Instituto Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Maestría Académica en Electrónica

Trabajo Final de Graduación

“SCAsrt: An On-line Assertion Library for SystemC TLM 2.0”

Para optar por el título de:

Maestría en Electrónica con Énfasis en Sistemas Empotrados

Con el grado académico de:

Magister Scientiae

José Andrés Pacheco Castro

Cartago, agosto 2022

Abstract:

Electronic systems have become more and more complicated in recent years, demonstrated by the increasing gate counts in designs and the invested effort validation and verification of the same. Early architectural choices are harder to make and have a larger impact in time to market, resulting in the widespread adoption of high level modeling for design prototyping and specification. The objective of this thesis is to design and implement an on-line assertion library for validating the correctness of these high level models. The library was designed to implement assertions for the SystemC TLM 2.0 modeling facilities, using native C++ syntax and checking the assertions live as the model is simulated and without modifying SystemC's kernel, being these two features the main differentiation factor from previous works. The library was validated using unit testing and a comparison to the existing SVA language. A proof of concept exercise was applied to an existing TLM LT model, where some features were validated using the assertion library. The performance impact of the assertion usage is of an increase of 30% on model simulation time, and a fixed increase in total memory usage.

Keywords:

PSL, CTL, LNL, ASIC, FPGA, VLSI, integrated circuits

Resumen:

Los sistemas electrónicos han venido aumentando en complejidad en los últimos años, hecho demostrado por el aumento en la cantidad de compuertas en diseños y el esfuerzo invertido en la verificación y validación de éstos. Decisiones arquitecturales al principio del proyecto se vuelven más difíciles de tomar y tienen un mayor impacto en el tiempo de llegada al mercado, resultando en la adopción general del modelado de alto nivel para el prototipado de diseños y su especificación formal. El objetivo de esta tesis es diseñar e implementar una biblioteca de aserciones en línea para validar la correctitud de estos modelos de alto nivel. La biblioteca fue diseñada para implementar aserciones para SystemC TLM 2.0, usando la sintaxis nativa de C++ y validando las aserciones en vivo conforme el modelo es simulado y sin modificar el kernel de SystemC, siendo estos dos aspectos la principal diferenciación con respecto a trabajos anteriores. La biblioteca fue validad usando pruebas unitarias y una comparación con el lenguaje SVA ya existente. Una prueba de concepto fue aplicada a un modelo TLM LT, donde algunas características fueron validadas usando la biblioteca de aserciones. El impacto en el desempeño por el uso de la biblioteca se midió en un incremento de un 30% al tiempo de ejecución, y un incremento fijo en el uso de memoria total.

Palabras clave:

PSL, CTL, LNL, ASIC, FPGA, VLSI, circuitos integrados

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
Maestría Académica en Electrónica
Trabajo Final de Graduación
Tribunal Evaluador
Acta de Aprobación de Tesis

Defensa del Trabajo Final de Graduación
Requisito para optar por el título de Máster en Ingeniería Electrónica
Grado Académico de Magister Scientiae

El Tribunal Evaluador aprueba la defensa del Trabajo Final de Graduación denominado "SCAsrt: An On-line Assertion Library for SystemC TLM 2.0", realizado por José Andrés Pacheco Castro Carné: 2018319396, y hace constar que cumple con las normas establecidas por la Unidad Interna de Posgrados de la Escuela de Ingeniería Electrónica del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal Evaluador

ENRIQUE COEN ALFARO (FIRMA)
Digitally signed by ENRIQUE COEN ALFARO (FIRMA)
DN: serialNumber=CPF-01-0951-0459, sn=COEN ALFARO, givenName=ENRIQUE, c=CR, o=PERSONA FISICA, ou=CIUDADANO, cn=ENRIQUE COEN ALFARO (FIRMA)
Date: 2022.09.10 11:34:57 -06'00'

M.Sc Enrique Coen Alfaro
Profesor Lector

[illegible]

M.Sc Mauricio Gurdián Murillo
Profesor Lector

ROBERTO CARLOS MOLINA ROBLES (FIRMA)
Digitally signed by ROBERTO CARLOS MOLINA ROBLES (FIRMA)
Date: 2022.09.16 01:15:57 -06'00'

M.Sc Roberto Carlos Molina
Evaluador Independiente

GERARDO JOSE CASTRO JIMENEZ (FIRMA)
Firmado digitalmente por GERARDO JOSE CASTRO JIMENEZ (FIRMA)
Fecha: 2022.09.09 23:23:36 -06'00'

M.Sc Gerardo Castro Jiménez
Director de Tesis

Cartago, agosto 2022

TEC | Tecnológico de Costa Rica

Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica

Maestría en Electrónica
con Énfasis en Sistemas Empotrados

Thesis director: Ing. Gerardo Castro. M.Sc

SCAsrt: An On-line Assertion Library for SystemC TLM 2.0

Author:
José Andrés Pacheco Castro

August 2022

Contents

List of Figures

List of Tables

1 Introduction

1.1 Thesis proposal and scope

SystemC is a systems-level modeling language, defined in the IEEE 1666-2011 standard [1]. Developed by the Accellera Systems Initiative, the SystemC C++ library can be used in high level modeling, simulation, and verification of hardware designs. In addition to the base library, extensions exist that provide facilities for modeling analog and mixed-signal designs, synthesis, configuration control & inspection (CCI), and simulation-driven verification.

SystemC is capable of modeling at different abstraction levels, from register transfer level (RTL) to transaction level modeling (TLM). For high level design the latter is most useful since it allows rapid prototyping while maintaining enough detail for producing useful simulation results.

Assertion-driven verification is an industry standard practice, used to capture design intent more accurately. It is a language that expresses relationships (rules) between different signals across a given time frame. It is used in combination with a formal validation tool or a simulation tool to root out issues in a model implementation.

As design complexity has grown over time, systems-level modeling has become an important tool for exploring different architectural options and their impact on performance and functionality. Validating that these models are without errors is an important concern.

This thesis intends to explore the possibility of using assertion-based verification methodologies in the validation of SystemC high-level models. A SystemC Assertions (SCAsrt) library will be developed that provides the capability of expressing relationships between different TLM 2.0 transactions across a span of time and validate that these relationships hold throughout the model simulation time.

The inspiration for this library will be the Property Specification Language (PSL), published under the IEEE standard 1850-2010 [2]. PSL has successfully been used as the basis for assertion implementations in System Verilog (SVA, IEEE standard 1800-2017 [3]), SystemC (same standard as previously mentioned) and Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL, IEEE standard 1076-2008 [4]). Note that SystemC's current assertion support is limited to its RTL modeling layer, not its TLM layer.

The syntax and capabilities of this new library will be based off of System Verilog's assertion language, but only a subset of these will be implemented. The intent is for the subset to be enough to significantly validate a design and make future expansion possible, taking into account the stated objectives of the transaction level modeling methodology. This means that the feature selection will prioritize the PSL language constructs with the most relevance to a high-level model.

1.2 Objectives

1.2.1 General Objectives

- Develop a library to express time-based relationships between different transactions (assertions), and validate them during simulation.

1.2.2 Specific Objectives

- Implement a subset of PSL for TLM 2.0 transactions. This subset will prioritize the constructs most relevant to a high level model.

- Implement a standalone library, avoiding external requirements to proprietary tools.
- Develop the library without any modification of base SystemC classes.
- Utilize the library to validate some features in an already existing SystemC model.
- Measure the performance impact of the library during model simulation.

1.3 Justification

The size and complexity of application specific integrated circuits (ASICs) and field programmable gate arrays (FPGAs) designs have been steadily increasing in the industry. This has been shown by surveys such as the 2020 Wilson Research Group Functional Verification Study [5], which points out that from 2014 to 2020 the number of designs with more than 500 million gates has increased from less than 15% to more than 25%.

The same survey shows the consequences of these trends. Effort spent in verification has overtaken and surpassed effort spent in design, with more engineers working in validation than design nowadays. This has pushed the industry to look into every part of the process where the chances of introducing errors in the design can be minimized.

The main cause of ASIC re-spins are logical or functional flaws, accounting for almost 50% of the total re-spins in the survey. As designs grow in complexity, architectural requirements increase in number, leading to a bigger chance of problems appearing either due to a bad architectural decision, or a bad interpretation of the architectural requirements leading to an erroneous implementation in RTL.

The development of system level modeling languages provides a solution to the trends seen above: Architectural models provide a means of testing solutions before spending a lot of effort in more detailed models, such as RTL. This rapid prototyping speeds up exploration and allows architects to more efficiently decide on the best possible solution to meet the design requirements. Additionally, these models become reference implementations down the line, capturing in a formal language the requirements of the design. Then, lower level implementations can be checked against the results obtained by the higher level models.

Of course, as higher level models become more complex the possibility of introducing errors in their implementation increases as well. Validation techniques developed to verify RTL can be adapted to validate higher level models as well, giving architects the confidence that their models accurately reflect their intent.

The importance of developing tools to validate these models is now clear. Assertion-based validation is one of these such techniques, particularly well suited to the task due to the low barrier of entry that writing an assertions poses, and the ability to capture part of the designer's intent itself.

As will be seen in the state of the art section, plenty of work has been done before in this field. Tools exist to perform some level or other of assertion based validation in SystemC TLM models. However, the proposed approach employed by SCAsrt is unique in its combination of online assertion checking, and no interference with the normal operation of the design and the SystemC library.

1.4 Problem Statement

As mentioned in the introduction, SystemC's high level modeling capabilities are well developed and in widespread use. However, validation facilities for this high level modeling has an unfulfilled gap that this thesis will aim to improve. This gap is the power of using assertions for validation.

Since SystemC is open source and its interfaces are standardized, it is possible to extend its functionality with external libraries created by third parties. Creating a separate library to define and implement assertions is therefore a good choice that many SystemC users will be able to adopt. Such an assertion library needs to provide two things: A means of expressing the expected behavior of a design, and a means of checking these expressions against the actual behavior of the design.

To express design intent, a syntax must be provided which captures the relationship between different TLM transactions. The first layer of this syntax is the boolean layer, which allows the creation of expressions that can be evaluated at a given instant of time. The second layer is the temporal layer, which takes the expressions of the previous layer and establishes their relationship across a given span of time. With these components, assertions can be created that describe how a design is expected to behave.

To check the expressions against design behavior, the library must provide a kernel and the means of monitoring the flow of transactions in the design. The monitors capture transactions as they flow from TLM initiator socket to TLM target socket. The kernel keeps a list of assertions describing the model, forwarding the monitored transactions to them and determining when the assertions hold or do not hold.

At the end of design simulation, a report must be generated that shows that for a given stimulus and design, which transactions were triggered, which passed, and which and failed.

1.5 Requirements

- SystemC library implementation, version 2.3.3, provided free of charge by the Accellera Systems Initiative, under the Apache 2.0 license.
- GCC C++ compilation toolchain, version 9.0.4, provided free of charge by the Free Software Foundation, under the GPL license.
- SystemC model to validate with SCAsrt. The model used in this thesis is the RISC-V based Virtual Prototype, published by Group of Computer Architecture of the University of Bremen, under the MIT license.
- VCS System Verilog compiler, published by Synopsys, under a nonfree license.

1.6 Methodology

The methodology applied in this project is experimental, following industry-standard software development practices. A software architecture to satisfy the library requirements will be proposed, such that the goals of an assertion-based validation methodology can be met by its use. Then, an implementation of the architecture will be developed. The correctness of this implementation will be confirmed by the use of unit testing and equivalence checking to existing industry tools. Finally, the implementation will be used to validate an already existing design. Statistical software performance measurement tools will be used to measure the impact that usage of this library has in comparison with the baseline execution of the original model.

The thesis will be successful when an existing design has some features validated with the developed library, and the performance impact of its use is measured.

1.6.1 Architectural considerations

The library architecture should not require any modifications of the base SystemC library. This will ensure compatibility with future releases of SystemC, and avoid the introduction of errors or bugs if users choose to use a SystemC implementation other that the reference one provided by Accellera.

SystemC Assertions will be provided as a set of classes and macros defined inside their own 'Scasrt' namespace. Information about the execution of the model will be obtained via the use of snooping macros, which will monitor TLM flow at the port connections between modules.

This design will allow users to instrument a given model with minimal effort, monitoring the transaction flow on the interesting TLM ports. The user will then define assertions based on the specification of the model of interest. This instrumented design can then be simulated with stimulus also provided by the user. Throughout the simulation the defined assertions will monitor the transaction flow. If at any point a property is violated, the simulation will be aborted with a message indicating the cause of failure. At the end of a successful simulation, a report will be generated with details on triggered assertions, non-triggered assertions, and general behavior of the model.

1.6.2 Implementation considerations

The proof-of-concept implementation will use unit testing methodology to validate its correctness. Each class, function and macro will have an associated suite of tests to check that responses are what is expected against the full range of possible inputs. Additionally, integration tests will validate interactions among the different parts, such that when the implementation is ready to be used in a model the library will perform as expected.

1.6.3 Usage considerations

It is of particular interest to measure the impact of using this library on simulation performance. Speedy simulation is of vital importance for architecture exploration, and too much of a cost here may impact the usefulness of SCAsrt.

2 Theoretical Framework

2.1 High level system modeling

The process of developing a digital circuit starts with gathering and defining requirements. Once this is done, the architect is tasked with coming up with an implementation plan that will meet the collected requirements and best practices criteria.

This criteria includes efficient use of resources, ease of design verification, ease of product validation, etc. These items often have conflicting interests, and achieving a balance between them is complicated.

High level system modeling shortens the feedback loop between coming up with a design and measuring its performance. A designer is able to implement different possible architectures, by modeling only the bare minimum details that are required for evaluating performance or other metrics.

This high level system model has other benefits besides architectural exploration. Once an architecture is chosen, the model may be used as a reference for the lower stages of implementation, acting as a more formal language for specifying the design's requirements.

The high level model (HLM) must model three aspects of a system in order to be of use[6]:

- Object and data modeling: Parameters of the system components.
- Activity modeling: Description of the processes that each system component performs.
- Behavioral modeling: Temporal description of relationships between the two previous items.

2.1.1 SystemC facilities for high level system modeling

SystemC is a library designed to create event-driven simulations, that is, where concurrent processes are executed in parallel. The classes it provides allow for modeling at different abstraction levels, from the low register transfer level to the high transaction-modeling level.

The transaction level modeling classes, known as TLM 2.0, are the collection of classes and interfaces which allow the latter.

