensive set of results that formed the basis for further analysis.

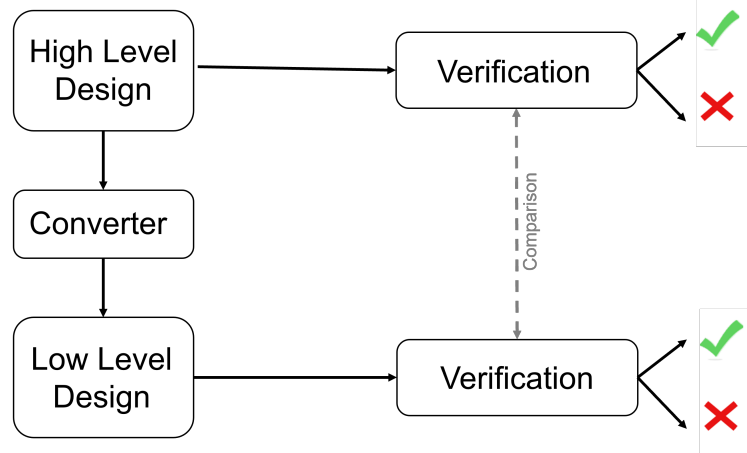


Figure 1.1: Implementation Data Flow

In the end, the results obtained from the verification processes were subjected to an analysis and discussion. This analysis focused on evaluating the effectiveness of the **HL** and **LL** verification methods, comparing their performance, and the time-consumed on the verification process. Our discussion of the results aims to elucidate the implications of the findings, providing insights into the advantages and limitations of each approach and offering recommendations for future research and practical applications in the field of formal verification. This process is summarized in Figure 1.1.

1.4 Structure

This thesis is structured as follows. Section 2 provides an overview of the state of the art, examining the origins of the **EDA** industries and **FV** techniques and offering an explanation of some of the principal **FV** tools. Section 3 clarifies the languages used in design and verification by the **EDA** companies. Section 4 outlines the implementation of the work, delineating the procedural steps and the instances of re-evaluation. It also identifies the challenges encountered and the strategies employed to address them. Section 5 presents the study results, along with an analysis of their limitations and a discussion of the results. Finally, section 6 provides a conclusion to the document, a review of the limitations, and suggestions for future research.

Chapter 2

Background

In recent years, there has been a notable advancement in technology worldwide, resulting in the development of more sophisticated and promising devices. One of the key drivers of this technological advancement is the work of companies specializing in **EDA**.

The **EDA**, as defined by Synopsys ¹, “*is a market segment consisting of software, hardware, and services with the collective goal of assisting in the definition, planning, design, implementation, verification, and subsequent manufacturing of semiconductor devices, or chips*” [52].

It has three main functions: design, simulation, and verification. Design tools facilitate the assembly and connection of circuit elements logically and physically, thereby creating interconnected geometric shapes for manufacturing purposes. Simulation and verification are typically conducted together, although they are distinct processes. Simulation tools predict the behavior of proposed circuits before implementation, whereas verification tools ensure the logical or physical accuracy and performance of the design [52].

The **EDA** history began in the early days of integrated circuits when designs were done manually. Automation came first with geometric software to generate tapes for photoplotters, with companies such as Calma leading this era. By the mid-70s, tools were available to automate design processes such as place and route [52].

The **EDA** industry formally began in 1981 with the founding of companies like Daisy Systems, Mentor Graphics, and Valid Logic Systems. This period saw the rise of the Application-Specific Integrated Circuit (**ASIC**) business, pioneered by Very Large Scale Integration (**VLSI**) Technology and LSI Logic, and the semiconductor model introduced by TSMC [52].

The current leading players in the **EDA** field are Synopsys, Siemens EDA ², and Cadence ³. Additionally, there is an important and independent non-profit organization

¹<https://www.synopsys.com/>

²<https://eda.sw.siemens.com/>

³<https://www.cadence.com/>

called *Accellera*⁴ whose objective is to create, support, promote, and advance system-level design, modeling, and verification standards ensuring its wide adoption and compatibility by the EDA companies [41].

Several concepts and ideas have been developed over the years, and it is necessary to explain them in detail. This will be done in the following sections.

2.1 Design

The ASIC design is a complex process that involves several critical stages. These stages are designed to ensure that the final product meets specific requirements and performs optimally [46].

The design process commences with the specification stage, during which the design's parameters, including area, functionality, timing, cost, and applications, are delineated. The next crucial stage is Register-Transfer Level (RTL) coding, which describes the design in terms of synchronous logic. The RTL code includes registers that hold state information, combinatorial logic for defining the inputs that will determine the next state and an input clock that controls the state changes.

This code is typically written in Verilog, VHSIC Hardware Description Language (VHDL), or SV, which are explained in detail in Section 3.

Synthesis follows, converting the RTL design into a gate-level netlist, which represents the interconnection of basic logic gates such as *nand*, *and*, and *nor* [46].

The RTL design is currently being studied in greater depth through the lens of the HLS approach. The HLS approach involves the translation of an abstract behavioral specification of a digital system [32], written in high-level languages like C++ or SystemC, into an RTL design [27].

A study that compares HLS with RTL in 2018 [27] suggests that the latest generation of HLS tools exhibits inferior performance and resource utilization compared to manual RTL. However, there is considerable variation in the outcomes, with HLS demonstrating equivalence or superiority to the RTL approach in approximately 40% of the evaluated cases. This study suggests the importance of HLS research which will be the focus in section 2.3 and will serve as a focal point of this thesis.

2.2 Verification

Verification is the process of ensuring the correct operation of the design. This process is complex and requires careful planning, as it spans the entire design life cycle. Despite variations in industrial practices, the fundamental steps of design verification remain consistent across the industry [8].

⁴<https://www.accellera.org/>

In the first years, verification was part of the design process, performed post-code completion by the designer. This worked for small designs but became inefficient as designs grew in complexity. Advanced teams recognized the need to treat verification as an independent task, improving process efficiency and result quality. Today, verification is conducted in parallel with design, allowing for immediate testing upon development. This approach, involving multiple complex testbenches and integration layers, reduces overall development time and enhances the effectiveness of the verification process [57].

One of the industry's most recent standards was defined in 2010 and is known as the Universal Verification Methodology (UVM) [21]. The UVM has been developed to verify ASICs, full custom designs, and Field-Programmable Gate Arrays (FPGA)-based designs. It provides a unified platform through UVM base class libraries, which verification engineers utilize to develop intricate test benches. UVM's adaptability enables its constructs and capabilities to be tailored to specific project requirements. The methodology places a strong emphasis on reusability, to reduce verification time, which is of crucial importance in meeting the industry's demand for rapid time-to-market [56].

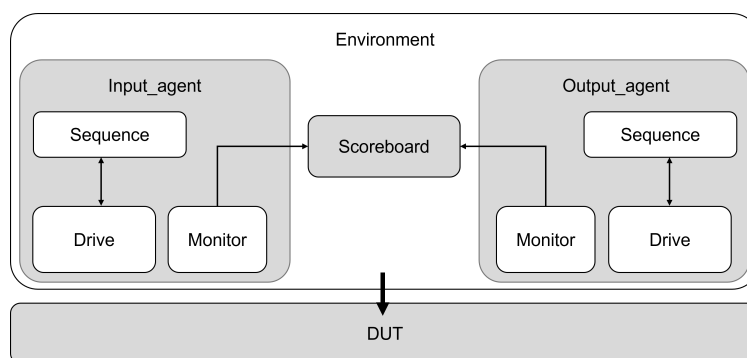


Figure 2.1: UVM diagram flow. Adapted from [56].

The UVM is composed of numerous elements, including the Design Under Test (DUT), Environment, Agent, Scoreboard, Sequence, Drive, and Monitor. These elements are illustrated in Figure 2.1. The DUT represents the hardware design being verified. The Environment is the primary verification component that encapsulates multiple Agents, providing configuration and reusability. In the figure example, two Agents are shown: Input and Output Agents. In some cases, the complexity of the tests necessitates the inclusion of additional Agents to facilitate the execution of additional tests. An agent is an abstract container comprising a Driver, Sequencer, and Monitor. It can be configured as active, driving transactions, or passive, only monitoring DUT activity. The Driver actively drives the DUT by applying stimuli based on the data items received. The Sequencer controls the order and generation of these data items, providing structured and meaningful test patterns. The Monitor passively observes DUT signals to collect coverage information and perform checking, ensuring protocol adherence and capturing transactional data. The

Scoreboard is used for data checking by comparing the expected outcomes with the actual **DUT** outputs, ensuring correct functionality [21].

This technique is a simulation-based verification, which is commonly referred to by the industry as functional verification. Another significant approach is **FV**, which employs mathematical techniques to demonstrate the functional correctness of a design. The **FV** is founded upon advanced theoretical and mathematical sciences, which often exceed the comprehension of typical verification engineers. The intricate nature of the underlying technology contributes significantly to the complexity and confusion surrounding discussions on it [57].

2.2.1 Formal Verification

FV is the process that employs mathematical techniques to rigorously analyze and affirm that a system, whether hardware or software, adheres to its specified requirements and operates correctly under all conditions [57].

The **FV** relevance for the **EDA** industry started in the mid-90s when a bug was identified in the Intel Pentium Processor in 1994, designated Pentium FDIV [15] and cost more than 400 million dollars change in to at that time. Following several months of investigation, the issue was resolved through the use of model-checking techniques. This represented a pioneering and pivotal advance in the evolution of **FV** [12].

Since then, a multitude of **FV** platforms have been developed to accommodate formal analysis. For example, the VC Formal from Synopsys ⁵ or Jasper from Cadence ⁶. In fact, “*FV techniques have reached a level of maturity that few believed possible when they were first introduced*” [45, p. 3].

The **FV** struggles with complexity issues, particularly the satisfiability problem, which can require exponential time, making it impractical to implement programs that can handle all conceivable cases efficiently. Additionally, mathematical constraints, such as Gödel’s Incompleteness Theorem and Turing’s Halting Problem, imply that a universal **FV** tool guaranteeing correctness for all specifications is distant [45].

These complexity challenges do not outweigh the inherent power and numerous benefits of this method which represents an engineering challenge [45]. This has driven the development of higher levels of abstraction, the **HLS**.

The use of **FV** is not always the optimal solution, as it is quite intricate to process data (involving mathematical operations and sequential programs), such as calculating floating point units and DSP designs. Conversely, it is a suitable solution for the design of arbiters, bus bridges, direct memory storage, schedulers, host units of buses, interrupts, memory controllers, tag generators, and credit managers [43].

⁵<https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html>

⁶https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html

The following section will present the two principal **FV** techniques: deductive verification and model checking.

2.2.1.1 Deductive Verification

Deductive verification is the method of confirming that the implemented system meets the design requirements through the use of mathematical reasoning. It is a proof-based approach to formal verification, which is also known as theorem proving [45].

The first-order theorem provers began in the late 1960s, with the work of Tony Hoare [24], McCarthy [31], Naur [36], Floyd [18], and, Manna [30] [55]. This prompted considerable academic interest and subsequent research throughout the 1970s and produced the first successful prover in 1972 by Robert S. Boyer and J. Strother Moore which became the basis for the nowadays A Computational Logic for Applicative Common Lisp (**ACL2**), a logic and programming language. However, these techniques were largely perceived as being unsuitable for practical use in industry due to their lack of scalability [45].

The industry's adoption started in the 1990s with the use of Formal Equivalence Verification (**FEV**), becoming the first commercial viable technique. It is a method of comparing two models and determining their equivalence. The models may be either high-level models, **RTL**, or gate-level netlists. Alternatively, the comparison may be conducted between two levels of abstraction [45]. This technique is based on equivalence checking, which is a type of deductive verification. Today, applications can be classified into three distinct categories:

- *RTL-Netlist FV* is used to check that a gate-level netlist results correctly from the synthesis [45].
- *Feature Change Verification* plays with the increasing reuse of existing **RTL** models. Often, some part is modified or adapted for new use cases while still expected to maintain functionality for existing features. The comparison between the original **RTL** and the modified version, ensures that the previously defined functionality continues to work even with the introduction of new features [45].
- *High-Level Model Equivalence Verification* refers to the comparison of a high-level model to an **RTL** model. This kind of comparison involves many challenges, given the potential for higher abstraction in the former. This task represents a new and emerging technology in the field [45].

2.2.1.2 Model checking

Model checking is an automated verification method for finite state systems, focusing on ensuring a system conforms to a temporal logic specification. Unlike earlier theorem-proving which requires user guidance, model checking utilizes efficient graph search algorithms to determine if a system's state graph satisfies its specifications. This approach

effectively reasons about infinite behaviors within finite state systems. If the system conforms to the specification, it confirms its correctness; otherwise, it identifies violations and provides counterexamples for debugging. Consequently, it serves to bridge the theoretical and mathematical verification with the practical automated analysis [17].

This research started during the 1970s and early 1980s with notable contributions from Owicki and Gries in 1976 [39], Pnueli in 1981 [42], Clarke and Emerson in 1981 [11], and Owicki and Lamport in 1982 [40].

It is noteworthy that model checking automates the verification process, thereby eliminating the need for users to construct correctness proofs, which can be time-consuming and complex. It is typically more expeditious, considerably reducing the time required for verification. Furthermore, it furnishes diagnostic counterexamples when specifications are not met, which are of paramount importance for debugging complex systems. Furthermore, it is capable of handling partial specifications effectively, thereby enabling verification to occur during the design phase [17].

Conversely, model checking is not without its shortcomings. A significant drawback of model checking is the state explosion problem, which arises when the number of global states in a concurrent system is vast. This can complicate the verification process. This issue has constituted a significant driving force behind much of the research and development in the field of model checking, intending to improve its scalability and efficiency [17].

Despite these challenges, model checking remains a valuable and widely used technique. In the current industrial context, the most prevalent methodologies that employ this technique are as follows:

- Assertion-Based Verification (**ABV**) involves utilizing assertions, typically articulated in languages such as System Verilog Assertion (**SVA**), to delineate properties that must hold true for **RTL** designs [45].
- Formal Property Verification (**FPV**) encompasses the utilization of formal tools to validate assertions. This technique is highly versatile and can be subdivided into various additional methodologies, like *Early Design Exercise FPV*, *Signoff Quality FPV*, *Bug Hunting FPV* and *Unreachable Coverage Elimination* [45].

2.3 The use of High-Level Synthesis

The recent advances in silicon chip technology have enabled the creation of highly dense Systems-on-Chip (**SoC**) [14]. This initiated a search for the most expedient methods to meet the deadlines, leading to the search for **HLS**.

