Generating High Coverage Tests for SystemC Designs Using Symbolic Execution

Bin Lin, Zhenkun Yang, Kai Cong, Fei Xie
Department of Computer Science, Portland State University, Portland, OR 97207, USA
{linbin, zhenkun, congkai, xie}@cs.pdx.edu

Abstract— SystemC is a system-level modeling language increasingly adopted by the semiconductor industry. Quality assurance for SystemC designs is important, since undetected errors may propagate to final silicon implementations and become very costly to fix. The errors, if not fixed, can cause major damage and even endanger lives. However, quality assurance for SystemC designs is very challenging due to their object-oriented nature, event-driven simulation semantics, and inherent concurrency. In this research, we have developed an approach to generating high coverage tests for SystemC designs using symbolic execution. We have applied this approach to a representative set of SystemC designs. The results show that our approach is able to generate tests that provide high code coverage of the designs with modest time and memory usage. Furthermore, the experiment on a RISC CPU design with more than 2K lines of SystemC code demonstrates that our approach scales to designs of practical sizes.

I. Introduction

The increasingly short time-to-market requirement for electronic devices, accompanied by the growing complexity of their designs, pushes designers to model systems at a high level of abstraction. Among high-level hardware description languages, SystemC [1] has gained popularity due to its fast simulation and strong support for hardware/software co-design.

SystemC can effectively describe a design at multiple abstraction levels. It enables step-by-step refinement of a high-level abstract design down to a low-level implementation. However, errors in the high-level design can also propagate down to the low-level implementation. Fixing the errors in high-level designs is critical, since the costs of correcting an error increases significantly with the refinement of the designs down to the lower levels. The errors in the final hardware implementations can cause major damage and even endanger lives. Thus, SystemC verification is necessary and important. Verification is not only a key step but also a major bottleneck of system design [2]. The challenges of SystemC verification are mainly due to the object-oriented nature, event-driven simulation semantics, and inherent concurrency of SystemC [3].

SystemC verification has two general paradigms: *formal verification* and *dynamic validation*. Although *formal verification*, particularly model checking [4], has been utilized in the industrial context for decades, it still remains a unit level verification technique [5]. The well-known technical limitation of model checking — state space explosion — prevents it from being applied to complex system-level designs. *Dynamic validation* is a simulation-based approach and the "workhorse" for validating SystemC designs [3]. To this end, designers usually simulate a design over a set of concrete test cases. Thus, high quality test cases are critical. However, manual test-case generation is very time-consuming because it requires deep understanding of the design under validation (DUV).

This paper presents an approach to automatically generating test cases that provide high code coverage for SystemC designs using symbolic execution [6]. Such symbolic execution is effective because one symbolic input can cover a set of concrete inputs. In addition, concrete test cases are generated for all paths through the designs that are exercised along with symbolic execution. Furthermore, our approach can rerun the generated test cases on the designs, which helps designers analyze SystemC designs better. The contributions of this research are summarized as follows.

- We have developed a framework for generating a test harness for each SystemC design.
- We have developed a symbolic execution engine named SESC (Symbolic Execution of SystemC) based on KLEE [7]. We have implemented a scheduler in SESC to handle SystemC concurrency and other features to handle SystemC hardware specific semantics, such as signal, FIFO, arbitrary-width data type and clock cycle.
- We have developed a framework for replaying the generated test cases on SystemC designs. Based on test case replay, various code coverage statistics are generated.
- We have implemented our approach and evaluated it on 11 SystemC designs. Our experimental results demonstrate that our approach is able to generate tests that provide high code coverage of the designs with modest time and memory usage, and to scale to practical designs with more than 2K lines of SystemC code.

II. Related Work

There have been several attempts [8, 9, 10, 11, 12] to formally verify SystemC designs by checking safety properties. In [8], a translation from SystemC designs to UPPAAL timed automata is proposed. Then the UPPAAL model checker and the UPPAAL tool suite can be applied to the resulting UPPAAL

978-1-4673-9569-4/16/$31.00 ©2016 IEEE

models. In [9], a methodology is proposed to translate a SystemC transaction level modeling (TLM) design into a sequential C model. Then, the induction-based formal method is used to check for C assertion violations. In [10], a translation from SystemC designs to threaded C models is proposed and then the Explicit-Scheduler/Symbolic Threads algorithm is applied. In [11], a symbolic model-checking technique that formalizes the semantics of SystemC designs in terms of Kripke structures is presented. In [12], an intermediate verification language (IVL) is proposed and a symbolic simulator is developed for the IVL. All of the aforementioned approaches are focused on translation of SystemC designs rather than direct SystemC verification. Furthermore, they only check limited properties and property formulation is challenging.