TLM 2.0 further provides two different levels of timing granularity. The two coding styles are as follows:

- LT (Loosely timed): This is the abstraction level with the least amount of timing detail. It supports only two timing points, at the start and end of a transaction. There is temporal decoupling between modules with this style.
- AT (Approximately-timed): This abstraction level has more timing detail. Each transaction exchange is modeled with 4 timing points. Here processes run in lockstep with the simulation time.

The classes and interfaces provided are as follows:

- Sockets: Used to model connections between modules. Route transactions from one place to another.
- Blocking interface: Used in the loosely timed coding style. The socket interrupts execution until the transaction is processed by the target.
- Non-blocking interface: Used in the approximately timed coding style. The socket queues the transaction without interrupting the current process.
- Generic Payload: Models the contents of a transaction.
- Phases: Represents the different stages through which a transaction passes.
- DMI: Short for direct memory access. Used to speed up large data transfers inside transactions.
- Quantum: Smallest unit of time. Used to temporally decouple modules in the loosely timed coding style.

With the above elements, systems are modeled as discrete components that exchange messages via sockets. A module simulates the desired behavior up to the point where communication with a separate module must occur, at which point the LT (see figure 1) or AT (see figure 2) protocol is used to send the information using timing annotations.

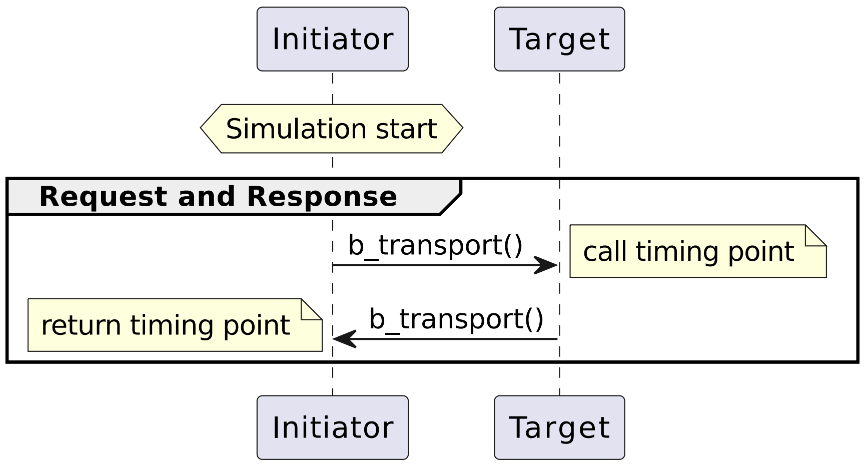


Figure 1: LT communication protocol

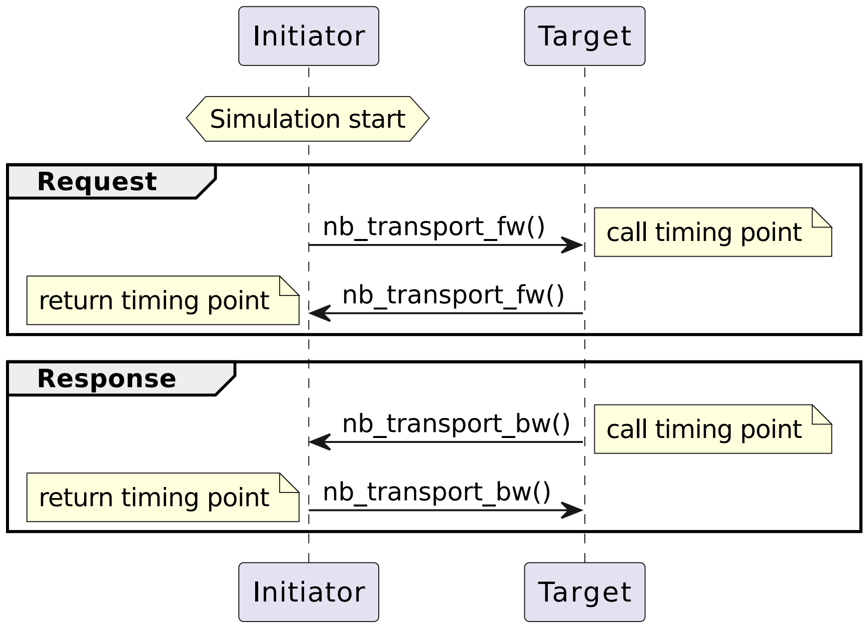


Figure 2: AT communication protocol

2.2 Assertion based validation

2.2.1 Definition of an assertion

An assertion is a construct that captures architectural intent [7]. It does not contribute to the implementation of a feature. Rather, it is used to validate consistency between what a designer created and what the specification required.

As opposed to traditional simulation-based approaches, where scoreboards check the output of a model against its input, assertions exist very close to the code they check. This characteristic makes assertions an invaluable addition to the validation toolkit, as they provide improved observability, reduce the debug time, check the code being executed during all stages of testing, and act as a more robust, formal way of documenting requirements and assumptions in the code.

In dynamic verification, assertions additionally help in determining the level of coverage achieved.

In formal verification, assertions are the basic building blocks which formal engines use to explore the state space of a model and generate proofs of correctness.

2.2.2 Mathematical foundations of assertions

Linear temporal logic (LTL) is a formal system proposed by Amir Pnueli in 1977 [8]. Its intent is to provide a means of temporal reasoning, where relationship between current and future events can be expressed in a formal language. Its grammar is composed of:

- Propositional variables
- Logical operators: negation ($\neg$), conjunction ($\wedge$), disjunction ($\neg$), implication ($\implies$), biconditional ($\iff$), true and false.
- Temporal operators: Always (**G**), finally (**F**), release (**R**), weak until (**W**) and strong release (**M**).

With this components an LTL formula can be constructed, such that a given sequence of events can be said to satisfy or not satisfy it.

Computational tree logic (CTL) is a formal system proposed by Edmund Clarke and Allen Emmerson in 1981 [9]. This system models time with a tree structure, where each path is a different future possibility. Its grammar is composed of:

- Propositional variables
- Logical operators: negation ($\neg$), conjunction ($\wedge$), disjunction ($\neg$), implication ($\implies$), biconditional ($\iff$), true and false.
- Paths quantifier: All ($\mathbf{A}\Phi$), exists ($\mathbf{E}\Phi$)
- Path quantifiers: Next ($\mathbf{N}\phi$), Globally ($\mathbf{G}\phi$), Finally ($\mathbf{F}\phi$), Until ($\phi\mathbf{U}\psi$), Weak until ($\phi\mathbf{W}\psi$),

With this components a CTL formula can constructed, such that a given system may be said to satisfy or not satisfy the formula for any possible inputs.

2.2.3 Existing assertion standards

The IEEE published the Property Specification Language (PSL) for the first time in 2005, with the latest revision published in 2010 [2].

This specification defines a formal language for describing electronic systems behavior. From this central specification the assertion languages for VHDL, SystemVerilog and SystemC were defined.

The stated goals for this standard is that the language is:

- Easy to learn, read and write.
- Concise.
- Rigorous.
- Expressive.

- Efficient to implement.

PSL is an extension of LTL and CTL. To implement the modeling capabilities of LTL and CTL, PSL uses a layered approach that compartmentalizes the functionality efficiently.

- Boolean layer: The boolean layer supplies boolean expressions to the upper layers. It allows the expression of signal relationships that happen in a single quantum of time. A boolean construct has two possible outcomes: The result is true or not true.
- Temporal layer: The temporal layer is used to express temporal relationships between signals. It builds on the boolean layer by adding operators that may span over multiple quantums of time. A temporal construct has three possible outcomes:
  - Holds strongly: For a given sequence of events, the temporal construct holds and no future stimulus may cause it to not hold.
  - Holds softly: For a give sequence of events, the temporal construct holds, but future stimulus may cause it to not hold.
  - Does not hold: For a given sequence of events, the temporal construct does not hold, and no future stimulus may cause it to hold.
- Verification layer: The verification layer is used to indicate the purpose of the temporal and boolean expressions. For example, the verification layer can indicate that a given assertion must hold at all times, or that a given sequence of events should never happen. A verification construct (in a simulation context) has two possible outcomes: the construct passes or the construct fails.
- Modeling layer: The modeling layer is used to model the behavior of the design input. For example, in a simulation context it applies stimulus to the model, while in a formal context it generates proofs regarding model behavior.

PSL can express properties that cannot be easily evaluated in simulation, as opposed to formal environments. In simulation-exclusive contexts, PSL defines a simple subset, where the operators conform to the notion of a monotonically advancing time. This means that time advances through the property from left to right, which eases their validation during simulation.

2.3 State of the Art

Multiple approaches have been tried in the past to allow assertion-based validation of SystemC TLM 2.0 models. These approaches have been classified based on the following criteria:

- Abstraction level: Whether the assertions support TLM or RTL, or something in between.
- Execution stage: Whether the assertions are evaluated as a post-process, on line during simulation, or as a formal analysis of the model.
- SystemC implementation interference: Whether the approach requires modification of the base SystemC library, either to the code itself or as an automatic instrumentation of the resulting binary.

- Model implementation interference: The level of modification that must be done to the target model itself in order to evaluate the defined assertions.
- Assertion language implemented: Language used by the approach to express assertions, be that a domain specific language (DSL) or the same language the model is defined in.
- Implemented properties: What kinds of assertions is the approach able to express. Boolean, time expressions, relationships between different ports, etc.

The existing approaches are summarized in table 1. These are grouped by author and sorted by date, as their work may span several years and multiple publications.

Paper	**Year**	**Summary**
Automated Test Generation for Validating SystemC Designs [10] [11] [12]	2021, 2018, 2016	Uses binary concolic execution to generate test cases and validate the entire model design space.
Scalable Simulation-Based Verification of SystemC-Based Virtual Prototypes [13]	2019	Off-line trace analysis of TLM 2.0 properties, instruments model automatically with Clang
Assertion-Based Verification for SoC Models and Identification of Key Events [14] [15] [16]	2017	Boolean properties written in PSL are automatically translated to checkers
Automatic equivalence checking for SystemC-TLM 2.0 models against their formal specifications [17] [18]	2017	Uses assertions to validate compliance with TLM 2.0 AT protocol, GDB used to extract model behavior
Reusing RTL Assertion Checkers for Verification of SystemC TLM Models [19] [20] [21]	2015, 2014	Used for RTL cycle-accurate System C models only.
Assertion-based flow monitoring of SystemC models [22]	2014	Even higher level modeling. Introduces concept of flows between modules that can be validated.
Assertion-Based Functional Consistency Checking between TLM and RTL Models [23]	2013	Uses assertions to validate consistency between low-level TLM and RTL models.
Chimp: a tool for assertion-based dynamic verification of SystemC models [24] [25] [26]	2013, 2012, 2010	Tool takes model and PSL assertions as input, produces an executable model that indicates PASS or FAIL.
Scalable fault localization for SystemC TLM designs [27] [28] [29] [30]	2013, 2012, 2010	Uses assertions and model checking, to generate multiple execution traces and pinpoint source of errors.
Exploiting UML based validation for compliance checking of TLM 2 based models [31]	2012	Checks for protocol compliance in TLM 2.0 models
Monitoring transaction level SystemC models using a generic and aspect-oriented framework [32] [33]	2012, 2010	Uses an aspect oriented tool to modify the base SystemC library and integrate monitors into it.
Concurrency-oriented verification and coverage of system-level designs [34]	2011	Uses fault insertion and test case mutation to quickly validate system-level designs.
TLM protocol compliance checking at the Electronic System Level [35]	2011	User specifies protocol sequences that are checked against a test trace.

An assertion-based verification method for SystemC TLM [36]	2012	Uses SimplePSL to describe static assertions and verify they hold.
ISIS: Runtime verification of TLM plataforms [37] [38] [39]	2010, 2009, 2008	Describes a prototype tool called ISIS, used to execute formal PSL property checking on SystemC models
Proving transaction and system-level properties of untimed SystemC TLM designs [40] [41]	2010, 2009	Implements properties using finite state machines, validates them with formal methodologies.
Transactions Sequence Tracking by means of Dynamic Binary Instrumentation of TLM Models [42]	2009	Proposes a process to inject faults in the binaries of SystemC models, to avoid source code modification.
Implementation of a Transaction Level Assertion Framework in SystemC [43]	2007	Requires profound changes to the SystemC library in its implementation.
Design and verification of SystemC transaction-level models [44]	2006	Methodology to automatically convert UML-described properties into AsmL for validating SystemC models

Table 1: Past approaches to verification in high-level SystemC models

After studying all of these in depth, a niche has been identified, which SCAsrt aims to fill. SCAsrt has the following combination of characteristics which make it a unique contribution to the field:

- Operates at the TLM 2.0 abstraction level.
- Executes assertions on-line during normal model simulation.
- Does not require modification to the base SystemC implementation.
- Does require some light instrumentation of the target model to observe transaction flow.
- Implements a subset of PSL, expressed as native C++ code.

Another study of the state of the art validation in SystemC is available at [45]. A paper on the performance of LNL logic is available at [46].

Overall, SCAsrt's contribution to the state of the art is the combination of the above mentioned features, which have been implemented partially by other projects in the past, but never all combined in the same library. Previous efforts have created assertions for the TLM 2.0 features, but as an off-line process checked after simulation. Still others require the use of a DSL, which introduces a layer of separation between the model and the assertions, reducing portability across the multiple environments in which the model may be exercised. Finally, other projects have implemented on-line checking, but for the RTL layer of SystemC, not the high level modeling layer. By allowing native C++ assertions to validate the correctness of a high level model as it executes, SCAsrt is uniquely positioned to become an efficient validation tool that operates in all parts of the ASIC development cycle.

3 Solution Architecture

3.1 Observability

The first step in defining SCAsrt's architecture is determining how the model will be instrumented for observability. The main concern of assertions is monitoring transaction flows. As such, the best place to probe is in the connections between modules, at the initiator and target sockets.

This probing should be implemented in as friction-less a way as possible. The library user should only have to do the minimum possible amount of modifications to their model. This will be achieved via a set of macros, which replaces the traditional b_transport and nb_transport calls inside the models.

With this approach, the user will only have to modify a single line of code in their models per connection that they wish to monitor.