The field of **HLS** was first investigated in the 1980s, with significant advances made throughout the 1990s. The initial tools were oriented towards data flow designs, addressing issues such as scheduling, which becomes exponentially more complex with system

complexity. Encountered difficulties in dealing with complex control flows and nested structures, frequently resulting in suboptimal hardware implementations [13].

The current tools have demonstrated better results and have already been used in numerous real-world applications [26]. However, they are not widely accepted by the industry, particularly for large applications with complex module and control-flow hierarchies. Despite their potential benefits for real-world applications, they often produce less optimal solutions due to synthesis heuristics aimed at reducing optimization and processing times [13]. Currently, there are a number of commercial tools available, including Catapult HLS, Symphony C Compiler, CyberWorkBench, and Stratus HLS [26]. Unfortunately, these tools do impose certain limitations, such as the exclusion of while loops, and complex user interfaces, which limit their accessibility for small and medium-sized enterprises. Also, academic tools like SPARK accept only small subsets of ANSI-C and focus on optimizing conditional source code methods [13].

Another aspect to consider is that the initial learning curve for adopting these tools can be considerable, as designers must become proficient in new methodologies and frameworks. While it facilitates numerous aspects of hardware design, it may not consistently perform as effectively as traditional methods when it comes to low-level optimizations [14].

Regarding the design effort of make HLS and synthesize in RTL compared to hand-written RTL, Joentaken, in [26], posits that the principal advantage is the enhanced productivity in frontend design, as the designer can concentrate on the functionality rather than the implementation details. Based on the outcomes of previous case studies where HLS has been contrasted with traditional RTL design, a considerable number of them have demonstrated a productivity increase of between two and five times.

The application of HLS for design and verification reveals that it is still limited which indicates that it is still in an early stage of development and requires further investigation [13].

2.4 The use of SystemC

Despite the efforts of the *Accellera*⁷, there is no consensus in the industry as to which HL language is the best. This situation raises the question of which language should be used.

A survey of FPGA HLS tools conducted by Razvan, and et al in 2016 [35] revealed that 17 tools are currently in use, with four of these being academic tools.

Table 2.1 presents an overview of the input languages supported by the 17 tools and the number of tools that use them. It is noteworthy that some tools are capable of supporting different types of input language types.

The information in Table 2.1 indicates that C is the most frequently used, being used in seven different tools. Additionally, C++ and SystemC are employed in five and three

⁷<https://www.accellera.org/>

Languages	Number
BSV	1
C	7
C++	5
C subset	3
Impulse-C	1
Handle-C	1
SystemC	4
MaxJ	1
BDL	1

Table 2.1: Input languages used by 17 tools and the number of tools that use each language, analyzed in the study [35]

tools, respectively, while the remaining tools, BSV, MaxJ, and BDL, are utilized in a single tool each.

Another study, in this case by Sakari Lahti et al. [27] in 2019, analyzed 46 papers that compare the HL with LL design flows. The HL languages used by these papers were employed in the construction of the table 2.2.

Languages	Number
Bluespec (BSV)	2
CAPH Language	2
C	8
C/C++/SystemC	23
HIPAcc language	2
HT language	1
Java	1
Matlab	4
OpenCL	3
Scala	2

Table 2.2: Input languages used by the 46 papers analyzed in the study [27].

Table 2.2 shows the number of articles that use a specific HL language. Also suggests that a considerable number of papers utilize languages such as C, C++, or SystemC as their input.

A comparative analysis of the most commonly used languages in both studies revealed that C, C++, and SystemC were the most prevalent. SystemC, developed and maintained by *Accellera*, has gained significant industry acceptance and is now the most commonly used language [41, 10]. It was therefore selected for this thesis.

2.5 Tools and Frameworks

This section examines the tools and frameworks that were developed in recent years for **FV** at both high and low levels of abstraction. It also presents some converters from SystemC to **SV**.

2.5.1 HL tools

The literature under review here explores formal approaches, although not all of these are in SystemC. Many of the approaches require a translation to some Intermediate Verification Language (**IVL**) representations, which usually represent only a subset of the features of SystemC [29]. This subsection examines those that have the greatest potential for future use.

The **CTSC** framework is an automated and scalable approach designed for the validation of SystemC designs at the binary level. The framework generates new test cases by executing binaries with an initial test case and integrating assertion-based verification techniques. This process ultimately produces a comprehensive test report that highlights assertion violations and test case status [29].

It offers several advantages, including its automation, scalability to practical design sizes, and compatibility with all SystemC language features without requiring extensive translation efforts. Additionally, CTSC streamlines the deployment process, requiring minimal engineering effort to apply to existing SystemC projects [29].

However, challenges such as constraint solver failures and over-concretization issues still need to be addressed to fully optimize the performance and effectiveness of the framework [29].

The **KRATOS** system is a SystemC model checker that translates SystemC designs into threaded C models, which are then subjected to a combination of an explicit scheduler and symbolic lazy predicate abstraction [29]. The system accepts inputs in threaded C and utilizes an explicit scheduler and symbolic lazy abstraction for model checking. Additionally, Partial Order Reduction (**POR**) is integrated into the scheduler to prune redundant schedules [23].

Furthermore, integrates both sequential and concurrent analyses, each meticulously tailored to verify SystemC designs with unparalleled precision. While the sequential analysis is comparable to existing software model checkers based on lazy predicate abstraction, the concurrent analysis represents a significant advance in the field through the Explicit-Scheduler/Symbolic-Threads (**ESST**) algorithm, which demonstrates superior performance in SystemC benchmark verification [9].

Nevertheless, the latest update to the software has resulted in a new version with a new language, K2, which necessitates additional steps to run the SystemC code⁸.

⁸<https://es-static.fbk.eu/tools/kratos/>

The **SCOOT** is a specialized model extractor tailored for SystemC, a system-level modeling language implemented as a C++ library. Leveraging a C++ front-end translates SystemC source files into a control flow graph enriched with assignments and guards, facilitating subsequent static analysis techniques to extract critical SystemC-specific information such as module hierarchy, process sensitivity lists, and port bindings [5].

A notable advantage of this approach is its capability to conduct static scheduling for dynamic verification, which has the potential to enhance performance compared to traditional dynamic scheduling methods. Moreover, its **IVL** originally designed for model checking, enables code re-synthesis by translating back to a flat C++ program independent of the SystemC library, thereby reducing the time required for simulation [5].

However, the complexity of the C++ language and the intricacies of SystemC pose obstacles in tool development, necessitating ongoing efforts to bolster SystemC support and explore additional formal techniques for further advancements in static scheduling [5]. It implements a simplified library due to the heavy usage of object-oriented features in the SystemC library [29].

The **SPIN** is an open-source software verification tool that has gained considerable recognition for its capacity to perform formal verification on multi-threaded software applications. Its development commenced in the 1980s and has since undergone numerous enhancements [16].

This tool was designed to check the distributed systems and features two primary data inputs: the code, written in the **IVL** Promela, which describes the representative automaton of the SystemC model, and Linear Temporal Logic (**LTL**) specifications delineating the properties to be verified [50] with a graphical front-end tool called JSPIN for user interaction [16].

The tool employs **POR** to verify models, aiming to mitigate the state space explosion in concurrent systems with asynchronous components. **POR** leverages the insight that numerous events are independent and can be executed in any order without altering the computation's outcome. However, despite these efforts, the verification speed does not exhibit a clear increase [50].

The **STATE** tool translates SystemC designs into timed automata, which are then verified using the UPPAAL model checker [29, 23, 22].

The transformation process preserves the structure of the original SystemC design, ensuring that it remains transparent to the designer. The tool supports a range of SystemC language elements, including process execution, interactions between processes, dynamic sensitivity, timing behavior, structs, arrays, pointers, static and dynamic memory management, and the Transaction Level Modeling (**TLM**) 2.0 standard [22].

However, it does not directly verify properties on SystemC designs; instead, properties must be formulated on the automata. STATE may not perform well with symbolic inputs and has limited compatibility with common SystemC benchmarks [23].

The **VerCors** is a deductive verification toolset designed to aid developers in writing reliable concurrent software by providing mechanized verification across various concurrency models [6]. This tool addresses the challenge of managing concurrent software by leveraging permission-based separation logic to ensure functional behavior and data race freedom. VerCors aims to verify the absence of data races and the proper handling of concurrent data structures [2].

VerCors is designed to support a range of languages, including Java, Prototypal Verification Language (**PVL**), OpenCL, and OpenMP. **PVL** enables the development of new features and facilitates the integration of new concurrent languages by developing parsers that convert their code into the intermediate language. This design feature enhances the extensibility of VerCors [6].

Tasche et al. paper [53] examine the potential of VerCors for use in SystemC verification. The parameterized SystemC design is automatically transformed into PVL. The designer can then formally analyze crucial safety properties by adding specifications to the transformed program, while the Reachable Abstract States Invariant (**RASI**) restricts the analysis to the relevant part of the state space [53]. This is illustrated in Figure 2.2.

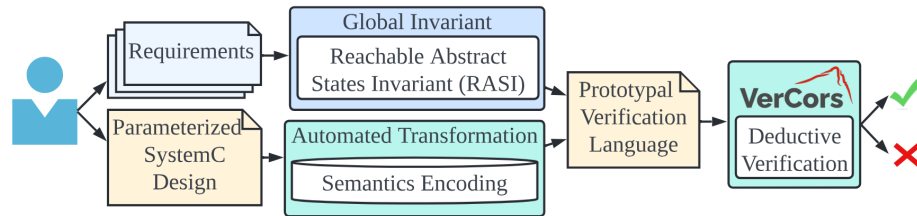


Figure 2.2: Vercors verification flow from [53].

One of the principal advantages of VerCors is its capacity to accommodate a multitude of concurrency mechanisms and sophisticated language features, rendering it adaptable and suitable for real-world programming languages. Furthermore, VerCors employs a design-by-contract methodology, whereby annotated code is transformed into verification problems, thereby guaranteeing the original program's correctness [6].

Nevertheless, there are also some disadvantages. While the tool is designed to support realistic Java programs and other languages, achieving complete verification for full programming languages represents a significant engineering challenge [6]. Despite these challenges, VerCors continues to evolve, aiming to provide comprehensive verification for concurrent software [2].

2.5.1.1 Common Aspect Between frameworks

There are numerous common elements between the frameworks. This section will analyze the key characteristics that differentiate them:

- **Scheduler Integration** - The scheduler oversees the set of execution states in a concurrent system, selecting and exploring unvisited execution states by handling all possible interleavings. It delegates control to the interpreter, which is responsible for executing each runnable thread. This implementation of symbolic execution enables the detection of errors. Control is returned to the scheduler when execution reaches the end of a thread or a wait statement, marking the state as visited and updating it. Alternatively, control is returned when encountering a conditional goto, which splits the state based on the condition [28]. It's used by SCOOT and VerCors to integrate into their verification approach.
- **POR** - is a technique employed to enhance the efficiency of thread scheduling sequence exploration by reducing the number of states that require examination while maintaining the integrity of the verification outcome [28]. It is used by KRATOS, integrated into its explicit scheduler, to eliminate redundant schedules during model checking. Similarly, it is employed by SPIN to mitigate the state space explosion in concurrent systems, thereby ensuring efficient model verification.
- **Delta-Cycle Simulation** - Delta cycles in SystemC simulations represent a specific iteration within the scheduler, employed to stabilise the system state prior to advancing the simulation time. They comprise three phases: evaluation, update and notification. During the evaluation phase, all executable processes are executed until no further processes remain. The update phase is responsible for processing any scheduled state changes, which typically affect signal assignments. In the notification phase, any processes that are sensitive to events that have occurred are identified and made runnable. Delta cycles are repeated until there are no longer any runnable processes, thus ensuring that all intermediate states are resolved before advancing the simulation time. This mechanism assists in maintaining the accuracy and consistency of the simulated model's state transitions [5]. SCOOT uses it to improve performance and reduce the time required for simulation.
- **Translation to IVL** - many tools convert SystemC into intermediate languages in the process of verification, like SPIN into Promela, and STATE into timed automata.
- **Automation and Scalability** - Especially by CTSC, efficiently generates new test cases and integrates assertion-based verification without requiring extensive translation efforts, and by VerCors, designed to support various concurrency mechanisms and languages, it ensures functional behavior and data race freedom through its automated verification process.

Each of the frameworks discussed offers unique strengths and approaches to verifying SystemC designs. Nevertheless, VerCors stands out for its comprehensive support for different concurrency models and languages, its robust approach to ensuring functional behavior and data race freedom, and its extensibility to new concurrent languages. These features make VerCors an ideal choice as it addresses the complexities of concurrent software verification while providing developers with a versatile and powerful toolset.

In contrast to the other tools, the VerCors was not developed with the specific purpose of verifying designs. Instead, it was designed to be more flexible and accommodate a wider range of verification techniques, including direct verification of Java programs. Besides, it can also be regarded as a valuable tool for verifying designs and was therefore selected as the most appropriate choice.

2.5.2 Converter

The necessity to test the same code at both the **HL** and **LL** led to the investigation of the most appropriate converters.

The **ICSC** is a tool designed to enhance the productivity of design and verification engineers by translating cycle-accurate SystemC into synthesizable **SV** [33].

The tool has several key advantages: support for arbitrary C++ at the design stage, compatibility with C++11/14/17 standards, and generation of human-readable Verilog code. This readability is maintained by preserving variable names, process labels, and control flow structures from the original SystemC source, facilitating rapid debugging and verification [33].

Another compiler that is worthy of consideration is the **SCCL**. It is an open-source tool designed to translate SystemC designs into synthesizable **RTL** code. It utilizes LLVM's Clang frontend to parse SystemC designs, creating an intermediate Abstract Syntax Tree (**AST**) representation called Hcode [59].

This tool just supports a subset of *Accellera's* SystemC synthesis standard, emphasizing complex C++ constructs such as templated types, class hierarchies, inheritance, and virtual functions [59].

Given the lack of support for SCCL and its limitations, the ICSC was selected as the converter tool to use.

2.5.3 LL tools

At the **LL**, there are already some highly efficient commercial frameworks to perform **FV**, such as the VC Formal from Synopsys⁹ or Jasper from Cadence¹⁰. However, due to licensing requirements, the emphasis in this discussion will be on open-source tools:

⁹<https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html>

¹⁰https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html

The **NuXmv** is a symbolic model checker designed for both finite and infinite-state synchronous transition systems. It is the evolutionary successor to the open-source NuSMV model checker, enhancing its predecessor by integrating state-of-the-art verification algorithms and extending its capabilities to support new data types such as integers and reals [7].

