Verifying SystemC TLM Peripherals using Modern C++ Symbolic Execution Tools

Pascal Pieper
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
Pascal.Pieper@dfki.de

Vladimir Herdt
Cyber-Physical Systems, DFKI GmbH
Institute of Computer Science, University of Bremen
Bremen, Germany
vherdt@uni-bremen.de

Daniel Große
Johannes Kepler University Linz
Linz, Austria
Daniel.Grosse@jku.at

Rolf Drechsler
Cyber-Physical Systems, DFKI GmbH
Institute of Computer Science, University of Bremen
Bremen, Germany
drechsler@uni-bremen.de

ABSTRACT

In this paper we propose an effective approach for verification of real-world SystemC TLM peripherals using modern C++ symbolic execution tools. We designed a lightweight SystemC peripheral kernel that enables an efficient integration with the modern symbolic execution engine KLEE and acts as a drop-in replacement for the normal SystemC kernel on pre-processed TLM peripherals. The pre-processing step essentially replaces context switches in SystemC threads with normal function calls which can be handled by KLEE. Our experiments, using a publicly available RISC-V specific interrupt controller, demonstrate the scalability and bug hunting effectiveness of our approach.

ACM Reference Format:
Pascal Pieper, Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2022. Verifying SystemC TLM Peripherals using Modern C++ Symbolic Execution Tools . In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC) (DAC '22), July 10–14, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3489517.3530604

1 INTRODUCTION

SystemC in combination with the *Transaction-Level Modeling* (TLM) style has become an industrial standard for creating advanced *Virtual Prototypes* (VPs). A VP is essentially an abstract executable model of the entire hardware platform which is leveraged for early software development and acts as a reference model for the subsequent hardware design flow steps. Early and thorough verification of SystemC-based VPs is very important to avoid propagation of errors and the associated costly iterations for fixing them. Beside the instruction set simulator, which is an abstract model of the processor, TLM peripherals, such as an interrupt controller, are a central part of the VP. TLM peripherals rely on common modeling standards to describe the register interface, according to a device memory map, and provide a TLM interface to implement (software-driven) read and write accesses. The actual functionality is implemented through SystemC threads that leverage the event driven semantics of the SystemC kernel for synchronization. Application of formal verification techniques on TLM peripherals is very challenging as it needs to support the intricate TLM periperhal modeling semantics in combination with the simulation semantics of the SystemC kernel. Existing methods commonly rely on formal intermediate representations to capture the TLM periperhal semantics, which require significant effort to derive, do not scale to advanced SystemC TLM peripherals, or do not support core features of the SystemC kernel[1].

To mitigate these issues, in this paper we propose an effective approach for verification of real-world SystemC TLM peripherals by using modern C++ symbolic execution tools. Our approach consists of three main parts: First, we perform a common SystemC thread transformation pre-processing step to enable replacement of context switching in threads with normal function calls, which is the main reason why an unmodified SystemC is incompatible with KLEE. Second, we designed a SystemC *Peripheral Kernel* (PK) that can essentially act as a drop-in replacement for the normal SystemC kernel on the pre-processed TLM peripherals. It implements all necessary interfaces which are used by advanced SystemC TLM peripherals. At the same time, the PK is much more lightweight by focusing only on relevant interfaces and integrating optimization procedures tailored to support symbolic execution engines. Third, we apply the existing state-of-the-art symbolic execution tool KLEE [2] to verify (symbolic) properties specified for the TLM peripheral by means of assertions and assumptions embedded in a testbench. As a case-study we report verification results for a RISC-V specific *Platform Level Interrupt Controller* (PLIC) [25] that is used in an open source virtual prototyping environment for the SiFive FE310 SoC [1]. The PLIC provides interrupt handling capabilities supporting several operating systems such as Zephyr and FreeRTOS. Our approach has been scalable and very effective in bug hunting. We found new previously unknown bugs in the PLIC and also demonstrate by means of fault-injection that other intricate bugs can be detected very quickly. To stimulate further research, we have made our PK together with our experimental setup available on GitHub[2].

2 RELATED WORK

Formal verification of SystemC [21] designs is very important and also very challenging [24]. Therefore, it has received significant attention from the research community.

Early efforts, for example [8, 13, 18, 23], have very limited scalability or do not model the SystemC simulation semantics thoroughly [14]. Furthermore, they are mostly geared towards RTL signal-based communication.

More recent approaches are specifically targeting high-level SystemC designs that are in general suitable to capture the TLM semantics [19]. As a result a set of SystemC verification tools have emerged.

