An Effective Methodology for Integrating Concolic Testing with SystemC-based Virtual Prototypes

Sören Tempel[1] Vladimir Herdt[1,2] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
tempel@uni-bremen.de, vherdt@uni-bremen.de, drechsler@uni-bremen.de

***Abstract*—In this paper we propose an effective methodology for integrating *Concolic Testing* (CT) with SystemC-based Virtual Prototypes (VPs) for verification of embedded SW binaries. Our methodology involves three steps: 1) integrating CT support with the *Instruction Set Simulator* (ISS) of the VP, 2) utilizing the standard TLM-2.0 extension mechanism for transporting concolic values alongside generic TLM transactions, and 3) providing lightweight concolic overlays for SystemC-based peripherals that enable non-intrusive CT support for peripherals and thus significantly reduce the CT integration effort. Our RISC-V experiments using the RIOT operating system demonstrate the effectiveness of our approach.**

I. Introduction

Embedded systems integrate numerous peripherals alongside the processor on the *Hardware* (HW) side and extensively rely on embedded *Software* (SW) for configuration and complex functionality. Verification of the embedded SW is crucial to avoid errors and mitigate the risk of security vulnerabilities.

Therefore, mainly simulation-based methods are employed that leverage *Virtual Prototypes* (VPs) for SW execution early in the design flow [1], [2]. VPs are essentially abstract models of the entire *Hardware* (HW) platform and predominantly created in SystemC TLM (*Transaction-Level Modeling*) [3]. Beside the *Instructions Set Simulator* (ISS), which is an abstract model of the processor, peripherals play a very important role for every platform. VPs are designed from the ground up to represent the whole HW platform and provide an industrial proven solution for analysis of complex HW/SW interactions. However, a comprehensive simulation-based verification requires integration of sophisticated test generation techniques.

Concolic Testing (CT) is such a technique that has been shown very effective in the SW domain for increasing the SW test coverage and finding intricate bugs, e.g. [4], [5]. CT works by successively exploring new paths through the SW by solving symbolic constraints that are tracked alongside the concrete execution. This combination of symbolic with concrete execution enables comprehensive testing of the SW by exploring a large set of different program paths very efficiently. Integrating CT with a VP-based simulation would enable comprehensive and accurate testing of embedded SW binaries that interact extensively with HW peripherals.

Contribution: We propose an effective methodology for such an integration. Essentially, our methodology involves three steps:

1) Integrating special CT support with the ISS and memory, both of which are generic components and thus can be reused henceforth to build VPs for different HW platforms.
2) Utilizing the standard TLM-2.0 extension mechanism for transporting concolic values alongside generic TLM transactions. This is very important for CT to seamlessly integrate with the existing TLM communication infrastructure, which is at the heart of every VP.
3) Providing lightweight CT overlays for SystemC-based peripherals. Overlays intercept TLM transactions to process the concolic extension before and after the transaction is routed to the existing SystemC-based peripheral. This significantly reduces the CT integration effort, as the existing SystemC-based peripherals can be reused.

Our RISC-V experiments using the RIOT operating system demonstrate the effectiveness of our approach in analyzing real-world embedded applications. Our methodology enables us to rapidly support new platforms for CT by reusing peripherals[1].

II. Related Work

CT and symbolic execution are very active research areas. They have been used for SW verification, based on intermediate representations [6] and recently at the binary level [4], [5], [7], as well as HW verification at RTL [8], [9] and VP level [10], [11].

To deal with embedded SW, specialized approaches are required that support complex HW/SW interactions. This gave rise to a number of approaches that mainly differ on how the underlying HW is being integrated. [12] uses virtual peripheral models manually extracted from QEMU. [13] on the other hand integrates HW Verilog models. [14] introduced instruction level abstraction to formally model SW-visible behavior of HW. [15] provides an MSP430-based symbolic execution environment based on KLEE. [16] introduces an assembly to LLVM-IR lifting approach. [17] allows hybrid binary CT with physical devices. [18], [19] use SW models extracted from SystemC-based peripherals.

To the best of our knowledge, an approach that directly integrates CT with SystemC-based VPs for the purpose of embedded SW verification is not available.

III. Concolic Testing with SystemC-based VPs

This section presents our proposed methodology on integrating CT with SystemC-based VPs. We start by an overview on CT in a VP-based setting (Section III-A), then present more details on the actual integration with SystemC TLM (Section III-B) and finally discuss our proposed CT overlays in more detail (Section III-C).