The NuXmv has been optimized for performance to ensure competitiveness with contemporary model checkers and has been effectively utilized in various industrial projects for verification, requirements analysis, contract-based design, hybrid systems model checking, and safety assessment [7].

The **Verilog2SMV** is an open-source tool designed to enhance the verification of hardware designs by converting Verilog designs, augmented with SystemVerilog assertions, into word-level model checking problems suitable for RTL verification [25].

This tool utilizes Yosys for Verilog synthesis and leverages the NuXmv model checker for advanced verification, creating a robust toolchain for ensuring design accuracy [25].

A significant advantage of this tool is its ability to handle memories without abstraction, preserving the detailed structure of the original design and thus facilitating more accurate verification [25].

The **Yosys** is a pioneering open-source software suite for Verilog HDL synthesis, designed to translate Verilog HDL code into digital circuits. It supports the majority of synthesizable Verilog features, providing a comprehensive solution for the synthesis of complex, real-world Verilog designs. The Yosys is built as an extensible framework, making it adaptable for custom synthesis flows and enabling research on new synthesis algorithms [58].

Additionally, there is another tool, **SBY** or SymbiYosys¹¹, which is the front end of Yosys, a tool developed by the same authors. SBY is constituted by an input file containing instructions to be applied in Yosys.

Finally, the **AutoSVA** tool is a framework designed to automate the generation of **FV** Test Bench (**TB**) for verifying forward progress in **RTL** modules. AutoSVA then generates properties to verify the transaction's liveness (ensuring that every request is eventually followed by a response) and safety (validating the response attributes). The **TB** generated can then be supplied to an **FV** tool like SBY. This approach significantly reduces the engineering effort required for **FV**, making it more accessible to hardware designers and promoting its wider adoption for ensuring **RTL** module interface correctness [38].

The thesis will present a test of the use of AutoSVA in automating **LL** tests. The formal tool SBY will be employed for this purpose, as it represents the front end of the most academically accepted tool.

¹¹<https://yosyshq.readthedocs.io/projects/sby/en/latest/>

Chapter 3

Programming Languages

This chapter will elucidate the principal languages used in the EDA companies and those most important in the context of the state of the art.

The language constitutes one of the principal elements of the design and verification process. A significant proportion of these languages have been standardized and are in use across the entire EDA companies.

Figure 3.1, which has been adapted from [4], presents a comparative analysis of the application of these languages, including requirements, architecture, hardware/software, behavior, verification, RTL, gate, and transistors. This illustration demonstrates the comprehensive scope of the subject matter.

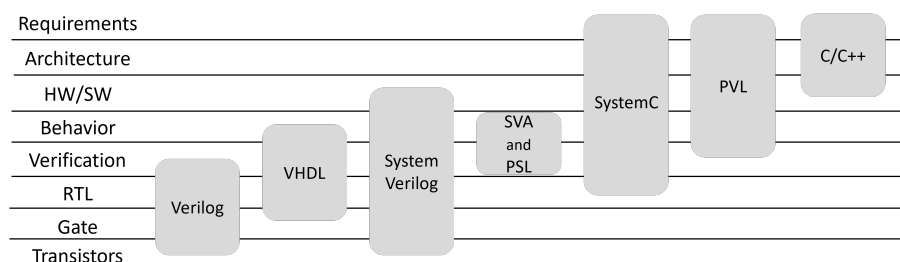


Figure 3.1: Language Comparison, adapted from [4]

The languages are divided into two main groups, LL and HL Languages, which are applied in different design and verification phases. The following section will discuss them.

3.1 LL Language

- **VHDL** - developed in the 1980s and offers designers a robust platform for describing digital systems. The use of this technology has been replaced by the use of Verilog [47].
- Verilog - is a Hardware Description Language (**HDL**) that became public domain in 1990 and the first IEEE standard was in 1995. It enjoys widespread adoption for both **ASIC** and **FPGA** [47].
- **SV** - represents a standardization initiative by *Accellera* to ensure that Verilog remains aligned with the evolving complexities of design and verification challenges. It is often referred to as Hardware Verification Language (**HVL**) and incorporates Object Oriented Programming (**OOP**) features that facilitate the execution of intricate testing procedures [44].
- Open Verification Methodology (**OVM**) - emerged as an open-source verification methodology for digital designs and **SoC**'s. It's a **HVL** based on **SV** [3].
- **UVM** - is the successor of **OVM**, built upon the concepts and principles of the latter, which was introduced in 2010 with its first release. Designed to offer a standardized and adaptable approach, **UVM** can be easily adapted to different verification environments and use cases [51].

3.2 HL Language

- C++ - mainly is a **OOP** language with a multitude of applications in various fields [20].
- SystemC - is a standard based on C++. Designed for use by designers and architects who need to address complex systems [1]. Includes module hierarchy, specific data types, time modeling, and communication mechanisms. Modules in SystemC interact via communication ports and channels, with support for a range of hardware data types and time units [19]. Further details can be found in section 3.4.
- PVL - is based on Java and serves as the input language for the VerCors tool¹. It will be employed in this thesis.

¹<https://vercors.ewi.utwente.nl/>

3.3 Assertion Based

The concept of assertions cannot be categorized as either **LL** or **HL** languages, even though they can be used in both cases. There are two main assertion languages:

- **SVA** - is a component of the **SV** language, providing a means to embed executable checks. It offers a more efficient approach to verification by capturing design behavior cycle-by-cycle. This real-time verification capability significantly improves bug detection and debugging, thereby reducing turnaround time in the design process. **SVA** supports two types of assertions: immediate and concurrent. Immediate assertions test expressions when executed in procedural code, behaving similarly to conditional statements. Concurrent assertions, in contrast, describe behavior over time and are evaluated only at clock ticks [48].
- **Property Specification Language (PSL)** - is a distinct language that has been specifically engineered to function seamlessly with various **HDLs** and their expression layers. In contrast to **SVA**, **PSL** cannot be integrated directly into **HDL**. However, it can be attached to **HDL** models using binding directives, thus enabling its inclusion in **HDL** with tool support. It is employed by system design firms to validate high-level system descriptions against architectural specifications. Furthermore, it has been extended to domains encompassing both analog and mixed-signal systems. In summary, **PSL** is a versatile, multi-level, and multi-flavor assertion language that provides flexibility and adaptability across various verification contexts [48].

3.4 SystemC

SystemC employs an object-oriented paradigm, which is a fundamental aspect of its design methodology. This approach improves abstraction, modularity, compositionality, and reuse, making it a robust tool for modeling complex systems. The object-oriented features of SystemC distinguish it from other modeling languages, facilitating more efficient and flexible system design [54].

The semantics of SystemC are derived from the combination of the semantics of C++ and the simulation semantics of the kernel [20]. Beyond this, SystemC functions as a comprehensive library of classes and macros within C++. An example of SystemC code is provided in Listing 3.1 shows.

```

1 int sc_main () {
2     Example exp("Exp0");
3     sc_start(20 ,SC_MS);
4     return(0);
5 }
6

```

```

7 SC_MODULE(Example) {
8     sc_in<int> a;
9     sc_out<bool> b;
10
11     int c;
12
13     SC_CTOR(example) {
14         SC_METHOD(function1);
15         SC_THREAD(function2);
16     }
17
18     int function1() {
19         //process
20     }
21
22     void function2() {
23         //process
24     }
25 };

```

Listing 3.1: Example SystemC design

Follow the Listing 3.1, the fundamental unit in SystemC is the module, *SC_MODULE*, which serves as a container for one or more processes that describe the parallel behavior of the design. Modules can also contain other modules, called sub modules, reflecting the hierarchical nature of the design [54]. Within the main routine, “sc_main”, modules and sub modules are instantiated and connected via “in”, “out”, and “inout” ports. Ports act as pointers to signals, and signal changes trigger events in the event-based simulation [34].

The behavioral structure is defined through three types of processes: *SC_METHOD*, *SC_THREAD*, and *SC_CTHREAD*, which communicate in parallel under the supervision of a simulation kernel. In essence, *SC_METHOD* is analogous to synthesizable HDL, it cannot suspend execution and are expected to terminate, returning control to the kernel. *SC_THREAD* and *SC_CTHREAD* correspond to behavioral HDL and can suspend execution using wait statements and run once during program execution [19, 54, 37].

The kernel coordinates the execution of the process and the progression of the simulation time. Processes only assign new signal values during the next simulation cycle, ensuring proper event synchronization. The “sc_start” command initializes the simulation by assigning initial values to the signals, after which the kernel and user-defined processes alternate in execution. Processes remain active until encountering a wait statement or completing execution. Depending on their classification (method, thread, or clocked thread), the processes follow different conditions to determine when to resume [34].

The SystemC simulation kernel operates within a discrete time model, where time is linearly ordered. It ensures that when all user-defined processes are suspended, the

kernel continues its task by advancing simulation time, updating signals, and reactivating processes when events are triggered [34]. In essence, it is executed in three phases: elaboration, execution, and cleanup. During elaboration, modules are instantiated and connections between them are established. Execution simulates process concurrency, with updates applied at each delta cycle until no further processes need evaluation [19].

Furthermore, it provides a unified design environment by providing C++ libraries, models, and tools that facilitate hardware-software co-design. The common platform enables the design process and enhances efficiency, allowing designers to work within a cohesive and integrated environment [41].

Its methodology facilitates the reuse of **TB** across different levels of modeling abstraction, thereby enhancing efficiency and reducing testing time [10]. It provides robust mechanisms for specifying abstract communication protocols, crucial for accurately representing complex system interactions and behaviors [41].

SystemC has emerged as the *de facto* standard for semiconductor industry modeling, with a substantial and diverse array of industry stakeholders endorsing its use. This widespread acceptance serves to illustrate the effectiveness and reliability of SystemC [10, 29].

3.4.1 Transaction Level Modeling (TLM)

TLM is one of the modern approaches for system level design [49], which allows for abstract communication between components [19]. Designing with **TLM** allows interactions between modules to be represented at a higher level of abstraction than traditional **RTL** design. By hiding implementation details, **TLM** significantly reduces simulation times, making it a popular choice for early system evaluation and performance analysis [49].

SystemC supports **TLM** through an additional library and a set of primitives collectively known as the interoperability layer. This layer includes sockets, interfaces, payloads, and a base protocol. Sockets act as the entry points for modules (initiators and targets) into the transaction mechanism. The interface provides blocking and non-blocking calls for data transport. The payload is the data structure that contains the data or commands exchanged between the modules. Finally, the base protocol is a set of rules that manage the communication to ensure high simulation performance [49].

The **TLM-2.0** introduces the concepts of transactions and temporal decoupling, enabling faster and more abstract system design and verification. Transactions represent abstract communication between modules, while temporal decoupling allows processes to run ahead in local time, improving simulation speed. **TLM** can be implemented with different timing models, such as loosely timed and approximately timed, each suited to different use cases [19].

Chapter 4

Implementation

This chapter details the methodology employed for the verification process, delineating the steps taken to obtain a systematic approach and the techniques used to ensure accuracy.

4.1 Methodology

The method employed starts with finding the most appropriate designs in SystemC. Then the design is parameterized and converted into **PVL**. This was followed by debugging and verifying using the VerCors tool.

Subsequently, the SystemC design was transformed into synthesizable **SV** using the ICSC tool. A **TB** was constructed and the SBY tool was employed.

Finally, a Python file was constructed to conduct a systematic verification of the designs for **HL** and then **LL**. This approach enabled the calculation of an average verification time and a comprehensive analysis.

The aforementioned process is depicted in Figure 4.1. Further details regarding the installation process can be found in Appendix A, all the SystemC designs in Appendix B, and more detail in the thesis's Artifact¹.

4.2 Case Studies

The selection of designs represents a critical stage in the implementation process. Key considerations included identifying reliable sources for the designs and ensuring that these sources could be trusted. Among the most prominent and diverse platforms identified was OpenCores². Additionally, GitHub was recognized as a valuable resource, however

¹https://github.com/Renato1818/ll_hl_verif.git

²<https://opencores.org/>

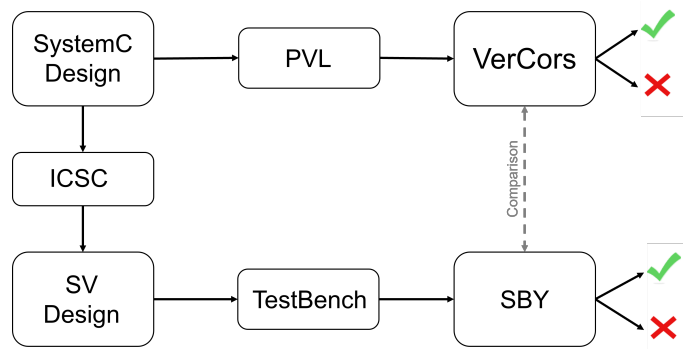


Figure 4.1: Final Implementation Data Flow

required further design verification, like the use of functional verification, to ensure correct implementation.

The first design was the *Robot*³, as is shown in the Listing 4.1. Compared to the initial design, the “*dist*” input did not exist, being internally randomly updated instead. This modification was needed by the fact that the design was not synthesizable, and therefore the conversion to *SV* would not be possible for the tool chosen.

This design is distinguished by two primary functions: a *controller* and a *sensor*. The first function detects whether the input variable, “*dist*”, is less than a specified parameter. The result is stored in a variable with a *true* value if the condition is true. The second function uses the value of this variable, and if it is *true*, activates the output variable. The selected design was chosen for its low complexity, which facilitates the testing of the tools in a shorter time.