DAC '22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530604

[1]We will discuss these existing methods and their limitations in Section 2.
[2]https://github.com/agra-uni-bremen/SymSysC

KRATOS [5] employs a model checking algorithm based on symbolic lazy abstraction and accepts an intermediate C input language with simple assertions. SCIVER [7] operates on sequential C models and leverages high-level induction techniques to check temporal properties [22]. SDSS[4] formalizes the semantics of SystemC designs in terms of Kripke structures and then applies a bounded model checking algorithm. In a follow-up work [3], the approach has also been optimized with state space reduction techniques based on *Partial Order Reduction* (POR). SISSI [11] defines the *Intermediate Verification Language* (IVL) format and employs stateful symbolic simulation techniques in combination with POR to deal efficiently with cyclic state spaces. For optimization purposes, native execution techniques have been leveraged [12]. STATE [10] translates SystemC designs to timed automata and verifies properties formulated on the timed automata using the UPPAAL model checker. In the context of these approaches, an extensive set of academic SystemC benchmarks is available. However, from a practical perspective, these approaches are still limited since due to their employed intermediate formalizations, they are not easily applicable to real-world VPs.

Other recent approaches have attempted to tackle this challenge. A first attempt has been made in [20], where the successful application of [9] on a simplified ARM AHB TLM-2.0 model is reported. In a follow-up work [16], slicing-based techniques are investigated to improve scalability and results on the verification of a packet switch are reported. However, the specific modelling challenges of TLM peripherals have not been considered.

Another recent approach [15] addresses this real world application issue specifically. The authors propose a *XIVL* formal intermediate representation that bridges the modelling gap of TLM peripherals with the formal language employed by the SISSI verification tool. While the approach has shown promising results in verifying formal properties on an interrupt controller, it still requires significant effort to (manually) transform a SystemC model into the XIVL. In contrast our approach operates directly on the C++ code and can thus also benefit from recent advances in modern symbolic execution engines tailored for C++.

We are also aware of the approach in [17], which leverages the Klee symbolic execution engine to generate test cases for SystemC modules that provide a high (branch) coverage. The approach also needs to integrate a customized scheduler to cover the SystemC simulation semantics and has reported very promising results in testing different SystemC designs. However, only the high-level synthesizable subset [6] of SystemC is supported by that approach. Moreover, it only supports static sensitivity to a single clock edge and does not allow the use of `sc_events`, which is a common modelling requirement for TLM peripherals. Therefore, this approach does not support the verification of TLM peripherals as considered in our case-study.

3 PRELIMINARIES

This section provides relevant background information on SystemC TLM (Section 3.1) and the RISC-V specific PLIC (Section 3.2).

3.1 SystemC TLM

SystemC [21] is a hardware modelling framework that is widely adopted in the industry. It offers a C/C++ style modelling framework with varying degrees of timing accuracy at the benefit of simulation speed. The structure of a SystemC design is described with ports and modules, whereas the behaviour is modelled in processes which are triggered by events. The execution of a process is non-preemptive, i.e. it uses co-operative user-space scheduling for processes of each module. This means that a process, once started, runs indefinitely until it either yields (`wait()`) or terminates forever (return). The process will be woken up when an event in its static sensitivity list triggers (e.g. a clock edge), or it can wait for a dynamic `sc_event`. This event may be triggered immediately or with a delay by, e.g., an asynchronous task, calling `event.notify(delay)`.

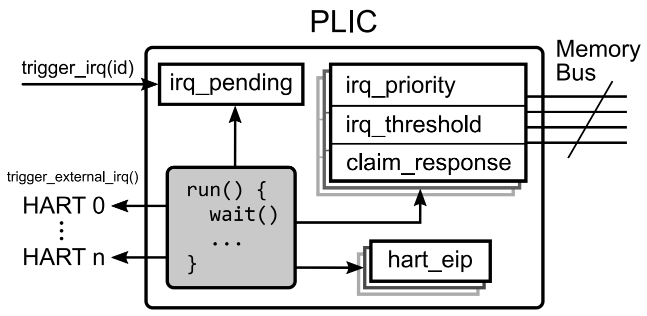

Figure 1: I/O Ports of the Platform Level Interrupt Controller. Elements with sharp corners are registers, managed by logic in the main `run()` method. `hart_eip` is a private variable used for suppressing interrupt re-triggers.