An example of how the macro operates can be seen in figure 3.

```
// Original socket transport call
socket->b_transport(*trans, delay);

// Replaced socket transport call
SNOOP_B_TRANSPORT(socket, *trans, delay);
```

(a) Loosely-timed macro

```
// Original socket transport call
status = socket->nb_transport_fw(*trans, phase, delay);

// Replaced socket transport call
SNOOP_NB_TRANSPORT_FW(status, socket, *trans, phase, delay);
```

(b) Approximately-timed macro

Figure 3: Macro usage example

The data captured at these observation points is the following:

- TLM transaction: This is the tlm_generic_payload struct that encapsulates a transaction in TLM 2.0.
- Transaction status: Field indicating whether the transaction was accepted, updated or completed.
- Transaction delay: In approximately-timed models, this field encodes the transaction processing delay.
- Transaction phase: In approximately-timed models, this field indicates the phase, it being BEGIN_REQ, END_REQ, BEGIN_RESP or END_RESP.
- Snooping phase: Phase where the transaction was snooped by the kernel. For every TLM 2.0 phase, there is a snoop before and after the transaction is processed.
- Socket and module hierarchy: This indicates the location in the model where the transaction was observed.
- Simulation time: This indicates the simulation time where the transaction was observed.

3.2 Adaptation of the PSL architecture

As discussed in the theoretical framework chapter, PSL uses the boolean, temporal and verification layers to capture the intent of assertion constructs. When applied to the TLM language, some parts of the standard must be adapted to better suit the higher-level modeling approach. Another important consideration is that this thesis will consider only the simple subset of PSL properties, as its purpose is to be applied in simulation contexts.

In regards to the boolean layer, PSL is capable of expressing relationships between signals using operators such as ==, ! =, <, ===. In SCAsrt, the boolean layer should express relationships between fields in a single TLM. Due to the nature of TLM modeling, there is no concept of synchronicity between TLMs: Different units may run ahead in the simulation, and execute multiple operations until a synchronization point is reached. This makes the concept of a boolean layer without a time component only meaningful inside the fields of a single TLM.

In the temporal layer, PSL expresses relationships between multiple cycles and sequences of cycles. For this purpose it uses operators such as **within**, **and**, **union**, **implication**, **clocking**, etc. These time relationships have a strong concept of a clocking event, where signal changes are synchronized. In TLM modeling, there is no clocking event. A concept of global time is present instead, which is used to control delay between the different execution phases inside a TLM and between TLMs. To implement the temporal layer, SCAsrt will provide a syntax to capture relevant TLMs, and the capability to express relationships between their fields. As seen in the previous section, since the data captured during a snoop includes the current simulation time, the necessary expressivity will be available to library users.

In the verification layer, PSL provides operators for simulation such as **assert**, **always** and **never**. There are additional operators such as **assume**, **restrict**, and **fairness**, but as these are not used in simulation context they fall outside the scope of this thesis. In SCAsrt, an assertion may be flagged as **PASS_ON_HOLD**, **PASS_ON_HOLD_STRONG**, **PASS_ON_NOTHOLD**, and **PASS_ON_NOTHOLD_STRONG**. This allows users to specify how they expect their models to interact with assertions.

Finally, the modeling layer is provided entirely by SystemC itself, SCAsrt does not implement anything in regards to this last item.

A summary table of the supported PSL features is available at table 2, presented in the section explaining the SCAsrt syntax.

3.3 Assertion kernel operation

The assertion kernel is the engine that powers the checking of assertions.

Its responsibility is to receive the monitored transactions, and for each such event:

1. Forward transaction data to each active assertion.
2. Spawn any new assertions that may have been triggered.
3. Clean up any assertions that may have become disabled.
4. At the end of the test, enumerate remaining assertions and generate a report of all interesting events that happened during simulation.

This operation applied to AT and LT models is seen in figure 4.

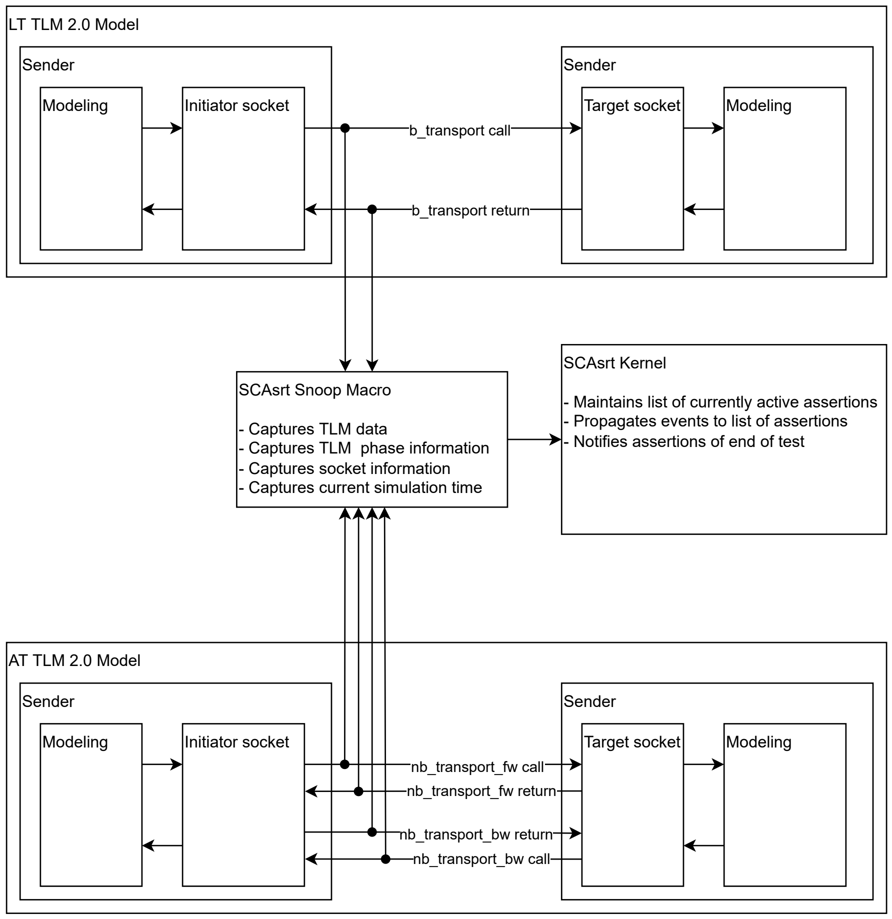


Figure 4: SCAsrt Kernel diagram

3.4 Assertion logic structure

To function, assertions need to implement the boolean and temporal layers of assertion logic. In the boolean layer, a condition is checked for a given instant of time. In the temporal layer, a chain of these previous conditions must hold over a given period of time.

The following notation will be used to describe a given assertion:

- a: a single letter represents a boolean property that may be true or false for a given TLM. The TLM may also be ignored by the property, in which case it does not return either false or true. An example of this last situation is when the current event is happening in a socket different than the one the property is interested in.
- $[a, b]c$: This is the smallest possible assertion unit, an assertion atom.
  - a represents a property that when true, starts the assertion execution.
  - b represents a property that when true, stops the assertion execution.
  - c represents a property that when true, the assertion holds.

  true denotes a property that always evaluates to true. **false** denotes a property that always evaluates to false. $<$**number**$>$ denotes a property that waits for the indicated amount of time. **main** denotes a start or stop property that is the same as the main body property.

- $[a, b]c \rightarrow [d, e]f$: The right arrow is used to chain assertion atoms together. When an assertion atom precedent holds, the consequent starts evaluation. Any number of assertion atoms may be chained in this way.
- $[a, b]c \rightarrow [d, e]f \rightarrow [g, h]i$ (pass criteria): At the end of a chain of evaluation atoms, a pass criteria is declared. This indicates whether the assertion passes or fails depending on its hold status at the end of the assertion lifetime.

The library will provide the means to declare these chains of assertion atoms and register them with the kernel at the start of simulation.

As the simulation progresses, the assertions will receive TLMs and update their status accordingly. The decision tree for assertion status can be seen in figure 5.

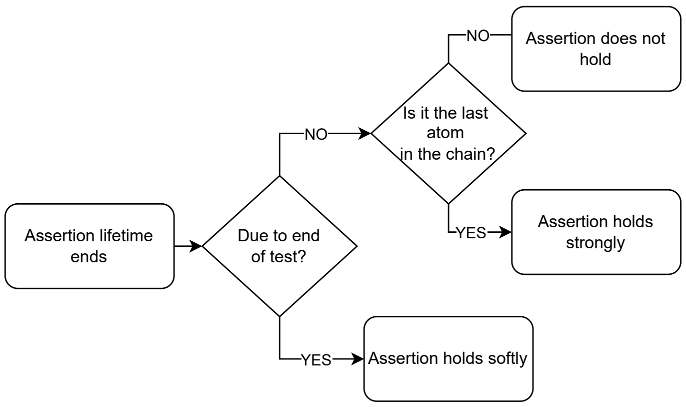


Figure 5: Assertion hold logic

After determining the assertion hold status, the kernel will check its corresponding pass criteria and emit a final judgment on whether the stimulus caused the assertion to pass or fail.

The described logic structure maps to different PSL features as seen in table 2.

PSL Feature	**Examples**	**SCAsrt support**
HDL operators	==, !=, >	Yes, C++ operators within TLM fields
Union operator	union	Yes, start and stop properties
Clocking operator	@	No analogue in TLM modeling
SERE repetition operator	[*], [+], [=], [->]	No
Sequence within operator	within	Yes, assertion atom chaining
Sequence AND operator	&, &&	Yes, assertion atom chaining
Sequence OR operator	|	Yes, assertion atom chaining
Sequence fusion operator	:	Yes, assertion atom chaining
Sequence concatenation operator	;	Yes, assertion atom chaining
FL termination operator	abort, sync_abort	No
FL occurrence operator	X, X!, F	Yes, start and stop properties
FL bounding operator	U, W	Yes, start and stop properties
Sequence implication operators	|->, |=>	Yes, assertion atom chaining
Boolean implication operators	->, <->	Yes, C++ operators within TLM fields
FL invariance operators	always, never, G	Yes, pass criteria

Table 2: SCAsrt PSL feature support

4 Solution Implementation

The above architecture has been implemented using native C++ classes, with the responsibilities divided as follows:

4.1 Types

The types component is a collection of enumerators and associated functions that are used by the SCAsrt classes to represent various states and messages that pass among them. The types are as follows:

- SnoopPhase: This enumerator encodes the phases at which transactions may be snooped
  - B_TRANSPORT_BGN
  - B_TRANSPORT_END
  - NB_TRANSPORT_FW_BGN
  - NB_TRANSPORT_FW_END
  - NB_TRANSPORT_BW_BGN
  - NB_TRANSPORT_BW_END
- AsrtStatus: This enumerator encodes the possible states that an assertion may have at the end of its lifetime:
  - ASRT_STATUS_INVALID
  - ASRT_HOLDS_STRONGLY
  - ASRT_HOLDS_SOFTLY
  - ASRT_DOESNT_HOLD
  - ASRT_NO_STATUS
- PropResult: This enumerator encodes the possible results a property may have after a transaction is probed:
  - PROP_PASS
  - PROP_FAIL
  - PROP_SKIP
- AsrtState: This enumerator encodes the lifetime status of a given assertion:
  - ASRT_STATE_INVALID
  - ASRT_DISABLED
  - ASRT_ENABLED
  - ASRT_EXPIRED
  - ASRT_MATCHED

- AsrtPass: This enumerator encodes the passing criteria assigned to a given assertion:
  - ASRT_PASS_INVALID
  - ASRT_PASS_ON_HOLD
  - ASRT_PASS_ON_HOLD_STRONG
  - ASRT_PASS_ON_NOTHOLD
  - ASRT_PASS_ON_NOTHOLD_STRONG

4.2 Classes

4.2.1 Tlm

This class encapsulates all the data captured at an observation point.

Its fields are the following:

- observation_point_: Point in the model hierarchy where the transaction was observed.
- time_stamp_: Simulation time at which transaction was probed.
- address_: Copy of tlm_generic_payload address field.
- command_: Copy of tlm_generic_payload command field.
- data_: Copy of tlm_generic_payload data field.
- length_: Copy of tlm_generic_payload length field.
- response_status_: Copy of tlm_generic_payload response status field.
- dmi_: Copy of tlm_generic_payload dmi field.
- byte_enable_: Copy of tlm_generic_payload byte enable field.
- byte_enable_length_: Copy of tlm_generic_payload byte enable length field.
- streaming_width_: Copy of tlm_generic_payload streaming width field.
- gp_option_: Copy of tlm_generic_payload gp option field.
- tlm_phase_: tlm_generic_payload phase when transaction was probed.
- snoop_phase_: Timing phase at which transaction was probed.
- status_: Status response captured at probing point.

Not all fields will be relevant for all assertion and probing possibilities. For the pointer values, library users are responsible of checking that the pointer is still valid when the evaluation is asserted.

4.2.2 Prop

This class name is short for property. Its responsibility is to implement the boolean layer of the assertion logic.

This class has two fields:

- probe_: This string indicates the point in the model hierarchy to which the property applies.
- eval_: This lambda function takes a Tlm object, and returns a PropResult.

Every time an assertion is evaluated, the Tlm object propagates down to Prop, which returns PASS if the property holds, FAIL if it doesn't, and SKIP if the Tlm object is irrelevant to the property.

4.2.3 Asrt

This class name is short for assertion. Its responsibility is to implement the temporal layer of the assertion logic.

Its main fields are as follows:

- name_ and id_: These fields are used to uniquely identify each assertion. The name is a mnemonic provided by the user, while id is a unique numerical identifier generated by the kernel.
- start_prop_ptr_: This field holds a pointer to a Prop object. When this property evaluates to true, the assertion moves from ASRT_DISABLED to ASRT_ENABLED.
- stop_prop_ptr_: This field holds a pointer to a Prop object. When this property evaluates to true, the assertion moves from ASRT_ENABLED to ASRT_EXPIRED.
- eval_prop_ptr_: This field holds a pointer to a prop object. When this property evaluates to true the assertion holds.
- state_: This field holds the assertion's current state. By default it is initialized to the ASRT_DISABLED state.
- spawn_next_asrt_: This boolean field indicates whether the current assertion's property has held. If true, the kernel will spawn the next assertion atom during its cleanup stage.
- spawn_count_: Indicates the number of times the assertion's property has held. If 0 at the end of the assertion's lifetime, the assertion evaluates accordingly. If it's any other value, the assertion atom doesn't evaluate at all.
- is_leaf_: Indicates whether the current assertion is the last in an assertion chain.
- next_asrt_: This field holds a pointer to the next Asrt object in a chain.
- history_: This field holds a list of all the TLMs that were evaluated in previous assertion atoms. This is what allows comparisons on the temporal layer.
- criteria_: This field indicates what the pass criteria is for the current assertion.

