# Symbolic Execution of Unmodified SystemC Peripherals[*]

Karl Aaron Rudkowski[1], Sallar Ahmadi-Pour[1], and Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
{karlaaron,sallar,drechsler}@uni-bremen.de

## Abstract

In the modern hardware design process, the ever-increasing complexity of designs poses a challenge. An important step in the workflow is the verification, which aims to identify errors early on. To this end, executable models of the design, called Virtual Prototypes (VPs), allow early and continuous verification. Current symbolic execution-based approaches rarely consider the characteristic differences of peripherals, and none consider cross-level verification of these. Additionally, SystemC, though a popular hardware modelling language, is either not targeted, or replaced by a heavily limited kernel implementation. We propose the first symbolic execution engine capable of verifying unmodified SystemC peripherals. Our tool leverages the threading behaviour specified in the SystemC standard for a light-weight PThread support. The evaluation of a RISC-V Platform Level Interrupt Controller (PLIC) shows our approach's effectiveness in multiple verification scenarios.

## 1 Introduction

In the modern hardware design workflow, VPs enable early verification, long before any chip is fabricated. This verification is an important step to minimise the cost and effort associated with repairing errors. However, it is challenged by the drastically increasing complexity of designs [1]. Model checking [2, 3, 4, 5] can offer a full verification, but is often based on custom intermediate representations, which sometimes even have to be generated manually. Simulation-based approaches like fuzzing [6, 7] or concolic execution [8, 9] work directly on the device, but are limited to generating concrete inputs. They do not offer any guarantees about the behaviour of the Design under Verification (DUV) outside of these test cases. In contrast, symbolic execution uses variables that represent sets of concrete values to explore the state space more effectively. It can thus offer a more complete reasoning about the design.

Approaches using symbolic execution either consider hardware designs in general [10, 11], or specifically processors [12]. However, another important verification target are the peripherals, which implement a wide range of tasks. Examples range from simple sensors to complex domain-specific accelerators [13]. They also require additional logic to communicate over bus systems. Due to these fundamental differences between the processor and the peripherals, they require separate attention in the verification workflow. Peripheral-specific symbolic execution approaches are currently limited to replacing the SystemC kernel with an alternative implementation [14]. This replacement is considered because the original kernel uses threads to execute `SC_THREAD`s, which hold all peripheral functionality. These are not supported by any state-of-the-art symbolic execution tool. In addition to the new kernel, these `SC_THREAD`s have to be modified. However, both the thread transformation and the replacement are themselves weak spots in the verification, and only support the most important SystemC Transaction Level Modelling (TLM) features. Their equivalence to the unmodified version of DUV and kernel are not proven. Furthermore, in the industry, these replacements often cannot be integrated into the verification process.

We introduce a symbolic execution engine which supports the *unmodified* SystemC kernel and peripherals. Unlike [14], we can thereby avoid any replacements or modifications. Our contributions are:

1. Extending the state-of-the-art tool KLEE [15] (version 3.1) by a SystemC-specific threading approach. We leverage the `SC_THREAD` behaviour specified in the standard [16] for an optimised verification of threading-based models.
2. A workflow offering verification at both TLM *and* Register Transfer Level (RTL), as well as a cross-level verification between the two.
3. An evaluation of a real-world peripheral in these three verification scenarios, using a RISC-V PLIC.

## 2 Preliminaries

### 2.1 SystemC [16]

The SystemC modelling language, implemented as a C++ library, encapsulates hardware components in modules. Each module's functionality is realised with processes, which can be either methods or threads. The simulation is event-driven, meaning the processes register themselves for execution on events such as clock edges, wait times, or incoming inputs.

[*]This work was supported in part by the German Federal Ministry of Education and Research (BMBF) under grant no. 16ME0127 and no. 16ME0135 (Scale4Edge), and no. 01IW24001 (EXCLplus).

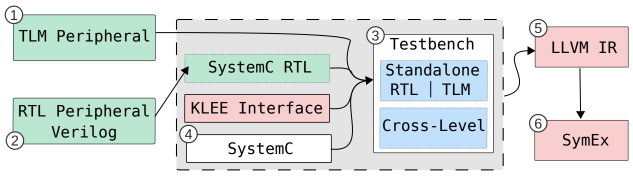

**Figure 1** Overview Verification Workflow.

Inter-module communication can be based on signals and ports, or, at a higher abstraction level, TLM sockets.

## 2.2 Symbolic Execution

Symbolic Execution is a verification technique from the software domain. There exist several software-focused engines like KLEE [15] and Angr [17]. The idea is to represent sets of concrete values with symbolic variables. During the execution, instructions involving these symbolic variables are evaluated using the Satisfiability Modulo Theory (SMT) solver. This can result in forking the execution (branch instructions), or generating concrete inputs that trigger errors. Thereby, the program can be checked for generic failures (e.g. divide by zero), and assertion violations.

