Concolic Testing of SystemC Designs

Bin Lin*, Kai Cong†, Zhenkun Yang†, Zhigang Liao‡, Tao Zhan§, Christopher Havlicek†, Fei Xie*
*Department of Computer Science, Portland State University, Portland, OR 97207, USA
{linbin, xie}@cs.pdx.edu
†Intel Corporation, Hillsboro, OR 97124, USA
{kai.cong, zhenkun.yang, christopher.havlicek}@intel.com
‡Virtual Device Technologies LLC, Portland, OR 97201, USA
zhigangliao@virtualdevicetech.com
§School of Computer Science and Engineering, NorthWestern Polytechnical University, Xi'an 710072, China
zhantao@nwpu.edu.cn

Abstract— SystemC is a system-level modelling language widely used in the semiconductor industry. SystemC validation is both necessary and important, since undetected bugs may propagate to final silicon products, which can be extremely expensive and dangerous. However, it is challenging to validate SystemC designs due to their heavy usage of object-oriented features, event-driven simulation semantics, and inherent concurrency. In this paper, we present CTSC, an automated, easy-to-deploy, scalable, and effective binary-level concolic testing framework for SystemC designs. We have implemented CTSC and applied it to an open source SystemC benchmark. In our extensive experiments, the CTSC-generated test cases achieved high code coverage, triggered 14 assertions, and found two severe bugs. In addition, the experiments on two designs with more than 2K lines of SystemC code show that our approach scales to designs of practical sizes.

Keywords— SystemC, concolic testing, code coverage, assertion-based verification, bug detection

I. Introduction

SystemC [1] is a system-level modelling language for systems that might be implemented in software, hardware, or combination of the two. A SystemC design is described as a set of modules that communicate through ports. A module includes at least one process that describes certain behaviors of a system. A module can also contain submodules to represent the hierarchy of a system. Processes in a design run concurrently. The SystemC scheduler has cooperative multitasking semantics that processes cannot be preempted. Each process yields when it runs to the end or to the predefined yield points, such as function `wait()` or its variations. The process is resumed by notification events to which it is sensitive.

SystemC has become a de-facto standard modelling language in the semiconductor industry. SystemC has been widely used for system-level modelling, architectural exploration, functional verification, and high-level synthesis [2]. It is critical to verify these high level SystemC designs during the system design life cycle, since undetected design errors may propagate to low-level implementations and become costly to fix. Bugs that remain uncovered in a final silicon product can be extremely expensive and dangerous, especially in safety-critical systems. This demands innovative approaches to SystemC validation.

There have been recent studies on formal verification of SystemC designs inspired by advances in the application of formal methods to software. However, it is very challenging to extend such methods to SystemC designs. Formal methods require formal semantics that describe the transition relation of a design. This is nontrivial for SystemC designs due to their heavy usage of object-oriented features, event-driven simulation semantics, and concurrency. Furthermore, state-space explosions limit the scalability of formal verification.

Dynamic validation, also known as the simulation-based approach, is the workhorse of SystemC validation [3]. SystemC simulation requires a set of concrete test cases. Generally, test cases for SystemC simulation are manually written, randomly generated or derived from symbolic execution [4]. Manual test writing requires the indepth knowledge of a design under validation (DUV), which is time-consuming, labor-intensive, and error-prone. Random testing, in contrast, is fast. However, many redundant test cases may be generated, which results in low code coverage and long simulation time. Symbolic execution can generate effective test cases and improve code coverage. Thus, symbolic execution has been widely used [5]–[8]. Unfortunately, for complex system-level designs in SystemC, traditional symbolic execution has its own limitations due to the path explosion problem.

Recently, concolic (a portmanteau of concrete and symbolic) execution that combines concrete execution and symbolic execution has achieved considerable success in both software and hardware domains [9]–[12]. Concolic execution runs a program by making input values symbolic in addition to concrete. The concrete execution part performs normal execution of the program. The symbolic execution part collects symbolic constraints over the symbolic inputs at each branch point along the concrete execution path. The collected constraints are negated one condition by one condition and sent to a constraint solver. If the solver can solve the negated constraints successfully, new test cases will be generated. Concolic execution can mitigate the path explosion problem of symbolic execution, as well as alleviating the redundancy problem of random testing. Thus, concolic execution has great potential to play an important role in validating SystemC designs.

In this paper, we present an effective concolic testing approach for SystemC designs. Central to this approach is CTSC

(Concolic Testing of SystemC), an automated, easy-to-deploy, scalable, and effective framework for SystemC validation.