<pre> 1 #define MIN_DIST 50 2 SC_MODULE(Robot) { 3 bool obs_detected; 4 int dist_in; 5 6 SC_CTOR(Robot) { 7 SC_THREAD(sensor); 8 SC_THREAD(controller); 9 } 10 11 void sensor() { 12 int dist; 13 while(true) { 14 wait(2, SC_MS); 15 // read from 0 to 255 16 dist = dist_in % 256; </pre>	<pre> 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 </pre>	<pre> if(dist < MIN_DIST) { obs_detected = true; }else{ obs_detected = false; } } } void controller() { bool alarm_flag = false; while (true) { wait(2, SC_MS); if (obs_detected) { alarm_flag = true; } else { alarm_flag = false; </pre>
---	---	--

³Robot original design available at: <https://data.4tu.nl/datasets/a7e780c9-87fa-486c-b484-a76a459a9d53>

```

34         }
35     }
36 }
37 };

```

Listing 4.1: Robot SystemC design

The second design is the **Half Adder**⁴, shown in Listing 4.2. It consists of a single function that computes the least significant bit, “*sum*”, and the most significant bit, “*carry*”, from the sum of two single bit input values.

```

1 SC_MODULE (half_adder) {
2     sc_in  <bool> a, b;
3     sc_out <bool> sum, carry;
4
5     void prc_half_adder(){
6         while (true) {
7             wait(5, SC_MS);
8             sum = !(a & b) & (a | b) ;
9             carry = a & b;
10        }
11    }
12    SC_CTOR (half_adder) {
13        SC_THREAD (prc_half_adder);
14    }
15 };

```

Listing 4.2: Half Adder SystemC design

The third design is the **Full Adder**⁴, shown in Listing 4.3. It leverages two instances of a *Half Adder* module to compute the sum and carry-out of three input bits. The design integrates these Half Adder outputs using one function which computes the design’s final outputs.

```

1 #include "half_adder.h"
2
3 SC_MODULE (full_adder) {
4     sc_in<bool> a, b, carry_in;
5     sc_out<bool> sum, carry_out;
6     sc_signal <bool> c1, s1, c2;
7     sc_signal <bool> sum_next;
8
9     void prc_or () {
10        while (true) {
11            wait(5, SC_MS);
12            carry_out = (c1 | c2);
13            sum = sum_next;
14        }
15    }
16    SC_CTOR (full_adder) {
17        half_adder ha1_ptr();
18        ha1_ptr.a (a);
19        ha1_ptr.b (b);
20        ha1_ptr.sum (s1);
21        ha1_ptr.carry (c1);
22
23        half_adder ha2_ptr();
24        ha2_ptr.a (s1);
25        ha2_ptr.b (carry_in);
26        ha2_ptr.sum (sum);

```

⁴Half Adder and Full Adder original designs available at: https://github.com/fabriziotappero/ip-cores/tree/other_systemc_to_verilog_synthesizable_subset_translator/examples

```

27     ha2_ptr.carry (c2);          31 };
28
29     SC_THREAD (prc_or);
30 }

```

Listing 4.3: Full Adder SystemC design

The original *Full Adder* design incorporated the *Half Adder* design. However, due to some tool limitations, subsequent modifications were made to this implementation. This will be presented in the following section.

This example was expanded to check how the **Full Adder** design would behave when instantiating more than one design. For that, the **Bit 2 Adder** was created, and later the **Bit 4 Adder**. The *Bit 2 Adder*, comprising five inputs and three outputs, is structured as two Full Adders, with the first carry-out connected to the carry-in of the second Full Adder. Similarly, the *Bit 4 Adder*, with nine inputs and five outputs, employs a similar logical structure, connecting two *Bit 2 Adder* units. The comparative analysis can be observed in Figure 4.2.

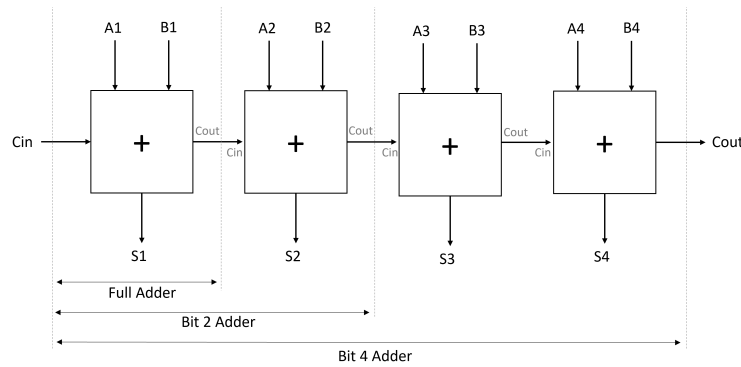


Figure 4.2: Comparison between Full Adder, Bit 2 Adder and Bit 4 Adder

The last design was the **ALU**⁵. The term “ALU” stands for “*Arithmetic Logic Unit*”. It is characterized by one function, has two data inputs, and another input for choosing the operation: addition, subtraction, increment, decrement, or Boolean operators (“and”, “or”, “nand”, or “xor”). The output is characterized by the result and two flags (detect “zero” and “carry”). The original design is shown in the Listing 4.4.

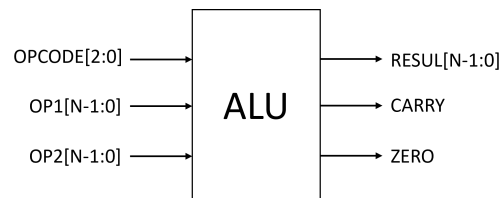


Figure 4.3: Diagram Block of the ALU for N bits

⁵ALU original design available at: <https://github.com/Muriukidavid/systemc-examples>

Initially, the *ALU* was tested using a 4-bit configuration (values between 0 and 16), as it was originally designed. Subsequently, it was adapted to support N-bit configurations. Figure 4.3 illustrates the generic *ALU*, modified to facilitate verification across various N-bit configurations. The results presented in Section 5 include the verification for 2, 4, 8, 16, 31, and 62 bits. Due to the limitations encountered for the 62-bit configuration, the circuit incorporates two variables for *OP1*, *OP2*, and *RESULT* at the HL.

Summarily, Table 5.1 presents a comprehensive overview of the key characteristics of each SystemC design. This includes the number of inputs, and the number of outputs separated as boolean and integer variables, respectively, and the number of functions.

Table 4.1: Designs used for the tests

Name	Inputs		Outputs		Nº Funct
	bool	int	bool	int	
Robot	0	1	1	0	2
Half Adder	2	0	2	0	1
Full Adder	3	0	2	0	2
Bit 2 Adder	5	0	3	0	4
Bit 4 Adder	9	0	5	0	8
ALU	0	3	2	1	1
ALU 62	0	5	2	2	1

```

1  #include <systemc.h>
2
3  SC_MODULE(ALU) {
4      sc_in < sc_uint<3> > OPCODE;
5      sc_in < sc_uint<4> > OP1, OP2;
6      sc_out < bool > CARRY, ZERO;
7      sc_out < sc_uint<4> > RESULT;
8
9      sc_uint<4> data1, data2;
10     sc_uint<5> result;
11
12     SC_CTOR(ALU){
13         SC_THREAD(operate);
14     }
15     void operate(){
16         while(true){
17             wait(5, SC_MS);
18             ZERO = false;
19             data1 = OP1.read();
20             data2 = OP2.read();
21             switch(OPCODE.read()) {
22                 case 0: //addition
23                     result = data1 + data2;
24                     break;
25                 case 1: //subtraction
26                     result = data1 - data2;
27                     break;
28                 case 2: //increment
29                     result = data1 + 1;
30                     break;
31                 case 3: //decrement
32                     result = data1 - 1;
33                     break;
34                 case 4: //and
35                     result = data1 & data2;
36                     break;
37                 case 5: //or
38                     result = data1 | data2;
39                     break;
40                 case 6: //nand
41                     result.range(3, 0) = ~(
42                         data1 & data2);
43                     break;
44                 case 7: //xor
45                     result = data1 ^ data2;
46                     break;
47             }
48         }
49     }
50 }

```

```

47     RESULT.write(result.range(3, 0)51     }
        );                               52     }
48     CARRY.write((bool) result[4]); 53 };
49     if(result == 0)
50         ZERO.write(true);

```

Listing 4.4: ALU SystemC design

4.3 Development

This section outlines the methodology employed throughout each phase of the verification process. It provides a comprehensive account of both the successful and unsuccessful procedures, beginning with the **HL** verification, followed by the conversion to **SV**, and concluding with the **LL** verification.

4.3.1 SystemC Verification: HL tool

In the context of the **HL** formal verification, the VerCors tool was selected based on its capacity to support different concurrency models and languages, its robust approach to ensuring functional behavior and data race freedom, as well as its extensibility to new concurrent languages. The approach used was based on the methodology outlined by Tasche et al. [53], as illustrated in the upper part of Figure 4.1, where the SystemC design is converted to **PVL** and verified with the VerCors tool.

The Tasche et al. paper’s artifact is a Virtual Machine (**VM**) with all the necessary installations. The **VM** was run in *Oracle VM Virtual Box*, with the following characteristics: memory of 1,9GB, processor *Intel Core I7*, and *Ubuntu 22.04.3 LTS (64-bit type) OS*. The Visual Studio Code was installed on the **VM** to facilitate debugging and the GitHub repository was created to save the entire environment.

The conversion to **PVL** presented certain challenges. This process is divided into two stages: initially, SystemC is converted to an **AST**, and then the **AST** is converted to **PVL**. The conversion to **AST** was accomplished using a converter component of the STATE tool and was completed successfully. The second stage, created by Tasche et al. [53], remains limited in scope, and does not support all the SystemC features. Consequently, further debugging was necessary to generate the desired **PVL**.

The modifications due to the conversion include defining the variable “*obs_detected*” (line 4) and adding the variable “*init*” (line 5) to ensure the verification part runs only after processing the code (lines 14, 25). Additionally, permissions were granted to write to these variables (lines 8-9, 20, 22, 26, 28, 32, and 34). The function responsible for obtaining the “*dist*” value was renamed, and its limit was constrained between 0 and 256 (lines 17, and 38). Furthermore, one assertion was added to confirm that “*obs_detected*” holds the correct value (line 42).

Listing 4.5 illustrates the *Sensor* function from the *Robot* design. The color components highlight the manually added elements, with the lighter color indicating partial edits. The

Sensor function corresponds to the lines 1-5, 18, 23, 29, and 38-41. The annotations correspond to the lines 7-9, 17, 20-22, 26-28, and 32-34. Lines 24, 30-31, and 35-37, deal with the scheduler.

```

1 class Sensor {
2     Main m;
3     int dist;
4     boolean obs_detected;
5     boolean init;
6
7     ensures Perm(m, write) ** Perm(dist, write) ** (m == m_param)
8         ** Perm(obs_detected, write) ;
9     ensures Perm(init, write) ** !init;
10    Sensor(Main m_param){
11        m = m_param;
12        dist = 0;
13        obs_detected = false;
14        init = false;
15    }
16
17    ensures \result >= 0 && \result < 256;
18    int getDistance ();
19
20    context ** (m != null) Perm(m, 1\2)
21        ** Perm(m.sensor, read) ** (m.sensor == this)
22        ** Perm(init, 1\2) ** !init;
23    void run(){
24        lock m;
25        init = true;
26        loop_invariant true ** (m != null) ** Perm(m, 1\2)
27            ** held(m) ** m.global_permission()
28            ** Perm(init, 1\2) ** !init;
29        while (true) {
30            m.process_state = m.process_state[0 -> 0];
31            m.event_state = m.event_state[0 -> 2];
32            loop_invariant true ** (m != null) ** Perm(m, 1\2)
33                ** held(m) ** m.global_permission()
34                ** Perm(init, 1\2) ** !init;
35            while (m.process_state[0] != -1 || m.event_state[0] != -2) {
36                unlock m;                lock m;
37            }
38            dist = getDistance();
39            if (dist < 50) { m.event_state = m.event_state[1 -> -1];
40                            obs_detected = true;                }
41            else {          obs_detected = false;                }
42            assert (dist < 50) == obs_detected;
43        }
44        unlock m;

```

```

45 }
46 }

```

Listing 4.5: Function Sensor in PVL final from Tasche et al.'s [53] converter

Once the verification of the *Robot* design was completed, the verification of the *Half Adder* and *Full Adder* started. During this process, a primary limitation was encountered with the input and output type variables, “*sc_in*”, and “*sc_out*”. Although the conversion process did not produce errors, these types were not supported in **PVL** and therefore were not converted, requiring their replacement with the “*bool*” type. This procedure was selected because it was more efficient than manually modifying the converted code. Another limitation was the lack of support for the “*xor*” operation in **PVL**, which was addressed by replacing it with a combination of “*not*”, “*and*”, and “*or*” operations. These limitations are illustrated in Listing 4.6, with the adopted solutions shown in Listing 4.7.

```

1 // Input and Output Limitation
2 sc_in  <bool> a, b;
3 sc_out <bool> sum, carry;
4
5 // Xor Limitation
6 sum.write(a ^ b);

```

Listing 4.6: Original Half Adder

```

// Solution
bool a, b;
bool sum, carry;

// Solution
sum.write(!(a & b) & (a | b));

```

Listing 4.7: Edit Half Adder

During the *Full Adder* verification, another issue occurred. This design instantiate two *Half Adder* modules, however, the converter did not support it. The solution involved integrating the *Half Adder* functions directly within the *Full Adder* with the result displayed on the Listing 4.8. The design now exhibits the same number of inputs and outputs and is constituted by two functions. Based on this, the *Bit 2 Adder* and *Bit 4 Adder* designs were created and subsequently verified.

<pre> 1 SC_MODULE (full_adder) { 2 bool Cin, A1, B1; 3 bool S1, Cout; 4 5 bool s11, c11, cin_1d; 6 7 void ha_1(){ 8 while (true) { 9 wait(5, SC_MS); 10 s11 = (!(A1 & B1)) & (A1 B1); 11 ; 12 c11 = (A1 & B1); 13 cin_1d = Cin; 14 } </pre>	<pre> 14 } 15 void fa_1 () { 16 while (true) { 17 wait(5, SC_MS); 18 S1 = (!(s11 & cin_1d)) & (s11 19 cin_1d); 20 Cout = (c11 (s11 & cin_1d)); 21 } 22 } 23 SC_CTOR (full_adder) { 24 SC_THREAD (ha_1); 25 SC_THREAD (fa_1); 26 } </pre>
--	---

Listing 4.8: Full Adder SystemC adapted design

Further constraints were identified during the *ALU* verification. These constraints were directly related to the converter and variable definitions. The converter does not support the “*sc_uint<N>*” format, necessitating a change in variable definitions to the “*int*” type and the implementation of a module to ensure the program only uses the *N* bits of the variable. These limitations are illustrated in Listing 4.9, with the adopted solutions shown in Listing 4.10.

```

1 // Input and Output Limitation
2 sc_in < sc_uint<3> > OP1, OP2;
3 sc_in < sc_uint<4> > OP1, OP2;
4 sc_out < bool > CARRY, ZERO;
5 sc_out < sc_uint<4> > RESULT;
6
7 // Write the variables
8 data1 = OP1.read();
9 data2 = OP2.read();

```

Listing 4.9: Original ALU Variables

```

// Solution
int OP1, OP2;
bool CARRY, ZERO;
int RESULT;

// Solution
data1 = OP1 % 16;
data2 = OP2 % 16;

```

Listing 4.10: Edited ALU Variables

Another limitation pertains to the inability of the *PVL* to accommodate Boolean operators, such as “*and*”, “*or*”, “*not*”, and “*xor*”, for an integer type variable. The solution adopted involved removing cases 4, 5, 6, and 7 from the switch-case structure (lines 35 to 46 in Listing 4.4). This adjustment was feasible since these cases did not influence the outcomes, as the modification was applied uniformly across both *HL* and *LL* verifications.

