2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
DECEMBER 6 - 7, 2022

A Cross-domain Heterogeneous ABV-Library for Mixed-signal Virtual Prototypes in SystemC/AMS*

Muhammad Hassan, DFKI GmbH, Bremen, Germany
(muhammad.hassan@dfki.de)
Thilo Vörtler, Karsten Einwich, COSEDA Technologies GmbH, Dresden, Germany
(thilo.voertler|karsten.einwich@coseda-tech.com)
Rolf Drechsler, University of Bremen & DFKI GmbH, Bremen, Germany
(drechsler@uni-bremen.de)
Daniel Große, Johannes Kepler University, Linz, Austria & DFKI GmbH Bremen, Germany
(daniel.grosse@jku.at)

***Abstract*—In this paper we discuss how assertions can be applied for the design and verification of mixed-signal virtual prototypes based on SystemC/AMS [1][2]. In particular, we propose a SystemC assertions library with a user-friendly API which enables specification of complex mixed-signal behaviors. We show that the assertions library is designed to handle heterogeneous systems containing mixed-signal characteristics, transactions, and software. Furthermore, we explain how assertions can be easily and intuitively used to specify the DUV behavior across multiple domains. Using ARM Fast Models-based temperature control system we show how our proposed SystemC assertions library can be used to verify the DUV.**

Keywords—SystemC/AMS; assertions; mixed-signal; temoporal; TLM; HW/SW co-design

I. INTRODUCTION

Heterogeneous systems have significantly altered the requirements of modern *System-on-Chips* (SOCs). Consequently, many semiconductor vendors are shifting their focus towards a more integrated solution of high-performance *Analog/Mixed-Signal* (AMS) design with *Software* (SW) running on top. In this regard, *Virtual Prototypes* (VPs) implemented using SystemC/AMS together with *Transaction Level Modeling* (TLM) are widely used for Hardware (HW)/SW co-design [3-5] [15-22]. The AMS part of SystemC allows it to model analog behavior through different *Models of Computation* (MOC) like *Timed Data Flow* (TDF) and *Electrical Linear Networks* (ELN).

Presently, digital designs are functionally verified with the help of SystemVerilog [9] using *Assertion Based Verification* (ABV) in combination with coverage analysis [1] and constrained randomization techniques [7][8]. ABV defines temporal properties to verify the functional correctness of the design with respect to expected behaviors. Several approaches have also been proposed for SystemC/AMS-based models/VPs [10-14]. However, they either run into state-space explosion problem or do not consider mixed-signal interactions, e.g., SW, TLM, and analog. Hence, one of the main challenges is the availability of a practical and publicly available assertions library for SystemC/AMS design verification which enables ABV methodologies. A SystemC/AMS assertions library should satisfy the following:

1. Expressiveness to represent complex behaviors.
2. SystemC, TLM, and SystemC/AMS compatibility.
3. Complex cross-domain interactions capturing, i.e., analog-digital-SW interactions.
4. Integration of complex heterogeneous characteristics like continuous time, frequency analysis etc.

In this paper we present a SystemC assertions library which satisfies the aforementioned features. The library provides an intuitive *Application Programming Interface* (API) for assertions specification and follows *SystemVerilog Assertions* (SVA) library closely. Furthermore, it supports SystemC/AMS, TLM, and continuous time evaluation. Additionally, the library enables capturing of complex behaviors, as well as HW/SW and TLM interactions.

* This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project AUTOASSERT under contract no. 16ME0117 and by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

Hence, the SystemC assertions library enables ABV for heterogeneous systems. The experiments on a real-world temperature control system model using ARM V8 based CPU (ARM Fast Models) demonstrates the capabilities of the library to improve the system verification in a significant way. Please note that the SystemC assertions library is still a work-in-progress. The API and some use cases are described in this paper.

II. Assertion Library Overview

For SystemC there is no assertion library in public, which has an expressiveness compared to SVA. A first effort was made in [14] to make assertions available for SystemC that can be translated to SVA, however only a small subset of SVA is supported, especially complex timing sequences are not possible. Also, the focus is set on synthesizable assertions.