For every monitored transaction, the assertion logic parses de property result and determines the resulting effect on the current assertion.

The states in an assertion progress according to the diagram in figure 6

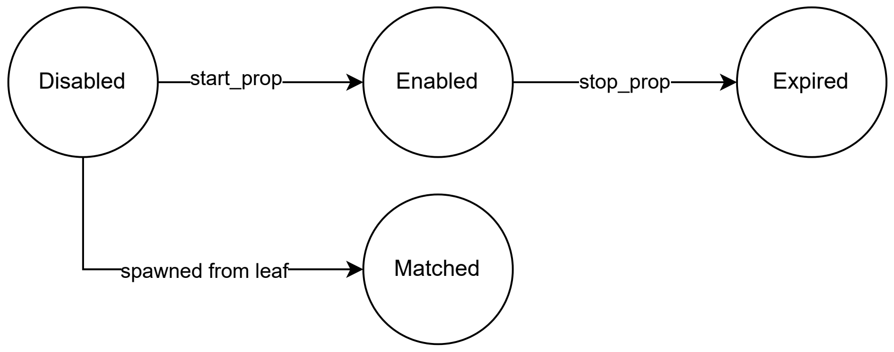


Figure 6: Assertion state logic

Three methods of this class must be highlighted:

- Main: This method receives the snooped TLM from the kernel, and evaluates the related properties.
- GenNextAsrt: This method returns the next assertion in an assertion chain. In the case of a leaf, it returns a copy of the current assertion with the ASRT_MATCHED state.
- Passes(): This method returns whether the assertion passes or fails, based on the current status and pass criteria.

4.2.4 Kernel

The kernel is a singleton class, responsible for maintaining the list of assertions and forwarding to each the transaction as it is captured from the model. Each transaction defined by the user is registered with the kernel.

Its fields are as follows:

- active_asrts_: This is the list of active assertions. Each time a transaction is monitored, its contents get forwarded to every transaction in this list for processing.
- inactive_asrts_: Assertions are moved to this list when their state becomes ASRT_EXPIRED or ASRT_MATCHED. This list is regularly purged throughout the simulation to reduce memory usage.
- next_asrts_: This is the list of recently spawned assertions. Whenever a transaction in a chain is triggered, the kernel uses its template to generate the next assertion in the chain and places it in this list. At the end of each monitoring event, the contents of this list are moved to the active_asrts_ list.
- next_id: This field keeps track of the next identification number that will be assigned to a newly registered assertion.
- snoop_count_: This field tracks the number of times transactions have been snooped.

- eval_count_: This field tracks the number of evaluated assertions for reporting purposes.
- pass_count_: This fields tracks the number of assertions that have passed for reporting purposes.
- fail_count_: this field tracks the number of assertions that have failed for reporting purposes.
- print_asrt_: This field controls whether a message is printed when an assertion passes or fails. In ordinary regressions, no message is printed, just a final report with the number of passes and fails. When a test fails, the test can be rerun with this field enabled to aid debugging.

At the end of the test, the kernel generates a report based on the contents of the lists it maintains. The following methods of this class must be highlighted:

- Snoop(B|NB)Transport[Fw|Bw](Bgn|End): These methods generate a TLM object from each snooped transactions and forward it to every assertion in the active_asrt list.
- Cleanup: This method is called after every snoop, its purpose is to update the active_asrts_, inactive_asrts_ and next_asrts_ lists, and determine whether any assertion is currently strongly holding or not holding.
- RunEot: This method is called at the end of simulation to cleanup any remaining active assertions, which are determined to be softly holding.

4.3 Usage guidelines

Applying the library to a model can be summarized in the following steps:

1. The user instruments the design with the provided macros on the points were observability is required.
2. The user defines the properties that define the behavior they expect to observe.
3. The user defines the assertions using the above properties, and chains them together to capture design intent.
4. The user registers these assertions with the kernel.
5. The user executes some interesting stimulus on the kernel and analyses the resulting report.

4.4 Unit and integration testing

For unit and integration testing the open-source library doctest [47] was used.

This allowed setting up tests for the functions of each class, validating the implementation before trying it out in a real model.

Besides per-class tests, integration tests were developed that verified that all parts of the library worked together as expected.

4.5 Reporting

For reporting and printing the library spdlog [48] was used.

This allows the SCAsrt library to print reports into a dedicated file, without polluting the normal output of the model itself.

5 Solution Validation

Now that the SCAsrt architecture is defined, and its implementation created, the next task is to determine whether the design objectives were achieved. This chapter will focus on the validation results for the solution correctness. That is, whether the library as designed and coded can be used to express properties of a design with a similar level of power as provided by PSL, and whether these properties are evaluated correctly in a simulation environment.

To perform this validation, two methodologies were applied. The following subsections will discuss the results.

5.1 Unit Testing

First, unit tests are used to validate all the internal methods used by the defined classes. The unit tests instance the defined classes, call their methods, and perform checks on the return values and internal fields for consistency.

In total, all of the library classes, their fields, and their internal methods are validated by 24 separate tests, with a total of 237 checks.

This effort assures us that the internal logic of the library is consistent, and its behavior is as expected by the architectural plan.

The list of tests is as follows:

- ‘Scasrt::Prop::Constructor’
- ‘Scasrt::Prop::Skip’
- ‘Scasrt::Prop::Pass’
- ‘Scasrt::Prop::Fail’
- ‘Scasrt::Prop::History’
- ‘Scasrt::Asrt::Leaf constructor’
- ‘Scasrt::Asrt::Non-leaf constructor’
- ‘Scasrt::Asrt::SetId’
- ‘Scasrt::Asrt::Name’
- ‘Scasrt::Asrt::GenNextAsrt’
- ‘Scasrt::Asrt::Main’
- ‘Scasrt::Asrt::GetStatus"

- ‘Scasrt::Asrt::GetCriteria’
- ‘Scasrt::Asrt::Passes’
- ‘Scasrt::Asrt::ToString’
- ‘Scasrt::Kernel::GetInstance’
- ‘Scasrt::Kernel::RegisterAsrt’
- ‘Scasrt::Kernel::Cleanup’
- ‘Scasrt::Kernel::SnoopBTransportBgn’
- ‘Scasrt::Kernel::SnoopBTransportEnd’
- ‘Scasrt::Kernel::SnoopNbTransportFwBgn’
- ‘Scasrt::Kernel::SnoopNbTransportFwEnd’
- ‘Scasrt::Kernel::SnoopNbTransportBwBgn’
- ‘Scasrt::Kernel::SnoopNbTransportBwEnd’

The above test suite achieved coverage of all the main conditions under which the SCAsrt library is expected to be used. All the tests are passing.

5.2 Comparison to existing implementations

To validate that the architectural design of SCAsrt is capable of reproducing the capabilities of PSL, an experimental comparison test was designed and executed.

In this evaluation, an assertion is programmed in both SCAsrt and SVA (System Verilog Assertion Language). Synthetic stimulus is generated and fed to both assertions, and the output of each simulation is compared. If the output of the assertions is the same given an equal input, then it can be said that SCAsrt is implementing the associated functionality correctly. Stimulus is generated to obtain both positive and negative results, to further ensure correctness.

The testbench is structured as seen in figure 7.

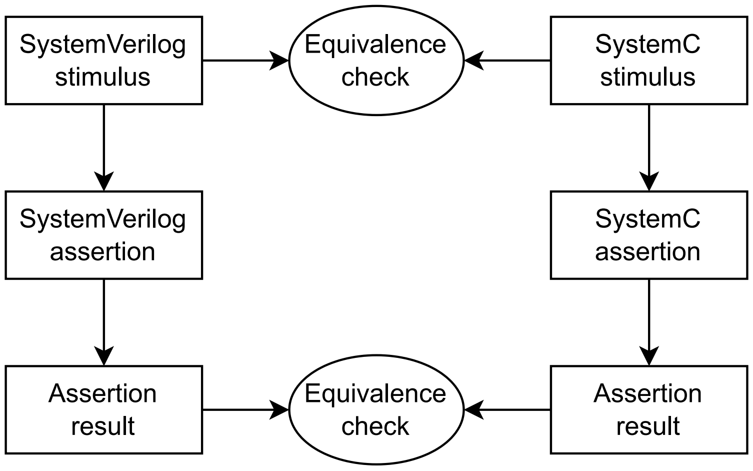


Figure 7: Testbench for SV equivalence validation

The equivalence checks are performed via textual comparisons of print messages in both the stimulus input and assertion output.

The following list shows the tested scenarios and their functionality.

- Concatenation: This scenario tests the capability to create a sequential connection of events. In PSL, the concatenation operator is ';', while in SVA the operator is '##'. The property layer uses implication to connect one event to the next with the operators '|=>' and '|->'.
- Rose: This scenario tests the ability to detect whether the signal rose from the last clock cycle to the current one. PSL and SVA use the operator '$rose()'.
- Fell: This scenario tests whether the signal rose from the last clock cycle to the current one. PSL and SVA use the operator '$fell()'.
- Stable: This scenario tests whether the signal is unchanged from the last clock cycle to the current one. PSL and SVA use the operator '$stable()'.
- Prev: This scenario tests the capability to read the signal value from previous clock cycles. PSL uses the operator '$prev()' and SVA uses the operator '$past()'.
- Intersect: This scenario tests the ability to detect two parallel sequences finishing on the same cycle. SVA uses de operator 'intersect'.
- Throughout: This scenario tests whether a property holds throughout the duration of given sequence. SVA uses de operator 'throughout'.
- Within: This scenario tests if a sequence starts and ends within the duration of another sequence. SVA uses de operator 'within'.

This results in sixteen total tests, 8 for the passing scenarios and 8 for the failing scenarios. All of the so defined tests determined that the SVA and SCAsrt produced equivalent results for an equivalent input, with the exception of the within scenario.

The following sections will further expound on the created scenarios, including the within test which failed the textual comparison.

5.2.1 Concatenation results and analysis

In the concatenation test, the assertion in this thesis's notation is depicted in figure 8, and in SVA in figure 9

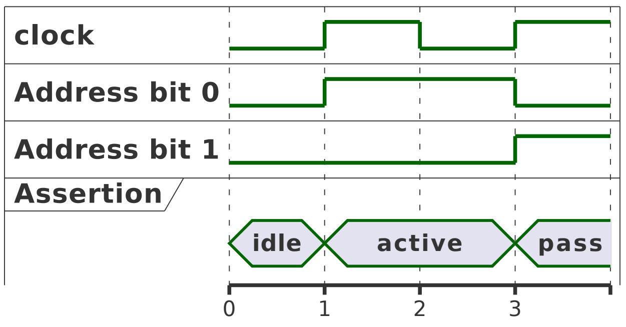
$$[\text{true}, \text{false}](\text{addr} == 1) \rightarrow [\text{true}, \text{false}](\text{addr} == 2) \text{ (holds strongly)}$$

Figure 8: Concatenation assertion in thesis notation

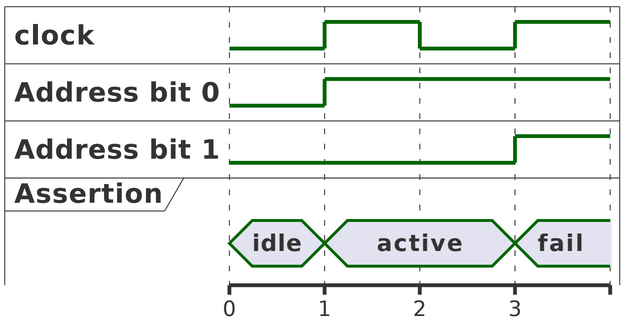
```
property asrt_1;
  @(posedge clk) (addr == 1) |=> (addr == 2);
endproperty : asrt_1;
assert property(asrt_1) $fdisplay(fd, $sformatf("PASS"));
else  $fdisplay(fd, $sformatf("FAIL"));
```

Figure 9: Concatenation assertion in SVA

The stimulus for this test can be seen in figure 10



(a) Passing



(b) Failing

Figure 10: Concatenation test stimulus

This test passed both pass and fail scenarios, since the textual output between SVA and SCAsrt was the same.

5.2.2 Rose results and analysis

In the rose test, the assertion in this thesis's notation is depicted in figure 11, and in SVA in figure 12

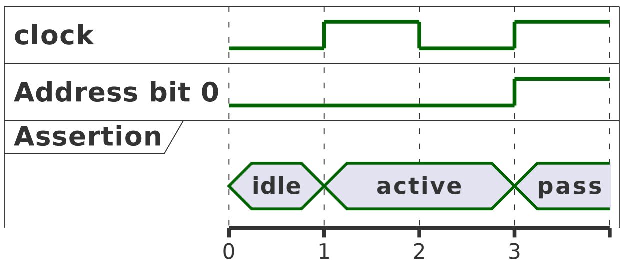
$$[\text{true}, \text{false}](\text{addr} == 0) \rightarrow [\text{true}, \text{false}](\text{addr} == 1 \&\& \text{prev\_addr} == 0) \text{ (holds strongly)}$$

Figure 11: Rose assertion

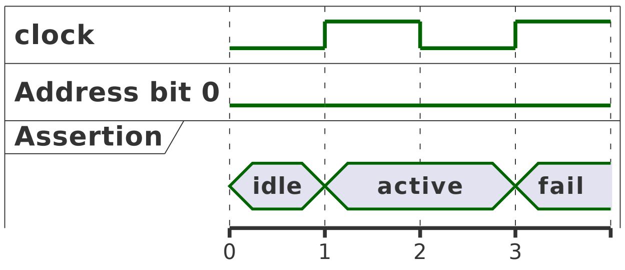
```
property asrt_1;
  @(posedge clk) (addr == 0) |=> $rose(addr[0:0]); //$
endproperty : asrt_1;
assert property(asrt_1) $fdisplay(fd, $sformatf("PASS"));
else  $fdisplay(fd, $sformatf("FAIL"));
```

Figure 12: Rose assertion in SVA

The stimulus for this test can be seen in figure 13



(a) Passing



(b) Failing

Figure 13: Rose test stimulus

This test passed both pass and fail scenarios, since the textual output between SVA and SCAsrt was the same.