Communication between SystemC modules can be abstracted using the TLM standard [19] at the cost of timing accuracy, but with significant improvements in simulation speed, i.e. up to a factor of 1,000 in comparison to RTL simulation. Especially in bus-like memory mapped communication networks, skipping interconnect procedures and signal resolutions will greatly reduce the execution time. Instead of taking the whole route through the VP, interactions can be initiated directly to a target port. These transactions can either *read* or *write* at a specified address carrying a generic payload along with a cumulative delay, and may return either `OK` or `ERROR`. This delay is increased by every model passing the transaction and added to a global quantum afterward. The global quantum tracks the time difference a transaction "jumped" in contrast to the actual simulated time. If this difference is bigger than the maximum allowed time, SystemC will initiate a global synchronization. This allows for a fine control over the trade-off between simulation speed and accuracy.

3.2 PLIC

The *Platform Level Interrupt Controller* (PLIC) is specified by the RISC-V instruction set architecture [25]. It manages incoming, 'global' interrupts and notifies the *hardware threads* (HARTs), i.e. the individual processor cores. It contains a set of registers for each HART where the processor can assign a priority and a notification threshold for each interrupt (see Fig. 1). When an external interrupt fires, it sets an *interrupt pending* bit to the corresponding position in an internal register. Then, the PLIC will decide, based on the interrupt's assigned priority and its threshold, if a notification is passed to the individual HARTs (via `trigger_external_irq()`).

After an interrupt notification, a HART may check pending interrupts in the claim/response register via the memory mapped interface. The HART finishes the completion of the interrupt by writing back the corresponding interrupt ID to the claim/response register. If other interrupts of less priority are pending, the PLIC will re-trigger all HARTs based on their individual threshold after that. Citing the official specification: "A priority value of 0 is reserved to mean *never interrupt* and effectively disables the interrupt. Priority 1 is the lowest active priority while the maximum level of priority depends on PLIC implementation. Ties between global interrupts of the same priority are broken by the interrupt ID; the lowest ID has the highest effective priority."[3]

[3]https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc

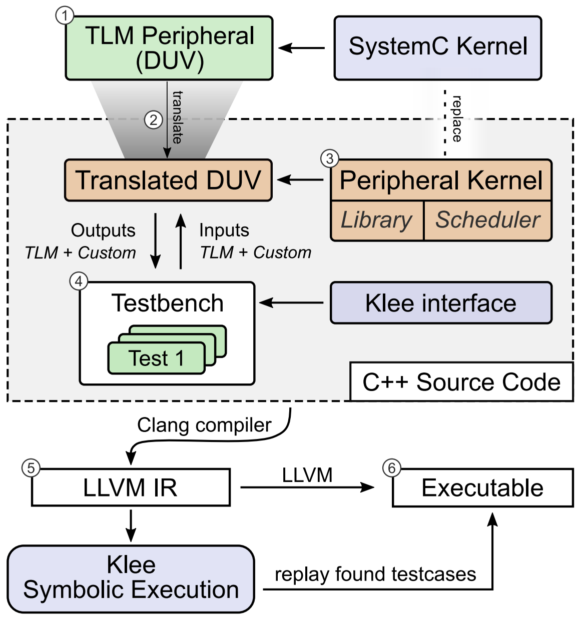

Figure 2: Overview of our approach. Highlighted in green are the user-defined parts, in brown are the provided elements, and blue are existing tools.

4 TLM PERIPHERAL VERIFICATION VIA SYMBOLIC EXECUTION

In this section, we present our proposed approach for verification of TLM peripherals via symbolic execution.

4.1 Overview

Fig. 2 shows an overview on our approach. Starting point is a SystemC *TLM peripheral* ① which is the *Device Under Verification* (DUV). It provides a TLM interface to communicate with other devices embedded in a virtual prototyping environment, and interacts with the SystemC kernel. The complexity of the SystemC kernel makes it very difficult to apply symbolic execution techniques for verification purposes of the DUV directly. In particular, the SystemC thread scheduling mechanism that relies on context switches and heavyweight data structures, such as floating point based *sc_time* implementations, lead to significant performance bottlenecks in symbolic execution tools, up to the point of being unsupported. Therefore, in a first pre-processing step, the DUV is *translated* ② by transforming its userspace-scheduling styled threads into classic function calls. In addition, we provide a *Peripheral Kernel* (PK, ③) as a drop-in replacement for the SystemC kernel on the translated DUV, with a compatible library and an optimized scheduling mechanism. Our PK provides all necessary interfaces which are used by advanced SystemC TLM peripherals. At the same time, it is much more lightweight by focusing only on relevant interfaces and integrating optimization procedures tailored to support symbolic execution engines. We will provide more details on the translation step and our PK in Section 4.2 and Section 4.3, respectively.