There has also been research on *dynamic validation* [13, 14, 15]. In [13], a methodology is proposed to automatically generate test cases based on code-coverage analysis. However, to use this framework, the DUV has to be instrumented manually. In [14], coverage metrics for SystemC verification using mutation testing are developed. In [15], an approach to generating RTL test cases from TLM specifications is presented.

We utilize symbolic execution to generate high coverage tests for SystemC designs. We focus on validating high-level synthesizable subset of SystemC [17]. High-level synthesis (HLS) is increasingly used for hardware designs in the industry over the last decade. However, the quality of high-level synthesizable SystemC designs is critical, since errors in these designs will propagate down to the synthesized designs. The earlier the errors are found, the less expensive to fix them. Thus, it is important to validate synthesizable SystemC designs.

III. Background

A. SystemC

SystemC is a hardware modeling language based on C++. It includes an event-driven simulation kernel, and extends the capabilities of C++ by enabling modeling of hardware designs. In addition, SystemC has clock cycle semantics.

The SystemC library provides powerful mechanisms for designing complex systems. It enables modeling hardware at multiple levels of abstraction. A SystemC design is modeled as a set of modules communicating through ports that are connected by channels. A module consists of at least one process. Each process is a block of statements that describe certain behavior of a system. Processes run concurrently and are managed by a non-preemptive scheduler. The semantics of SystemC concurrency is co-operative multitasking. A process suspends itself when encounters function `wait()` or runs to the end, and is resumed by one or more notified events of its sensitivity list. SystemC provides three types of processes: SC_METHOD, SC_THREAD, and SC_CTHREAD.

Figure 1 shows a simple SystemC design. It has one module containing two processes P1 and P2 that are registered as SC_CTHREAD. They are both sensitive to the positive edge of the clock, which means that they will be executed at the positive edge of each clock. The design has three input ports: *en*, *clk*, and *din*; one output port *dout*; and one shared signal *b*. P1 reads the input data *din* according to the signal *en*, while P2 do the computation based on the input data and output the result.

```
1  SC_MODULE(example){
2    sc_in<bool> en;
3    sc_in<bool> clk;
4    sc_in<int> din;
5    sc_out<int> dout;
6
7    sc_signal<int> b;
8
9    void P1() {
10     wait();
11     while(true) {
12       if(en.read())
13         b = din.read();
14
15       wait();
16     }
17   }
18
19   void P2() {
20     int c;
21     wait();
22     while(true) {
23       if(b < 0)
24         c = -b;
25       else if(b % 2)
26         c = b / 2;
27
28       dout.write(c);
29       wait();
30     }
31   }
32   SC_CTOR(example){
33     SC_CTHREAD(P1, clk.pos());
34     SC_CTHREAD(P2, clk.pos());
35   }
36 };
```

Fig. 1. A SystemC example

The synthesizable subset of SystemC [17] defines some restrictions on the standard SystemC [1] so that designs following the restrictions are appropriate for input to HLS tools. The three types of processes are all supported but with restrictions. Processes cannot be created dynamically. An SC_METHOD process can only describe combinational circuit and its sensitivity list shall be static. Thread processes, SC_THREAD and SC_CTHREAD, have the same expressiveness for the synthesis purposes. A thread process shall be statically sensitive to exactly one clock edge. The thread process body of a design shall only use `wait()` or `wait(`*int*`)`. The structure *sc_event* is not supported.

B. Symbolic Execution and KLEE Engine

Symbolic execution [6] exercises a program by taking symbolic values as inputs, which are symbols representing arbitrary values allowed by the types of corresponding variables. Consequently, the results are represented as symbolic expressions over the inputs. A symbolic execution state includes values of program variables, a path condition, and a program counter. The path condition is a boolean formula that must be satisfied to reach current execution state from the initial state. The program counter denotes the next statement to execute.

KLEE is a symbolic execution engine built upon the LLVM infrastructure [16]. KLEE exercises C programs symbolically and generates test cases automatically, aiming to achieve high code coverage. First, programs are compiled to the LLVM assembly language. Then, KLEE interprets LLVM instructions directly. We developed our own engine, SESC, to enable symbolic execution of SystemC designs, based on KLEE.