We bring the idea of binary-level concolic execution into SystemC validation. CTSC works directly on binaries compiled from SystemC designs by linking the SystemC library [23]. Therefore, CTSC requires no translation of SystemC designs, no modelling of dependent libraries, and thus supports all SystemC features provided by the SystemC library.

We have implemented the proposed framework as a prototype tool. A SystemC design and its testbench that can be generated automatically are compiled into a binary with a standard compiler by linking the SystemC library. Then, the binary is executed with an initial test case by our binary-level concolic execution engine which generates new test cases. The newly generated test cases are used to validate the design directly with integration of assertion-based verification (ABV) techniques. A test report, which mainly indicates assertion violations and the statuses of test cases (failure or pass), is generated when the testing process terminates.

Furthermore, we have evaluated the effectiveness of CTSC on a SystemC benchmark [22]. The benchmark comprises a variety of application domains, such as security, CPU architecture, image processing, network, and digital signal processing (DSP). These designs cover most core features and data types of the SystemC language. Our experimental results demonstrate the effectiveness of our approach. It is able to achieve high code coverage, as well as detecting severe bugs. Moreover, the experiments on `RISC CPU` and `DES` illustrate that our approach scales to designs of practical sizes.

The rest of this paper is organized as follows. Section II reviews related work. Section III presents the key idea of our approach to SystemC validation using binary-level concolic testing. Section IV elaborates on the experiments that we have conducted and the evaluation results. We conclude this research and discuss future work in Section V.

II. Related work

There have been several attempts for SystemC formal verification. Scoot [24] extracts formal models from SystemC designs for formal analysis. However, it implemented a simplified library because the SystemC library [23] makes heavy usage of object-oriented features. STATE [14] translates SystemC designs into UPPAAL timed automata, which can be verified by the UPPAAL tool chain. KRATOS [15] translates SystemC designs to threaded C models, which are checked by combining an explicit scheduler and symbolic lazy predicate abstraction. Chou *et al.* [16] present a symbolic model checking technique that conducts reachability analysis on Kripke structures formulated from SystemC designs. SISSI [17] proposes an intermediate verification language for SystemC. Due to the complexity and characteristics of SystemC, it is challenging to describe the transition relation of a SystemC design directly using formal semantics. Thus, all these approaches translate SystemC designs into intermediate representations (IR), which are verified afterwards. However, these IRs usually represent only a subset of SystemC features.

There are also a handful of simulation-based approaches to SystemC verification. The SystemC verification library [23] provides APIs only for transaction-based verification using weighted, constrained, or unconstrained randomization techniques. Chen *et al.* [19] present an approach for generating register transfer level (RTL) test cases from TLM specifications. A SystemC TLM design is first translated into an IR that can represent only a subset of SystemC TLM. Our previous work, SESC [20], generates test cases for SystemC designs automatically using symbolic execution. Although SESC is able to achieve high code coverage, it is focused on high-level synthesizable subset of SystemC. Mutation testing has also been applied to SystemC verification [18]. However, for mutation testing, users may develop any number of mutants, and exercising these mutants is time consuming.

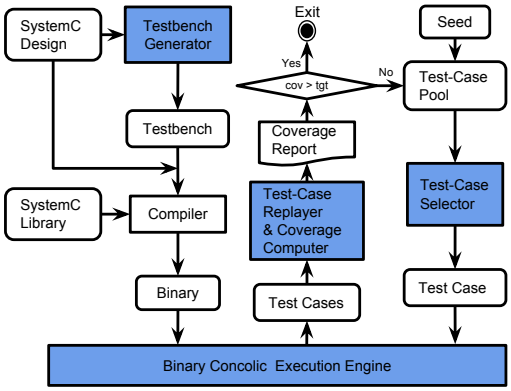

Fig. 1. Workflow for binary-level concolic testing of SystemC designs

III. Concolic testing of SystemC designs

In this section, we describe the key idea of our approach. Before discussing details, we first introduce the definition of a test case. A **test case** of a SystemC design, denoted as $\tau \triangleq I_1, I_2, \ldots, I_n$, is a sequence of inputs, such that input I_i $(1 \leq i \leq n)$ is applied to the design at clock cycle i, where I_i is a set of concrete stimuli corresponding to all input ports. Note that our approach determines whether a test case passes or fails according to whether it triggers an assertion, since our framework integrates the ABV techniques.