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0127 and within the project VerSys under contract no. 01IW19001, and by the German Research Foundation (DFG) as part of the Collaborative Research Center (Sonderforschungsbereich) 1320 EASE – Everyday Activity Science and Engineering, University of Bremen (http://www.ease-crc.org/) in subproject P04.

[1]Visit http://www.systemc-verification.org/risc-v to find our most recent RISC-V related approaches.

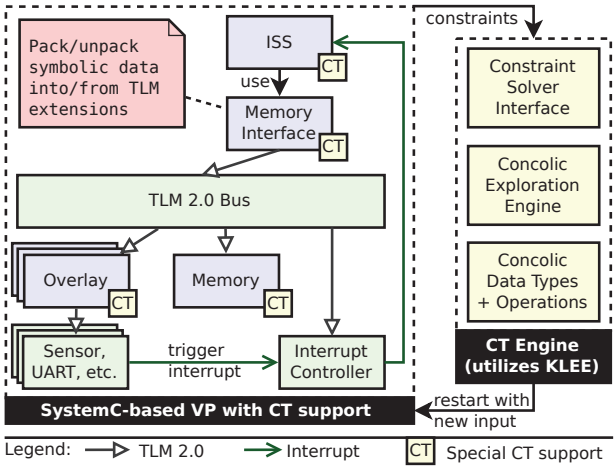

Fig. 1. Overview on our CT integration with VPs

A. Overview Concolic Testing

Fig. 1 shows an overview on our CT integration for VPs. Essentially, it consists of two parts: 1) the SystemC-based VP with CT extensions (left side), and 2) the CT engine (right side).

A central component of the CT engine is the exploration engine. It is the main entry point and orchestrates CT by successively generating new inputs for the testing process. Each input is processed by the VP. That means the VP re-executes the embedded SW binary from the beginning and provides the input to the SW (via input peripherals such as a sensor or UART). The VP tracks symbolic (input) constraints alongside the concrete execution. The collected constraints are solved by the exploration engine to generate new inputs that will drive the execution towards different paths in the embedded SW binary. Numerous search heuristics have been proposed in the literature to guide the exploration faster towards interesting cases. We effectively implemented a standard approach based on *dynamic symbolic execution*, that flips branch conditions to generate new inputs and utilizes *address concretization* in case of symbolic memory indices [20].

The CT engine provides concolic data types and operations for the VP as well as a solver interface to reason about constraints. We have build this part of our engine on top of KLEE [6], a state-of-the-art symbolic execution engine, by providing thin wrapper around the respective KLEE APIs. Technically, a concolic data type is a pair of (mandatory) concrete value and (optional) symbolic expression.

So far we conceptually follow the existing CT approaches. The main novelty lies in our integration of CT with SystemC TLM, which we discuss in the following.

B. Integration with SystemC TLM based VPs

In order to track symbolic constraints alongside the VP-based execution, we need to integrate that support into the ISS, memory, TLM bus and peripherals. The memory is a generic component that is necessary in any VP. Similarly, the ISS needs to be provided once per supported processor ISA (*Instruction Set Architecture*) and can be re-used henceforth. Thus, investing more time to build a highly specialized CT integration for the memory and ISS is reasonable. However, peripherals are often different between embedded systems and thus require special consideration to reduce the CT integration effort and subsequent maintenance effort due to providing duplicate models (for a concrete execution and for a symbolic execution). Therefore, we propose a methodology to build special CT models for the memory and ISS but try to facilitate the integration of new peripherals with CT as much as possible.

To enable CT support in the memory we essentially replace the native with concolic data types. Similarly, in the ISS we modify the register bank and execution unit to use concolic instead of native data types and operations. Many parts of the ISS, such as the decoder logic, processing of incoming interrupts or the debugger interface can be re-used without modification. In addition, we adapt the memory interface, that translates load/store instructions to TLM transactions before the bus system, to pack and unpack symbolic constraints into a TLM extension and pass it alongside the normal TLM transaction. This is a very important design choice that naturally fits into the TLM idea of interoperability. The peripherals are unaware of the TLM extensions and simply process the normal TLM transaction. For each peripheral we introduce an optional CT overlay that is placed between the bus and the peripheral to provide non-intrusive CT support for the peripheral. The overlay can modify the TLM extension before and after routing the TLM transaction to the peripheral. In addition, it is possible to start with a very simple (or even none if the device does not need to handle symbolic constraints) overlays and refine them on demand as necessary to support the embedded SW. We further illustrate the overlay idea and discuss it in more detail in Section III-C.