5.2.3 Intersect results and analysis

In the intersect test, the assertion in this thesis's notation is depicted in figure 14, and in SVA in figure 15

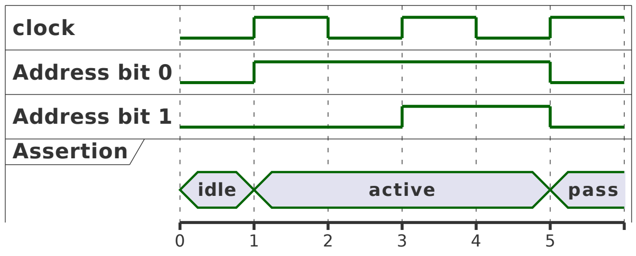
$$[\text{true}, \text{false}](\text{addr} == 1) \to [\text{true}, \text{false}](\text{addr} == 3) \to [\text{true}, \text{false}](\text{addr} == 0)\ (\text{holds strongly})$$

Figure 14: Intersect assertion in thesis notation

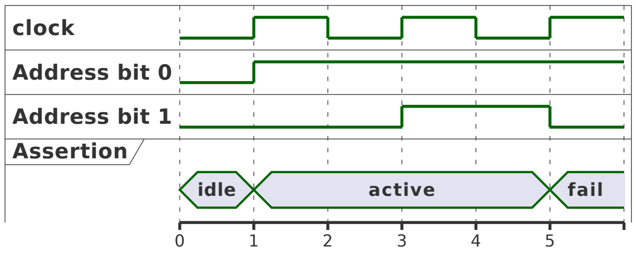
```
property asrt_1;
  @(posedge clk)
  addr[1:1] == 1 |-> ((addr[1:1] == 1 ##1 addr[1:1] == 0)
    intersect (addr[0:0] == 1 ##1 addr[0:0] == 0));
endproperty : asrt_1;
assert property(asrt_1) $fdisplay(fd, $sformatf("PASS"));
else  $fdisplay(fd, $sformatf("FAIL"));
```

Figure 15: Intersect assertion in SVA

The stimulus for this test can be seen in figure 16



(a) Passing



(b) Failing

Figure 16: Intersect test stimulus

This test passed both pass and fail scenarios, since the textual output between SVA and SCAsrt was the same.

5.2.4 Within results and analysis

In the within test, the assertion in this thesis's notation is depicted in figure 17, and in SVA in figure 18

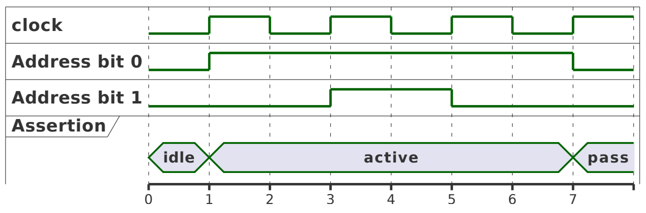
$$[(\text{addr}[0{:}0] == 1), \text{false}](\text{addr}[1{:}1] == 1) \rightarrow [\text{true}, (\text{addr}[0{:}0] == 0)](\text{addr}[1{:}1] == 0) \text{ (holds strongly)}$$

Figure 17: Within assertion in thesis notation

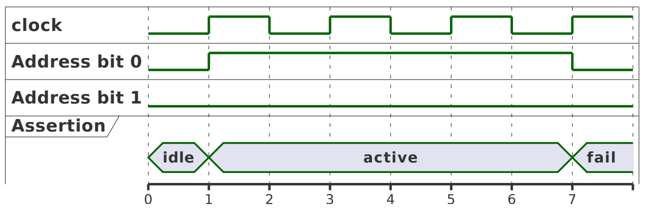
```
property asrt_1;
  @(posedge clk)
  addr == 1 |=> (addr[1:1] == 1 ##1 addr[1:1] == 0)
    within (addr[0:0] == 1 ##2 addr[0:0] == 0);
endproperty : asrt_1;
assert property(asrt_1) $fdisplay(fd, $sformatf("PASS"));
else  $fdisplay(fd, $sformatf("FAIL"));
```

Figure 18: Within assertion in SVA

The stimulus for this test can be seen in figure 19



(a) Passing



(b) Failing

Figure 19: Within test stimulus

This test failed the textual comparison scenario due to an implementation detail difference between the SVA and SCAsrt. In the passing scenario, the PASS message is printed before the last stimulus message in SCAsrt, while in SVA the last stimulus message is printed first. In the failing scenario the opposite happens, where the FAIL message is printed before the last stimulus message in SVA, while in SCAsrt the last stimulus message is printed before the FAIL message.

This shows that the implementations are not exactly equivalent, an area where SCAsrt may be improved.

6 Proof of concept application and simulation performance impact

6.1 SystemC model

To perform a proof of concept validation exercise, the RISC-V based Virtual Prototype was selected.

This is a SystemC TLM 2.0 LT model, that provides multiple components that can be connected together to model a complete RISC-V system. This model is maintained by the University of Bremen, and has been cited by over 12 papers (such as [49], [50] and [51]), published in venues like JSA (Journal of Systems Architecture), ATVA (Symposium on Automated Technology for Verification and Analysis), FDL (Forum on Specification & Design Languages), DAC (Design Automation Conference), and DATE (Design, Automation and Test in Europe Conference).

Its main components are a 32-bit and a 64-bit RISC-V core, a configurable generic memory bus, a Core Local Interrupt Controller (CLINT), a Platform-Level Interrupt Controller (PLIC), and multiple peripherals such as timers, displays, sensors, ethernet and Direct Memory Access (DMA) controllers.

6.2 System architecture

The system architecture used for the proof of concept has the structure show in figure 20.

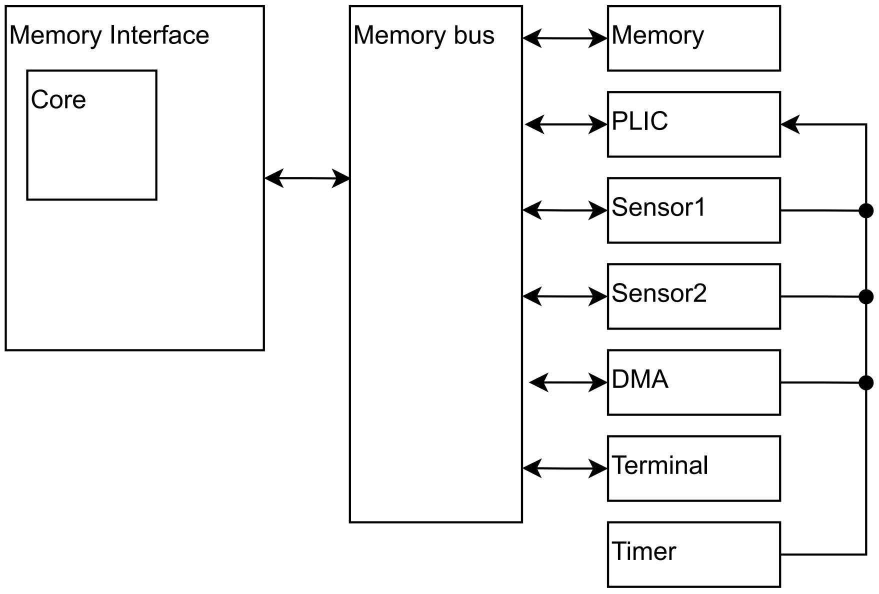


Figure 20: System architecture

The components of the system are as follows:

- Core: This is the RISC-V core, simulates instruction execution.
- Memory interface: The memory interface is the connection between the core and the memory bus.
- Memory bus: The memory bus holds the memory map, and is responsible for correctly routing memory operations to the different peripherals.

- Memory: Main system memory, can be read and written by the core and peripherals.
- PLIC: Platform-level interrupt controller, receives interrupts from devices and routes them to the core.
- Sensor 1 and Sensor 2: These are models of any possible sensing device. These periodically generate interrupts when data is sensed. They have scale and filter control registers.
- DMA: Direct memory access controller, can be used to offload memory operations from the core. It has registers to control the start and end address, length and type of operations.
- Terminal: This is a write only device. It does not generate any interrupts, and any bytes written to it are displayed as characters.
- Timer: This basic timer generates an interrupt every millisecond. It has not configuration registers.

6.3 Validation strategy

The SCAsrt proof of concept will validate two areas of the system.

6.3.1 Memory map address translation and transaction routing

The memory map will be validated with assertions that check the correct propagation of memory operations. The assertions will compare the input of the memory interface against the input of each peripheral, confirming that every transaction is delivered to the correct peripheral with the correct address translation.

The assertion template used here can be seen in figure 21.

$$\begin{gathered}[\text{true}, \text{false}](\text{addr} >= \text{device\_start\_addr} \ \&\& \ \text{addr} < \text{device\_end\_addr}) \rightarrow \\ [\text{true}, \text{main}](\text{previous\_addr} == \text{addr} + \text{device\_start\_addr})(\text{holds strongly})\end{gathered}$$

Figure 21: Memory map assertion

This assertion template was replicated for each device connected to the memory bus, resulting in 6 total assertions. An example of its operation can be seen in figure 22, were a transaction directed to the memory device is checked.

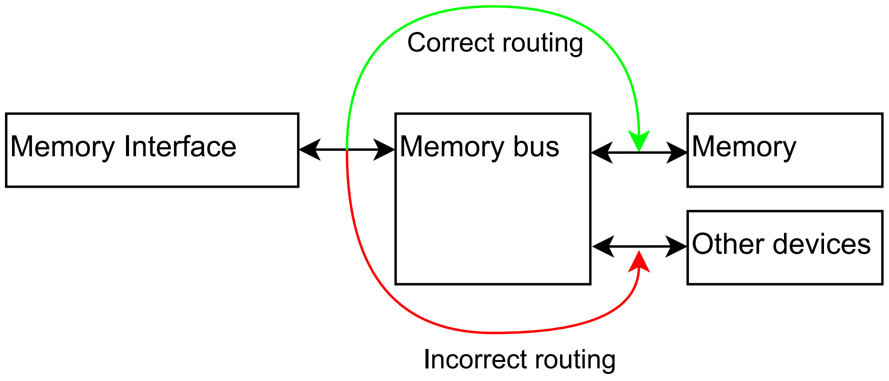


Figure 22: Memory map check

6.3.2 Interrupt propagation

The PLIC connection between devices will be validated for interrupts coming from the connected devices, namely the sensors, DMA and timer.

For the DMA, two assertions were created to validate correct interrupt propagation during normal operation, and to validate no spurious interrupt generation when the DMA is idle.

The first assertion (figure 23 and figure 24) checks for the conditions that enable the DMA engine. If those conditions are met, then it expects a single interrupt to eventually arrive in the PLIC. If the interrupt arrives, the assertion passes, if it doesn't, the assertion fails.

$$[\text{true}, \text{false}](\text{addr} == \text{dma\_start\_register} \ \&\& \ \text{data} == 1) \rightarrow [\text{true}, \text{main}](\text{addr} == \text{PLIC\_interrupt\_register} \ \&\& \ \text{data} == \text{dma\_int\_id})(\text{holds strongly})$$

Figure 23: Assertion for DMA enabled

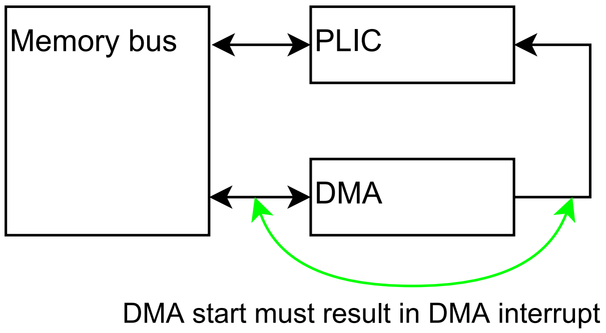


Figure 24: DMA enabled interrupt propagation

The second assertion (figure 25 and figure 26) checks for conditions that enable the DMA engine. If those conditions are met, the assertions ends. If a DMA interrupt arrives while this assertion is active, the assertion fails.

$$[\text{true}, (\text{addr} == \text{dma\_start\_register} \ \&\& \ \text{data} == 1)](\text{addr} == \text{PLIC\_interrupt\_register} \ \&\& \ \text{data} == \text{dma\_int\_id})(\text{doesn't hold strongly})$$

Figure 25: Assertion for DMA disabled

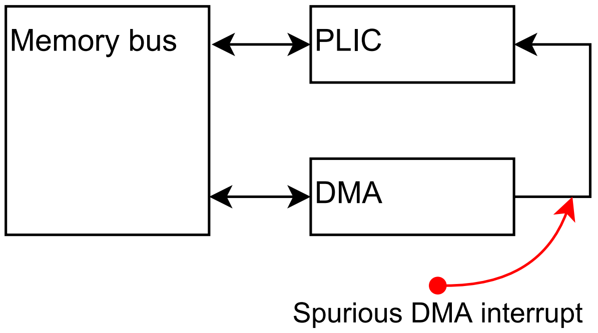


Figure 26: Spurious DMA event

For the sensors, one assertion was created (figure 27 and figure 28). This assertion checks for the sensor configuration sequence, and after it is detected, it expects at least one interrupt to arrive. Every time an interrupt arrives from the sensor, the assertion passes.

$$[\text{true}, \text{false}](\text{addr} == \text{sensor\_scale\_register} \ \&\& \ \text{data} \neq 0) \rightarrow [\text{true}, \text{main}](\text{addr} == \text{sensor\_filter\_register} \ \&\& \ \text{data} \neq 0) \rightarrow [\text{true}, \text{false}](\text{addr} == \text{PLIC\_interrupt\_register} \ \&\& \ \text{data} == \text{sensor\_int\_id})(\text{holds strongly})$$

Figure 27: Assertion for sensor enabled

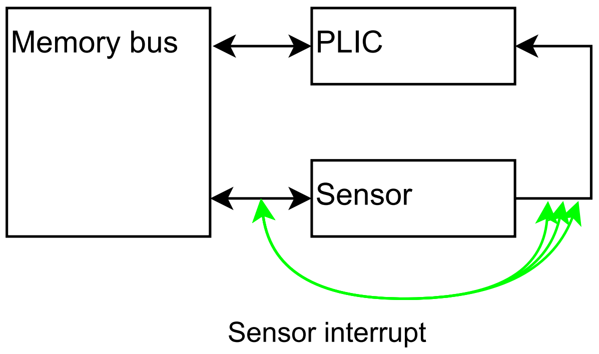


Figure 28: Sensor assertion

For the timer, one assertion was created (figure 29 and figure 30). This assertion passes every time an interrupt from the timer arrives, since there is no mechanism to disable the timer.