A. Workflow for Binary Concolic Testing of SystemC Designs

To validate SystemC designs effectively, we propose a framework using binary-level concolic testing. Figure 1 shows the workflow of our framework. It has four key steps: (1) testbench generation, (2) binary-level concolic execution, (3) test-case selection, and (4) testing with generated test cases. For a given SystemC design, its testbench is generated first and compiled with the design into a binary by linking the SystemC library. Then, the binary and an initial test case, called the *seed*, are fed to a binary concolic execution engine that includes two parts, concrete execution and symbolic execution. Concrete execution simulates a design concretely during which an execution trace is obtained. Symbolic execution exercises the trace symbolically to generate new test cases. Subsequently, a test case is selected from the newly generated test cases for a new iteration of concolic execution. Finally, the newly generated test cases are used to validate the SystemC design. This process repeats until a termination condition is achieved.

Algorithm 1 illustrates the validation process using concolic testing. For most complex system designs, due to path explosion, the common usage model for automated test generation by symbolic or concolic execution is to run for a fixed amount of time or run until a target coverage goal is reached. We also follow such a model. SC-CON-TESTING takes four parameters, a SystemC design duv, an initial test case $seed$, a target coverage tgt, and a time bound β, as inputs. The outputs are the generated test cases TC and achieved coverage cov.

Algorithm 1: SC-CON-TESTING(duv, $seed$, tgt, β)

```
1  TC ← {seed},  TCP ← {seed}
2  cov ← 0
3  while (TCP ≠ ∅) ∧ (cov < tgt) ∧ (time < β) do
4      τ ← TEST-CASE-SELECTOR(TCP)
5      TCP ← TCP \ {τ}
6      NTC ← CONCOLIC-EXECUTION(duv, τ)
7      foreach t ∈ NTC do
8          TEST-CASE-REPLAYER(duv, t)
9          TCP ← TCP ∪ {t}
10     TC ← TC ∪ NTC
11     cov ← COVERAGE-COMPUTER()
12 return TC, cov
```

Here, TC saves all generated test cases and TCP is a test-case pool accessed dynamically during the validation process. At the beginning, the concolic execution engine executes the design with the $seed$ (line 6). When the engine terminates, it generates new test cases that are saved in a temporary set, NTC. For each newly generated test case, our framework reruns it on the SystemC design (line 8) to look for unusual behavior, such as assertion violations, and generate information for computing coverage. Each test case that is replayed is added to TCP (line 9) for the next iteration. Subsequently, the newly generated test cases are added to TC (line 10), and COVERAGE-COMPUTER is called to compute coverage (line 11). If the coverage satisfies the coverage target tgt, the concolic testing terminates. Otherwise, the test-case selector selects a new test case (line 4) and removes it from TCP (line 5). Then, the binary concolic engine runs again. This process repeats until the target or other termination conditions are achieved. The time bound β guarantees the termination of the validation process in case the target coverage cannot be achieved within the given time. The variable $time$ denotes the total time elapsed since the start of the validation process. In the following, we will discuss the details of each key step.

B. Testbench Generation

The main purposes of a testbench are to generate stimuli and apply them to the design, as well as recording and monitoring the output of the design. To apply CTSC to an existing SystemC project, the existing testbench of the SystemC project can be reused with slight modification. Instead of generating concrete stimuli, the function `CTSC_make_concolic` is used to construct stimuli as symbolic in addition to keeping their concrete values, so-called *concolic stimuli*. Additionally,

```
1  SC_MODULE(driver){
2    sc_in<bool> clk;
3    sc_out<int> data1;
4    sc_out<int> data2;
5
6    void run() {
7      int data1_tmp, data2_tmp;
8
9      wait();
10     while(true){
11       CTSC_make_concolic(&data1_tmp,
12               sizeof(data1_tmp), "data1_tmp");
13       CTSC_make_concolic(&data2_tmp,
14               sizeof(data2_tmp), "data2_tmp");
15       data1.write(data1_tmp);
16       data2.write(data2_tmp);
17
18       wait();
19     }
20   }
21   SC_CTOR(driver){
22     SC_CTHREAD(run, clk.pos());
23   }
24 };
```

Fig. 2. An example of stimuli generation module

we have developed a GUI to generate a testbench for a DUV automatically. Users simply specify names and types for both stimuli and outputs of DUVs. Users can also set the properties of a clock signal, such as period and duty cycle. A complex testbench may demand slight modification manually.

Suppose there are two integral inputs, $in1$ and $in2$, for a SystemC design, Figure 2 illustrates the stimuli generation module[1] in a testbench for the design. The module has one clock input clk, and two data outputs, $data1$ and $data2$ that are connected to the inputs, $in1$ and $in2$, of the design. The function `CTSC_make_concolic` constructs concolic stimuli for the design. The recording and monitoring of the output and the definition of the clock signal are straightforward. Thus, they are not presented.