A last challenge that needs to be addressed to integrate CT is restarting the VP-based simulation which in turn is based on SystemC. The open source reference implementation of SystemC does not provide such a restart mechanism. However, we were able to essentially emulate a restart behavior with the following procedure: Besides the usual *sc_main* function we provide a custom *main* function from which we call the SystemC *sc_elab_and_sim* function for each CT input. In addition, we reset the global SystemC simulation context (*sc_core::sc_curr_simcontext*) before each call to ensure that a new SystemC simulation context is created[2]. A new VP instance is initialized and executed per simulation context.

C. Peripheral Overlay Example

This section discusses a theoretical example scenario to illustrate the design and refinement of SystemC peripheral overlays.

As an example peripheral we will consider a SiFive UART (e.g. used on the FE310-G000 SoC of the HiFive1 board, see [21] Chapter 17). Essentially, it provides memory mapped registers to receive and transmit data as well as configuration registers to enable interrupts, configure the watermark level (how many elements should be received before triggering an interrupt) and other important functionality. The UART provides a small buffer to store received and process transmitted characters. To receive data, the *rxdata* register is read. It returns a 32 bit value where the highest bit indicate if the UART is empty (the bit is set) and the lower 8 bit are the received character (if not empty).

Fig. 2 shows a basic embedded SW driver, that copies data from an RX to a TX UART, in three different variants from simple to more sophisticated. The driver function is interrupt driven and triggered whenever the RX UART receives new data. Fig. 3 shows the

[2]This approach currently relies on implementation details and is not standard-conform. Apart from the simulation context, SystemC contains additional global variables which cannot be properly reset at this time. Even though we are currently unaware of functional issues with our current reset approach, we are striving towards a reduction of global state in SystemC. See https://github.com/accellera-official/systemc/issues/8 for more information.

```
1  void copy_driver() {
2    //-------------------------[ VARIANT-1 ]---------
3    char c = read_rx_uart();
4    write_tx_uart(c);
5    //-------------------------[ VARIANT-2 ]---------
6    char c = read_rx_uart();
7    assert (c != '#');  // assume sender filters '#'
8    write_tx_uart(c);
9    //-------------------------[ VARIANT-3 ]---------
10   while (!empty_rx_uart()) {
11     char c = read_rx_uart();
12     assert (c != '#'); // assume sender filters '#'
13     write_tx_uart(c);
14   }
15   //-------------------------[ END ]---------------
16 }
```

Fig. 2. Example embedded SW to illustrate overlay design and refinement

corresponding (SystemC concolic) overlay for the UART in order to support the respective variant of the SW driver. For clarity and brevity we employ a slight pseudo-code notation (in particular with respect to the *assume* function that adds symbolic constraints) and omit irrelevant details (such as the constructor). Our methodology idea is to start with a coarse overlay and refine it on demand as necessary to support the embedded SW.

The first SW variant simply copies a single character from RX to TX UART (Line 3–4 in Fig. 2). To support such a scenario the overlay simply returns an arbitrary value on a read access (Line 10 in Fig. 3). In the next variant we assume a scenario where the received characters are constrained according to an environment model, i.e. we assume that the character '#' is never received. With variant 1 of the overlay, the variant 2 SW would hit a spurious error, because the assertion in Line 7 should not fail under the current environment assumptions[3]. To support such a scenario, the outgoing UART data can be constrained accordingly by the overlay (Line 13) as shown in variant 2. The SW variant 3 adds another layer of complexity (Line 10–14). Now, the copy process continues until the RX UART is empty (which makes this driver more generic)[4]. Using variant 2 of the overlay, the SW could spin infinitely in the copy loop because the empty bit is an unconstrained symbolic independent of the actual UART status. This abstraction is fixed by the variant 3 refinement (Line 17–21) that also demonstrates the benefits of having access to the actual peripheral from the overlay.

This last variant covers the main functionality from the receive part of the UART which is the interesting functionality with regard to a symbolic evaluation. The remaining functionality is mainly responsible to configure the UART and process interrupts. Thus, the concolic overlay can be very compact and yet enable full CT of embedded SW binaries that extensively interact with the UART. Furthermore, peripherals and overlays can be developed side by side which makes maintanance and testing much more easy (the peripherals are still normally used in the concrete VP and bug fixes are immediately available).

IV. Evaluation

We evaluate our methodology based on the open source RISC-V VP that is implemented in SystemC TLM [22], [23] and available at GitHub [24]. In particular, we use the HiFive1 configuration of the VP. The HiFive1 is a RISC-V board from SiFive that features

[3]A rather interesting fact is that using an abstract overlay can even lead to the detection of SW errors that would otherwise be masked by the concrete peripheral implementation.