$$[\text{true}, \text{false}](\text{addr} == \text{PLIC\_interrupt\_register} \ \&\& \ \text{data} == \text{timer\_int\_id})(\text{holds strongly})$$

Figure 29: Assertion for timer enabled

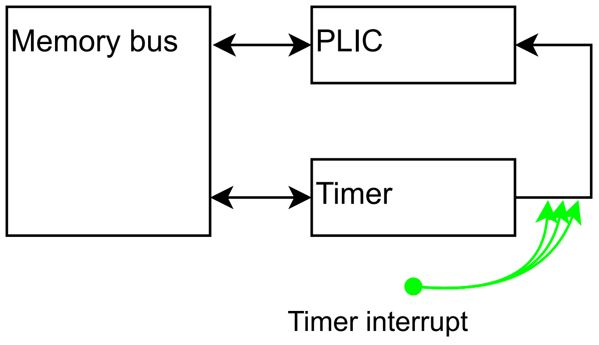


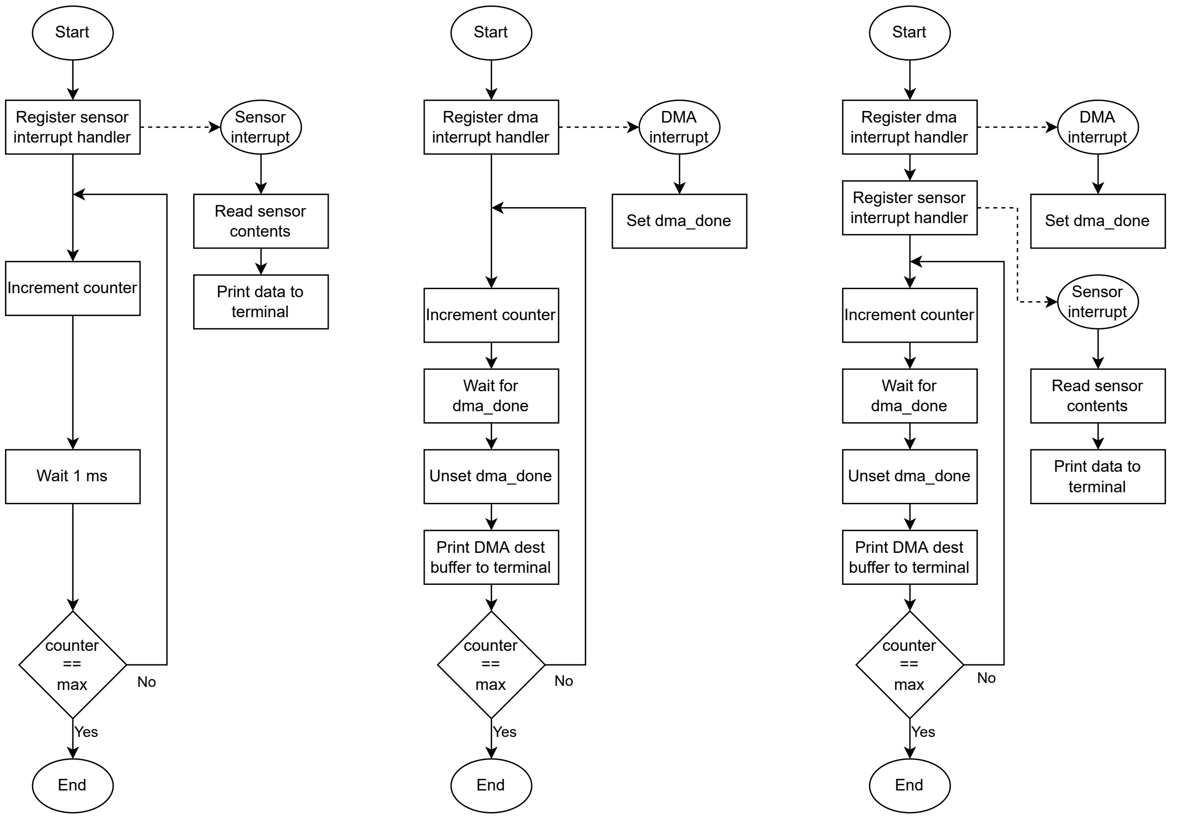
Figure 30: Timer interrupt

In total, five interrupts validate the connectivity between devices and the PLIC.

6.4 Stimulus scenarios

Three stimulus scenarios were simulated using the model and related assertions.

- Simple sensor: In this test, an interrupt handler is registered to respond to sensor interrupts and print the sensor data to the terminal. The main program flow can be seen in figure 31a.
- Basic DMA: In this test, a basic dma copy operation is performed between two regions in the local memory. The main program flow can be seen in figure 31b.
- Performance: This test is a combination of the above. One interrupt handler is registered per available sensor, and the DMA is configured to repeatedly execute operations. The main program flow can be seen in figure 31c.



(a) Simple sensor program (b) Basic DMA program (c) Performance program

Figure 31: Stimulus scenarios

These tests were run multiple times each to collect all the validation and performance results.

6.5 Validation results

6.5.1 Simple sensor test

The validation simple sensor test ran for 10 iterations, and the following results were obtained:

For the memory bus validation assertions, the DMA and second sensor assertions held softly once. This is expected as this test does not have any traffic to these devices. The local memory, PLIC, first sensor and terminal devices assertions held strongly for all transactions observed in the test.

For the interrupt validation assertions, the first sensor assertion held strongly exactly ten times, just as expected. The timer assertion held strongly fifty times, as expected given the duration of the test. The spurious DMA assertion ended the test softly holding, just as expected given the stimulus generated.

When an error was injected in the model, breaking the memory map and the interrupt connections, the assertions failed as expected.

6.5.2 Basic DMA test

The validation basic DMA test ran for 10 iterations, with the following results obtained:

For the memory bus validation assertions, the sensor assertions held only once, as no traffic was directed to them. The local memory, PLIC, terminal and DMA device assertions did hold strongly for the duration of the test, triggering hundreds of times.

For the interrupt validation, the DMA propagation assertion was correctly triggered 10 times, all holding strongly. The spurious DMA interrupt assertion was correctly disabled when the DMA is configured.

When an error was injected in the model, breaking the memory map and the interrupt connections, the assertions failed as expected.

6.6 Performance results

For performance measurements the performance stimulus scenario was utilized. The scenario was run with a variable number of iterations, and the execution times and memory utilization information was collected.

Each scenario was run three times:

- The original model, without any of the SCAsrt components.
- The model instrumented with assertions, in regression mode, where only a final report is generated on total passing and failing assertions.
- The model instrumented with assertions, in debug mode, where a message is printed each time an assertion passes or fails.

The test was then executed repeatedly with a variable amount of transactions. Data was collected until clear trends in execution time were observed, and trying to execute even more transactions resulted in prohibitively long execution times. In total, the test was run 47 times, with ten thousand transactions in the shortest test and two hundred million transactions in the longest test. These amounts of transactions correspond to five milliseconds of simulated time for the least amount of transactions, and one hundred seconds of simulated time for the most amount of transactions. In wallclock time, the shortest test took less than a second, and the longest test took almost fifty minutes.

The execution time results can be seen in figure 32.

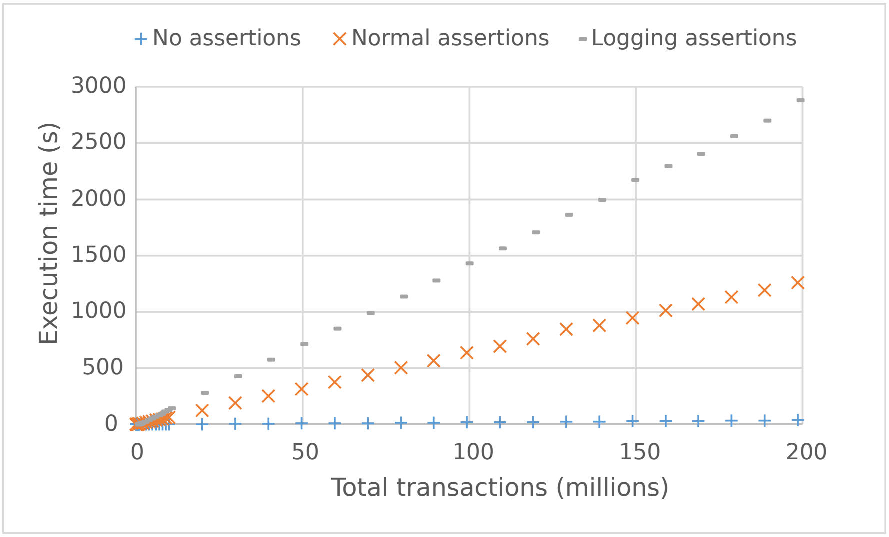


Figure 32: Execution time

As can be seen in the plot, well-constructed transactions impose a fixed overhead per transaction, resulting in a linear increase of total simulation time. On average, each transaction takes 30 times as long to be processed in the normal mode compared to the baseline, and each transaction takes 70 times as long to be processed in the logging mode compared to the baseline.

The memory usage results can be seen in figure 33.

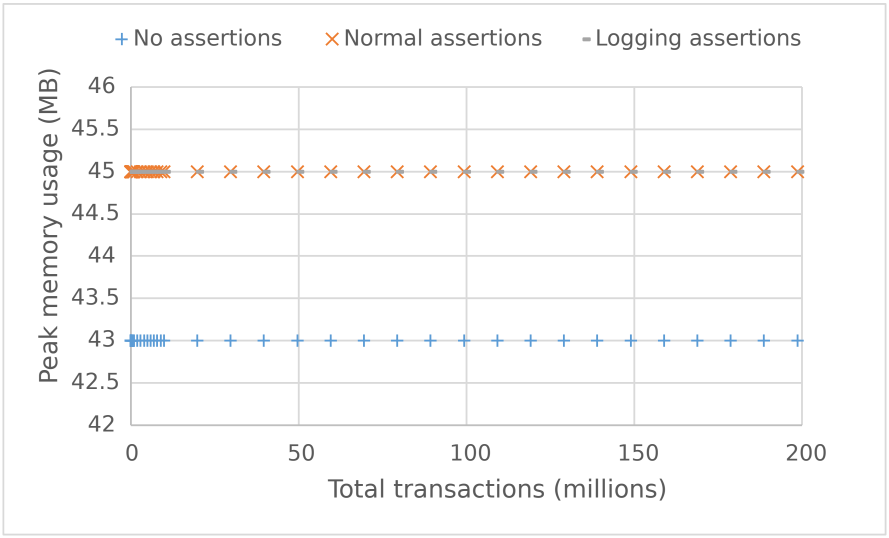


Figure 33: Memory usage

The plot shows that SCAsrt consumes a fixed amount of memory over what the baseline model demands. As long as created assertions are well behaved, the memory usage is constant and does not impose a limit on possible applications of this library. Logging has no impact on the amount of memory consumed by SCAsrt.

6.6.1 Performance impact of degenerate assertions

An inexperienced user of the library may inadvertently create assertions that consume vast computational resources. For example, in the assertion shown in figure 34, every time an assertion is snooped a new assertion chain will be launched that does not terminate.

$$[\text{true}, \text{false}](\text{true}) \rightarrow [\text{true}, \text{false}](\text{false})(\text{holds strongly})$$

Figure 34: Assertion that never completes and multiplies itself

This active assertion list will quickly expand to consume all memory and available CPU cycles. The library doesn't stop this behavior because in some situations such assertions may be desired.

The results of adding the above assertion to a performance test run may be observed in figures 35 and 36.

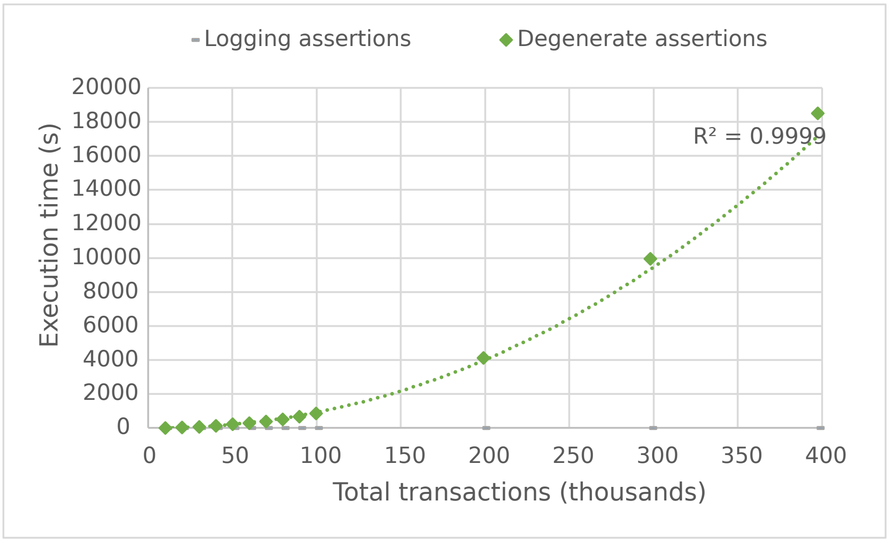


Figure 35: Execution time with degenerate assertion

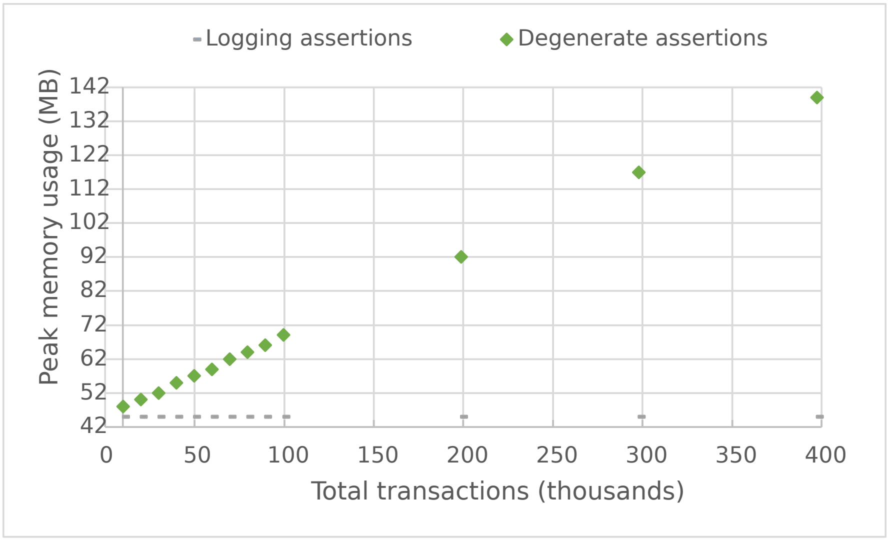


Figure 36: Memory usage with degenerate assertion

The execution time plot shows that with a degenerate assertion, execution times goes from having a linear dependence on the number of transactions to having a polynomial time dependence. The memory usage plot shows that memory usage goes from having a constant overhead to having a linear overhead in relation to the number of transactions. However running out of memory is not a concern since the execution times grows so much faster than the memory consumption.