Testbenches ④ are user-provided for verification purposes. They interact with the translated DUV using the standard TLM interface (e.g. to read/write TLM registers) as well as custom interface functions (e.g. an interrupt line in an interrupt controller). Assumptions and assertions can now be embedded in the testbench to specify symbolic input parameters and check the output behaviour, respectively. This setup enables verification engineers to write very fine-grained yet generalized tests to enable a broad coverage and search for previously unknown corner-cases via symbolic execution. In this work we leverage Klee, a state-of-the-art symbolic execution engine for C/C++, which provides a set of interface functions to declare and reason about symbolic expressions.

Each testbench is compiled together with the translated DUV, PK and Klee interface into a single LLVM *Intermediate Representation* (IR, ⑤) using the Clang C++ compiler. The LLVM IR is analysed using the Klee symbolic execution engine. Klee performs a symbolic state space exploration searching for errors on the symbolic execution paths. An error may be an assertion evaluated to *false*, an invalid memory access (segmentation fault, array-out-of-bounds), a software trap such as a division by zero, or an unhandled exception. For every error, a counterexample, i.e. concrete assignment for symbolic inputs, is generated by Klee. It allows to reproduce the error and replay the testbench execution for debugging purposes. For convenience, the IR bytecode can be compiled into a machine-native *Executable* ⑥ so that a classical debugger can be attached to analyse the counterexamples.

In the following, we provide more details on the *translation step* ② and our *PK* ③ in Sections 4.2 and 4.3, respectively.

4.2 Thread to Function Translation

The thread to function translation is the key idea in enabling the symbolic execution through Klee, as the SystemC userspace-scheduling implementations are incompatible with Klees interpreter. It essentially works by moving local into static variables to preserve them across function calls and embedding *Finite State Machine* (FSM) logic with goto statements to interrupt and resume the function at the right position on each context switch. This translation allows to preserve the execution context across multiple function calls and thus models the SystemC thread semantic. For illustration, Fig. 3 shows a SystemC thread (from the PLIC TLM peripheral) called *run* and Fig. 4 the resulting thread function after the translation process. The translated function consists of a header (Lines 15-27) and body (Lines 29-46) part. The header consists of goto statements to dispatch execution according to the context switch semantic. The current position in the thread function is stored in the newly introduced static `position` variable, which is an enum of type `Label` (Line 20). A label is provided for the first execution (`init`) and each wait function call (`lbl1` in this example). The body is a copy of the SystemC thread body where each wait function is annotated with appropriate context switch logic. It saves the current position (Line 33) before exiting the function (Line 34). A corresponding label is added for this position (Line 18). To support the translation process we developed a Python script that automates these steps for the DUV threads.

```
1  void run() {
2    while (true) {
3      sc_core::wait(e_run);
4      for (unsigned i=0; i<NumberCores; ++i) {
5        if (!hart_eip[i]) {
6          if (hart_has_pending_enabled_interrupts(i)) {
7            hart_eip[i] = true;
8            target_harts[i]->trigger_external_interrupt();
9          }
10       }
11     }
12   }
13 }
```

Figure 3: Original SystemC `run` process of the PLIC from the open source RISC-V VP. The `e_run` event is used for synchronization with a new incoming interrupt. The function on Line 6 implements the priority calculation.

```
void run() {
  //--[ header begin ]-----
  enum class Label {
    init,
    lbl1,
  };
  static Label position = Label::init;
  switch (position) {
    case Label::lbl1:
      goto LBL1;
    default:
      break;
  }
  //--[ header end ]-----

  //--[ body begin ]-----
  while (true) {
    // context switch (i.e. wait) transformation
    sc_core::wait(e_run);
    position = Label::lbl1;
    return;
LBL1:
    // unmodified logic of the original run thread
    for (unsigned i=0; i<NumberCores; ++i) {
      if (!hart_eip[i]) {
        if (hart_has_pending_enabled_interrupts(i)) {
          hart_eip[i] = true;
          target_harts[i]->trigger_external_interrupt();
        }
      }
    }
  }
  //--[ body end ]-----
}
```

Figure 4: Translated SystemC `run` process of the PLIC.

4.3 Peripheral Kernel (PK)

The PK is designed to be used as a drop-in replacement for the actual SystemC kernel. Fig. 5 shows an overview of the PK architecture and integration. It consists of a SystemC compatible library (top left of Fig. 5), matching wrapper macros (top of Fig. 5), and the PK scheduler (bottom left of Fig. 5) itself. As SystemC modules are designed to be modular and interact with the environment via defined functions and interfaces, our PK library can connect to these with custom, lightweight, SystemC classes that the DUV in the testbench will link to. Symbolic execution engines typically save and re-start the execution context of individual branches of the program, so our slimmed down PK library enables faster spawning of states. Especially the `sc_time` calculation routines need to be re-designed to use integer arithmetic wherever possible, to both speed up the symbolic execution and expand the possibilities for symbolic propagation. This is necessary, as KLEE currently does not support floating-point operations and concretizes these values.