[4]We assume in this illustration scenario that the TX UART has a sufficiently large buffer and processes incoming characters fast enough.

```
1  class UARTOverlay : public sc_core::sc_module {
2    UART &uart; // reference to concrete UART
3    // ...omit constructor...
4    void transport(tlm::tlm_generic_payload &trans,
5                   sc_core::sc_time &delay) {
6      // process symbolic extension
7      auto addr = trans.get_address();
8      if (addr == RX_ADDR) {
9  //-------------------------[ VARIANT-1 ]----------
10       auto data = SymbolicUint32();   // any value
11 //-------------------------[ VARIANT-2 ]----------
12       auto data = SymbolicUint32();   // any value
13       assume ((data & 0xff) != '#');  // avoid '#'
14 //-------------------------[ VARIANT-3 ]----------
15       auto data = SymbolicUint32();   // any value
16       assume ((data & 0xff) != '#');  // avoid '#'
17       if (uart.empty()) {
18         assume (data & (1 << 31));    // empty
19       } else {
20         assume (!(data & (1 << 31))); // not empty
21       }
22 //-------------------------[ END ]----------------
23       // pack the symbolic value into a TLM extension
24       auto ext = new SymbolicExtension(data);
25       trans.set_extension(ext);
26     }
27     // call the normal UART
28     isock->b_transport(trans, delay);
29   }
30 };
```

Fig. 3. Example overlay to illustrate overlay design and refinement

a 32 bit RISC-V processor alongside a set of essential peripherals, including UARTs (receive and transmit), interrupt controller and GPIO [25]. It is particularly well suited for small embedded applications that are built on top of lightweight operating systems such as FreeRTOS, Zephyr or RIOT. The HiFive1 configuration of the VP is binary compatible to the real board, i.e. can execute SW binaries compiled for the real board.

Following our methodology we modified the existing ISS and memory in the VP to operate on concolic data types and integrated our CT engine as described in Section III. Peripheral overlays will be added and refined on demand (though overlays can obviously be reused once they have been implemented and fully refined for a specific peripheral). As a case-study we consider an example application for the HiFive1 that queries and processes sensor data on top of the RIOT operating system. In the following we present more details on the application (Section IV-A), then discuss our test setup (Section IV-B) and results (Section IV-C), and conclude with an outlook for future work (Section IV-D).

A. RIOT-based Example Application

As an example we consider an interrupt driven application that periodically processes sensor data. An interrupt is triggered when new sensor data is available. A producer-consumer scheme is employed by the application to process this data. For this purpose, two RIOT threads are employed which communicate with each other using RIOT's interprocess communication mechanism. The producer thread reads data from the sensor and passes received data to the consumer thread which processes the data and ultimately writes it to the UART. Assertions are used to perform sanity checks on the data, for instance to check that sensor values are within a certain expected range. Considering that RIOT is a multithreading operating system, we believe these kinds of producer-consumer patterns to be common in embedded applications utilizing RIOT. The compiled binary of this example application consists of 4061 assembler instructions.

TABLE I
EVALUATION RESULTS FOR DIFFERENT OVERLAY REFINEMENTS

iteration	#instrs	run-time	#paths	bug
I1	127,477	1.11 sec	1	SB
I2	1,370,415	13.58 sec	13	RB
I3	341,805,972	170 min	3270	-

B. Test Setup

To ensure termination of the testing process, because the application is interrupt driven and hence non-terminating, we bounded the number of processing iterations in the application. The testing process continues until all bugs are fixed. Since we start with coarse overlays that over-approximate the behavior of the real peripherals, spurios bugs are possible. In this case a refinement of the overlays is necessary.

For this example application, an overlay is only required for the sensor peripheral (it acts as the only input device). Overlays for the other HiFive1 peripherals are not necessary. This includes the UART, since it only acts as an output device, but also other peripherals which are accessed by RIOT during the operating system initialization phase. Furthermore, overlays are much more compact than the real peripheral and can be integrated non-intrusively as discussed in Section III-C. This is a major benefit of our approach, since we can leverage the power of the existing SystemC-based peripherals (which are integrated into full platforms such as the HiFive1 VP).

C. Test Results

Table I shows the results. The first column shows the test iteration. The next three columns show the number of executed instructions (column: #instrs), overall run-time (column: run-time) and number of concolic execution paths (column: #paths). Please note, by using symbolic expressions, CT can cover a large set of different inputs on a single path. The last column shows if a bug is found and if the bug is spurios (SB) or a real (RB). In total, three test iterations I1, I2 and I3 are performed until no more bug is detected (in I3). All experiments are performed on a Linux system with an Intel i7-8565U processor.