IV. Proposed Approach

A. Overview

Figure 2 shows the framework of our approach to SystemC validation. It has three important steps: (1) test-harness generation, (2) symbolic execution, and (3) test-case generation.

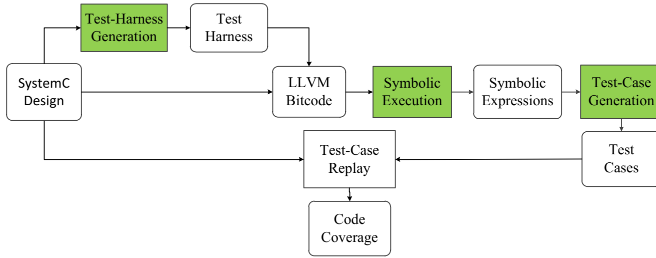

Fig. 2. Framework for SystemC test generation

For a given SystemC design, its test harness is generated and compiled with the design together to LLVM bitcode. The execution engine SESC takes LLVM bitcode as input and exercises designs symbolically to explore as many execution paths as possible. When an execution path terminates or encounters an error, SESC sends the path conditions that are represented by symbolic expressions to a constraint solver, which returns concrete values that satisfy the expressions. Then SESC generates a concrete test case for the path. The generated test cases are then rerun on the SystemC design to compute coverage statistics. We will discuss the detail in the following.

B. Test-Harness Generation

A SystemC design by itself is not a stand-alone program. It invokes SystemC library and communicates with its environment in the simulation. SystemC simulation requires the SystemC library that provides predefined structures and simulation kernel, so does symbolic execution. However, the whole SystemC library is too complex for symbolic execution. Hence, a test harness that models the environment must be provided to enable the symbolic execution of SystemC designs. A key challenge here is how to generate a test harness. The test harness should be simple enough so that SESC can efficiently execute a design symbolically.

A test harness includes global variables definitions, synchronization mechanisms, symbolic variables constructions, and process registrations. Figure 3 shows the skeleton of the test harness for the design shown in Figure 1. Shared signals are defined as global structure *globalVars*. It only has one member *b* in this case. The harness defines two variables of type *globalVars*. The variable *currState* contains the synchronized value, while *LStates* is an array that each element is modified by one process. The function `SESC_make_symbolic(···)` constructs symbolic variables and `SESC_thread(···)` registers a SystemC process with SESC so that SESC can schedule it when required. The function `SESC_start`($numCycles$) starts symbolic execution, where *numCycles* specifies how many clock cycles to simulate.

C. Scheduler

SystemC is widely used to model concurrent systems consisting of multiple processes. To deal with the SystemC concurrency, we have implemented a scheduler in SESC to manage multiple processes, as shown in Algorithm 1. Currently, our approach supports two types of processes, SC_THREAD and SC_CTHREAD, in the synthesizable subset of SystemC.

```
1  typedef struct Globals{
2    int b;
3  }globalVars;
4
5  globalVars currState, LStates[2];
6  ......
7  void PREPROCESS(currState) {  ......  }
8  void SYNC(LStates) {  ......  }
9
10 int main(int argc, char **argv) {
11   ......
12   SESC_make_symbolic(&en, sizeof(en), "en");
13   SESC_make_symbolic(&in, sizeof(in), "in");
14   SESC_thread("P1", &en, &clk, &din, &LStates[1].b, &dout);
15   SESC_thread("P2", &en, &clk, &din, &LStates[2].b, &dout);
16   ......
17   SESC_start(numCycles);
18
19   return 0;
20 }
```

Fig. 3. Skeleton of the test harness for the design shown in Figure 1

Algorithm 1: SYM-EXE-SCHEDULER(P, $numCycles$)

Data: $currState$ and $LStates$ are global variables.
Result: scheduling the symbolic execution of processes P for $numCycles$ cycles.