C. Binary-Level Concolic Execution of SystemC Designs

SystemC designs usually include hierarchical structures, object-oriented features, and hardware-oriented data types. Generally, a SystemC design also contains multiple processes that run concurrently, which requires a scheduler to simulate the design. These features are implemented in libraries. Thus, SystemC designs are not stand-alone programs. SystemC simulation invokes libraries that provide those features.

Therefore, to analyze a design's behavior accurately, it often requires taking the dependent libraries into account. Due to the complexity of the SystemC library, most existing SystemC verification approaches either translate SystemC designs into other IRs, which can represent only a subset of SystemC usually, or handle the SystemC library by writing a simplified one. Thus, those approaches cover only a subset of SystemC features. Although SystemC comes with a well-written user's manual and a reference implementation, it lacks formal specification and leaves out some implementation choices deliberately. Hence, even carefully writing a simplified library can easily result in a dialect. Moreover, such modelling is time-consuming, error-prone, and labor-intensive.

[1]The infinite *while* loop follows the SystemC semantics.

Our framework, in contrast, requires no translation of SystemC designs, no modelling of dependent libraries, and therefore supports all kinds of SystemC designs without restrictions. First, our concolic execution engine concretely simulates a SystemC binary in a native operating system for a fixed number of clock cycles by linking the SystemC library [23], the reference implementation of the Standard [1]. So, concrete execution follows the scheduling mechanism provided by the library. During concrete execution, the current concrete execution path is recorded as a trace. The recorded trace is in the format of LLVM [21] bitcode including all runtime information that is required by symbolic execution afterwards. This way, a concurrent SystemC design is unwound as a sequential self-contained execution trace. Then, the trace is analysed by the engine symbolically to generate new test cases. The symbolic execution follows the scheduling orders of concrete execution. In the future, we will perform scheduling-space exploration to detect concurrency related bugs, such as data races.

A SystemC program is a hardware design that can be simulated for arbitrary clock cycles. Thus, a concrete execution trace of a design can be intimidating long. If we capture the entire execution trace, the symbolic engine will take a long time to exercise the trace and may use up the memory after loading the trace, even before symbolic execution starts. To relieve the strain on memory and reduce the time usage of symbolic execution, we need an intelligent way of capturing a trace. We introduce *selective instruction tracing* technique to reduce the size of a captured trace. CTSC provides a mechanism to capture instructions only from a design itself, excluding the SystemC library and other libraries. Usually, users are not interested in the library code. In addition, if the entire trace is captured, the symbolic engine may explore numerous paths in the library while only a small number of paths are explored from the design perspective.

We built our binary-level concolic execution engine based on CRETE [13], a recently open-sourced concolic execution engine that targets software programs. We will not present the details of binary-level concolic execution here, such as how to capture a concrete trace and what it consists of. Users of interests may refer to CRETE for detailed information.

D. Test-Case Selection

Concolic execution requires a concrete test case each time. At the beginning, there is only one test case, the *seed*, which is simply selected by TEST-CASE-SELECTOR. After the first iteration, multiple test cases may be generated. Thus, different test-case selection strategies can be adopted. Currently, we have developed three test-case selection strategies in terms of the time stamp of test cases: (1) first-come-first-serve (FCFS) that the earliest generated test case is selected first; (2) last-come-first-serve (LCFS) that the last generated test case is selected first, and (3) random selection that a test case is selected among all generated test cases randomly. In the future, we will use SystemC features to guide test-case selection.

E. Testing with Generated Test Cases

It is only half of the story to generate test cases that explore as many paths as possible. A generated test case follows the exact same code path that the concolic engine exercised. Thus, if an error is encountered by the concolic engine, the generated test case can reproduce the error afterwards. Therefore, our framework validates SystemC designs by replaying the generated test cases (line 8 in Algorithm 1). This replay is checked for unusual behavior or errors. If TEST-CASE-REPLAYER detects an error when replaying a test case, it saves the test case and generates a report that records the detailed information of the error, such as error type and error location. This helps users better analyze the design and fix the error.

To check whether a test case passes or fails, we have also integrated the ABV techniques in which designers use assertions to capture specific design intents. Assertions are used to improve the ability to observe bad behavior once they are triggered by specific test cases. By getting an assertion triggered, users can easily identify if there are bugs or invalid inputs. This helps users fix the bug or further constrain the input ranges.

Currently, our framework is focused on generating test cases, not focused on generating assertions for designs. Instead, we utilize the existing assertions in the SystemC designs to evaluate the effectiveness of the CTSC-generated test cases.