Despite these limitations, the process yielded valuable insights and demonstrated promising results, underscoring the potential of the chosen methodology.

Tasche et al. [53] verified also the *RASI*. The goal was to replicate the SystemC design in Python, execute it, and derive the reachable abstract process states. However, this was not pursued due to the absence of automation, support, and the lack of necessity for this application.

4.3.2 SystemC to SV conversion

To convert the SystemC to *SV* design, the ICSC tool was selected and installed in the *VM*. However, the tool proved to be incompatible with the *VM*, resulting in its failure to operate and subsequent crash. This incompatibility issue with ICSC within the *VM*, was solved using the tool through Windows Subsystem for Linux (*WSL*) with a Debian distribution.

The typical procedure for using this converter involves creating a folder with the same name as the design to convert. Within this folder, the SystemC design file is placed, and

a new file named *CMakeLists.txt* is created, containing the necessary configuration for the conversion process. A template for this file is provided in Listing 4.11. The final step is to execute the required commands in the terminal, all within the ICSC installation directory. Detailed instructions for the terminal commands are available in Appendix A.3.

```
cmake_minimum_required(VERSION 3.12)

enable_testing()

if(NOT DEFINED ENV{ICSC_HOME})
    message("ICSC_HOME is not defined!")
    return()
endif()

# Design template
project(mydesign)

find_package(SVC REQUIRED)
set(CMAKE_CXX_STANDARD 17)

# All synthesizable source files must be listed here (not in libraries)
add_executable(mydesign example.cpp)

# svc_target will create @mydesign_sctool executable that runs code
# generation
# and @mydesign that runs general SystemC simulation
# ELAB_TOP parameter accepts hierarchical name of DUT
# (that is SystemC name, returned by sc_object::name() method)
svc_target(mydesign INIT_LOCAL_VARS ELAB_TOP tb.dut_inst)
```

Listing 4.11: Template CMakeLists.txt

Following this, the *Robot* design conversion was initialized. However, the design was not synthesizable due to a lack of both input and output variables, except for the clock input. Consequently, the translation required additional modifications to the design as is already illustrated in Listing 4.1. For the others SystemC designs, they did not need relevant additional modifications.

In general, the converter revealed good results. The final *SV Robot* design is in the Listing 4.12.

```
1 module robot #(
2     parameter DATA_IN_WIDTH      = 16,
3     parameter MIN_DIST            = 16'd50
4 )
5 (
6     input wire clk,
```

```

7      input wire rstn,
8      input wire [DATA_IN_WIDTH-1:0] dist_v,
9      output wire alarm_flag,
10     output wire obs_detected_out
11 );
12
13 reg obs_detected;
14 reg out_alarm_flag;
15 reg [DATA_IN_WIDTH-1:0] buff_dist_v;
16
17 reg obs_detected_next;
18 reg alarm_flag_next;
19
20 assign buff_dist_v = dist_v;
21 assign obs_detected_out = obs_detected;
22
23
24 assign obs_detected_next = (buff_dist_v < MIN_DIST) ? 1'd1 : 1'd0;
25
26 always @(posedge clk or negedge rstn) begin : sensor_ff
27     if ( !rstn ) begin
28         obs_detected <= 1'd0;
29     end else begin
30         obs_detected <= obs_detected_next;
31     end
32 end
33
34
35 assign alarm_flag_next = obs_detected ? 1'd1 : 1'd0;
36
37 always @(posedge clk or negedge rstn) begin : controller_ff
38     if ( !rstn ) begin
39         out_alarm_flag <= 1'd0;
40     end else begin
41         out_alarm_flag <= alarm_flag_next;
42     end
43 end
44
45 assign alarm_flag = out_alarm_flag;
46 endmodule

```

Listing 4.12: Robot SV design

To guarantee consistency and fairness in the results, it was imperative to run all tools within the same environment. Consequently, all software installations were carried out within the [WSL](#). A GitHub repository was cloned in Windows, and both the VerCors tool and Visual Studio were installed, verifying the positive outcomes previously achieved on the [VM](#).

4.3.3 SV Verification: LL tool

To perform the **LL** analysis, the open-source SBY tool was selected. The installation process entailed a multitude of steps, including the preliminary installation of the Yosys and the solvers, all documented in the Appendix A.3. In this phase, the AutoSVA approach, developed by Orenes [38] was also tested.

This approach enables the automatic generalization of the **FV TB**. The **TB** comprises two files: the bind file and the property file. The first file is where the design is connected to the verification. The property file is where the assertions are written.

Following this, the *Robot* design was subjected to the verification. The **TB** automatically generated, with no assertions added, was tested. However, the results were inconclusive. Further investigation revealed that the issue was related to the limitations of the free version of Yosys, specifically the lack of support for multiple file tests.

This limitation was critical to achieving the desired **LL** results. The solution involved taking a step back and the use of a single file directly implemented in the SBY tool, bypassing the use of AutoSVA. To further refine the process and allow for the selective activation of assertions, these were encapsulated within a conditional flag, as shown in Listing 4.13.

```

1 'ifdef FORMAL
2     always @(posedge clk) begin
3         if (~rstn) begin
4             assert_obs_detected_reset: assert (!obs_detected);
5             assert_alarm_false_reset: assert (!alarm_flag);
6         end
7     end
8 'endif

```

Listing 4.13: Robot reset assertions for SBY tool

Another significant limitation was the restricted support for concurrent assertions, with only basic constructs from immediate assertions like *assume*, *assert*, and *cover* within an *always* block being supported. For the designs verified, some of the most useful features that were not supported included the *default clock*, *disable_iff*, *sequences*, and properties such as $\rightarrow$, $\Rightarrow$, and $\#\#N$. For example, Listing 4.14 illustrates one of the *Robot* assertions related to the *Sensor* function, presenting the difference between the original assertion and its equivalent supported by SBY.

```

1 assign f_obs = (dist_v < MIN_DIST) ? 1'd1 : 1'd0;
2
3 //Original Assertion
4 assert_alarm_true: assert property ( @(posedge clk) disable iff(!rstn)
5     (f_obs) ==> obs_detected_out);

```

```

6
7
8 //Assertions with SBY limitations
9 always @(posedge clk) begin
10     if (f_past_valid && rstn) begin
11         assert_obs_detected: assert ( !(f_obs_past) || (obs_detected_out) )
12         ;
13     end
14 end

```

Listing 4.14: Robot example of original assertion and with the SBY limitation

The process for the *Half Adder*, *Full Adder*, *Bit 2 Adder*, *Bit 4 Adder*, and *ALU* designs were similar with the same limitations presented. As with the **HL** tool, some debugging was required before a positive result was obtained.

4.4 Systematic Verification

To optimize the verification process, save the verification time, and produce statistics, like the average and standard deviation, three Python programs were developed: two for performing **HL** and **LL** verification, respectively, and a third to compute and compare the results, ultimately generating a visual plot. All the programs are available at thesis's **Artifact**.

The first two programs are organized into three main components: configuration, process data, and saving information. The initial part allows the selection of the number of tests, the command to execute, and the files to verify. The second part generates the statistics, like maximum, minimum, average, and standard deviation. The last part creates two output filenames. The first output is stored the verification time of all simulations organized according to the list of files given. The second output file includes the details statistics in a table format.

The third program is structured with a configuration section that allows to specify the names of the input and output files, as well as the labels, titles, and image names for the plot. This program reads the results from the two earlier programs, calculates the time differences between without and with assertion verification, and stores this information in two new files for **HL** and **LL** verification. Each file includes the design name, average, and standard deviation for the without and with assertion, and the difference between them. Finally, the program generates a single plot that visually compares the results of **HL** and **LL** verification.

Chapter 5

Results Analysis

This chapter presents the results obtained from the implementation, also providing an examination of the results and a detailed discussion.

5.1 Test Environment

All the tests were realized on the **WSL** with a Debian Distribution in Windows 11 version 22H2. The computer was a *Dell Latitude 7400* with the following characteristics: *Intel(R) Core(TM) i7-8665U CPU @ 1.90GHz 2.11 GHz*, RAM *16.0 GB*, and *64-bit operating system*. Also, each verification was executed 40 times, both with and without assertions.

5.2 Experimental Results

After implementation, the design exhibited the characteristics detailed in Table 5.1. For each design, the following attributes are presented: the number of inputs, outputs, and functions in the SystemC file, and the number of assertion properties in the **PVL** and **SV** files.

Table 5.1: Characteristics Designs Used

Design	Inputs	Outputs	Functions	Nº Assertions
Robot	1	1	2	2
Half Adder	2	2	1	8
Full Adder	3	2	2	16
Bit 2 Adder	5	3	4	16
Bit 4 Adder	9	5	8	24
ALU	3	3	1	12
ALU 62	5	4	1	12

The *Robot* and *Half Adder* designs are the simpler constructs. The *Robot* design consists of a single input and output with two functions. And, the *Half Adder* design has two inputs, two outputs, and one function, reflecting a simple level of complexity.

In contrast, the *ALU 62* and *Bit 4 Adder* designs are the most complex designs examined. The *ALU* has five inputs and four output integer type variables, indicating a higher level of computational and structural complexity. Furthermore, the *Bit 4 Adder* has nine inputs and five outputs and takes 8 clock cycles to obtain the output values.

Table 5.2: Verification Time Experimental Results - HL Verification

Designs	Without Assert		With Assert		Difference	
	Avg	Std Dev	Avg	Std Dev	Avg	Std Dev
Robot	30,203s	2,588s	29,555s	2,061s	0,648s	3,308s
Half Adder	21,987s	1,555s	22,772s	2,210s	0,785s	2,702s
Full Adder	30,211s	2,969s	29,000s	2,497s	1,211s	3,879s
Bit 2 Adder	47,575s	2,843s	108,290s	5,675s	60,715s	6,347s
Bit 4 Adder	142,874s	10,166s	1927,622s	31,678s	1784,748s	33,269s
ALU 2	49,805s	3,662s	56,924s	5,160s	7,119s	6,327s
ALU 4	49,975s	3,117s	55,437s	3,189s	5,462s	4,459s
ALU 8	50,890s	3,508s	55,468s	2,664s	4,578s	4,405s
ALU 16	50,516s	3,758s	55,750s	3,629s	5,234s	5,224s
ALU 31	51,133s	3,287s	55,211s	2,602s	4,078s	4,192s
ALU 62	78,480s	5,070s	130,587s	8,321s	52,107s	9,744s

Table 5.3: Verification Time Experimental Results - LL Verification

Designs	Without Assert		With Assert		Difference	
	Avg	Std Dev	Avg	Std Dev	Avg	Std Dev
Robot	0,836s	0,059s	1,079s	0,109s	0,243s	0,124s
Half Adder	1,030s	0,154s	1,389s	0,227s	0,386s	0,274s
Full Adder	1,042s	0,104s	1,501s	0,083s	0,459s	0,133s
Bit 2 Adder	1,088s	0,121s	1,677s	0,223s	0,589s	0,254s
Bit 4 Adder	1,173s	0,109s	3,738s	0,822s	2,565s	0,829s
ALU 2	1,665s	0,089s	2,343s	0,271s	0,678s	0,285s
ALU 4	1,682s	0,142s	2,901s	0,346s	1,219s	0,374s
ALU 8	1,329s	0,159s	4,099s	1,373s	2,770s	1,382s
ALU 16	1,219s	0,113s	4,831s	0,434s	3,612s	0,448s
ALU 31	1,389s	0,175s	10,478s	1,510s	9,089s	1,520s
ALU 62	1,548s	0,172s	26,148s	2,449s	24,600s	2,455s

After the initial verification time measurements, it became evident that the time required for verification at the **HL** was significantly longer than at the **LL** for most of the designs. The underlying reasons for this discrepancy will be discussed later in this chapter. To minimize the potential impact of pre-processing time on each verification, they were

verified without and with assertions, and the difference in time was calculated to isolate the time consumed by the assertions and enable a fair comparison.

To ensure reliable and accurate results, each design underwent 40 verification iterations, both without and with assertions. The average time consumption and standard deviation were calculated for each design and each type of verification, encompassing both **HL** and **LL** approaches. The results of these calculations are presented in Table 5.2 for **HL** verification and Table 5.3 for **LL** verification. The “Difference” columns, “Avg” and “Std Dev”, indicate respectively, the average time, the difference between executions without and with assertions, and the standard deviation, calculated using the square root of the sum of the squares of the individual standard deviations.

5.3 Discussion

Based on the information calculated present in Tables 5.2 and 5.3, the diagram on Figure 5.1 was created using the average time of the “Difference” column. It also shows the standard deviation of mean, SEM, obtained for each design represented by the black line on each.

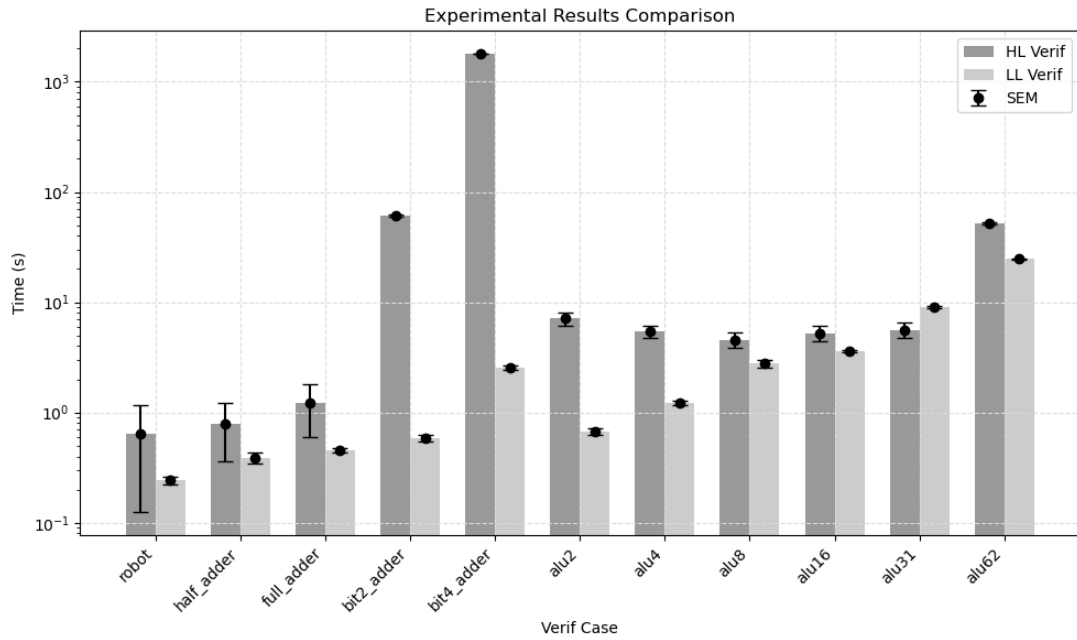


Figure 5.1: Diagram of the Difference Average Experimental Results at HL and LL

A comparative analysis of the diagram indicates that the *Robot* design demonstrates the shortest assertion verification time, while the *Bit 4 Adder* and *ALU 62 bits* exhibits the longest.