1 $cycles \leftarrow 0$
2 $runnable \leftarrow \emptyset$
3 **foreach** $p \in P$ **do**
4   ENQUEUE($runnable$, p)
5 **while** $cycles \leq numCycles$ **do**
6     $next_runnable \leftarrow \emptyset$
7     $LStates \leftarrow$ PREPROCESS($currState$)
8   **while** $runnable \neq \emptyset$ **do**
9         $q \leftarrow$ DEQUEUE($runnable$)
10   SYM-EXE-PROCESS(q)
11   ENQUEUE($next_runable$, q)
12     $runnable \leftarrow next_runnable$
13     $currState \leftarrow$ SYNC($LStates$)
14     $cycles \leftarrow cycles + 1$

According to the SystemC specification [1], access to shared storage should be synchronized explicitly to avoid non-deterministic behavior, although the scheduler is non-deterministic. Thus, different scheduling sequences should not affect the simulation result for a well-formed design, which means that it is sufficient for the design to execute only one scheduling sequence for each clock cycle. Therefore, we simulate designs deterministically using symbolic execution and synchronize shared storage explicitly. If a design is not well-formed, such as variables other than signals are used as inter-process communication, there are potential races. In this work, we assume that there are no races for the DUV.

As shown in Algorithm 1, before execution starts, the scheduler initializes simulation cycle as zero and puts all runnable

processes into the runnable queue *runnable*. When execution starts, for each clock cycle, the scheduler first empties the queue *next_runnable*. Then PREPROCESS is called to make N replicas of $currState$ and store them in the array $LStates$, where N is the number of processes. After that, the scheduler removes each process from *runnable*, and calls SYM-EXE-PROCESS to execute the selected process symbolically. Note that each process modifies its local state. When the process encounters `wait`, the scheduler puts the process into $next_runnable$. When $runnable$ is empty, the execution is finished for this clock cycle. So the scheduler puts every process into $runnable$ for the next clock cycle and synchronizes all local states resulting in a new global state $currState$, followed by advancing the simulation cycles. If the number of simulation cycles reaches *numCycles*, the simulation is done, and the execution engine terminates.

D. Symbolic Execution of SystemC Designs

To symbolically execute SystemC designs, besides the concurrency addressed in the previous section, SESC needs to address the following three technical challenges.

First, the path explosion problem is a major limitation of symbolic execution to exercise a complex program thoroughly. The number of paths is approximately exponential to the number of branches in a program. Not surprisingly, this problem also exists with symbolic execution of SystemC designs.

We apply two bounds to address this problem. One is the time bound that is the maximum time for symbolic execution engine to run. The time bound ensures that SESC will terminate in a given amount of time. If SESC does not finish within the given time, there may be unfinished paths. For such paths, SESC still generates test cases with the path constraints collected before termination. The other bound is the clock cycle bound that specifies how many clock cycles to simulate.

Second, a SystemC design has a hierarchical modular structure usually and include the object-oriented features, such as inheritance and polymorphism. Our framework flattens these features first by preprocessing SystemC designs. Then, they are compiled to LLVM bitcode using the *clang* compiler [21]. Each process is flattened as a function whose name is the same as the process name. All the input and output ports of the module, as well as shared signals among processes, become the pointer parameters of the function.

Figure 4 illustrates the skeleton of the flattened result for the design shown in Figure 1. Processes P1 and P2 are interpreted as function P1 and P2. The input ports *en, clk*, and *din*, output port *dout*, and shared signal *b* become the pointer parameters of the functions.

Third, a SystemC design may contain hardware specific data structures, such as port, signal, FIFO, arbitrary-width data, and bit-precise operation. We address these structures by either implementing their stubs or adapting the symbolic execution engine. Ports are interpreted as the function parameters as described above. Predefined channels, such as *sc_signal* and *sc_fifo*, the preprocessor replaces these structures with our implementations. Arbitrary-width data types and bit-precise operations, such as bit selection and bit set, are ubiquitous in hardware designs. KLEE does not support such data types and bit operations. We have implemented the functionalities that support arbitrary-width data types and bit-precise operations, such as part selection and part set.

```
1  void P1(*en, *clk, *din, *b, *dout) {
2    ......
3  }
4
5  void P2(*en, *clk, *din, *b, *dout) {
6    ......
7  }
```

Fig. 4. Skeleton of flattened result for the design shown in Figure 1

With the aforementioned challenges addressed, a SystemC design can be executed symbolically now. SESC collects path constraints along the execution of each path. For the example shown in Figure 1, suppose we set the clock cycle bound as three and the corresponding symbolic inputs for three clock cycles are $\{en_1,\ in_1\}$, $\{en_2,\ in_2\}$, and $\{en_3,\ in_3\}$. After symbolic execution of the design, SESC can collect constraints for all explored paths. Three sample constraints are as follows.