IV. EXPERIMENTAL RESULTS

This section presents our experimental results on the SystemC benchmark [22] that includes a number of application domains, such as security, CPU architecture, network and DSP. The benchmark designs cover most core features and data types of the SystemC language. We currently have performed the experiments on 19 total designs, among which 16 designs are from the SystemC benchmark [22]. The other three designs, `RISC_CPU_bdp`, `RISC_CPU_crf`, and `sync_mux81`, are taken from SESC [20] for comparison. Two out of 19 designs have practical sizes, `RISC CPU` containing 13 processes and 2056 lines of code (LoC), and `DES` consisting of 14 processes and 2410 LoC.

The summary of 19 designs are shown in the first three columns of Table I. It lists the names of the designs, the number of processes, and LoC, respectively. Note that only the code in a design itself is taken into account, excluding the testbench code. All experiments were performed on a desktop with a 4-core Intel(R) Core(TM) i7-4790 CPU, 16 GB of RAM, and running the Ubuntu Linux OS with 64-bit kernel version 3.19.

Appropriate coverage metrics are required to evaluate the effectiveness and confidence in the verification results. Functional coverage needs to be redeveloped for new designs and built by engineers with indepth knowledge of both the design specification and implementation. Thus, it is not automatic. SystemC is widely used for modelling functionalities of systems at high-level abstraction. Therefore, we adopt the typical code coverage, line coverage and branch coverage, which are widely used and understood. We choose line and branch coverage reported by `LCOV` [25]. We also adopt assertion coverage to show the ability of our approach to detect bugs.

A. Code Coverage Improvement over Seeds

In our experiments, we developed the seed for each design. We set a 24-hour time bound and 100% branch coverage target

TABLE I
SUMMARY OF DESIGNS, TIME AND MEMORY USAGE

Designs	# of Proc.	LoC	ET (s)	MEM (MB)	# of TCs
DES	14	2401	186	3928	35
RISC_CPU_bdp	3	148	3077	505	149
RISC_CPU_mmxu	1	187	1291	240	258
RISC_CPU_exec	1	126	251	230	44
RISC_CPU_floating	1	127	576	243	122
Qsort	1	86	88	175	41
UsbTxArbiter	5	144	234	265	298
Sync_mux81	1	52	73	180	28
MIPS	1	255	207	124	474
IDCT	1	244	335	574	494
MD5C	1	271	21	465	37
RSA	1	324	1944	8103	131
RISC_CPU_crf	5	927	10863	331	1220
RISC_CPU_control	1	826	12005	347	1246
ADPCM	1	134	25	220	40
Y86	11	301	493	480	67
Pkt_switch	17	376	3189	333	385
RISC CPU	13	2056	17520	1303	386
Master/Slave Bus	5	974	24	205	88

for all designs. The actual execution time (ET) of each design in seconds is listed in the fourth column of Table I, after which the branch coverage could not be improved within the 24-hour time bound. The corresponding maximum memory usage and the number of generated test cases are presented in column five and six. As shown, the time and the memory usage was modest. Since `RSA` and `DES` are cipher algorithms that do computation on large numbers, they used more memory.

Figure 3 and Figure 4 show the code coverage improvement on 19 total designs over the seeds with the time usage listed in Table I. In our current experiments, we adopted FCFS test-case selection strategy. In the future, we will evaluate the effects of different test-case selection strategies. As illustrated, the CTSC-generated test cases are able to improve code coverage substantially. For most designs, high code coverage is achieved in a short time. CTSC achieves 100% line coverage on ten designs and 100% branch coverage on eight designs. Table II shows the overall code coverage improvement of CTSC over the seeds. On average, CTSC achieves 97.3% line coverage and 91.8% branch coverage (Column 2 of Table II). The maximum improvement of line coverage and branch coverage are 84% and 91.5% (Column 3 of Table II), respectively. On average, line coverage and branch coverage are improved by 32.3% and 50.2% (Column 4 of Table II), respectively.