As in SystemC, macros like `SC_HAS_PROCESS()` are used to register threads or processes to the simulation context of our PK scheduler. The scheduler keeps track of waiting processes, scheduled events and the simulation time. E.g., when a translated process waits for a specified time or an event, it will be placed into a *wakelist*. These waiting processes are managed in a sorted list. Every simulation step advances the global time by the maximum amount possible without skipping a waiting event, calling all threads that are scheduled for that time. As the SystemC scheduler is non-deterministic [21], our PK scheduler does not need to incorporate a special order within multiple threads waiting for the same simulation event.

In summary, the PK is a lightweight implementation focusing on relevant interfaces and integrating well-designed and optimized data-structures for SystemC process scheduling. It serves as foundation to enable an efficient symbolic verification process.

5 EXPERIMENTS

We have implemented our approach for TLM peripheral verification. For evaluation purposes we consider the PLIC from the open source

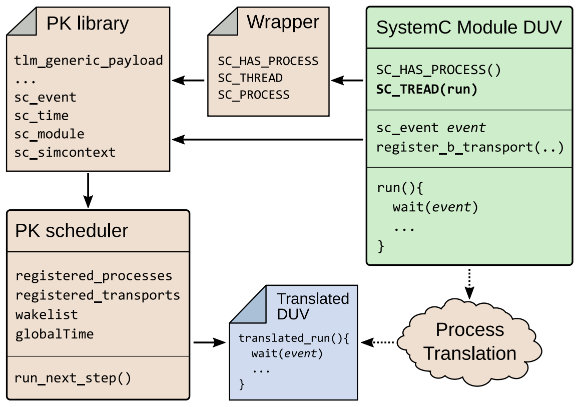

Figure 5: PK architecture overview with different interfaces for connecting to the (translated) DUV.

RISC-V VP which is available on GitHub[4]. In particular, we use the FE310 configuration of the PLIC[5] which is based on the respective FE310 SoC from SiFive [1]. Implementation-wise, this PLIC uses a dynamic, synchronous `run`-method that is sensitive to an `sc_event` which in turn is triggered when new interrupts arrive.

For evaluation, we created a set of symbolic unit tests to assess the PLIC against behaviour, timing, and conformance to interface specifications. In addition to testing the original PLIC with SystemC 2.3.3 and our PK, we also performed a fault-injection evaluation to further demonstrate the ability of our approach in finding intricate TLM peripheral bugs very efficiently. All experiments have been performed on a Linux Fedora 31 with an Intel Xeon 5122 with 3.6 GHz. We use KLEE in version 2.2 with the SMT solver STP.

In the following, we first describe our symbolic tests (Section 5.1). Then, we present the obtained results in testing the original PLIC (Section 5.2) as well as the PLIC with faults injected (Section 5.3).

5.1 Tests

In total, we have created five symbolic tests. Each test feeds symbolic input data through the standard TLM interface in order to access the TLM registers of the PLIC, or triggers interrupts for processing using a custom interface function. Assertions are placed in each respective test to check correct output behaviour and (internal) state changes of the PLIC. In addition, KLEE also searches for generic errors such as buffer overflows or null pointer dereferences.

In the following, we provide more details on five symbolic tests chosen to verify the sanity of the in- and output interface and the interrupt sequence assumption mentioned in Section 3.2.

T1 performs a basic interaction test. It triggers a symbolic interrupt and checks if the correct interrupt is fired within the specified latency, the corresponding `pending interrupt`-bit is set, claimable through a TLM transaction, and cleaned up afterwards.

T2 performs an interrupt sequence test. For illustration purposes, an excerpt of this test is shown in Fig. 6. It configures two symbolic (but different) interrupt lines (Lines 55-61) with symbolic priorities (Lines 63-64) and triggers them simultaneously in zero (simulation) time (Lines 66-67). After that preparation, it advances the time to the next event and checks if the interrupt with the higher priority was fired first (Line 78). If they have the same priority, the one with lower interrupt ID shall fire first. The test goes then on to check the second,

[4]https://github.com/agra-uni-bremen/riscv-vp
[5]FE310 PLIC: one HART, 51 interrupt sources with 32 priority levels.

Table 1: Test results for the original PLIC. For a *Fail*ed result, the number of detected failures is given in parentheses.