One of the main challenges when implementing assertions for SystemC is that the assertions are bound to the restrictions of the C++ language. We choose to implement our language as a standard C++ library based on SystemC/AMS and TLM. In addition to the challenge of designing an easy understandable API for assertions, SystemC/AMS covers a different level of abstraction as SystemVerilog. SystemC/AMS allows to describe heterogeneous characteristics like continuous time, frequency analysis, slopes, and equation systems. Such systems often lack the notion of a clock. The usage of simulation events is at the moment insufficiently integrated in known specification languages. Therefore, our SystemC assertions library considers all these conditions for bridging the gap of ABV for heterogeneous designs. The SystemC assertions library is developed with an intuitive, user-friendly, and an expressive API. It is not intended to use any formal methods at the back-end rather it focuses on the simulations only.

The library itself follows the principles of SVA extending it to heterogenous system use-cases. It is imperative to mention that to enable the API expressiveness, a layered architecture inline with SVA layered architecture is used, i.e., boolean layer, sequence layer, property layer, and verification layer. At the back-end, first the assertion is divided into different layers and expressions, then multiple SystemC processes are spawned to monitor the signals and events specified in the expressions. Each assertion is synchronized to simulator calculation points (notion of discretized time) of DUV as defined by SystemC/AMS semantics. Assertions are evaluated over time based on an event tick (clock edge or repeated time). If the specified expressions evaluate to *true*, the assertion is satisfied. The library is developed in pure C++ without external library dependencies and is compatible with the IEEE SystemC/AMS standards. It is simulator agnostic, hence, it can be directly used in any SystemC project.

III. Assertion Library Feautures

A. Introductionary example

A complete example assertion using our SystemC assertions library is shown in Figure 1. The assertion checks the behavior of two SystemC discrete event signals *sig_a* and *sig_b* (Line 1-2). It says after *sig_a* is true within 1 or 2 clock cycles *sig_b* has to be smaller than 10. To use signals in the assertion language, they must be wrapped in an *expr* object (see also Section III.B) corresponding to their type (Line 4-5). In Line 7 a temporal sequence *seq* is defined that describes that the *sig_b* has to be smaller than value 10 after a delay of one/two clock cycles. The

```
1.  sc_core::sc_signal<bool> sig_a("a"), clk("clk");
2.  sc_core::sc_signal<int> sig_b("b");
3.
4.  expr<decltype(sig_a)> expr_a (sig_a);
5.  expr<decltype(sig_b)> expr_b (sig_b);
6.
7.  sequence seq = true | delay(1,2) | expr_b < 10;
8.  property example_prop = expr_a ->* seq;
9.  example_prop.default_sampling(clk.posedge_event());
10. ASSERT_PROPERTY (example_prop);
```

Figure 1: Assertion library example

final property, which is checked is an implication that says if sig_*a* is *true* the sequence *seq* has to hold (Line 8). To run an assertion a default sampling event can be specified, this states when the assertion is started and how to interpret delays in the delay operator. To actually schedule assertion the macro ASSERT_PROPERTY is used. In the following sections the different building blocks of the assertion library are described.

B. Boolean layer

The boolean layer of the SystemC assertions library describes the atomic behavior of signals, events, and variables. The signals are related to each other using various operators, e.g., arithmetic operators. The SystemC assertions library supports all SystemC/AMS datatypes. To use them within an assertion they must be wrapped within objects of type *expr*. A non-comprehensive list of supported boolean operators is shown in Table 1. The *expr* objects are combined using the boolean operators to make an expression which returns a value that must be comparable. The *expr* class implements thereby the access to the value of the wrapped object e.g., it calls the *read ()* function of a *sc_port*. Additionally, it supports C++ *lambda* expressions to define complex functionality inside an assertion. The Lambda expression can be defined as shown in Figure 2. In this example the lambda returns an integer by calling the arbitrary function. This expression is always triggered when the assertion gets evaluated at its corresponding time point

```
1. expr<std::function<int(void)>> lambda { []() {
2.         return calc_average();
3. } };
4. auto expr_lambda = lambda < 3.0;
5. ASSERT_PROPERTY (expr_lambda);
```

Figure 2: Lambda expression operator example

Table 1: Non-comprehensive list of supported boolean expressions by SystemC assertions library

Operator	Name
+= -= /= *= &= |=	Binary assignment operators
< <= > >=	Binary relational operators
+ - ∗ =	Binary arithmetic operators
&& || == !=	Binary logical operators
+ - ! ++ –	Unary operators

C. Sequence layer

Following the layered architecture of SVA, the sequence layer builds on top of boolean layer to specify the temporal relationship between boolean expressions. Furthermore, the sequence layer also specifies sequences as either a combination of simpler sequences using sequence operators or as basic boolean expressions over events. The SystemC assertions library provides several operators like *delay ()* and *repeat ()* to specify the behavior over time. A non-comprehensive list of supported sequence operators is shown in Table 2. To chain expressions within a sequence in C++ a special operator is required. We use therefore the pipe operator (|) to represent the continuity of a sequence.