## 3 Symbolic Execution of SystemC Peripherals

In the SystemC kernel, the `SC_THREAD`s are implemented using system threads. Threading is not supported by symbolic execution due to the associated overhead of verifying multi-threaded software, e.g. checks for data races. However, in SystemC, the standard specifies *cooperative multitasking* [16]. That means that all processes, including `SC_THREAD`s, are executed without interruption, and sequentially. The developer defines the synchronisation points, e.g. with `wait()` statements. System threads enable the context switches between `SC_THREAD`s, but they do not execute in parallel. Therefore, many threading issues are not a topic for the symbolic execution of SystemC peripherals. Instead, it is only necessary to model the thread calls and suspensions according to the SystemC standard.
The following sections present the verification workflow using the proposed symbolic execution tool, as well as a description of how SystemC threading is integrated into the symbolic execution.

### 3.1 Verification Workflow

In Figure 1, we give an overview of the proposed verification workflow. The desired DUV can be provided at both the RTL and TLM level. The TLM description ① is usually readily available in SystemC. Similarly, the RTL description ② can be written in SystemC. However, more commonly, it would be available in Verilog, in which case an automatic transcompilation to SystemC using Verilator [18] is possible. A symbolic execution test bench ③ is written to define the desired behaviour of

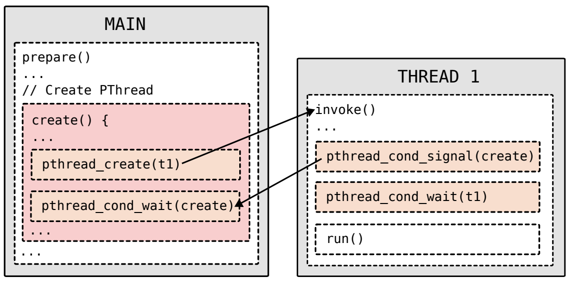

**Figure 2** Overview SystemC Thread Creation (PThread module).

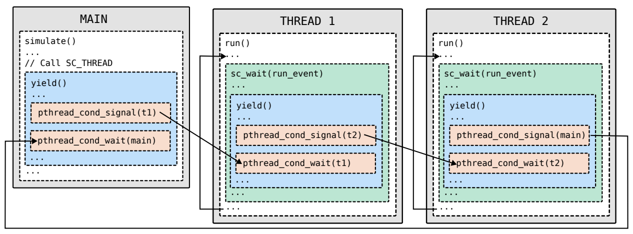

**Figure 3** Overview SystemC Thread Simulation (PThread module).

the DUV using assertion statements, and declare symbolic variables with KLEE interface methods. The tests can realise three possible verification scenarios:

(a) Standalone verification of a TLM device

(b) Standalone verification of a RTL device

(c) Cross-level verification RTL⇔TLM

If both RTL and TLM implementations of a device are available, the test bench can cover all three scenarios in separate tests.
The SystemC device(s), optimised kernel ④, and test bench are compiled into the LLVM *Intermediate Representation*, using the Clang C++ compiler ⑤. Finally, the peripheral can be verified using our symbolic execution tool ⑥.

### 3.2 SystemC Threading

SystemC offers multiple implementations of the threading behaviour defined in the standard. These differ in the threading they are based on, but the general attributes remain the same. We focused on the PThread-based version. In it, each `SC_THREAD`, as well as the scheduler, is it's own PThread, associated with a unique `pthread_condition` (in the following `pt_cond`). All threads are created before the simulation start, as illustrated in Figure 2. During the setup, the new thread does not execute the `SC_THREAD`-declared function (here: `run`). Instead, it starts waiting on it's newly created `pt_cond` after setup.
During the simulation, the thread execution is managed over a `pt_signal/_wait` handshake, as illustrated in Figure 3. To switch from one thread to another, the callee thread first signals the called thread's `pt_cond`. Then, it starts waiting on it's own `pt_cond`, and only continues

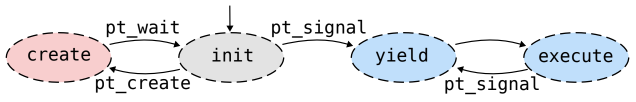

**Figure 4** Overview Symbolic Execution Engine States.

execution after it is signalled again. Therefore, at any time, only one thread is actually executing.