Test	Result	# Exec. Instr.	Time [s]	Paths	Solver
T1	Fail (1)	4 330 418	1293	64	98.02 %
T2	Pass	8 975 783	78755	3162	98.82 %
T3	Pass	7 027 481	66576	967	98.62 %
T4	Fail (3)	38 062 265	67	1168	74.17 %
T5	Fail (4)	102 992 556	93383	62 017	97.58 %

lower prioritized interrupt for integrity, which is omitted in this listing for readability reasons.

T3 performs an interrupt masking test. It configures a symbolic interrupt line with a symbolic priority and sets the *consider_threshold* to a symbolic value. It checks if the interrupt is only fired if its priority is both not zero and above the configured threshold.

T4 performs a TLM read interface test. It triggers an interrupt and starts a TLM *read*-transaction at a symbolic address using a symbolic length parameter. This test allows to check that the TLM periperhal can handle generic TLM read transactions and is not missing the handling of specific address ranges.

T5 is similar to T4 but performs a TLM write interface test. It also triggers an interrupt but then starts a TLM *write*-transaction at a symbolic address using a symbolic length parameter and writes up to 1000 bytes of symbolic data.

```
void functional_test_itr_priority(
  PLIC<1, numberInterrupts, maxPriority>& dut) {

  Interrupt_target hart(dut); // mock-up hart
  //interrupt line plic -> hart
  dut.target_harts[0] = &hart;

  uint32_t i = klee_int("i_interrupt");
  uint32_t j = klee_int("j_interrupt");

  // generate two valid different interrupt ids
  klee_assume(i < numberInterrupts && i > 0);
  klee_assume(j < numberInterrupts && j > 0);
  klee_assume(i != j);

  uint32_t lower_itr = i < j ? i : j;
  uint32_t highr_itr = i > j ? i : j;

  dut.trigger_interrupt(i);
  dut.trigger_interrupt(j);

  pkernel_step(); //advance time to next event

  // PLIC should have triggered an external interrupt
  assert(hart.was_triggered);

  // Is correct Interrupt claimable?
  uint32_t first_itr = hart.claim_interrupt();

  // Was the itr with the highest prio chosen first?
  assert(first_itr == lower_itr);

  assert(hart.was_cleared &&
  "Interrupt␣was␣not␣cleared␣after␣claim");

  pkernel_step(); //advance time to next event
  [...]
}
```

Figure 6: Part of the *interrupt priority test* (T2). This test contains multiple logic checks in the form of assertions.

5.2 Test Results: Original PLIC

Table 1 shows an overview on the test results for the original PLIC. The first column reports the performed test. The second column provides the test result. Each test can either *Pass* with no errors or *Fail* with at least one detected error. In case of a *Fail*, the number of detected errors by the respective test is given in parantheses. Please note, KLEE does not terminate after the first error is found but completes the symbolic state space exploration[6]. The next column *# Exec. Instr.* provides the overall number of executed LLVM bytecode instructions. The remaining columns show the total execution time in seconds (column: *Time*), the number of explored symbolic execution paths by KLEE (column: *Paths*) and how much of the overall execution time was spent in the SMT solver engine of KLEE to process SMT queries (column: *Solver*). It can be observed that the solver time vastly dominates the overall execution time in most tests. Only in T4 the solver queries are less complex performance-wise, resulting in a (symbolic) execution speed of up to 568 thousand instructions per second. The overall runtime varies between 67 seconds for T4 and around 26 hours for T5. Please note that this is the time required to perform the complete state space exploration, errors are typically found much faster, which we will discuss further in Section 5.3. When using the normal SystemC kernel, the initialization phase could be completed, but right at the first scheduling event, KLEE crashed with a segmentation fault right after a `mprotect()` syscall in the quickthreads implementation. Even after manually patching the kernel without this syscall, a successful context switch could not be performed, rendering this approach unsuccessful.

Based on our five tests, we found six errors in total. We describe them in the following:

F1 is a forgotten assertion in the `trigger_interrupt` routine. This assertion checks if the passed interrupt id is valid; i.e. between one and the maximum number of interrupts. However, this assertion throws an unhandled error that terminates the program which does not fit into a production grade environment. Also, when built in release mode, such assertions would not be checked and thus the program would produce a segmentation fault.

F2 describes a failed assertion checking the 4-byte alignment of a TLM register access. The correct way to handle failed assertions would be to return a TLM error state instead of terminating the program. This way, a transaction initiator like a processor can handle this with a correct exception handler.

F3 defines a failed assertion, similarly to F2, that checks the existence of a TLM register mapping that can handle the required address.

F4 characterizes a failed assertion, similarly to F2, checking the TLM target register is registered as writeable in case of a write transaction.