$$Constraint\,1 : en_2 \neq 0\ \wedge in_2 < 0 \wedge en_3 \neq 0$$
$$Constraint\,2 : en_2 \neq 0\ \wedge in_2 > 0 \wedge (in_2\%2 \neq 0) \wedge en_3 \neq 0$$
$$Constraint\,3 : en_2 \neq 0\ \wedge in_2 > 0 \wedge (in_2\%2 = 0) \wedge en_3 \neq 0$$

E. Test-Case Generation

A test case $\mathcal{T} \triangleq I_1, I_2, \ldots, I_n$ of a SystemC design is a sequence of inputs, such that cycle input I_i $(1 \leq i \leq n)$ will be applied to the design at clock cycle i as inputs. A cycle input $I \triangleq \{\langle p, v\rangle \mid p$ is an input port, v is the concrete value of $p\}$ of a design is a set of concrete inputs.

The path constraints collected by SESC are denoted as symbolic expressions. However, symbolic expressions are hard to understand for general designers. So when an execution path terminates, the symbolic expressions are sent to a constraint solver, which returns concrete values that satisfy the expressions. As the constraints shown at the end of previous section, the corresponding test cases generated by SESC are as follows.

$\mathcal{T}_1 \triangleq I_1, I_2, I_3$ where $I_1 \triangleq \{\langle en_1, 0\rangle,\ \langle in_1, 0\rangle\}$, $I_2 \triangleq \{\langle en_2, 1\rangle,\ \langle in_2, -1\rangle\}$, $I_3 \triangleq \{\langle en_3, 1\rangle,\ \langle in_3, 0\rangle\}$.

$\mathcal{T}_2 \triangleq I_1, I_2, I_3$ where $I_1 \triangleq \{\langle en_1, 0\rangle,\ \langle in_1, 0\rangle\}$, $I_2 \triangleq \{\langle en_2, 1\rangle,\ \langle in_2, 1\rangle\}$, $I_3 \triangleq \{\langle en_3, 1\rangle,\ \langle in_3, 0\rangle\}$.

$\mathcal{T}_3 \triangleq I_1, I_2, I_3$ where $I_1 \triangleq \{\langle en_1, 0\rangle,\ \langle in_1, 0\rangle\}$, $I_2 \triangleq \{\langle en_2, 1\rangle,\ \langle in_2, 2\rangle\}$, $I_3 \triangleq \{\langle en_3, 1\rangle,\ \langle in_3, 0\rangle\}$.

V. EVALUATION

This section describes our experiments on a benchmark of 10 representative SystemC designs and a design of practical size. The 10 designs as listed in Table I were collected from open source benchmarks [18, 19] of hardware designs. All experiments were performed on a desktop with 4-core Intel(R) Xeon(R) CPU, 8 GB of RAM, and running the Debian Linux operating system with 64-bit kernel version 3.16. The time and

TABLE I
TIME AND MEMORY USAGE, AND COVERAGE RESULTS

Designs	# Proc.	LoC	Time(s)	Memory(MB)	# TestCases	LCov(%)	BCov(%)
risc_cpu_exec	1	126	3.23	46.9	35	100	100
risc_cpu_mmxu	1	187	11.38	15.6	95	99.4	97.9
risc_cpu_control	1	826	0.57	17.8	76	100	100
risc_cpu_bdp	3	148	0.15	17.5	36	100	100
risc_cpu_crf	5	927	300	61.1	1759	98.2	95.7
usbArbStateUpdate	2	85	0.05	13.7	10	100	100
mips	1	255	178.23	27.6	39	100	97.9
adpcm	1	134	1.88	16.2	25	100	100
idct	1	244	180	134.0	135	100	100
sync_mux81	1	52	0.04	13.5	10	100	100

memory usage of our approach is shown in Table I. The first column gives the names of designs. The second column shows the number of processes of each design. Lines of code for each design are listed in the third column. The subsequent two columns present time and memory usage, respectively. The sixth column gives the number of generated test cases. As we can see, the time and memory usage are modest.

A. Coverage Methodology

To measure the effectiveness of SESC-generated test cases, our framework reruns them on the unmodified SystemC designs by supplying them to a replay harness that we developed. We use line and branch coverage reported by gcov [20], because code coverage is widely used in white box testing and there is a positive relation between code coverage and reliability [22]. Note that the coverage results only consider code in the design itself. We measure sizes in terms of executable lines of code (LoC) in the SystemC designs.