6.6.2 Profiler results and analysis

GProf [52] was used to profile the model execution. This tool instruments the compiled binary to collect statistical usage data of the functions and methods that compose a program.

In the baseline model, the top three time-consuming functions were sc_core::sc_time::sc_time(), sc_core::sc_event::trigger() and std::shared_ptr_access::M_get().

In comparison, in the regression model the above calls are outside the top fifty. The top time consuming functions in this case are gnu_cxx::exchange_and_add(), Sca::Asrt::Main(), Sca::Kernel::Cleanup(), and std::shared_ptr<Sca::Asrt>::get().

For the logging model, most of the time was consumed by functions related to printing to buffer.

These profiling results show that most of cycles consumes by SCAsrt are related to the reference counting types and heap allocation subroutines.

7 Conclusions and Recommendations

The state of the art investigation revealed the need in the industry for high level assertion validation solutions. The work done before in this field is pretty extensive, but the niche of on-line assertion validation in a native language remained unaddressed.

The development of SCAsrt met its stated objectives.

- SCAsrt is capable of expressing time based relationships between transactions, and evaluating their validity during the course of a simulation.
- When an assertion expectation is violated, the error is detected the moment it happens, at which point the simulation may be terminated. This saves simulation cycles that may be wasted reproducing a model that is behaving outside normal expectations.
- Memory utilization: SCAsrt has highly optimized memory usage. When an assertion outcome is known, a report is generated and the associated memory freed.
- Storage utilization: The on-line processing provided by SCAsrt means that very little logging is produced. It is not necessary to capture a trace of the entire simulation. Rather, a report with only the relevant pass/fail assertions is produced.
- The proof of concept application of this library successfully validated features in a popular model used across the industry.
- Performance testing revealed that careless assertion definition by users may greatly increase simulation time. Memory usage is also affected, but as it increases linearly while simulation time increases exponentially, memory exhaustion is unlikely to happen before the execution time makes simulation impractical.

7.1 Future work

- The proof of concept validation effort revealed that the verification layer in SCAsrt has room for improvement and expansion. Declaring that a chain of assertions should **PASS_ON_HOLD**, **PASS_ON_HOLD_STRONG**, **PASS_ON_NOTHOLD**, and **PASS_ON_NOTHOLD_STRONG** does not provide the verifier with enough granular control of trace evaluation. The current solution falls short when differentiating cases where the library user may want an assertion to fail or pass depending on which assertion atom is still active at the end of test.

- The performance results of SCAsrt point to areas where it may be greatly improved. Currently, assertion lists use shared pointer types, which imposes a costly execution time penalty when propagating events from the kernel down to the assertions. This cost is caused by the use of atomic operations on pointer copies. With a careful analysis of the implementation, most of the shared pointers can be replaced with unique pointer and move semantics, which should reduce the performance overhead.

- Another area where execution performance can be improved is in heap memory management. Currently, an object of the internal Tlm type is allocated from the heap every time a transaction is snooped. The repeated small allocations could be avoided by using a custom memory management class, which allocates large pools of memory at once and distributes them during execution.

References

[1] "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, Jan. 2012, Conference Name: IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005). DOI: 10.1109/IEEESTD.2012.6134619.

[2] "IEEE Standard for Property Specification Language (PSL)," *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pp. 1–182, Apr. 2010, Conference Name: IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005). DOI: 10.1109/IEEESTD.2010.5446004.

[3] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, Feb. 2018, Conference Name: IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012). DOI: 10.1109/IEEESTD.2018.8299595.

[4] "IEEE Standard VHDL Language Reference Manual," *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pp. 1–640, Jan. 2009, Conference Name: IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002). DOI: 10.1109/IEEESTD.2009.4772740.

[5] H. Foster, *Prologue: The 2020 Wilson Research Group Functional Verification Study*, en-US, Oct. 2020. [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2020/10/27/prologue-the-2020-wilson-research-group-functional-verification-study/ (visited on 04/14/2022).

[6] R. Waxman, J.-M. Bergé, O. Levia, and J. Rouillard, *High-level System Modeling*. Kluwer Academic Publishers, 1996, ISBN: 978-1-4612-8561-8.

[7] C. N. Coelho and H. D. Foster, "Assertion-Based Verification," en, in *Advanced Formal Verification*, R. Drechsler, Ed., Boston, MA: Springer US, 2004, pp. 167–204, ISBN: 978-1-4020-2530-3. DOI: 10.1007/1-4020-2530-0_5. [Online]. Available: https://doi.org/10.1007/1-4020-2530-0_5 (visited on 04/14/2022).

[8] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, ISSN: 0272-5428, Oct. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[9] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," en, in *Logics of Programs*, D. Kozen, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1982, pp. 52–71, ISBN: 978-3-540-39047-3. DOI: 10.1007/BFb0025774.

[10] B. Lin, "Automated Test Generation for Validating SystemC Designs," en, Ph.D. dissertation, Portland State University, Jan. 2021. [Online]. Available: https://www.proquest.com/openview/0485b638c3a2c51bd8a27e4c6e5ea5c6/1?pq-origsite=gscholar&cbl=18750&diss=y (visited on 07/08/2021).

[11] B. Lin, K. Cong, Z. Yang, *et al.*, "Concolic testing of SystemC designs," in *2018 19th International Symposium on Quality Electronic Design (ISQED)*, Mar. 2018, pp. 1–7. DOI: 10.1109/ISQED.2018.8357256.

[12] B. Lin, Z. Yang, K. Cong, and F. Xie, "Generating high coverage tests for SystemC designs using symbolic execution," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, ISSN: 2153-697X, Jan. 2016, pp. 166–171. DOI: 10.1109/ASPDAC.2016.7428006.

[13] M. Goli and R. Drechsler, "Scalable Simulation-Based Verification of SystemC-Based Virtual Prototypes," in *2019 22nd Euromicro Conference on Digital System Design (DSD)*, Aug. 2019, pp. 522–529. DOI: 10.1109/DSD.2019.00081.

[14] L. Pierre and M. Chabot, "Assertion-Based Verification for SoC Models and Identification of Key Events," in *2017 Euromicro Conference on Digital System Design (DSD)*, Aug. 2017, pp. 54–61. DOI: 10.1109/DSD.2017.75.

[15] L. Pierre and L. Ferro, "Enhancing the assertion-based verification of TLM designs with reentrancy," in *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, Jul. 2010, pp. 103–112. DOI: 10.1109/MEMCOD.2010.5558642.

[16] ——, "A Tractable and Fast Method for Monitoring SystemC TLM Specifications," *IEEE Transactions on Computers*, vol. 57, no. 10, pp. 1346–1356, Oct. 2008, Conference Name: IEEE Transactions on Computers, ISSN: 1557-9956. DOI: 10.1109/TC.2008.74.

[17] M. Goli, J. Stoppe, and R. Drechsler, "Automatic Protocol Compliance Checking of SystemC TLM-2.0 Simulation Behavior Using Timed Automata," in *2017 IEEE International Conference on Computer Design (ICCD)*, ISSN: 1063-6404, Nov. 2017, pp. 377–384. DOI: 10.1109/ICCD.2017.65.

[18] ——, "Automatic equivalence checking for SystemC-TLM 2.0 models against their formal specifications," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, ISSN: 1558-1101, Mar. 2017, pp. 630–633. DOI: 10.23919/DATE.2017.7927064.

[19] N. Bombieri, F. Fummi, V. Guarnieri, *et al.*, "Reusing RTL Assertion Checkers for Verification of SystemC TLM Models," en, *Journal of Electronic Testing*, vol. 31, no. 2, pp. 167–180, Apr. 2015, ISSN: 1573-0727. DOI: 10.1007/s10836-015-5514-8. [Online]. Available: https://doi.org/10.1007/s10836-015-5514-8 (visited on 07/08/2021).

[20] N. Bombieri, R. Filippozzi, G. Pravadelli, and F. Stefanni, "RTL property abstraction for TLM assertion-based verification," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, ISSN: 1558-1101, Mar. 2015, pp. 85–90. DOI: 10.7873/DATE.2015.0121.

[21] N. Bombieri, F. Fummi, V. Guarnieri, *et al.*, "On the reuse of RTL assertions in SystemC TLM verification," in *2014 15th Latin American Test Workshop - LATW*, ISSN: 2373-0862, Mar. 2014, pp. 1–6. DOI: 10.1109/LATW.2014.6841903.

[22] S. Dutta and M. Y. Vardi, "Assertion-based flow monitoring of SystemC models," in *2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE)*, Oct. 2014, pp. 145–154. DOI: 10.1109/MEMCOD.2014.6961853.

[23] M. Chen and P. Mishra, "Assertion-Based Functional Consistency Checking between TLM and RTL Models," in *2013 26th International Conference on VLSI Design and 2013 12th International Conference on Embedded Systems*, ISSN: 2380-6923, Jan. 2013, pp. 320–325. DOI: 10.1109/VLSID.2013.208.

[24] D. Tabakov, "Chimp: A tool for assertion-based dynamic verification of systemc models," *CEUR Workshop Proceedings*, vol. 1130, pp. 38–45, Oct. 2013.

[25] D. Tabakov, K. Y. Rozier, and M. Y. Vardi, "Optimized temporal monitors for SystemC," en, *Formal Methods in System Design*, vol. 41, no. 3, pp. 236–268, Dec. 2012, ISSN: 1572-8102. DOI: 10.1007/s10703-011-0139-8. [Online]. Available: https://doi.org/10.1007/s10703-011-0139-8 (visited on 07/07/2021).

[26] D. Tabakov, "Dynamic assertion-based verification for systemc," ISBN-13: 9781124776521, phd, Rice University, USA, 2010.

[27] H. M. Le, D. Große, and R. Drechsler, "Scalable fault localization for SystemC TLM designs," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, ISSN: 1530-1591, Mar. 2013, pp. 35–38. DOI: 10.7873/DATE.2013.022.

[28] H. M. Le, D. Große, V. Herdt, and R. Drechsler, "Verifying SystemC using an intermediate verification language and symbolic simulation," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13, New York, NY, USA: Association for Computing Machinery, May 2013, pp. 1–6, ISBN: 978-1-4503-2071-9. DOI: 10.1145/2463209.2488877. [Online]. Available: https://doi.org/10.1145/2463209.2488877 (visited on 03/21/2022).

[29] H. M. Le, D. Grosse, and R. Drechsler, "Automatic TLM Fault Localization for SystemC," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 8, pp. 1249–1262, Aug. 2012, Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, ISSN: 1937-4151. DOI: 10.1109/TCAD.2012.2188800.

[30] H. M. Le, D. Große, and R. Drechsler, "Automatic Fault Localization for SystemC TLM Designs," in *2010 11th International Workshop on Microprocessor Test and Verification*, ISSN: 2332-5674, Dec. 2010, pp. 35–40. DOI: 10.1109/MTV.2010.15.

[31] V. Jain, A. Kumar, and P. Panda, "Exploiting UML based validation for compliance checking of TLM 2 based models," en, *Design Automation for Embedded Systems*, vol. 16, no. 2, pp. 93–113, Jun. 2012, ISSN: 1572-8080. DOI: 10.1007/s10617-012-9089-7. [Online]. Available: https://doi.org/10.1007/s10617-012-9089-7 (visited on 07/08/2021).

[32] M. Kallel, Y. Lahbib, A. Baganne, and R. Tourki, "Monitoring transaction level SystemC models using a generic and aspect-oriented framework," *International Journal of Computer Aided Engineering and Technology*, vol. 4, no. 3, pp. 229–249, Jan. 2012, Publisher: Inderscience Publishers, ISSN: 1757-2657. DOI: 10.1504/IJCAET.2012.046635. [Online]. Available: https://www.inderscienceonline.com/doi/abs/10.1504/IJCAET.2012.046635 (visited on 07/08/2021).

[33] M. Kallel, Y. Lahbib, R. Tourki, and A. Baganne, "Verification of SystemC transaction level models using an aspect-oriented and generic approach," in *5th International Conference on Design Technology of Integrated Systems in Nanoscale Era*, Mar. 2010, pp. 1–6. DOI: 10.1109/DTIS.2010.5487605.

[34] A. Sen, "Concurrency-oriented verification and coverage of system-level designs," *ACM Transactions on Design Automation of Electronic Systems*, vol. 16, no. 4, 37:1–37:25, Oct. 2011, ISSN: 1084-4309. DOI: 10.1145/2003695.2003697. [Online]. Available: https://doi.org/10.1145/2003695.2003697 (visited on 07/08/2021).

[35] M. Bawadekji, D. Große, and R. Drechsler, "TLM protocol compliance checking at the Electronic System Level," in *14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, Apr. 2011, pp. 435–440. DOI: 10.1109/DDECS.2011.5783132.

[36] Z. Xiong, J. Bian, and Y. Zhao, "An assertion-based verification method for SystemC TLM," in *2010 International Conference on Communications, Circuits and Systems (ICCCAS)*, Jul. 2010, pp. 842–846. DOI: 10.1109/ICCCAS.2010.5581859.

[37] L. Ferro and L. Pierre, "Formal semantics for PSL modeling layer and application to the verification of transactional models," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, ISSN: 1558-1101, Mar. 2010, pp. 1207–1212. DOI: 10.1109/DATE.2010.5456991.

[38] ——, "ISIS: Runtime verification of TLM platforms," in *2009 Forum on Specification Design Languages (FDL)*, ISSN: 1636-9874, Sep. 2009, pp. 1–6.

[39] L. Ferro, L. Pierre, Y. Ledru, and L. du Bousquet, "Generation of test programs for the assertion-based verification of TLM models," in *2008 3rd International Design and Test Workshop*, ISSN: 2162-061X, Dec. 2008, pp. 237–242. DOI: 10.1109/IDT.2008.4802505.

[40] D. Große, H. M. Le, and R. Drechsler, "Proving transaction and system-level properties of untimed SystemC TLM designs," in *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, Jul. 2010, pp. 113–122. DOI: 10.1109/MEMCOD.2010.5558643.