In general, the **HL** approach exhibits inferior performance relative to the **LL** approach, as evidenced by the longer runtime required for verification. However, this comparison is

inherently complex due to the fundamentally different nature of the verification processes involved.

To explore this performance discrepancy, the `--progress` flag was employed in the **HL** verification command. This flag provided a detailed terminal output, capturing the time taken at each stage of the simulation process. Appendix C illustrates the terminal output for the *Robot* and *Bit 4 Adder* designs. A further analysis of these outputs is shown in Figure 5.2.

For the *Robot* design, the entire verification process took 34.55 seconds, with the verification phase, denoted as *Task Viper Verification*, accounting for 25.01 seconds, representing 82.8% of the total time. In contrast, the *Bit 4 Adder* design took 2,385.58 seconds, with the verification phase consuming 2,370.55 seconds, which constitutes 99.6% of the overall simulation time.

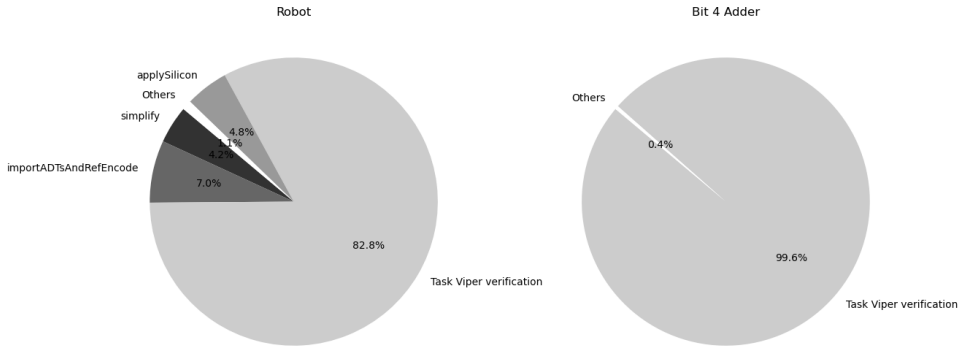


Figure 5.2: Analysis of Terminal Output from the Robot and Bit 4 Adder Assertion HL Verification Test

These percentages show significantly dependence on the complexity of the design. As the design complexity increases, the verification phase consumes a greater proportion of the total time. For smaller simulations, the verification phase represents a substantial portion of the overall simulation time.

5.4 Performance

Focus on the results they can be categorized into two main types. The first category, termed *Adders Results*, includes the *Half Adder*, *Full Adder*, *Bit 2 Adder*, and *Bit 4 Adder*. The second category, referred to as *ALU Results*, comprises the *ALU 2 bits*, *ALU 4 bits*, *ALU 8 bits*, *ALU 16 bits*, *ALU 31 bits*, and *ALU 62 bits*.

The *Adders Results* demonstrate a consistent doubling in complexity, with the designs requiring 1, 2, 4, and 8 clock cycles, respectively, to generate the first valid output. Analysis of the data in Tables 5.2 and 5.3, alongside Figure 5.1, shows an exponential increase in verification time, particularly at the **HL** abstraction level. The significant performance

impact begins to emerge with the *Bit 2 Adder*, while for **LL** verification, these time effects become notable only with the *Bit 4 Adder*.

In the *ALU Results*, complexity also doubles with increasing bit sizes, except between the *ALU 16 bits* and *ALU 31 bits*. At the **HL**, the verification times for *ALU 2 bits* through *ALU 31 bits* remain relatively consistent. This pattern does not hold at the **LL** level, where results increase exponentially, and reach a time bigger than at **HL** at the *ALU 31 bits*. For the *ALU 62 bits* the results reveal again that the **HL** time is bigger than at **LL**. This behavior can be attributed to the use of integer data types at **HL**.

The integer type has a 31 bit capacity and therefore has two consequences. First, the cases of 2, 4, 8, and 16 bits needed to use the module to limit the scope of the integer variable, but for the tool, all the cases were tested and led the simulation to have similar verification times. Second, the *ALU 62 bit* needed two variables for each data inputs and output, which increased the verification time.

In general, the **HL** verification had worse results than at **LL**. Especially at the *Adders Results*. For the *ALU Results*, these differences are small revealing an interesting and possible better efficiency at **HL** when using variables that are intrinsic to the **HL** verification tool (32 bit variables, int unit).

5.5 Efficiency

The time required to obtain a positive verification was consistent across tests. The **HL** tool was employed to conduct an efficiency test, automatically generating the **PVL** code. This automation streamlines the process by generating the main annotations needed for testing, while the remaining annotations were manually added.

For **LL** verification, the design required certain corrections, and the assertions had to be adapted due to the limitations of the free SBY version. These adjustments were necessary to align with the constraints imposed by the tool and ensure accurate verification.

Chapter 6

Conclusions

This thesis provides a comprehensive exploration of **FV** within the **EDA** domain. It defines **FV**, explains its fundamental principles and highlights its critical role in hardware design reliability. The research reviews existing open-source **FV** tools, detailing their features and applications. Subsequently, the VerCors as **HL** tool and the SBY as **LL** were chosen for the design verification.

6.1 Results

Given that **HL** verification operates at a higher level of abstraction, it was anticipated that this approach would be faster due to the reduced number of state explosions. However, the expected results were not achieved. In fact, the **HL** approach exhibited inferior performance for almost all the designs analyzed compared to the **LL** approach, as evidenced by the longer verification run times.

The exception occurred in a specific design, where the use of an integer type variable appeared to improve efficiency in the **HL** context. This suggests that integer types may offer a performance advantage at higher abstraction levels, particularly to 31-bit variables.

The **HL** verification approach does not use the SystemC **TLM-2.0**, which operates at a higher level of abstraction. This level of abstraction would simplify the design process by modeling the interactions between components without requiring detailed implementation specifics and therefore the verification process. However, the conversion limitations and complexities associated with transitioning from high-level abstractions to low-level verifiable models have hindered its broader adoption. This constraint may contribute to the challenges in achieving more effective verification outcomes at the **HL** verification.

Nonetheless, this comparison is inherently complex due to the fundamental differences in their methodologies and tools. At **HL** the **PVL** proves a significantly larger number of properties compared to **LL** approach. This comprehensive property verification in **PVL**

indicates a thorough examination of the system’s behavior, like verifying the allocation memory, and the use of annotations, and therefore the increased verification time.

Lastly, the maturity level of **LL** tools surpasses that of **HL** tools. **LL** verification tools have undergone extensive development and optimization over the years, resulting in more efficient and faster verification processes. These tools are finely tuned to handle the intricacies of low-level design verification, providing a robust framework for identifying and rectifying design flaws.

Despite these current challenges, a promising aspect is the existence of tools and an open-source ecosystem that facilitate the initial stages of this analysis. This availability suggests a future potential where formal verification on SystemC models could be performed entirely at the SystemC level, leveraging advances in **HL** fields to complete the design verification process effectively. However, the current limitations of **HL** tools and their support, along with the challenges associated with using **IVL**, present significant obstacles that need to be addressed to fully realize this potential.

6.2 Limitations

The research was constrained by some limitations, which had a considerable impact on the overall progress and outcomes.

A significant challenge was the considerable time required for the installation and configuration of the necessary tools. As a result, the time allocated to these preliminary procedures was in direct conflict with the time available for the actual verification processes and analysis.

Another notable limitation was the lack of comprehensive support for the **HL** verification in a specific context of SystemC to **PVL** converter, which was developed and only discussed in the original paper. This lack of comprehensive support manifested in several ways, including inadequate handling of certain language features, insufficient documentation, and an increased time requirement for debugging.

Furthermore, achieving a fair and direct comparison between **HL** and **LL** verification approaches also presented a significant challenge. The fundamental differences in the abstraction levels, methodologies, and tool capabilities between **HL** and **LL** verification made it inherently difficult to establish a uniform basis for comparison. Moreover, the **HL** tool, being relatively newer and less mature, often exhibited longer runtimes and less efficient processing compared to the well-established **LL** tool.

This inherent complexity required careful consideration and methodological adjustments to ensure that the comparative analysis was as fair and objective as possible, taking into account the inherent differences between the two verification paradigms.

6.3 Further Work

Future work should focus on several key aspects to improve the reliability and robustness of our conclusions. Firstly, improving the reliability of the results requires testing a wider variety of designs, including, for example, SystemC **TLM-2.0** designs, which may offer advantages over traditional clock-based representations. This expansion would be particularly valuable if the tool would be updated to support **TLM-2.0** in the future. Or a tool is found with better performance and support for it.

Furthermore, expanding the range of designs tested would provide a more comprehensive understanding of the strengths and limitations of both **HL** and **LL** formal verification approaches. By including a diverse set of design types, it would be possible to assess generality and identify any design-specific behaviors or issues that may not have been apparent in the initial set of designs.

Secondly, addressing scalability is essential to extend the applicability of the verification methods to more complex designs. Future work should include testing increasingly complex designs to assess how well the verification tools and methods perform at higher levels of complexity and larger state spaces. Evaluating scalability will also be critical to understanding the practical applicability of **HL** and **LL** verification techniques in real-world scenarios, where designs are often intricate and large.

Overall, focusing on a broader range of design tests and exploring scalability with more complex designs will provide a deeper and more nuanced understanding of the verification processes, ultimately leading to more reliable and applicable results in the field of formal verification.

In conclusion, the **HL** verification was expected to be faster, but, in general, it showed inferior performance compared to **LL** verification. This disparity arises from the fundamental differences in methodologies and tools. **HL** verification via **PVL** proves more properties, enhancing system behavior examination but increasing verification time. Conversely, **LL** tools, benefiting from extensive optimization, offer faster verification processes. Despite these challenges, the availability of **HL** tools and an open-source ecosystem suggests a promising future for formal verification entirely at the SystemC level, contingent on overcoming current limitations.

References

- [1] IEEE standard for standard systemc language reference manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, 2012.
- [2] Afshin Amighi, Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. The vercors project: Setting up basecamp. In *Proceedings of the sixth workshop on Programming languages meets program verification*, pages 71–82, 2012.
- [3] Michael Baird. Coverage driven verification of an unmodified dut within an ovm testbench. In *Design & Verification Conference*, 2010.
- [4] David C. Black, Jack Donovan, Bill Bunton, and Anna Keist. *SystemC: From the Ground Up*. Springer US.
- [5] Nicolas Blanc, Daniel Kroening, and Natasha Sharygina. Scoot: A tool for the analysis of systemc models. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 467–470. Springer, 2008.
- [6] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The vercors tool set: verification of parallel and concurrent software. In *Integrated Formal Methods: 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings 13*, pages 102–110. Springer, 2017.
- [7] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*, pages 334–342. Springer, 2014.
- [8] Wen Chen, Sandip Ray, Jayanta Bhadra, Magdy Abadir, and Li-C Wang. Challenges and trends in modern soc design verification. *IEEE Design & Test*, 34(5):7–22, 2017.
- [9] Alessandro Cimatti, Alberto Griggio, Andrea Micheli, Iman Narasamdya, and Marco Roveri. Kratos - a software model checker for systemc. In *International Conference on Computer Aided Verification*, pages 310–316. Springer, 2011.
- [10] Alessandro Cimatti, Iman Narasamdya, and Marco Roveri. Software model checking systemc. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(5):774–787, 2013.

- [11] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on logic of programs*, pages 52–71. Springer, 1981.
- [12] Edmund M Clarke, Manpreet Khaira, and Xudong Zhao. Word level model checking—avoiding the pentium fdiv error. In *Proceedings of the 33rd Annual Design Automation Conference*, pages 645–648, 1996.
- [13] M Dossis. High-level synthesis integrated verification. *Engineering, Technology & Applied Science Research*, 5(5):864–870, 2015.
- [14] Michael Dossis. Converging formal verification in a high-level synthesis environment. In *2021 6th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM)*, pages 1–5, 2021.
- [15] Alan Edelman. The mathematics of the pentium division bug. *SIAM review*, 39(1):54–67, 1997.
- [16] Lamia Eljadiri and Ismail Assayad. "Generic Framework Architecture for Verifying Embedded Components". *International Journal of Advanced Computer Science and Applications*, 11(6), 2020.
- [17] E Allen Emerson. The beginning of model checking: A personal perspective. *25 Years of Model Checking: History, Achievements, Perspectives*, pages 27–45, 2008.
- [18] RW Floyd. Assigning meaning to programs. mathematical aspects of computer science. XIX. *Amer. Math. Soc., Providence, Rhode Island*, 1967.
- [19] Mehran Goli and Rolf Drechsler. *"Automated analysis of virtual prototypes at the electronic system level: design understanding and applications"*. Springer Nature, 2020.
- [20] Nesrine Harrath, Bruno Monsuez, and Kamel Barkaoui. A framework for verification of systemc designs using systemc waiting state automata. *Integration of Reusable Systems*, pages 77–104, 2014.
- [21] Hannibal Height. *A practical guide to adopting the universal verification methodology (UVM)*. Lulu. com, 2012.
- [22] Paula Herber, Marcel Pockrandt, and Sabine Glesner. "STATE—A SystemC to Timed Automata Transformation Engine". In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 1074–1077. IEEE, 2015.
- [23] Vladimir Herdt, Daniel Große, and Rolf Drechsler. "Formal Verification of SystemC-Based Designs using Symbolic Simulation". In *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*, pages 59–117. Springer International Publishing.
- [24] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

- [25] Ahmed Irfan, Alessandro Cimatti, Alberto Griggio, Marco Roveri, and Roberto Sebastiani. Verilog2smv: A tool for word-level verification. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1156–1159, 2016.
- [26] Tero Joentakanen. Evaluation of hls modules for asic backend. Master’s thesis, 2017.
- [27] Sakari Lahti, Panu Sjövall, Jarno Vanne, and Timo D. Hämäläinen. Are we there yet? a study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):898–911, 2019.
- [28] Hoang M Le, Daniel Große, Vladimir Herdt, and Rolf Drechsler. Verifying systemc using an intermediate verification language and symbolic simulation. In *Proceedings of the 50th Annual Design Automation Conference*, pages 1–6, 2013.
- [29] Bin Lin, Kai Cong, Zhenkun Yang, Zhigang Liao, Tao Zhan, Christopher Havlicek, and Fei Xie. "Concolic testing of SystemC designs". In *2018 19th International Symposium on Quality Electronic Design (ISQED)*, pages 1–7, 2018.
- [30] Zohar Manna. The correctness of programs. *Journal of Computer and System Sciences*, 3(2):119–127, 1969.
- [31] John McCarthy. A basis for a mathematical theory of computation, preliminary report. In *Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*, pages 225–238, 1961.
- [32] M.C. McFarland, A.C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, 1990.
- [33] Mikhail Moiseev, Roman Popov, and Ilya Klotchkov. Systemc-to-verilog compiler: a productivity-focused tool for hardware design in cycle-accurate systemc. *Proceedings of DVCON Europe*, page 15, 2020.
- [34] W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiehl. The simulation semantics of systemc. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 64–70, 2001.
- [35] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.
- [36] Peter Naur. Proof of algorithms by general snapshots. *BIT Numerical Mathematics*, 6(4):310–316, 1966.
- [37] Van Chan Ngo and Axel Legay. Formal verification of probabilistic systemc models with statistical model checking. *Journal of Software: Evolution and Process*, 30(3):e1890, 2018.
- [38] Marcelo Orenes-Vera, Aninda Manocha, David Wentzlaff, and Margaret Martonosi. Autosva: democratizing formal verification of rtl module interactions. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 535–540. IEEE, 2021.