TABLE II
COVERAGE IMPROVEMENT OVER SEEDS

Coverage	Ave. (%)	Max Δ (%)	Ave. Δ (%)
Line	97.3	84	32.3
Branch	91.8	91.5	50.2

Besides the final coverage our approach achieved, we also graphically demonstrate the cumulative progression as test cases were generated. Here, we selected three designs, `RISC CPU`, `RISC_CPU_control`, and `RISC_CPU_crf`, which

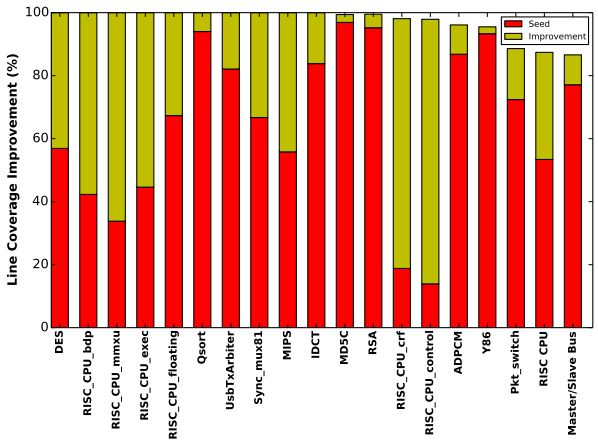

Fig. 3. Line coverage improvement on 19 total designs

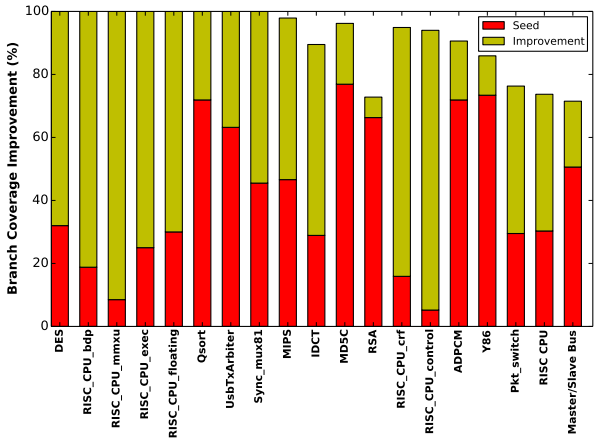

Fig. 4. Branch coverage improvement on 19 total designs

use relatively longer time. Figure 5 and Figure 6 illustrate the cumulative progression of line coverage and branch coverage, respectively. The figures demonstrate the 10-hour time line visually. As shown, the line coverage and the branch coverage are improved substantially within the first hour based on the seed, after which improvement tapers off in a few hours.

There are three possible reasons that our approach is unable to achieve 100% code coverage for some designs. First, the constraint solver may fail to solve complex symbolic expressions. Since there are symbolic inputs at each simulation cycle, the collected path constraints can be very complicated. Second, the concolic execution engine records instructions from discrete parts of an execution path. Symbolic variables may be concretized from the captured part to the uncaptured part. Third, there are unreachable statements and branches in certain designs.

B. Comparison with the State-of-the-Art Approaches

In this section, we compare the results of our approach with the state-of-the-art approaches. To the best of our knowledge, no other existing SystemC verification approaches except our previous work, SESC [20], provide such code coverage. Therefore, we compare the results of the 11 shared designs achieved by CTSC and SESC, as shown in the Table III. LCov and BCov denote line coverage and branch coverage, respectively.

As illustrated, compared with SESC, CTSC achieves higher line and branch coverage for `RISC_CPU_mmxu`. CTSC

TABLE III
COMPARISON WITH SESC

Designs	ET (s)		MEM (MB)		# of TCs		LCov (%)		BCov (%)	
	SESC	CTSC	SESC	CTSC	SESC	CTSC	SESC	CTSC	SESC	CTSC
RISC_CPU_mmxu	11.38	1291	15.6	240	95	258	99.4	100	97.9	100
RISC_CPU_exec	3.23	251	49.6	230	35	44	100	100	100	100
RISC_CPU_bdp	0.15	3077	17.5	505	36	149	100	100	100	100
UsbTxArbiter	0.05	234	13.7	265	10	298	100	100	100	100
Sync_mux81	0.04	73	13.5	180	10	28	100	100	100	100
MIPS	178.23	207	27.6	124	39	474	100	100	97.9	97.9
IDCT	180	335	134	574	135	494	100	100	100	89.5
ADPCM	1.88	25	16.2	220	25	40	100	96.1	100	90.6
RISC_CPU_control	0.57	12005	17.8	347	76	1249	100	97.9	100	94
RISC_CPU_crf	300	10863	61.1	331	1759	1220	98.2	98.1	95.7	94.9
RISC CPU	169	17520	264	1303	2099	386	96.3	87.4	93.2	73.7