In the symbolic execution engine, each thread needs it's own context, represented by stack and program counter. The stack is initially empty, and contains only variables allocated while the respective thread is active. During the symbolic execution, the `CallInsts` targeting `pthread_*` functions are intercepted. Their original implementation is ignored, which avoids the creation of real system threads. Instead, they mark an internal state change, as illustrated in Figure 4. In *init*, the main thread's context is active. In *create*, `sysc::invoke` is executed in a new context, as seen in Figure 2. After `pt_wait`, this context is associated with the new `pt_cond`. Back in *init*, the main context continues with the instruction after `pt_create`. A `pt_signal` triggers *yield*, as shown in Figure 3. During *yield*, the current context is saved, and the desired one is loaded. Afterwards, regular *execution* continues.

Outside of these three types of `CallInst`, all LLVM instructions are handled as specified. Thus, the original implementations of SystemC and DUV are considered. This stands in contrast with the current state of the art [14], where both SystemC and DUV were modified.

## 4 Evaluation

We evaluate our approach by applying it to a RISC-V TLM PLIC, specifically the FE310 configuration based on the SiFive FE310 SoC [19]. It manages global interrupts, for example from Input/Output devices, and notifies the targets, usually Hardware Threads. The order of the interrupts is determined by their individual priorities, as well as the per-target priority threshold.

The TLM implementation from the open source RISC-V VP[1] was already used as a case study in [14], who used a custom SystemC replacement kernel. We additionally consider an in-house implementation of this device at the RTL abstraction level, which allowed for tests across three different verification scenarios: the standalone at each abstraction level, and the cross-level verification between the two. For each scenario, multiple test cases were defined. At both TLM and RTL, there are three that consider the firing of an interrupt (T1-3,T6-8). Tests #1/#6 models the basic case of triggering one input with a symbolic interrupts number, tests #2/#7 add to this a symbolic priority and priority threshold. Tests #3/#8 check the correct interrupt order given two symbolic interrupts, each with a symbolic priority. For the TLM PLIC, cases #4 and #5 read/write symbolic data over the TLM socket, respectively. Finally, for the cross-level verification, tests #9 and #10 mirror tests #2 and #3.

[1]https://github.com/agra-uni-bremen/riscv-vp

**Table 1** PLIC Evaluation (TO=Timeout).

| Test | | Paths | | Time | Memory | Comment |
|---|---|---|---|---|---|---|
| | | compl. | partial | | | |
| **1** | TLM | 0 | 5 | TO | 448.55 | F1 |
| **2** | TLM | 1 | 6 | TO | 453.76 | |
| **3** | TLM | 1 | 14 | TO | 457.06 | |
| **4** | TLM | 970 | 198 | 221.43 | 459.47 | F2,F3 |
| **5** | TLM | 2733 | 261731 | TO | 6279.18 | F2-5 |
| **6** | RTL | 0 | 5 | TO | 566.65 | |
| **7** | RTL | 0 | 530 | 566.69 | 608.07 | F6,F7 |
| **8** | RTL | 2460 | 5889 | TO | 2221.87 | |
| **9** | EQ | 0 | 8 | TO | 607.21 | |
| **10** | EQ | 0 | 7 | TO | 609.83 | |

We configured our symbolic execution tool to use STP [20] as the solver for SMT queries. In order to achieve results within an acceptable time frame, some limits were imposed: an execution runs a maximum of 24 h, and can use at most 4000 MiB of working memory. A single solver query can take at most 120 s, so that the run time is not dominated by few solver queries. The search strategy for choosing execution states during the exploration is Breadth First Search (BFS).

Table 1 shows the results for each test case. First, the number of explored paths is given, divided into those that were exhaustively explored, and those that were prematurely abandoned, e.g. due to the time limit or assertion violations. Following, we list the run's complete duration (in seconds), and the average memory usage (MiB). Finally, the result of the test case is summarised.

At the TLM level, our tool found the same errors as [14] (**F1-5**). At the RTL, the relationship between the priority and the threshold was inverse (**F6,7**). Instead of firing an interrupt if it's priority was greater than the threshold, it was fired if it was less than or equal. However, this error was not found in the cross-level scenario. The cause for this are the timeouts (**TO**), in which the tool could not exhaustively explore a test's state space in under 24 h. They appear even in the standalone test cases mirroring the cross-level ones. Consequently, if the tool has to find a path through both implementations, timeouts are more likely to occur. This limits the results that can be attained.

There are multiple causes of these timeouts. First, larger object sizes directly lead to larger arrays in SMT queries. Current solvers, however, struggle with large arrays. The object size is influenced by the involved SystemC features. For example, the TLM PLIC generates a 2336 B sized array, of which over 1000 B come from the `tlm::simple_target_socket`. This problem could be addressed by including only relevant parts of the array.

A second source of origin is a known limitation of symbolic-execution-based approaches called the *state space explosion* [21]. The number of possible symbolic states grows exponentially with the number of branch instructions. Under this constraint, achieving interesting results with limited resources is important. Our tool was able to find all known errors in the DUV implementations. The results match those of the current state-of-the-art [14], despite the greater number of processed branch instructions. Evaluating the advantages and disadvantages of these two opposing approaches, including a detailed performance comparison, would be an interesting direction for future work.