- [39] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.
- [40] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):455–495, 1982.
- [41] Preeti Ranjan Panda. Systemc: a modeling platform supporting multiple design abstractions. In *Proceedings of the 14th international symposium on Systems synthesis*, pages 75–80, 2001.
- [42] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical computer science*, 13(1):45–60, 1981.
- [43] Gautham Prabhakar. "Validation of efficiency of formal verification methodology for verification closure". Master's thesis, EECS, KTH, Stockholm, Sweden, 2022.
- [44] D.I. Rich. The evolution of systemverilog. *IEEE Design Test of Computers*, 20(4):82–84, 2003.
- [45] Erik Seligman, Tom Schubert, and M V Achutha Kiran Kumar. *Formal Verification: An Essential Toolkit for Modern VLSI Design*. 2nd ed. Cambridge: Elsevier, 2015.
- [46] Dilip Singh and Rajeevan Chandel. Register-transfer-level design for application-specific integrated circuits. *Nanoscale VLSI: Devices, Circuits and Applications*, pages 295–319, 2020.
- [47] Douglas J Smith. Vhdl & verilog compared & contrasted—plus modeled example written in vhdl, verilog and c. In *Proceedings of the 33rd annual Design Automation Conference*, pages 771–776, 1996.
- [48] Ashley T Sonny and S. Lakshmiprabha. "OVL, PSL, SVA: Assertion based verification using checkers and standard assertion languages". In *2013 International Conference on Advanced Computing and Communication Systems*, pages 1–4, 2013.
- [49] Manuel F Soto, J Agustin Rodriguez, and Pablo R Fillottrani. "SystemC/TLM flow for SoC design and verification". In *2015 Argentine School of Micro-Nanoelectronics, Technology and Applications (EAMTA)*, pages 37–42. IEEE, 2015.
- [50] Elbouanani Soumia, Assayad Ismail, and Sadik Mohammed. "Formal Method for the Rapid Verification of SystemC Embedded Components". In *2019 5th International Conference on Optimization and Applications (ICOA)*, pages 1–5, 2019.
- [51] Stuart Sutherland and Tom Fitzpatrick. Uvm rapid adoption: A practical subset of uvm. *Proceedings of DVCon*, 2015.
- [52] Synopsys. What is EDA? Available at <https://www.synopsys.com/glossary/what-is-electronic-design-automation.html> (accessed mai. 2024).
- [53] Philip Tasche, Raúl E Monti, Stefanie Eva Drerup, Pauline Blohm, Paula Herber, and Marieke Huisman. Deductive verification of parameterized embedded systems modeled in systemc. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 187–209. Springer, 2023.

- [54] Moshe Y Vardi. Formal techniques for systemc verification. In *Proceedings of the 44th annual Design Automation Conference*, pages 188–192, 2007.
- [55] Moshe Y. Vardi. Program verification: a 70+- year history. In *2021 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 1–2, 2021.
- [56] Pankaj S Vitankar and AK Kureshi. Uvm architecture for verification. *International Journal of Electronics and communications Engineering and Technology*, 7(3):29–37, 2016.
- [57] Paul Wilcox. *Professional verification: a guide to advanced functional verification*. Springer Science & Business Media, 2004.
- [58] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, volume 97, 2013.
- [59] Zhuanhao Wu, Maya Gokhale, Scott Lloyd, and Hiren Patel. Sccl: An open-source systemc to rtl translator. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 23–33, 2023.

Appendix A

Guide Installation

This appendix explains all the procedures for installing the tools used. For the SystemC standards and implementation, all the information is available [here](#).

A.1 WSL

The steps for the installation are the follows, more information can be found on the official website [here](#).

1. Open the Windows Command Prompt as administrator and enter the command
`wsl --install`
2. Install Debian Distribution with the command `wsl --install -d Debian`

A.2 VerCors

The official VerCors website can be found [here](#). The steps for installation are the following:

1. Open the Debian Terminal
2. Install the requires pre-installation, namely Java (≥ 17) and Clang:
`sudo apt install clang openjdk-17-jre`
3. Download the 1.4.1 VerCors version from the GitHub from [here](#). File named `Vercors_1.4.1_all.deb`.
4. Then install the VerCors: `sudo dpkg -i Vercors_1.4.1_all.deb`
5. (Optional) To run the VerCors tool, go to a file with PVL and execute the command:
`vercors --silicon *.pvl`

A.3 ICSC

Intel® Compiler for SystemC* (ICSC) translates synthesizable SystemC design into equivalent SystemVerilog code. The official website can be found [here](#). The steps for installation are the following:

1. Open the Debian Terminal and install the requires prr-installation, Git, C++ (=>17) and CMake (=>3.13)
2. Define ICSC folder `export ICSC_HOME=/home/username/project`, where `/home/username` is the home folder
3. Install [Protobuf](#) (=>3.6.1) and [LLVM/Clang](#) 12.0.1. on the ICSC folder
4. Clone ICSC repository

```
git clone https://github.com/intel/systemc-compiler $ICSC_HOME/icsc
```
5. On the terminal enter the ICSC folder installation: `cd $ICSC_HOME`
6. Download and install all required components `icsc/install.sh`
7. On the terminal enter the ICSC folder and setup `PATH` and `LD_LIBRARY_PATH` with the command `source setenv.sh`
8. Build and install ICSC

```
cd $ICSC_HOME/icsc
mkdir build && cd build
cmake ../ -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=$ICSC_HOME
make -j8
make install
```
9. (Optional) Generate SV code for the available examples (counter is the design example):

```
cd $ICSC_HOME
source setenv.sh
cd build
cmake ../
ctest -R counter
```

A.4 SBY

The SBY also known as SymbiYosys has its official website [here](#). The installation steps are the follows:

1. Open the Debian Terminal and install the requires pre-installation

```
sudo apt-get install build-essential clang bison flex
sudo apt-get install libreadline-dev gawk tcl-dev libffi-dev git
sudo apt-get install graphviz xdot pkg-config python3 zlib1g-dev
python3 -m pip install click
```
2. Install **Yosys**

```
git clone https://github.com/YosysHQ/yosys
cd yosys
make -j$(nproc)
sudo make install
```
3. Install Boolector

```
git clone https://github.com/boolector/boolector
cd boolector
./contrib/setup-lingeling.sh
./contrib/setup-btor2tools.sh
./configure.sh
make -C build -j$(nproc)
sudo cp build/bin/{boolector,btor*} /usr/local/bin/
sudo cp deps/btor2tools/bin/btorsim /usr/local/bin/
```
4. Install Z3 builder

```
git clone https://github.com/Z3Prover/z3.git
cd z3
python scripts/mk_make.py
cd build
make
sudo make install
```
5. Install SBY

```
git clone https://github.com/YosysHQ/sby
cd sby
sudo make install
```
6. (Optional) Install GTKWave `sudo apt install gtkwave`
7. (Optional) On the SBY website you can find the FIFO example [here](#) and the example can be reproduce.

Appendix B

Source Code

In this Appendix are available the SystemC files of all the designs verified. All the designs in **PVL** and **SV** were created based on these files which can be found on GitHub¹.

B.1 SystemC

B.1.1 Robot

```
1 //main.cpp
2
3 #include <systemc.h>
4 #include "Robot.h"
5
6 int sc_main (int argc, char* argv[]) {
7     Robot robot("Robot0");
8     sc_start(20, SC_MS);
9     return(0);
10 }
11
12
13
14 //Robot.h
15
16 #define MIN_DIST 50
17 SC_MODULE(Robot) {
18     bool obs_detected;
19     bool dist_in;
20
21     SC_CTOR(Robot) {
22         SC_THREAD(sensor);
23         SC_THREAD(controller);
24     }
25
26     void sensor() {
27         int dist;
28
29         while(true) {
30             wait(2, SC_MS);
31             // read from 0 to 255
32             dist = dist_in % 256;
33             if(dist < MIN_DIST) {
34                 obs_detected = true;
35             }else{
36                 obs_detected = false;
37             }
38         }
39
40         void controller() {
41             bool alarm_flag = false;
42             while (true) {
43                 wait(2, SC_MS);
44                 if (obs_detected) {
45                     alarm_flag = true;
46                 }
47                 else {
48                     alarm_flag = false;
49                 }
50             }
51         }
52     };
53 }
```

Listing B.1: The Robot design

B.1.2 Half Adder

```
1 //main.cpp
2
3 #include <systemc.h>
4 #include "half_adder.h"
5
6
7 int sc_main(int argc, char* argv[]) {
8     half_adder half("HA");
9 }
```

¹https://github.com/Renato1818/ll_hl_verif.git

```

9   sc_start(5000, SC_NS);
10  return 0;
11 }
12
13 //half_adder.h
14
15 #include <systemc.h>
16
17 SC_MODULE (half_adder) {
18     sc_in  <bool> a, b;
19     sc_out <bool> sum, carry;
20
21     bool s_nand, s_or;
22
23     void prc_half_adder(){
24         wait(2, SC_MS);
25
26         while (true) {
27             s_nand = !(a & b);
28             s_or = a | b ;
29             sum.write(s_nand & s_or);
30             carry.write(a & b);
31             wait(5, SC_MS);
32         }
33     }
34
35     SC_CTOR (half_adder) {
36         SC_THREAD (prc_half_adder);
37     }
38 };

```

Listing B.2: The Half Adder design

B.1.3 Full Adder

```

1 //main.cpp
2
3 #include <systemc.h>
4 #include "full_adder.h"
5
6 int sc_main(int argc, char* argv[]){
7     full_adder adder("adder");
8     sc_start(5000, SC_MS);
9     return 0;
10 }
11
12 //full_adder.h
13 #include <systemc.h>
14
15 SC_MODULE(full_adder) {
16     bool Cin, A1, B1;
17     bool S1, Cout;
18
19     bool s11, c11, cin_1d;
20
21     void ha_1() {
22         s11 = (!(A1 && B1)) && (A1 || B1);
23         c11 = (A1 && B1);
24
25         cin_1d = Cin;
26     }
27
28     void fa_1() {
29         S1 = (!(s11 && cin_1d)) && (s11 || cin_1d);
30         Cout = (c11 || (s11 && cin_1d));
31     }
32
33     void process() {
34         while (true) {
35             wait(5, SC_MS);
36             ha_1();
37             fa_1();
38         }
39     }
40
41     SC_CTOR (full_adder) {
42         SC_THREAD (process);
43         SC_METHOD (ha_1);
44         SC_METHOD (fa_1);
45     }
46 };

```

Listing B.3: The Full Adder design

B.1.4 Bit 2 Adder

```

1 //main.cpp
2
3 #include <systemc.h>
4 #include "bit2_adder.h"
5
6 int sc_main(int argc, char* argv[]){
7     bit2_adder adder("adder");
8     sc_start(5000, SC_MS);
9     return 0;
10 }
11
12 //bit2_adder.h
13 #include <systemc.h>
14
15 SC_MODULE (bit2_adder) {
16     sc_in  <bool> Cin, A1, A2, B1, B2;
17     sc_out <bool> S1, S2, Cout;
18
19     //Auxiliar variables to ha1 and fa1
20     sc_signal <bool> s11, c11, cin_1d;
21
22     sc_signal <bool> s1, c1;
23
24     //Auxiliar variables to ha2 and fa2
25     sc_signal <bool> s21, c21, c1_1d;
26     sc_signal <bool> s2, c2;
27
28     //Delay 2nd Inputs
29     sc_signal <bool> a2_1d, a2_2d, b2_1d, b2_2d;
30     //Delay 1st Output
31     sc_signal <bool> s1_1d, s1_2d;
32
33     void ha_1(){
34         s11 = (!(A1 && B1)) && (A1 || B1);
35         c11 = (A1 && B1);
36         cin_1d = Cin;
37
38         //Delay 2nd inputs
39         a2_1d = A2;
40         b2_1d = B2;
41     }
42
43

```

```

44 void fa_1 () {
45     s1 = (!(s11 && cin_1d)) && (s11 || cin_1d);
46     c1 = (c11 || (s11 && cin_1d));
47
48     //Delay 2nd inputs
49     a2_2d = a2_1d;
50     b2_2d = b2_1d;
51 }
52
53
54 void ha_2() {
55     s21 = (!(a2_2d && b2_2d)) && (a2_2d ||
56         b2_2d);
57     c21 = (a2_2d && b2_2d);
58     c1_1d = c1;
59
60     //Delay 1st Output
61     s1_1d = s1;
62 }
63
64 void fa_2 () {
65     s2 = (!(s21 && c1_1d)) && (s21 || c1_1d);
66     c2 = (c21 || (s21 && c1_1d));
67
68     //Delay 1st Output
69     s1_2d = s1_1d;
70
71     //Final Outputs
72     S1 = s1_2d;