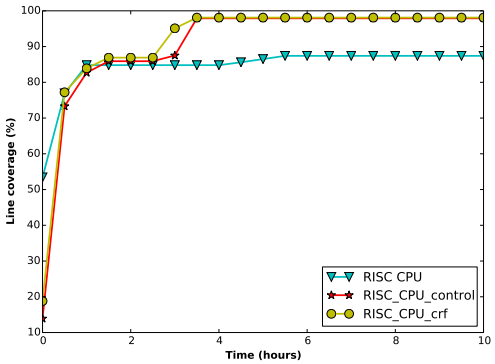

Fig. 5. The cumulative progression of line coverage

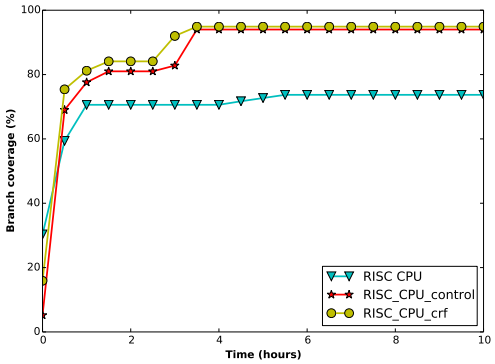

Fig. 6. The cumulative progression of branch coverage

achieves the same line coverage for six designs and the same branch coverage for five designs as achieved by SESC. For other designs, CTSC achieves a little lower code coverage compared with SESC. In terms of the number of test cases and time usage, CTSC has slightly larger number of generated test cases and a bit longer time usage in general. The reason is that SESC symbolically executes a design only once, while CTSC is an iterative process. A test case generated by one iteration may cover the same code with a test case generated by another iteration. CTSC also costs a little more memory, which is because the concolic execution in this framework involves virtual machine. Thus, the memory usage is reasonable and modest. However, SESC only works on high-level synthesizable SystemC that is a subset of SytemC. CTSC supports all features of the SystemC language provided by the SystemC library and has better scalability.

C. Comparison with Random Testing

We also compare the code coverage results of CTSC with random testing that randomly generates test cases automatically at each simulation cycle. We set the branch coverage of each design achieved by CTSC as the target for random testing. In case random testing could not achieve the target, we set a 24-hour time bound. We conducted the experiments of random testing 10 times for each design and computed the average. Note that the inputs of `Master/Slave Bus` has rigorous restrictions. It is hard to generate valid inputs randomly. Thus, we excluded this design.

The target is achieved on eight designs, but with many more test cases. Random testing does not achieve the target for other ten designs after running 24 hours, although numerous random test cases were generated and simulated. Note that if regression testing is performed, it is time-consuming to simulate hundreds of thousands of test cases or more each time. Compared with random testing, CTSC has the advantage of generating much fewer test cases to achieve high code coverage and to cover corner cases efficiently where bugs are likely to appear, as illustrated in the following section.

D. Bug Detection

In addition to computing code coverage, we also show the capability of our approach to trigger assertions and detect bugs. When the validation process terminates, a validation report is generated automatically. The report is in a plain text format that mainly indicates the statuses of test cases (failure or pass) and assertion violations. Upon triggering an assertion, CTSC generates a test case automatically leading to the assertion, which helps designers find the root cause easier when debugging the design.

Among the 19 total designs, five designs contain assertions, as shown in the first column of Table IV. The second column shows the total number of assertions. The last three columns present the number of assertions triggered by the seed, by

the CTSC-generated test cases, and by random testing, respectively. Although random testing can trigger some assertions, it generates many more test cases than CTSC. Note that some assertions always hold. For instance, although there are 15 total assertions in the design `RSA`, we have verified manually that 11 assertions can never be triggered. For example, the violation of assertion `assert(a == d)` directly following the assignment `a = d` cannot be triggered.

TABLE IV
ASSERTION COVERAGE

Designs	# of Assertions			
	total	by seed	by CTSC	by random
RISC_CPU_exec	2	0	2	0
MD5C	1	0	1	0
RSA	15	0	3	3
Master/Slave Bus	11	0	6	N/A
RISC CPU	2	0	2	2

During our experiments, we found an interesting bug in the design `RSA`, an asymmetric cryptographic algorithm. The first step of `RSA` is to find two *different* large prime numbers, p and q. Note that the algorithm relies on the fact that p and q are different. If they are equal, the algorithm does not work correctly. However, this implementation does not check whether or not p and q are equal. In addition, we also found an out-of-bound access to an array in the design `Y86`. The bugs found by CTSC underlines the importance of performing automated concolic testing and the effectiveness of CTSC.