F5 is an unhandled memory access in which a TLM read transaction was accepted by a register mapping if the address matched a register with a 4-byte aligned transaction size, that could exceed the actual register boundaries. This leads to a `memcopy` with the source exceeding valid memory addresses.

F6 labels a failed assertion inside the TLM transport register access callback that was previously thought never to be false. In this case, the address was set to the interrupt `claim_response` register. Normally, a interrupt target writes to the register only after being notified. In this case however, the test initiated the transaction just after triggering the interrupt before the periodic PLIC thread was scheduled. This race condition was previously not found in normal operation because of the high PLIC thread frequency compared to the processor.

We found **F1** with T1; **F2** to **F4** with T4; and **F3** to **F6** with T5. In the following, we provide more details on how fast each error was found and we present results on finding additional injected faults that represent other common TLM peripheral errors.

5.3 Test Results: PLIC with Injected Faults

For further evaluation purposes, we injected six additional common (TLM peripheral) bugs into the PLIC: **IF1** to **IF6**. These include off-by-one faults (**IF1**, **IF6**), selectively dropping functional parts (**IF2**, **IF4**, **IF5**) and a race-condition (**IF3**). Of these, **IF1**, **IF3** and **IF6** have

[6]Only the single execution path that triggers the error is terminated.

Table 2: Overview on how fast the errors in the original PLIC (F1 to F6) and the PLIC with injected faults (IF1 to IF6) have been found by the respective tests. The runtime is given in minutes and rounded to the next highest integer.

	F1	F2	F3	F4	F5	F6	IF1	IF2	IF3	IF4	IF5	IF6
T1	1m	–	–	–	–	–	1m	3m	–	19m	4m	–
T2	–	–	–	–	–	–	–	60m	24h	–	105m	–
T3	–	–	–	–	–	–	–	–	–	–	–	7m
T4	–	1m	1m	1m	–	–	–	–	–	–	–	–
T5	–	–	1m	1m	16m	147m	–	–	–	–	–	–

been present in earlier versions of the PLIC, as can be observed in the GitHub logs. In the following, we first provide more details on these six bugs and then show how fast they are detected with our approach:

IF1 changes a check for the highest allowed interrupt number from `irq_id < NumberInterrupts` to `irq_id <= NumberInterrupts`, resulting in a buffer overflow in the array storing pending interrupts.

IF2 explicitly drops the notification of interrupts with the id 13 after writing to the correct pending interrupt register.

IF3 skips a necessary re-trigger for another simultaneously waiting interrupt after claiming the first one. This behavior is particularly hard to debug without well-suited unit tests.

IF4 artificially increases the event notification for the main thread if interrupt number is over 32. This shall emulate an error or misspecification in the timing model of the DUT.

IF5 returns the interrupt clear routine early if a specific interrupt is being cleared.

IF6 originates in a misinterpretation of the specification that checks if a pending interrupt priority is greater or equal to the configured threshold, while it shall be strictly greater.

Table 2 shows how fast the errors in the original PLIC (F1 to F6) and the PLIC with injected faults (IF1 to IF6) have been found by the respective tests. It can be observed that all original bugs are found in less than 3 hours with most bugs being found in just a few minutes or even less than a minute. The efficiency can be explained by Klee's symbolic exploration heuristics, which attempt to solve the most promising paths first and by tracking extensive symbolic constraints among these paths. The results demonstrate the effectiveness of our approach in finding relevant bugs in real-world TLM peripherals quickly.

6 CONCLUSION

This paper proposed an effective approach for verification of real-world SystemC TLM peripherals by using modern C++ symbolic execution tools. The foundation of our approach is a lightweight PK that acts as drop-in replacement for the SystemC kernel and is tailored for enabling the symbolic execution of TLM peripherals. The PK combines optimized data structures with a simplified function-based scheduling mechanism that relies on a thread to function transformation process. As a case-study, we reported verification results for a RISC-V specific PLIC that is used in an open source virtual prototyping environment for the SiFive FE310 SoC. We found new previously unknown bugs in the PLIC and also demonstrate by means of fault-injection that other intricate bugs can be detected very quickly using Klee, a state-of-the-art symbolic execution engine for C/C++. To stimulate further research we will make our PK together with the experimental setup available as open source.

For future work, we plan to investigate additional optimizations of our PK to further boost symbolic execution performance; and to evaluate our approach, beyond TLM peripherals, both for verification of other SystemC IP components such as a co-processor and the feasibility to verify whole SystemC projects with a high number of individual components.