Additionally, the timeouts, as well as the problems in the cross-level scenario, highlight the importance of choosing the most relevant states to follow. While there are surveys regarding the (dis)advantages of search strategies for symbolic execution of software, we are not aware of similar studies for symbolic execution of hardware. Consequently, this too is a knowledge gap to be filled in the future.

## 5 Conclusion

We propose a symbolic execution engine capable of verifying unmodified SystemC peripherals. We leverage the `SC_THREAD` behaviour as defined in the standard to model the expected control and data flow between threads. This domain-specific approach avoids the overhead associated with verifying general multi-threaded software. In an experimental evaluation, our approach's effectiveness in verifying at two different abstraction levels, TLM and RTL, is shown using a RISC-V PLIC. This showed our approach's functional improvement over the current state-of-the-art [14], a replacement kernel limited to TLM. However, the performance penalty associated with SystemC's complexity revealed three objectives for future work. They consist of an improved object handling, a detailed comparison to [14], and appropriate exploration of the state space.

## 6 Literature

[1] W. Chen *et al.*, "Challenges and Trends in Modern SoC Design Verification," *IEEE Design & Test*, vol. 34, no. 5, pp. 7–22, 2017.

[2] P. Herber *et al.*, "STATE – A SystemC to Timed Automata Transformation Engine," in *2015 IEEE 17th ICHPCC, 2015 IEEE 7th CSS, and 2015 IEEE 12th ICESS*, 2015, pp. 1074–1077.

[3] H. M. Le *et al.*, "Towards Formal Verification of Real-World SystemC TLM Peripheral Models - A Case Study," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 1160–1163.

[4] S. Deng *et al.*, "Bounded Model Checking for RTL Circuits Based on Algorithm Abstraction Refinement," in *2006 8th International Conference on Solid-State and Integrated Circuit Technology Proceedings*, 2006, pp. 2082–2084.

[5] P. Bavonparadon *et al.*, "RTL Formal Verification of Embedded Processors," in *2002 IEEE International Conference on Industrial Technology, 2002. IEEE ICIT '02.*, vol. 1, 2002, pp. 667–672.

[6] N. Bruns *et al.*, "Efficient Cross-Level Processor Verification Using Coverage-Guided Fuzzing," in *Proceedings of the Great Lakes Symposium on VLSI 2022*, ser. GLSVLSI '22, 2022, p. 97–103.

[7] S. Ahmadi-Pour *et al.*, "Synergistic Verification of Hardware Peripherals through Virtual Prototype Aided Cross-Level Methodology Leveraging Coverage-Guided Fuzzing and Co-Simulation," *Chips*, vol. 2, pp. 195–208, 2023.

[8] A. Ahmed *et al.*, "Directed Test Generation Using Concolic Testing on RTL Models," in *2018 DATE*, 2018, pp. 1538–1543.

[9] B. Lin *et al.*, "Concolic Testing of SystemC Designs," in *2018 19th International Symposium on Quality Electronic Design (ISQED)*, 2018, pp. 1–7.

[10] Y. Zhang *et al.*, "Automatic Generation of High-Coverage Tests for RTL Designs Using Software Techniques and Tools," in *2016 IEEE 11th Conference on Industrial Electronics and Applications (ICIEA)*, 2016, pp. 856–861.

[11] B. Lin *et al.*, "Generating High Coverage Tests for SystemC Designs Using Symbolic Execution," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2016, pp. 166–171.

[12] N. Bruns *et al.*, "Processor Verification using Symbolic Execution: A RISC-V Case-Study," in *2023 DATE*, 2023, pp. 1–6.

[13] J. L. Hennessy *et al.*, "A New Golden Age for Computer Architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[14] P. Pieper *et al.*, "Verifying SystemC TLM peripherals Using Modern C++ Symbolic Execution Tools," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 1177–1182.

[15] C. Cadar *et al.*, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, 2008, p. 209–224.

[16] "IEEE Standard for Standard SystemC® Language Reference Manual," *IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011)*, pp. 1–618, 2023.

[17] Y. Shoshitaishvili *et al.*, "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157.

[18] (2004) Verilator Compiler. [Online]. Available: https://www.veripool.org/verilator/

[19] (2020) SiFive FE310-G000 Manual. [Online]. Available: https://static.dev.sifive.com/FE310-G000.pdf

[20] V. Ganesh *et al.*, "A Decision Procedure for Bit-Vectors and Arrays," in *Computer Aided Verification*, W. Damm *et al.*, Eds., 2007, pp. 519–531.

[21] R. Baldoni *et al.*, "A Survey of Symbolic Execution Techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.