As the sequence layer builds on top of boolean layer, it evaluates the boolean expressions at evaluation time points by the SystemC events. This can either be a default event described by a SystemC event, but also a simulator event specified using a SystemC time.

Table 2 Non-comprehensive list of supported sequence operators by SystemC assertions library

Operator	Description
delay	Specifies delay from current sampling point until the next
and	Sequence *and* operation
or	Sequence *or* operation
repeat	Repetition operator
|	Sequence continue operator

Delay Operator The SystemC assertions library introduces *delay* operator as shown below

```
1. // delay (delay_cycles)
2. sequence seq1 = expr_a | delay (3) | expr_b
3. //delay (min_delay_cycles, max_delay_cycles)
4. sequence seq2 = expr_a | delay (sc_time(1, SC_MS), sc_time (3, SC_MS)) | expr_b
```

The function of *delay* operator is to create a relationship between boolean expressions over a period of time or between the given time constraints. The sequence *seq1* in above figure (Line 2) defines a sequence where *expr_b* is evaluated *3 default sampling events* after *expr_a*. Similarly, sequence *seq2* (Line 4) defines a sequence where the delay between two expressions can be between 1 and 3 milliseconds.

Repeat Operator The library supports *repeat* operator which is used to specify a consecutive repetition of the left-hand side operand as shown below

```
1. // repeat (value)
2. sequence seqr1 = expr_a | delay (3) | expr_b | repeat (2)
3. //repeat (min_value, max_value)
4. sequence seqr2 = expr_a | delay (3) | expr_b | repeat (1,3)
```

It helps in cases when a certain set of expressions are expected to be true over multiple time points. E.g., seqr1 in (Line 2) defines a sequence which is repeated twice, i.e., seq1 is equivalent to

expr_a | delay (3) | expr_b | delay(1) | expr_a | delay (3) | expr_b

D. Property layer

The property layer allows for more general behaviors to be specified, i.e., specification of properties as either a combination of simpler properties using property operators or as an implication built up from several sequences. The properties and their respective sequences (including boolean expressions) are evaluated on each sampling event of the system's default sampling time, unless specified. A non-comprehensive list of supported property operators is shown in Table 3.

Implication Operator An implication refers to a scenario in which in order for a behavior to occur, a preceding sequence must have occurred. This preceding sequence in this case is known as *antecedent*. The succeeding behavior is known as *consequent*. Evaluation of an implication starts through repeated attempts to evaluate the antecedent. When the antecedent succeeds, the consequent is required to succeed for the property to hold. It is defined as

antecedent - > ∗ consequent
where - > ∗ = overlapping implication operator

```
1. expr expr1 = a_int < 42;
2. sequence expr2 = true | delay(1,2) | expr_a + expr_b;
3. property non_overlap = expr1 ->* expr2;
4. ASSERT_PROPERTY (non_overlap);
```

Please note, currently the SystemC assertions library does not support the non-overlapping operator unlike SVA. However, the API can be leveraged to create non-overlapping operator's functionality, e.g., it can be built using delay (…) operator.

Overlapping Implication In the assertions library we introduce an overlapping implication operator *(->*)*. This means that if the antecedent sequence is evaluated to *true*, the consequent sequence is evaluated at the same time point. An example of assertion using overlapping implication operator is shown in above example.

Table 3 Non-comprehensive list of supported property operators.

Operator	Description
not	the evaluation of the property returns the opposite of the evaluation of the underlying property expression
and	The property evaluates to true if, and only if, both property expression 1 and property expression 2 evaluate to true.
or	The property evaluates to true if, and only if, at least one of property expression 1 and property expression 2 evaluates to true

E. User functions

The API of SystemC assertions library provides assertion evaluation function so that the defined assertion can be evaluated at run-time. A macro *ASSERT_PROPERTY (…)* is used in this regard as shown below:

```
1. expr expr1 = a_int < 42;
2. sequence expr2 = true | delay(1,2) | expr_a + expr_b;
3. property non_overlap = expr1 ->* expr2;
4. non_overlap.default_sampling(1_SC_US);
5. ASSERT_PROPERTY (non_overlap);
```

The advantage of the macro is that it provides the access to code line numbers during debugging. Consequently, if any assertion fails, exact line number is known. Furthermore, the API provides functions to set default clock of the assertions library if all assertions use the same clock course. Additionally, a default sampling function (*default_sampling(…)*) is also provided to setup the sampling frequency. This is shown in above code snippet (Line 4).