In **I1** the sensor overlay returns fully unconstrained data. While this abstraction is too coarse (since the real sensor filters data according to a configuration setting), we left it for demonstration purposes. As expected, a spurios bug is detected very fast, in fact on the very first path after executing around 127K instructions. We refined the sensor overlay accordingly to consider the filter setting of the real sensor, resulting in **I2**. Now a real application bug is detected (caused by a false assumption regarding potential values returned by the sensor in the application code) after 13 paths and around 1.4M instructions. It shows, that our approach can be very effective in finding bugs (proving correctness is more difficult, as it requires to explore all possible paths). For **I3** we fixed the aforementioned bug and discovered no further bugs. In total 3270 paths were explored in I3 with around 341M executed instructions in 170 minutes.

D. Outlook

Our experiments demonstrate the applicability and effectiveness of our proposed methodology in integrating CT with SystemC-based VPs to test real-world embedded applications. To further boost our methodology we plan to:

- Investigate automated methods to derive peripheral overlays from existing SystemC-based peripherals. A starting point might be a half-automated approach that leverages peripheral interface descriptions.
- Devise automated methods to refine overlays or localize the exact source of a spurios error (currently it is a manual approach though the existing VP-based debug infrastructure can be used).
- Investigate how to minimize the performance impact on restarting the SystemC-based simulation and consider parallelization or snapshoting to boost performance. Further, integrate advanced symbolic query optimizations.

REFERENCES

[1] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, March 2014.
[2] V. Herdt, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer, 2020.
[3] *IEEE Standard SystemC Language Reference Manual*, 2011.
[4] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *IEEE S & P*, 2012, pp. 380–394.
[5] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE S & P*, 2016.
[6] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008, pp. 209–224.
[7] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," in *ASPLOS*, 2011, pp. 265–278.
[8] S. Pinto and M. S. Hsiao, "RTL functional test generation using factored concolic execution," in *ITC*, 2017, pp. 1–10.
[9] A. Ahmed, F. Farahmandi, and P. Mishra, "Directed test generation using concolic testing on RTL models," in *DATE*, 2018, pp. 1538–1543.
[10] B. Lin, K. Cong, Z. Yang, Z. Liao, T. Zhan, C. Havlicek, and F. Xie, "Concolic testing of SystemC designs," in *ISQED*, 2018, pp. 1–7.
[11] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Verifying SystemC using intermediate verification language and stateful symbolic simulation," *TCAD*, vol. 38, no. 7, pp. 1359–1372, July 2019.
[12] A. Horn, M. Tautschnig, C. G. Val, L. Liang, T. Melham, J. Grundy, and D. Kroening, "Formal co-validation of low-level hardware/software interfaces," in *FMCAD*, 2013, pp. 121–128.
[13] R. Mukherjee, M. Purandare, R. Polig, and D. Kroening, "Formal techniques for effective co-verification of hardware/software co-designs," in *DAC*, 2017, pp. 35:1–35:6.
[14] B. Huang, S. Ray, A. Gupta, J. M. Fung, and S. Malik, "Formal security verification of concurrent firmware in SoCs using instruction-level abstraction for hardware," in *DAC*, 2018, pp. 91:1–91:6.
[15] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *USENIX Security*, 2013, pp. 463–478.
[16] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: System-wide security testing of real-world embedded systems software," in *USENIX Security*, 2018, pp. 309–326.
[17] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares," in *NDSS*, 2014.
[18] 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 *DAC*, 2019, pp. 188:1–188:6.
[19] V. Herdt, D. Große, and R. Drechsler, "RVX - a tool for concolic testing of embedded binaries targeting RISC-V platforms," in *ATVA*, 2020, pp. 543–549.
[20] R. Baldoni, E. Coppa, D. C. Delia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, May 2018.
[21] "SiFive FE310-G000 manual," https://sifive.cdn.prismic.io/sifive%2F500a69f8-af3a-4fd9-927f-10ca77077532_fe310-g000.pdf
[22] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Extensible and configurable RISC-V based virtual prototype," in *FDL*, 2018, pp. 5–16.
[23] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *JSA*, 2020.
[24] "RISC-V virtual prototype," https://github.com/agra-uni-bremen/riscv-vp
[25] "Hifive1," https://www.sifive.com/boards/hifive1