ACKNOWLEDGMENTS

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project VerSys under contract no. 01IW19001, within the project Scale4Edge under contract no. 16ME0127 and by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

REFERENCES

[1] 2020. *SiFive FE310-G000 Manual.* Retrieved 2020-09-17 from https://sifive.cdn.prismic.io/sifive%2F500a69f8-af3a-4fd9-927f-10ca77077532_fe310-g000.pdf
[2] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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* (San Diego, California) *(OSDI'08)*. USA, 209–224.
[3] C. Chou, C. Chu, and C. Huang. 2013. Conquering the Scheduling Alternative Explosion Problem of SystemC Symbolic Simulation. In *ICCAD*.
[4] Chun-Nan Chou, Yen-Sheng Ho, Chiao Hsieh, and Chung-Yang Huang. 2012. Symbolic model checking on SystemC designs. In *DAC*. 327–333.
[5] Alessandro Cimatti, Iman Narasamdya, and Marco Roveri. 2013. Software Model Checking SystemC. *TCAD* 32, 5 (2013), 774–787.
[6] Philippe Coussy, Andres Takach, Michael McNamara, and Mike Meredith. 2010. An Introduction to the SystemC Synthesis Subset Standard. 183–184. https://doi.org/10.1145/1878961.1878993
[7] D. Große, H. M. Le, and R. Drechsler. 2010. Proving Transaction and System-level Properties of Untimed SystemC TLM Designs. In *MEMOCODE*. 113–122.
[8] P. Herber, J. Fellmuth, and S. Glesner. 2008. Model Checking SystemC Designs Using Timed Automata. In *CODES+ISSS*. 131–136.
[9] Paula Herber, Marcel Pockrandt, and Sabine Glesner. 2011. Transforming SystemC Transaction Level Models into UPPAAL timed automata. In *Ninth ACM/IEEE MEMPCODE 2011*. 161–170.
[10] P. Herber, M. Pockrandt, and S. Glesner. 2015. STATE – A SystemC to Timed Automata Transformation Engine. In *HPCC-CSS-ICESS*.
[11] Vladimir Herdt, Hoang M. Le, Daniel Große, and Rolf Drechsler. 2019. Verifying SystemC using Intermediate Verification Language and Stateful Symbolic Simulation. *IEEE Transactions on Computer Aided Design of Circuits and Systems* 38, 7 (July 2019), 1359–1372.
[12] Vladimir Herdt, Hoang M. Le, Daniel Große, and Rolf Drechsler. 2016. Compiled Symbolic Simulation for SystemC. In *ICCAD*. 52:1–52:8.
[13] D. Karlsson, P. Eles, and Z. Peng. 2006. Formal Verification of Systemc Designs Using a Petri-net Based Representation. In *DATE*. 1228–1233.
[14] D. Kroening and N. Sharygina. 2005. Formal Verification of SystemC by Automatic Hardware/Software Partitioning. In *MEMOCODE*.
[15] Hoang M. Le, Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2016. Towards formal verification of real-world SystemC TLM peripheral models - a case study. In *2016 DATE*. 1160–1163.
[16] Timm Liebrenz, Verena Klös, and Paula Herber. 2017. Automatic Analysis and Abstraction for Model Checking HW/SW Co-Designs Modeled in SystemC. *Ada Lett.* 36, 2 (May 2017), 9–17.
[17] Bin Lin, Zhenkun Yang, Kai Cong, and Fei Xie. 2016. Generating high coverage tests for SystemC designs using symbolic execution. In *2016 21st ASP-DAC*. 166–171. https://doi.org/10.1109/ASPDAC.2016.7428006
[18] M. Moy, F. Maraninchi, and L. Maillet-Contoz. 2005. LusSy: An open tool for the analysis of systems-on-a-chip at the transaction level. *ACSD* 10, 2-3 (2005), 73–104.
[19] OSCI 2009. *OSCI TLM-2.0 Language Reference Manual.* OSCI.
[20] Marcel Pockrandt, Paula Herber, and Sabine Glesner. 2011. Model checking a SystemC/TLM design of the AMBA AHB protocol. In *2011 9th IEEE Symposium on Embedded Systems for Real-Time Multimedia*. 66–75.
[21] Std. 1666 2011. *IEEE Standard SystemC Language Reference Manual.* Std. 1666.
[22] D. Tabakov, M.Y. Vardi, G. Kamhi, and E. Singerman. 2008. A Temporal Language for SystemC. In *FMCAD*. 1–9.
[23] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi. 2007. A SystemC/TLM Semantics in Promela and Its Possible Applications. In *SPIN*.
[24] M. Y. Vardi. 2007. Formal techniques for SystemC verification. In *DAC*.
[25] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. 2014. The RISC-V Instruction Set Manual.