```

```

73     S2 = s2;
74     Cout = c2;
75 }
76
77 void process() {
78     while (true) {
79         wait(5, SC_MS);
80
81         ha_1();
82         fa_1();
83         ha_2();
84         fa_2();
85     }
86
87     SC_CTOR (bit2_adder){
88         SC_THREAD (process);
89
90         SC_METHOD (ha_1);
91         SC_METHOD (fa_1);
92
93         SC_METHOD (ha_2);
94         SC_METHOD (fa_2);
95     }
96 }
97

```

Listing B.4: The Bit 2 Adder design

B.1.5 Bit 4 Adder

```

1 //main.cpp
2
3 #include <systemc.h>
4 #include "bit4_adder.h"
5
6
7 int sc_main(int argc, char* argv[]) {
8     bit4_adder adder("adder");
9     sc_start(5000, SC_MS);
10    return 0;
11 }
12
13 //bit4_adder.h
14 #include <systemc.h>
15
16 SC_MODULE (bit4_adder) {
17     sc_in <bool> Cin;
18     sc_in <bool> A1, A2, A3, A4;
19     sc_in <bool> B1, B2, B3, B4;
20
21     sc_out <bool> S1, S2, S3, S4;
22     sc_out <bool> Cout;
23
24     //Auxiliar variables to ha1 and fa1
25     sc_signal <bool> s11, c11, cin_1d;
26     sc_signal <bool> s1, c1;
27
28     //Auxiliar variables to ha2 and fa2
29     sc_signal <bool> s21, c21, c1_1d;
30     sc_signal <bool> s2, c2;
31
32     //Auxiliar variables to ha3 and fa3
33     sc_signal <bool> s31, c31, c2_1d;
34     sc_signal <bool> s3, c3;
35
36     //Auxiliar variables to ha4 and fa4
37     sc_signal <bool> s41, c41, c3_1d;
38     sc_signal <bool> s4, c4;
39
40     //Delay Inputs to prevent miss match
41     sc_signal <bool> a2_1d, a2_2d, b2_1d, b2_2d;
42
43     sc_signal <bool> a3_1d, a3_2d, a3_3d, a3_4d,
44         b3_1d, b3_2d, b3_3d, b3_4d;
45     sc_signal <bool> a4_1d, a4_2d, a4_3d, a4_4d,
46         a4_5d, a4_6d;
47     sc_signal <bool> b4_1d, b4_2d, b4_3d, b4_4d,
48         b4_5d, b4_6d;
49
50     //Delay 1st Output
51     sc_signal <bool> s1_1d, s1_2d, s1_3d, s1_4d,
52         s1_5d, s1_6d;
53     sc_signal <bool> s2_1d, s2_2d, s2_3d, s2_4d;
54     sc_signal <bool> s3_1d, s3_2d;
55
56     void ha_1() {
57         s11 = (!(A1 && B1)) && (A1 || B1);
58         c11 = (A1 && B1);
59         cin_1d = Cin;
60
61         //Delay Inputs
62         a2_1d = A2;
63         b2_1d = B2;
64         a3_1d = A3;
65         b3_1d = B3;
66         a4_1d = A4;
67         b4_1d = B4;
68     }
69
70     void fa_1() {
71         s1 = (!(s11 && cin_1d)) && (s11 || cin_1d);
72         c1 = (c11 || (s11 && cin_1d));
73
74         //Delay inputs
75         a2_2d = a2_1d;
76         b2_2d = b2_1d;
77         a3_2d = a3_1d;
78         b3_2d = b3_1d;
79         a4_2d = a4_1d;
80         b4_2d = b4_1d;
81     }
82 }

```

```

78                                     138
79                                     139
80 void ha_2() {
81     s21 = (!(a2_2d && b2_2d)) && (a2_2d || 140
        b2_2d);
82     c21 = (a2_2d && b2_2d);
83     c1_1d = c1;
84
85     //Delay inputs
86     a3_3d = a3_2d;
87     b3_3d = b3_2d;
88     a4_3d = a4_2d;
89     b4_3d = b4_2d;
90
91     //Delay output
92     s1_1d = s1;
93 }
94
95 void fa_2() {
96     s2 = (!(s21 && c1_1d)) && (s21 || c1_1d 156
97     c2 = (c21 || (s21 && c1_1d));
98
99     //Delay inputs
100    a3_4d = a3_3d;
101    b3_4d = b3_3d;
102    a4_4d = a4_3d;
103    b4_4d = b4_3d;
104
105    //Delay output
106    s1_2d = s1_1d;
107 }
108
109 void ha_3() {
110     s31 = (!(a3_4d && b3_4d)) && (a3_4d || 170
        b3_4d);
111     c31 = (a3_4d && b3_4d);
112     c2_1d = c2;
113
114     //Delay inputs
115     a4_5d = a4_4d;
116     b4_5d = b4_4d;
117
118     //Delay output
119     s1_3d = s1_2d;
120     s2_1d = s2;
121 }
122
123 void fa_3() {
124     s3 = (!(s31 && c2_1d)) && (s31 || c2_1d 186
125     c3 = (c31 || (s31 && c2_1d));
126
127     //Delay inputs
128     a4_6d = a4_5d;
129     b4_6d = b4_5d;
130
131     //Delay output
132     s1_4d = s1_3d;
133     s2_2d = s2_1d;
134 }
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

Listing B.5: The Bit 4 Adder design

B.1.6 ALU

```

1 //main.cpp
2
3 #include <systemc.h>
4 #include "alu.h"
5
6
7 int sc_main(int argc, char* argv[]) {
8     ALU alu("ALU");
9
10    sc_start(5000, SC_NS);
11    return 0;
12 }
13
14 //alu.h
15 #include <systemc.h>
16
17 SC_MODULE(ALU) {

```

```

18  int OPCODE;
19  int OP1, OP2;
20  bool CARRY, ZERO;
21  int RESULT;
22
23      int data1, data2;
24      int result;
25
26  void operate() {
27      while(true){
28          wait(5, SC_MS);
29          ZERO = false;
30          data1 = OP1 % 16;
31          data2 = OP2 % 16;
32
33          switch(OPCODE % 8){
34              case 0: //addition
35                  result = data1 + data2;
36                  break;
37
38              case 1: //subtraction
39                  result = data1 - data2;
40                  break;
41
42              case 2: //increment
43                  result = data1 + 1;
44                  break;
45
46              case 3: //decrement
47                  result = data1 - 1;
48                  break;
49
50          }
51
52          RESULT = (result % 16); // Ensuring
53                          RESULT is 4-bit
54          CARRY = ((result / 16) % 2);
55          if((result % 16) == 0)
56              ZERO = (true);
57      }
58  }
59
60  SC_CTOR(ALU){
61      SC_THREAD(operate);
62  }
63  };

```

Listing B.6: The ALU design

B.1.7 ALU 62

```

1  //main.cpp
2
3  #include <systemc.h>
4  #include "alu62.h"
5
6
7  int sc_main(int argc, char* argv[]) {
8      ALU alu("ALU");
9      sc_start(5000, SC_NS);
10     return 0;
11 }
12
13 //alu62.h
14 #include <systemc.h>
15
16 //N_BITS = 31
17 #define MAX_RESULT 2147483648 // => 2^N_bits
18
19 SC_MODULE(ALU) {
20     int OPCODE;
21     int OP10, OP11;
22     int OP20, OP21;
23     bool CARRY, ZERO;
24     int RESULT0, RESULT1;
25
26     //internal variables
27     int data10, data11;
28     int data20, data21;
29     int opcode;
30     int result0, result1;
31     int i;
32
33     void operate() {
34         while (true) {
35             wait(5, SC_MS);
36             ZERO = false;
37             data10 = OP10 % MAX_RESULT;
38             data11 = OP11 % MAX_RESULT;
39             data20 = OP20 % MAX_RESULT;
40             data21 = OP21 % MAX_RESULT;
41             opcode = (OPCODE % MAX_RESULT);
42
43             switch (opcode) {
44                 case 0: // Addition
45                     result0 = data10 + data20;
46                     result1 = data11 + data21;
47                     if (data10 > (MAX_RESULT - data20)) {
48                         result1 = result1 + 1;
49                     }
50                     break;
51
52                 case 1: // Subtraction
53                     result0 = data10 - data20;
54                     result1 = data11 - data21;
55                     if (result0 < 0) {
56                         result0 = MAX_RESULT - result0;
57                         result1 = result1 - 1;
58                     }
59                     break;
60
61                 case 2: // Increment
62                     result0 = data10 + 1;
63                     result1 = data11;
64                     if ((data10 == (MAX_RESULT - 1))) {
65                         result1 = result1 + 1;
66                     }
67                     break;
68
69                 case 3: // Decrement
70                     result0 = data10 - 1;
71                     result1 = data11;
72                     if (result0 < 0) {
73                         result0 = MAX_RESULT - result0;
74                         result1 = result1 - 1;
75                     }
76                     break;
77             }
78             RESULT0 = result0 % MAX_RESULT;
79             RESULT1 = result1 % MAX_RESULT;
80             CARRY = ((result1 / MAX_RESULT) % 2) == 1 ? true : false;
81             ZERO = (result0 % MAX_RESULT) == 0 ? ((result1 % MAX_RESULT) == 0 ? true : false) : false;
82         }
83     }

```

```
84     }
85
86     SC_CTOR(ALU) {
87         SC_THREAD(operate);
88     }
```

```
89
90     };
```

Listing B.7: The ALU design

Appendix C

Terminal Output PVL

In this Appendix are available the Terminal Output from the *Robot* and the *Bit 4 Adder* Assertion [HL](#) Verification Test.

C.1 Robot

```
1 [progress] Parsing inputs...
2 [progress] Parsing pvl file robot_assert.pvl
3 [progress] Parsing pass took 3410ms
4 [progress] AST conversion pass took 180ms
5 [progress] Read robot_assert.pvl successfully
6 [progress] Parsed 1 file(s) in: 3620ms
7 [progress] [03%] splitCompositeDeclarations took 20 ms
8 [progress] [07%] loadExternalClasses took 20 ms
9 [progress] [11%] standardize took 10 ms
10 [progress] [15%] encodeLockInvariantProof took 10 ms
11 [progress] [19%] encodeForkLockWait took 10 ms
12 [progress] [23%] specifyImplicitLoopLabels took 10 ms
13 [progress] [26%] propagateAbstractMethodContracts took 30 ms
14 [progress] [30%] arrayNullValuesToNone took 20 ms
15 [progress] [34%] addRequirementSatCheck took 10 ms
16 [progress] [38%] inferADTElementTypes took 10 ms
17 [progress] [42%] checkAssignInPar took 0 ms
18 [progress] [46%] inline took 30 ms
19 [progress] [50%] dummy-InvariantsPropagatedHere took 0 ms
20 [progress] First parsing pass took 630ms
21 [progress] AST conversion took 70ms
22 [progress] First parsing pass took 130ms
23 [progress] AST conversion took 10ms
24 [progress] [53%] simplify took 1280 ms
25 [progress] First parsing pass took 10ms
26 [progress] AST conversion took 0ms
27 [progress] [57%] optimizeForSilver took 10 ms
```

```

28 [progress] [61%] desugarArrayOps took 20 ms
29 [progress] [65%] flattenNestedExpressions took 20 ms
30 [progress] [69%] encodeInheritanceToDomain took 20 ms
31 [progress] [73%] importADTsAndRefEncode took 2110 ms
32 [progress] [76%] returnTypeToOutParameter took 20 ms
33 [progress] [80%] reduceQuantifierNesting took 10 ms
34 [progress] [84%] inlinePatternsToTriggers took 10 ms
35 [progress] [88%] scaleAllPredicateApplications took 20 ms
36 [progress] [92%] collectInnerDeclarations took 0 ms
37 [progress] [96%] collectDeclarations took 20 ms
38 [progress] Conversion to silver AST took 80ms
39 [progress] Verification with builtin silicon backend starting at 13:45:01
40 Success!
41 [progress] Task Viper verification took 25012 ms
42 [progress] [100%] applySilicon took 1460 ms
43 The final verdict is Pass
44 [progress] entire run took 34552 ms

```

Listing C.1: Terminal Output from the Robot Assertion HL Verification Test

C.2 Bit 4 Adder

```

1 [progress] Parsing inputs...
2 [progress] Parsing pvl file bit4_adder_assert.pvl
3 [progress] Parsing pass took 3510ms
4 [progress] AST conversion pass took 200ms
5 [progress] Read bit4_adder_assert.pvl successfully
6 [progress] Parsed 1 file(s) in: 3730ms
7 [progress] [03%] splitCompositeDeclarations took 30 ms
8 [progress] [07%] loadExternalClasses took 30 ms
9 [progress] [11%] standardize took 10 ms
10 [progress] [15%] encodeLockInvariantProof took 30 ms
11 [progress] [19%] encodeForkLockWait took 20 ms
12 [progress] [23%] specifyImplicitLoopLabels took 10 ms
13 [progress] [26%] propagateAbstractMethodContracts took 30 ms
14 [progress] [30%] arrayNullValuesToNone took 20 ms
15 [progress] [34%] addRequirementSatCheck took 30 ms
16 [progress] [38%] inferADTElementTypes took 10 ms
17 [progress] [42%] checkAssignInPar took 10 ms
18 [progress] [46%] inline took 110 ms
19 [progress] [50%] dummy-InvariantsPropagatedHere took 0 ms
20 [progress] First parsing pass took 590ms
21 [progress] AST conversion took 70ms
22 [progress] First parsing pass took 110ms
23 [progress] AST conversion took 10ms
24 [progress] [53%] simplify took 1830 ms

```

```
25 [progress] First parsing pass took 0ms
26 [progress] AST conversion took 10ms
27 [progress] [57%] optimizeForSilver took 70 ms
28 [progress] [61%] desugarArrayOps took 60 ms
29 [progress] [65%] flattenNestedExpressions took 40 ms
30 [progress] [69%] encodeInheritanceToDomain took 70 ms
31 [progress] [73%] importADTsAndRefEncode took 2470 ms
32 [progress] [76%] returnTypeToOutParameter took 70 ms
33 [progress] [80%] reduceQuantifierNesting took 40 ms
34 [progress] [84%] inlinePatternsToTriggers took 40 ms
35 [progress] [88%] scaleAllPredicateApplications took 40 ms
36 [progress] [92%] collectInnerDeclarations took 50 ms
37 [progress] [96%] collectDeclarations took 40 ms
38 [progress] Conversion to silver AST took 200ms
39 [progress] Verification with builtin silicon backend starting at 11:57:21
40 Success!
41 [progress] Task Viper verification took 2370551 ms
42 [progress] [100%] applySilicon took 4620 ms
43 The final verdict is Pass
44 [progress] entire run took 2385577 ms
```

Listing C.2: Terminal Output from the Bit 4 adder Assertion HL Verification Test