[41] D. Grosse, H. M. Le, and R. Drechsler, "Induction-Based Formal Verification of SystemC TLM Designs," in *2009 10th International Workshop on Microprocessor Test and Verification*, ISSN: 2332-5674, Dec. 2009, pp. 101–106. DOI: 10.1109/MTV.2009.16.

[42] K. Tomasena, J. Sevillano, J. Perez, A. Cortes, and I. Velez, "A Transaction Level Assertion Verification Framework in SystemC: An Application Study," in *2009 Second International Conference on Advances in Circuits, Electronics and Micro-electronics*, Oct. 2009, pp. 75–80. DOI: 10.1109/CENICS.2009.24.

[43] W. Ecker, V. Esen, T. Steininger, M. Velten, and M. Hull, "Implementation of a Transaction Level Assertion Framework in SystemC," in *Automation Test in Europe Conference Exhibition 2007 Design*, ISSN: 1558-1101, Apr. 2007, pp. 1–6. DOI: 10.1109/DATE.2007.364406.

[44] A. Habibi and S. Tahar, "Design and verification of SystemC transaction-level models," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 1, pp. 57–68, Jan. 2006, Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, ISSN: 1557-9999. DOI: 10.1109/TVLSI.2005.863187.

[45] B. Lin and F. Xie, "A Systematic Investigation of State-of-the-Art SystemC Verification," *Journal of Circuits, Systems and Computers*, vol. 29, no. 15, p. 2 030 013, Dec. 2020, Publisher: World Scientific Publishing Co., ISSN: 0218-1266. DOI: 10.1142/S0218126620300135. [Online]. Available: https://www.worldscientific.com/doi/10.1142/S0218126620300135 (visited on 07/01/2021).

[46] G. De Giacomo and M. Y. Vardi, "Linear temporal logic and linear dynamic logic on finite traces," in *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, ser. IJCAI '13, Beijing, China: AAAI Press, Aug. 2013, pp. 854–860, ISBN: 978-1-57735-633-2. (visited on 07/07/2021).

[47] *Doctest/doctest*, original-date: 2014-08-05T21:43:51Z, Apr. 2022. [Online]. Available: https://github.com/doctest/doctest (visited on 04/16/2022).

[48] G. Melman, *Spdlog*, original-date: 2014-11-01T01:28:53Z, Apr. 2022. [Online]. Available: https://github.com/gabime/spdlog (visited on 04/16/2022).

[49] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Early Concolic Testing of Embedded Binaries with Virtual Prototypes: A RISC-V Case Study," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, ISSN: 0738-100X, Jun. 2019, pp. 1–6.

[50] ——, "Verifying Instruction Set Simulators using Coverage-guided Fuzzing," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, ISSN: 1558-1101, Mar. 2019, pp. 360–365. DOI: 10.23919/DATE.2019.8714912.

[51] ——, "Extensible and Configurable RISC-V Based Virtual Prototype," in *2018 Forum on Specification & Design Languages (FDL)*, ISSN: 1636-9874, Sep. 2018, pp. 5–16. DOI: 10.1109/FDL.2018.8524047.

[52] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *ACM SIGPLAN Notices*, vol. 17, no. 6, pp. 120–126, Jun. 1982, ISSN: 0362-1340. DOI: 10.1145/872726.806987. [Online]. Available: https://doi.org/10.1145/872726.806987 (visited on 08/02/2022).

[53] J. Pacheco, *Jeosadn/scasrt*, original-date: 2022-07-25T02:05:07Z, Aug. 2022. [Online]. Available: https://github.com/jeosadn/scasrt (visited on 08/07/2022).

[54] ——, *SCAsrt library POC application*, original-date: 2022-08-07T16:53:45Z, Aug. 2022. [Online]. Available: https://github.com/jeosadn/scasrt_poc (visited on 08/07/2022).

8 Appendixes

8.1 SCAsrt testing guide

To get the SCAsrt library source code, you may clone the repository from its public GitHub page at SCAsrt [53].

To build its dependencies: SystemC, doctest and spdlog, execute the command:

```
make deps
```

To run the unit test suite, execute the command:

```
make
```

To run the SVA validation tests, execute the command:

```
./run_validation.sh
```

Note that you must have a valid VCS installation to run the validation tests.

8.2 SCAsrt usage guide

This section describes the steps needed apply the SCAsrt library to a SystemC model, using the proof of concept results as a reference.

This Proof of Concept (POC) application is available at SCAsrt_POC [54].

The changes are as follows:

- Update your compilation scripts to include the SCAsrt library. In the case of this POC, this meant an update to CMakeLists.txt:

  ```
  diff --git c/vp/CMakeLists.txt w/vp/CMakeLists.txt
  index 1725d43..e0b13b3 100644
  --- c/vp/CMakeLists.txt
  +++ w/vp/CMakeLists.txt
  @@ -12,6 +12,8 @@ set(CMAKE_CXX_FLAGS "-Wall -Wextra -Wno-unused-parameter") #
   ↪  TODO: -Wpedantic
   set(CMAKE_CXX_FLAGS_DEBUG "-g3")        #"-fsanitize=address
    ↪  -fno-omit-frame-pointer"
   set(CMAKE_CXX_FLAGS_RELEASE "-O3")

  +add_compile_options(-I/scasrt/external -I/scasrt/inc -I/scasrt/src)
  +
   # Allows running tests without invoking `make install` first.
   set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/bin") \\$
  ```

- Update the model components to snoop the transaction flow. This means that for every module that calls the blocking or non-blocking transport functions, an import to the 'Probe.h' header file must be added and the transport function call should be replaced by the snooping macro.

  - Memory device:

```
diff --git c/vp/src/core/rv32/mem.h w/vp/src/core/rv32/mem.h
index 1ef645d..e317b17 100644
--- c/vp/src/core/rv32/mem.h
+++ w/vp/src/core/rv32/mem.h
@@ -4,6 +4,8 @@
 #include "iss.h"
 #include "mmu.h"

+#include "Probe.h"
+
 namespace rv32 {

 /* For optimization, use DMI to fetch instructions */
@@ -60,7 +62,7 @@ struct CombinedMemoryInterface : public
 ↪  sc_core::sc_module,

                sc_core::sc_time local_delay =
                 ↪  quantum_keeper.get_local_time();

-               isock->b_transport(trans, local_delay);
+               SNOOP_B_TRANSPORT(isock, trans, local_delay);

                assert(local_delay >= quantum_keeper.get_local_time());
                quantum_keeper.set(local_delay);
```

– DMA device:

```
diff --git c/vp/src/platform/basic/dma.h w/vp/src/platform/basic/dma.h
index 7d137ae..aa9b10c 100644
--- c/vp/src/platform/basic/dma.h
+++ w/vp/src/platform/basic/dma.h
@@ -9,6 +9,8 @@
 #include <unordered_map>
 #include <array>

+#include "Probe.h"
+
 struct SimpleDMA : public sc_core::sc_module {
        tlm_utils::simple_initiator_socket<SimpleDMA> isock;
        tlm_utils::simple_target_socket<SimpleDMA> tsock;
@@ -151,7 +153,7 @@ struct SimpleDMA : public sc_core::sc_module {
                trans.set_data_ptr(data);
                trans.set_data_length(num_bytes);

-               isock->b_transport(trans, delay);
+               SNOOP_B_TRANSPORT(isock, trans, delay);

                if (delay != sc_core::SC_ZERO_TIME)
```

```
                        sc_core::wait(delay);
```

- Memory bus:

```
diff --git c/vp/src/platform/common/bus.h w/vp/src/platform/common/bus.h
index 7cf9a46..49431a0 100644
--- c/vp/src/platform/common/bus.h
+++ w/vp/src/platform/common/bus.h
@@ -8,6 +8,8 @@
 #include <tlm_utils/simple_target_socket.h>
 #include <systemc>

+#include "Probe.h"
+
 struct PortMapping {
        uint64_t start;
        uint64_t end;
@@ -57,7 +59,7 @@ struct SimpleBus : sc_core::sc_module {
                }

                trans.set_address(ports[id]->global_to_local(addr));
-               isocks[id]->b_transport(trans, delay);
+               SNOOP_B_TRANSPORT(isocks[id], trans, delay);
        }

        unsigned transport_dbg(tlm::tlm_generic_payload &trans) {
@@ -92,7 +94,7 @@ struct PeripheralWriteConnector : sc_core::sc_module {
        void transport(tlm::tlm_generic_payload &trans, sc_core::sc_time
         ↪  &delay) {
                bus_lock->wait_until_unlocked();

-               isock->b_transport(trans, delay);
+               SNOOP_B_TRANSPORT(isock, trans, delay);

                if (trans.get_response_status() ==
                 ↪  tlm::TLM_ADDRESS_ERROR_RESPONSE)
                        throw std::runtime_error("unable to find target
                         ↪  port for address " +
                         ↪  std::to_string(trans.get_address()));
```

- Update the model's main component to set up the kernel before running the simulation:

```
diff --git c/vp/src/platform/basic/main.cpp w/vp/src/platform/basic/main.cpp
index b9fb70e..3ba19d1 100644
--- c/vp/src/platform/basic/main.cpp
+++ w/vp/src/platform/basic/main.cpp
@@ -108,7 +108,27 @@ std::ostream& operator<<(std::ostream& os, const
 ↪  BasicOptions& o) {
        return os;
 }

+#include "spdlog/sinks/basic_file_sink.h"
+#include "spdlog/spdlog.h"
+
+#include "Tlm.cpp"
+#include "Asrt.cpp"
+#include "Kernel.cpp"
+#include "Prop.cpp"
+#include "Types.cpp"
+
+auto sca_log = spdlog::basic_logger_mt("sca_log", "sca.log", true);
+auto sca_test_log = spdlog::basic_logger_mt("sca_test_log", "sca_test.log",
 ↪  true);
+
 int sc_main(int argc, char **argv) {
+  spdlog::get("sca_log")->set_pattern(
+      "[%-12!n] [%L] [%-10!s] [%-15!!] [%-3!#] %v");
+  spdlog::get("sca_test_log")
+      ->set_pattern("[%-12!n] [%L] [%-10!s] [               ] [%-3!#] %v");
+  SPDLOG_LOGGER_INFO(spdlog::get("sca_test_log"), "Running on
 ↪  platform/basic/main.cpp");
+
+  Scasrt::Kernel::GetInstance()->reset();
+
        BasicOptions opt;
        opt.parse(argc, argv);

@@ -225,6 +245,8 @@ int sc_main(int argc, char **argv) {

        sc_core::sc_start();

+  Scasrt::Kernel::GetInstance()->RunEot();
+
        core.show();

        if (opt.test_signature != "") {
```

- Finally, assertions may be registered with the kernel at any point before the simulation starts via the 'sc_core::sc_start()' function invocation. Additionally, after the simulation completes the user should call the kernel end of test function: 'Scasrt::Kernel::GetInstance()->RunEot()'.

  An assertion atom implementation requires 4 objects to be instanced: three instances of the 'Scasrt::Prop' class for the start, stop and evaluation properties, and one instance of the 'Scasrt::Asrt' class.

  The 'Scasrt::Prop' constructor takes two arguments: a string representing the point in the hierarchy where the transactions will be snooped, and a lambda function that will perform the evaluation.

The ‘Scasrt::Asrt’ constructor takes four arguments: 3 ‘Scasrt::Prop’ pointers (they may point to the same object instance), and either a pass or fail criteria in the case of a leaf assertion, or another assertion instance, in the case of a non-leaf assertion.

The first assertion in a chain is the only one that needs to be registered to the kernel, via a call to ‘Scasrt::Kernel::GetInstance()->RegisterAsrt()’.

```
if (1) { // MEM interface to DMA mapping assertion
  // A section
  auto mem_to_dma_strt_prop_ptr =
   ↪  std::make_shared<Scasrt::Prop>(MEM_INTF_PROBE, L_PROP { return
   ↪  Scasrt::PROP_TRUE; });
  auto mem_to_dma_stop_prop_ptr =
   ↪  std::make_shared<Scasrt::Prop>(MEM_INTF_PROBE, L_PROP { return
   ↪  Scasrt::PROP_FALSE; });
  auto mem_to_dma_eval_prop_ptr =
   ↪  std::make_shared<Scasrt::Prop>(MEM_INTF_PROBE, L_PROP {
    if (tlm.snoop_phase_ != Scasrt::B_TRANSPORT_BGN) {
      return Scasrt::PROP_SKIP;
    }

    if (tlm.address_ >= 0x70000000 && tlm.address_ < 0x70001000) { //
     ↪  dma_start_addr
      return Scasrt::PROP_TRUE;
    } else {
      return Scasrt::PROP_FALSE;
    }
  });

  // B section
  auto dma_to_mem_strt_prop_ptr = std::make_shared<Scasrt::Prop>(DMA_PROBE,
   ↪  L_PROP { return Scasrt::PROP_TRUE; });
  auto dma_to_mem_eval_prop_ptr = std::make_shared<Scasrt::Prop>(DMA_PROBE,
   ↪  L_PROP {
    if (tlm.snoop_phase_ != Scasrt::B_TRANSPORT_END || tlm.response_status_
     ↪  != tlm::TLM_OK_RESPONSE) {
      return Scasrt::PROP_SKIP;
    }

    if (history[0]->address_ == tlm.address_ + 0x70000000) { //
     ↪  dma_start_addr
      return Scasrt::PROP_TRUE;
    } else {
      return Scasrt::PROP_FALSE;
    }
  });
```

```
    // Assembly
    auto mem_intf_to_dma_asrt_B =
     ↪  std::make_shared<Scasrt::Asrt>("mem_intf_to_dma_asrt_B",
     ↪  dma_to_mem_eval_prop_ptr, dma_to_mem_strt_prop_ptr,
     ↪  dma_to_mem_eval_prop_ptr, Scasrt::ASRT_PASS_ON_HOLD_STRONG);
    auto mem_intf_to_dma_asrt_A =
     ↪  std::make_shared<Scasrt::Asrt>("mem_intf_to_dma_asrt_A",
     ↪  mem_to_dma_eval_prop_ptr, mem_to_dma_strt_prop_ptr,
     ↪  mem_to_dma_stop_prop_ptr, mem_intf_to_dma_asrt_B);
    Scasrt::Kernel::GetInstance()->RegisterAsrt(mem_intf_to_dma_asrt_A);
  }
```