B. Experimental Results on a Benchmark

The coverage results of our approach on the benchmark are shown in the last two columns of Table I. As we can see from the table, our approach can achieve 100% line and branch coverage for most designs. Our approach does not achieve 100% coverage for design risc_cpu_mmxu since one branch is eliminated by compiler optimization. For design risc_cpu_crf, the path explosion problem is severe. After we set the time bound as 300 seconds, SESC generates 1759 test cases that achieve 98.2% line coverage and 95.7% branch coverage. There is a dead branch in the design mips, so 100% branch coverage can never be achieved.

We compared the coverage results of SESC with two groups of random testing. They are denoted as Random10 and Random100 that represent the number of random tests with 10 times and 100 times of the number of test cases generated by SESC. For each group, we conducted 10 times experiments and calculated the average. The comparisons of line coverage and branch coverage are shown in Figure 5 and Figure 6, respectively. As we can see from the figures, SESC gains the best coverage results. Especially, our approach beats both by a significant margin for mips, idct and sync_mux81.

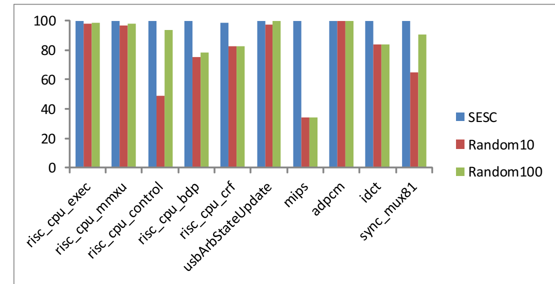

Fig. 5. Line coverage of SESC testing vs. Random10 and Random100 for 10 SystemC designs. SESC beats both by at most 66%.

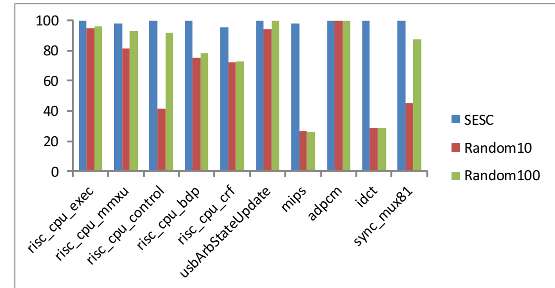

Fig. 6. Branch coverage of SESC testing vs. Random10 and Random100 for 10 SystemC designs. SESC beats both by at most 71%.

Based on the comparisons, the SystemC designs can be divided into three groups. In the first group, designs can be easily achieved high code coverage using random testing, such as adpcm. Designs in this group contain only a few branches. The second group contains designs that can be achieved relatively high code coverage by random testing with much more test cases, such as risc_cpu_control. Designs in this group consist of relatively more branches than the first group, but there are barely any nested branches. In the third group, it is hard to achieve high code coverage using random testing even with much more test cases, such as mips. This is mainly because designs in this group include nested branches and compound conditions of branches. As we have seen, our approach can achieve very high coverage for relatively complex designs, while random testing cannot. If designs are more larger and includes more complex conditions of branches, the advantage of our approach will be more obvious, which we will show in the following.

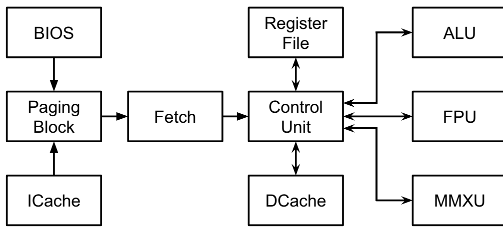

Fig. 7. Architecture of RISC CPU

TABLE II
COVERAGE RESULTS OF RISC CPU

Method	# TestCases	LCov(%)	BCov(%)
SESC	2,099	96.3	93.2
Random10	20,990	81.2	66.5
Random100	209,900	91.1	76.4

C. Experimental Results on a Design of Practical Size

We also applied our approach on a design of practical size — RISC CPU [18]. The design consists of 10 modules, 13 processes, and 2056 LoC in total. Figure 7 shows the architecture of the CPU design. The CPU reads in program instructions and executes them and then writes the results back to registers or data memory. The instruction set is defined based on commercial RISC processor together with MMX-like instructions. `BIOS` stores system bios data. `ICache` caches program instructions, while `DCache` caches data. `Fetch` fetches instructions from `Paging`. `Control Unit` decodes instructions. `Register File` models registers. `ALU` is an integer execution unit, while `FPU` is a floating point execution unit. `MMXU` is an MMX-like execution unit.