IV. LIBRARY IMPLEMENTATION

The SystemC assertions library is implemented in C++ and it leverages the C++ features, i.e., operator overloading is used for different operators, e.g., arithmetic operators. This enables a user-friendly syntax. The architecture of library is shown in Figure 3. Once the assertion is specified using C++ constructs, it is traversed using a visitor pattern. The visitor pattern provides the functionality to transform the defined assertion into an *Intermediate Representation (IR)*. The IR is in the form of data structures defined to store and use assertions information at

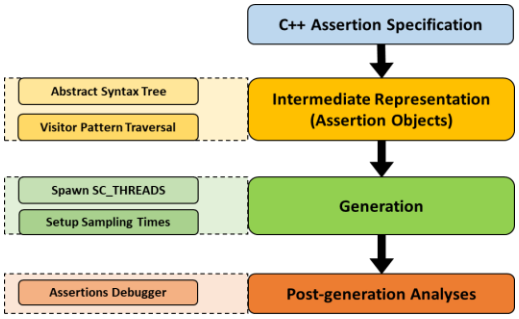

Figure 3: SystemC Assertions library architecture overview

different stages of evaluation. Depending on the assertion, SystemC threads are created for parallel execution. Each assertion is evaluated in a separate SystemC thread so that *AND/OR* sequence and property operators also function properly.

V. Case Study – Temperatrue Control System

A. *Case-study Setup*

We present here a real world complex heterogeneous system as a case-study (Figure 4) to showcase the features of the library. The system models a temperature control system covering multiple domains, i.e. SW, digital HW, and analog behavior. The system is modeled in SystemC/AMS using different *Models of Computation* (MoC), in particular *Timed Data Flow* (TDF) and *Electrical Linear Networks* (ELNs). The overall system as shown in Figure 4 consists of the following components:

- an ARM V8 based CPU using ARM Fast Models implemented as SystemC TLM with Linux operating system and SW running on top,
- four ADT7420 temperature sensors implemented as SystemC/AMS TDF and discrete event model,
- an *Advanced Microcontroller Bus Architecture* (AMBA) bus that acts as a bridge device to connect temperature sensors and ARM processor (created in SystemC TLM) – (*COS_AMBA_DEVICE* in Figure 4),
- an environment model (*Thermal_Network*) that builds 3 connected rooms and an ambient temperature modeled as a sinus (*SIN_SRC_TDF*), i.e. each sensor senses a different temperature (implementation as SystemC/AMS ELN and discrete event model),
- a heater model implemented as SystemC/AMS ELN that can be used to increase the temperature.

The communication between SW running on the processors and the connected sensors is done via registers connected to the bus of the processor. The SW configures the sensors by writing to addresses on the bus, which in turn creates TLM transactions. These TLM transactions are written into the corresponding registers of the ADT7420

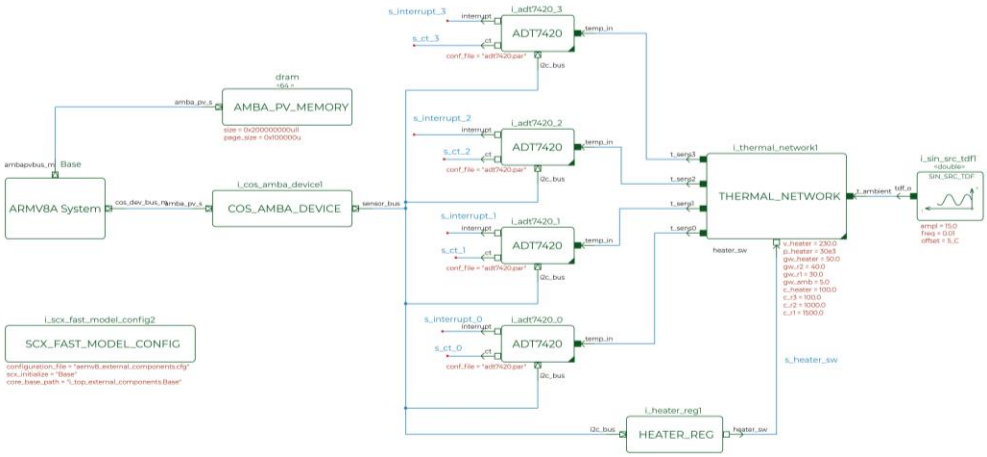

Figure 4: Case Study: Temperature control system

sensors. The COS_AMBA_DEVICE translates the AMBA-PV transactions used by ARM Fast Models to I2C transactions of the sensor model. To showcase the features of proposed system-level assertions library, the running example considers the following scenario for demonstration purposes:

- booting a Linux operating system on the ARM processor,
- a control SW is executed on top of Linux. The control SW continuously measures (monitors) the temperature sensor output,

- if the SW detects that the temperature value falls below a programmed threshold value, it switches the heater to ON state,
- otherwise, when the temperature exceeds a certain programmed threshold, the heater is switched to OFF state.