V. CONCLUSIONS

In this paper, we have presented an automated, easy-to-deploy, scalable, and effective binary-level concolic testing framework for SystemC designs. Our extensive experiments illustrate that CTSC is able to achieve high code coverage and detect bugs effectively, as well as scaling to designs of practical sizes. Our framework can handle object-oriented features, hardware-oriented structures, event-driven simulation semantics, and concurrency effectively. Four major advantages of our approach are summarized as follows. First, CTSC requires no translation of SystemC designs and no modelling of dependent libraries, while most existing work does. Second, CTSC supports *all* features of the SystemC language, while most existing approaches support only a subset of SystemC features. Third, CTSC provides an easy deployment model. It requires minimum engineering effort to apply CTSC to existing SystemC projects. Fourth, once a testbench is configured, the whole validation process is *fully automated*.

The current experimental results are promising. We will evaluate the performance of our framework on the remaining designs from the SystemC benchmark [22]. Besides, there are three directions that we want to explore in the future. First, we will utilize SystemC specific features to guide test-case selection. Second, we will explore the scheduling space systematically to detect concurrency related bugs. Third, we will work on the constraint solver failures and over concretization problems.

ACKNOWLEDGEMENT

This research received financial support from National Science Foundation (Grant #: CNS-1422067).

REFERENCES

[1] IEEE Standards Association, "Standard SystemC Language Reference Manual," IEEE Standard 1666-2011, 2011.

[2] https://en.wikipedia.org/wiki/SystemC

[3] M. Y. Vardi, "Formal Techniques for SystemC Verification," in *Proceedings of the 44th annual Design Automation Conference*, 2007.

[4] J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, 1976.

[5] C. Cadar, D. Dunbar and D. Engler, "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*, 2008.

[6] V. Chipounov,V. Kuznetsov and G. Candea, "S2E: A Platform for In-vivo Multi-path Analysis of Software Systems," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[7] K. Cong, F. Xie, and L. Lei, "Symbolic Execution of Virtual Device," in *Proc. of the 13th International Conference on Quality Software*, 2013.

[8] B. Lin and D. Qian, "Regression Testing of Virtual Prototypes Using Symbolic Execution," *International Journal of Computer Science and Software Engineering (IJCSSE)* 4, no. 12, 2015.

[9] P. Godefroid, N. Klarlund and K. Sen, "DART: Directed Automated Random Testing," *ACM SIGPLAN Notices*, 2005.

[10] K. Sen, D. Marinov and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

[11] K. Cong, F. Xie and L. Lei, "Automatic Concolic Test Generation with Virtual Prototypes for Post-silicon Validation," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, 2013.

[12] G. Guglielmo, M. Fujita, F. Fummi, G. Pravadelli, and S. Soffia, "EFSM-based Model-driven Approach to Concolic Testing of System-level Design," in *Proceedings of the 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, 2011.

[13] B. Chen, C. Havlicek, Z. Yang, K. Cong, R. Kannavara and F. Xie, "CRETE: A Versatile Binary-Level Concolic Testing Framework," to appear in *Proceedings of FASE*, 2018.

[14] P. Herber, J. Fellmuth and S. Glesner, "Model Checking SystemC Designs Using Timed Automata," in *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2008.

[15] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri, "KRATOS: A Software Model Checker for SystemC," in *Proceedings of International Conference on Computer Aided Verification*, 2011.

[16] C.-N. Chou, Y.-S. Ho, C. Hsieh, and C.-Y. Huang, "Symbolic Model Checking on SystemC Designs," in *Proceedings of the 49th Annual Design Automation Conference*, 2012.

[17] H. M. Le, D. Große, V. Herdt, and R. Drechsler, "Verifying SystemC Using an Intermediate Verification Language and Symbolic Simulation," in *Proceedings of the 50th Annual Design Automation Conference*, 2013.

[18] A. Sen and M. S. Abadir, "Coverage Metrics for Verification of Concurrent SystemC Designs Using Mutation Testing," in *Proceedings of International High Level Design Validation and Test Workshop*, 2010.

[19] M. Chen, P. Mishra, and D. Kalita, "Automatic RTL Test Generation from SystemC TLM Specifications," *ACM Transaction on Embedded Computing System*, 2012.

[20] B. Lin, Z. Yang, K. Cong and F. Xie, "Generating High Coverage Tests for SystemC Designs Using Symbolic Execution," in *Proceedings of the 21st Asia and South Pacific Design Automation Conference*, 2016.

[21] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004.

[22] B. Lin, and F. Xie, "SCBench: A Benchmark Design Suite for SystemC Verification and Validation," in *Proceedings of the 23rd Asia and South Pacific Design Automation Conference*, 2018.

[23] http://www.accellera.org/downloads/standards/systemc/files

[24] B. Nicolas, K. Daniel, and S. Natasha, "Scoot: A Tool for the Analysis of SystemC Models," in *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of System*, 2008.

[25] http://ltp.sourceforge.net/coverage/lcov/readme.php