The symbolic execution overhead of the CPU design is still modest. It takes 169 seconds and 264 MB. SESC generated 2099 test cases. The coverage results of SESC, as well as Random10 and Random100 (averaged over 10 runs), are illustrated in Table II. Our approach achieved 96.3% line coverage and 93.2% branch coverage which is much better than random testing. Moreover, random testing generated more than 200,000 test cases which is very time-consuming to run.

VI. CONCLUSIONS AND DISCUSSIONS

In this paper, we have presented an approach to validating SystemC designs by generating high coverage tests using symbolic execution. We have developed the symbolic execution engine, SESC, to symbolically execute SystemC designs. We have evaluated the proposed approach on a set of 11 SystemC designs. The results of our experiments demonstrate that the proposed approach is able to generate test cases that achieve high code coverage with modest time and memory usage. Moreover, our experiment on a RISC CPU design with more than 2K lines of SystemC code illustrates that our approach scales to designs of practical sizes.

Currently, our approach is focus on synthesizable SystemC designs with the assumption that there are no races for the DUV. The major limitation is that the process type SC_METHOD is not supported yet. Our future research will explore the following three directions. First, we will support the process type SC_METHOD. Second, we will develop an algorithm to detect data races. Third, we will enlarge the set of SystemC designs that can be validated by our framework, such as transaction level models.

ACKNOWLEDGEMENT

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

REFERENCES

[1] IEEE Standards Association, "Standard SystemC Language Reference Manual," IEEE Std. 1666-2011, 2011.
[2] J. Bhasker, *A SystemC Primer,* Star Galaxy, 2002.
[3] M. Y. Vardi, "Formal Techniques for SystemC Verification," in *DAC*, 2007.
[4] E. M. Clarke Jr., O. Grumberg and D. A. Peled, *Model Checking,* MIT Press, 1999.
[5] Y. Wolfsthal and R. M. Gott, "Formal Verification: Is It Real Enough?" in *DAC*, 2005.
[6] J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, 1976.
[7] C. Cadar, D. Dunbar and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *OSDI*, 2008.
[8] P. Herber, J. Fellmuth and S. Glesner, "Model Checking SystemC Designs Using Timed Automata," in *Proc. 6th Int. Conf. Hardware/Software Codesign and System Synthesis*, 2008.
[9] D. Große, H. M. Le and R. Drechsler, "Proving Transaction and System-level Properties of Untimed SystemC TLM Designs," in *MEMOCODE*, 2010.
[10] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri, "KRATOS: A Software Model Checker for SystemC," in *CAV*, 2011.
[11] C.-N. Chou, Y.-S. Ho, C. Hsieh, and C.-Y. Huang, "Symbolic Model Checking on SystemC Designs," in *DAC*, 2012.
[12] H. M. Le, D. Große, V. Herdt, and R. Drechsler, "Verifying SystemC Using an Intermediate Verification Language and Symbolic Simulation," in *DAC*, 2013.
[13] A. D. Junior and D. J. Cecilio da Silva, "Code-coverage Based Test Vector Generation for SystemC Designs," in *Proc. IEEE Computer Society Annu. Symp. on VLSI*, 2007.
[14] A. Sen and M. S. Abadir, "Coverage Metrics for Verification of Concurrent SystemC Designs Using Mutation Testing," in *HLDVT*, 2010.
[15] M. Chen, P. Mishra, and D. Kalita, "Automatic RTL Test Generation from SystemC TLM Specifications," *ACM Transaction on Embedded Computing System*, 2012.
[16] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. Int. Symp. Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004.
[17] Accellera Systems Initiative, *SystemC Synthesizable Subset Version 1.4 Draft*, 2015.
[18] http://www.cprover.org/hardware/sequential-equivalence
[19] B. Schafer and A. Mahapatra, "S2CBench: Synthesizable SystemC Benchmark Suite for High-Level Synthesis," *IEEE Embedded Systems Letters*, 2014.
[20] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[21] http://clang.llvm.org/docs/UsersManual.html
[22] Y. K Malaiya, N. Li, J. Bieman, R. Karcich, and B. Skibbe, "The Relationship between Test Coverage and Reliability," in *Proc. 5th Int. Symp. Software Reliability Engineering*, 1994.