For demonstrating the features of the proposed SystemC assertions library, we use the following assertion:

- When the temperature of Room 1 *t_r1* (SystemC TDFsignal) is above the threshold *t_threshold* (SW-controlled TLM register value), the heater must be switched off (*heater_sw*) within *1 ms*.

This assertion can be specified using our proposed SystemC assertions library as follows:

```
1. auto heater_off = (t_r1 > t_threshhold) ->* (true | delay(1_SC_MS) | (heater_sw==false));
2. heater_off.default_sampling(1_SC_MS);
```

B. Simulation Results

Partial simulation results of the temperature control system SW are shown in Figure 5. The orange sinus signal is the ambient temperature (SIN_SRC_TDF) which oscillates between 262 K and 293 K. The green waveform signal (t_r1) is the temperature of room 1. The blue waveform signal (t_r2) is the temperature of room 2. The purple waveform signal (t_r3) is the temperature of room 3. At the bottom of Figure 3, digital signals – *heater_switch* and *interrupts (irq0-irq3)* from temperature sensors are displayed.

After booting the Linux OS (approx. 30s) the control SW gets started. The heater (*heater_switch*) gets turned on as the temperature in room one (t_r1) is below the minimum temperature of 292 K. It can be seen how the

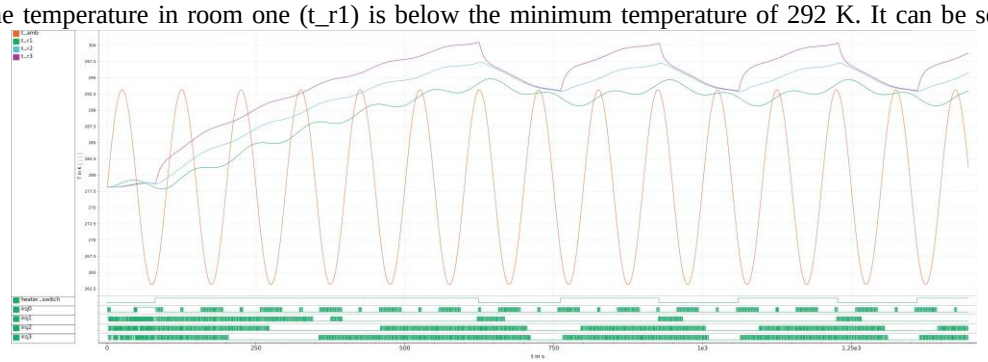

Figure 5: Simulation results running the temperature control SW

temperature slowly increases in all rooms. When the temperature is above the maximum threshold of 294.15 K the heater gets turned off. As a consequence, the room temperatures start to decrease. The sensors have been programmed to generate an interrupt whenever the temperature is above or below a threshold value (stored in register). We could see the assertion was satisfied throughout the simulation. However, if we decreased the *delay(...)* from 1 ms to a smaller value, the assertion was always violated. This is expected and in accordance with the specifications. They require that the *heater_switch* should be turned off within 1 ms after the threshold temperature is crossed. The reason for 1 ms is because of the inherent delays due to reading and writing of registers in different connected devices, and can be summarized as follows:

- the temperature sensor senses the temperature,
- the sensed temperature is written into the register,
- SW reads the temperature from the ARM processor,
- SW checks whether the sensed temperature value is above the threshold value,
- and writing the heater switch control register depending on the comparison result.

Hence, using the proposed intuitive SystemC assertions library, it is possible to check the timing of complex behaviors of the heterogeneous systems, e.g., digital, analog, and SW behavior.

VI. Conclusions

In this paper, we proposed a SystemC assertions library for heterogeneous systems. The library comprises of an intuitive and user-friendly API and offers full compatibility with SystemC, TLM, and SystemC/AMS. As a result, the library supports specification of complex interactions, necessary to represent complex AMS behavior of a System. The SystemC assertions library prototype was used to verify the industrial model using ARM Fast Models, a temperature control system SW, environment models, temperature sensors, and assertions. The library is still a work-in-progress with a focus on building an easy-to-use assertion language within SystemC. Currently, it only supports a subset of SVA operators as a proof of concept. Next, the goal is to support all SVA features and extensions.

References

[1] "IEEE Standard for Standard SystemC® Analog/Mixed-Signal Extensions Language Reference Manual," in IEEE Std 1666.1-2016, vol.,no.,pp.1-236, April 6 2016 doi: 10.1109/IEEESTD.2016.7448795

[2] "IEEE Standard for Standard SystemC Language Reference Manual," in IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005) , vol., no., pp.1-638, Jan. 9 2012 doi: 10.1109/IEEESTD.2012.6134619

[3] C. Grimm, M. Barnasconi, A. Vachoux,and K. Einwich, "An introduction to modeling embedded analog/mixed-signal systems using SystemC AMS extensions," in *DAC*, vol. 23, 2008.

[4] F. Pecheux, C. Grimm, T. Maehne, M. Barnasconi, and K. Einwich, "SystemC AMS based frameworks for virtual prototyping of heterogeneous systems," in ISCAS, 2018, pp. 1–4.

[5] M. Barnasconi and S. Adhikari, "ESL design in SystemC AMS: Introducing a top-down design methodology for mixed-signal systems," in *DAC*, 2017, pp. 1–5.

[6] A. B. Mehta, *System Verilog Assertions and Functional Coverage: Guide to Language, Methodology and Applications*. Springer Nature, 2019.

[7] J. Yuan, C. Pixley, and A. Aziz, Constraint-based Verification. Springer, 2006.

[8] F. Haedicke, H. M. Le, D. Große, and R. Drechsler, "CRAVE: An advanced constrained random verification environment for SystemC," in SoC, 2012, pp. 1–7.

[9] D. A. S. Committee *et al.*, "IEEE standard for systemverilog unified hardware design, specification, and verification language standard ieee 1800," *http://www. edastds. org/sv/*, 2005.

[10] S. Lammermann, J. Ruf, T. Kropf, W. Rosenstiel, A. Viehl, A. Jesser, and L. Hedrich, "Towards assertion-based verification of heterogeneous system designs," in *DATE*, 2010, pp. 1171–1176.

[11] N. Bombieri, F. Fummi, V. Guarnieri, G. Pravadelli, F. Stefanni, T. Ghasempouri, M. Lora, G. Auditore, and M. N. Marcigaglia, "Reusing rtl assertion checkers for verification of systemc tlm models," *Journal of Electronic Testing*, vol. 31, no. 2, pp. 167–180, 2015.

[12] Ničković, Dejan, et al. "AMT 2.0: qualitative and quantitative trace analysis with extended signal temporal logic." International Journal on Software Tools for Technology Transfer 22.6 (2020): 741-758.

[13] Mukherjee, Rajdeep, et al. "Formal techniques for effective co-verification of hardware/software co-designs." 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC). IEEE, 2017.

[14] Moiseev, Mikhail, et al. "Temporal Assertions in SystemC". DVCon Europe 2020.

[15] Muhammad Hassan, Daniel Große, and Rolf Drechsler. *Enhanced Virtual Prototyping for Heterogeneous Systems*. Springer, 2022.

[16] Vladimir Herdt, Daniel Große, and Rolf Drechsler. *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer, 2020.

[17] Daniel Große and Rolf Drechsler. *Quality-Driven SystemC Design*. Springer, 2010.

[18] Vladimir Herdt, Hoang M. Le, Daniel Große, and Rolf Drechsler. Verifying SystemC using intermediate verification language and stateful symbolic simulation. TCAD, 38(7):1359-1372, July 2019.

[19] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. Early concolic testing of embedded binaries with virtual prototypes: A RISC-V case study. In DAC, pages 188:1-188:6, 2019.

[20] Daniel Große and Rolf Drechsler. Checkers for SystemC designs. In MEMOCODE, pages 171-178, 2004.

[21] Thilo Vörtler, Karsten Einwich, Muhammad Hassan, and Daniel Große. Using constraints for SystemC AMS design and verification. In DVCon Europe, 2018.

[22] Muhammad Hassan, Daniel Große, Thilo Vörtler, Karsten Einwich, and Rolf Drechsler. Functional coverage-driven characterization of RF amplifiers. In FDL, pages 1-8, 2019.