Towards Verifying Determinism of SystemC Designs

Hoang M. Le[1] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{hle,drechsle}@informatik.uni-bremen.de

***Abstract*—Ensuring the correctness of high-level SystemC designs is an important and challenging problem in today's Electronic System Level (ESL) methodology. Prevalently, a design is checked against a functional specification given by e.g. a testcase with reference output or a user-defined property. Another research direction takes the view of a SystemC design as a piece of concurrent software. The design is then checked for common concurrency problems and thus, a functional specification is not required. Along this line, several methods for deadlock detection and race analysis have been developed.**

In this work, we propose to consider a new concurrency verification problem, namely input-output determinism, for SystemC designs. That means for each possible input, the design must produce the same output under any valid process schedule. We argue that determinism verification is stronger than both deadlock detection and race analysis. Beside being an attractive correctness criterion itself, proven determinism helps to accelerate both simulative and formal verification. We also present a preliminary study to show the feasibility of determinism verification for SystemC designs.

I. Introduction

The so-called *Electronic System Level* (ESL) methodology [1] has become state of the art for the design and verification of today's complex electronic systems. The essential idea is to start the design and verification process at a high level of abstraction using a system modeling language such as the de facto standard SystemC [2]. Here, the functionality of the system is realized and evaluated in an abstract fashion, typically using algorithmic modeling or *Transaction Level Modeling* (TLM) [3] techniques. From the first abstract design, the RTL implementation is obtained by successive refinement steps across different levels of timing accuracy. During this process, it is important to detect errors in the SystemC models as early as possible to prevent costly late changes or product delay.

Functional verification of SystemC designs is therefore of major interest. The main challenge is the huge verification space of a SystemC design that consists of all valid inputs and all possible process schedules. Among existing academic and industrial approaches, simulation is most widely employed due to its scalability and ease of use. Simulation-based approaches apply test vectors to the design and then check produced outputs against reference outputs, or alternatively monitor user-defined temporal properties during simulation [4], [5]. The shortcoming of simulation is that it considers only one possible schedule for a given data input, resulting in a poor coverage of the verification space. Methods based on *Partial Order Reduction* (POR) have been proposed [6], [7] to address this issue. They explore all possible scheduling sequences of SystemC processes, however, only for a given data input. The complete coverage of both inputs and process schedules can only be ensured by formal verification approaches [8], [9], [10], [11]. These verify a design exhaustively against a given property, but do not yet scale to large designs.

In contrast to conventional functional verification, which requires a functional specification (reference outputs or properties), another research direction takes the view of SystemC designs as concurrent software programs and checks them for common concurrency problems. It is advantageous since a correct design should be free of these problems and the check can be applied even if a functional specification is not yet available. So far, deadlock detection [12], [13] and data race analysis [7], [14] have been considered.

Along this line of research, the paper makes two contributions:

1) We propose to examine a new concurrency verification problem, namely input-output determinism, for SystemC designs. That means for each possible input, the design must produce the same output under any valid process schedule. Determinism verification has been considered for concurrent software [15], [16], however, approaches for SystemC requires special consideration of its concurrency semantics. In SystemC context, we also show that determinism verification is stronger than both deadlock detection and race analysis, and discuss the benefits of determinism regarding enhancement of both simulative and formal verification.
2) We propose and evaluate a first solution to demonstrate the feasibility of determinism verification for SystemC designs. The implemented solution executes symbolically two versions of a design: a version with only one single schedule and an encoding of all possible schedules, and asserts the equivalence of produced outputs. Hence, the verification result is complete, i.e. either determinism is proved or a counter-example is found.

The remainder of the paper is organized as follows: Section II summarizes SystemC semantics and its encoding for formal verification. Section III motivates determinism verification by an example. Section IV discusses the usefulness of

978-3-9815370-2-4/DATE14/

```
1  while (runnable_count > 0) { // time loop
2    while (runnable_count > 0) { // delta cycle loop
3      while (runnable_count > 0) { // evaluation loop
4        choose_one_runnable_process();
5        runnable_count−−;
6        if (process 1 is chosen) process_1();
7        ...
8        if (process n is chosen) process_n();
9      }
10     // delta notification
11     if (event 1 has been delta notified)
12       make_all_waiting_processes_runnable();
13     ...
14     if (event m has been delta notified)
15       make_all_waiting_processes_runnable();
16   }
17   // timed notification
18   t = get_smallest_notification_delay();
19   advance_simulation_time_by(t);
20   reduce_{all}_delays_by(t);
21   if (notification delay of event 1 == 0)
22     make_all_waiting_processes_runnable();
23   ...
24   if (notification delay of event m == 0)
25     make_all_waiting_processes_runnable();
26 }
```

Fig. 1. Generated SystemC scheduler for n processes and m events

proven determinism in more detail. Section V presents our preliminary study. Finally, Section VI concludes the paper.

II. Preliminaries

A. SystemC Concurrency Semantics

SystemC follows a non-preemptive semantics which allows a process to execute until it finishes or explicitly calls wait() for synchronization. This semantics can be summarized as the following steps [17]:

1) Initialization: Processes are set to be runnable.
2) Evaluation: A runnable process is executed or resumes its execution. In case of immediate notification a waiting process becomes runnable immediately. This step is repeated until no more processes are runnable.
3) Update: Updates of signals and channels are performed.
4) Delta notification phase: If there are delta notifications, the waiting processes are made runnable, and then it is continued with Step 2.
5) Timed notification phase: If there are timed notifications, the simulation time is advanced to the earliest one, the waiting processes are made runnable, and it is continued with Step 2. Otherwise the simulation is stopped.

B. Encoding of SystemC Semantics for Formal Verification

Existing formal verification approaches such as [8], [11] first translate a SystemC design to C. In this section we briefly summarize such a translation, which consists mainly of three steps:

1) The design structure is statically resolved. Modules, channels, and other objects are flattened into global variables and functions. At the end of this step, the design becomes a set of SystemC processes communicating over shared variables.
2) A static scheduler implementing the SystemC semantics is generated. The scheduler skeleton is illustrated in Fig. 1. Note that before the depicted scheduler loop is entered, each process gets a global variable indicating its status (e.g. runnable or waiting). Non-deterministic choice of which runnable process to be executed next, is embedded into the evaluation loop (Line 4 in Fig. 1). This allows a model checker to explore *all interleavings* implicitly.
3) The handling of events is implemented by manipulating a set of Boolean and integer variables (e.g. notification flag and delay). The suspension and resumption of processes is mapped to jumping between appropriate labels.

After the translation, a C model checker can be applied to verify the translated model formally and relate the verification result back to the original SystemC model. Alternatively, such a translation can omit the encoding of the scheduler. In this case, the concurrency semantics of SystemC must be directly integrated into the model checking algorithms (see for example [10], [11]).

III. Motivating Example

Fig. 2 shows a SystemC example that would benefit from determinism verification. The example is loosely based on the benchmark *B1* from [7]. The behavior of the design can be summarized by the following loop: it receives an integer input and then non-deterministically selects one of two possible computation paths. The input value is increased by two and then by three along the first path and in reverse order along the second. Afterwards the computed value is outputted and the design is ready again to receive a new input. The implementation uses three clocked processes *c*, *p1*, and *p2*. The process *c* performs the actual computation using a state machine, while *p1* and *p2* are responsible for the non-deterministic path selection. Two computation paths are identified by the state sequences (0, 1, 3, 5) and (0, 2, 4, 5), respectively. The shared variable *locked* ensures that only one path can be selected. Due to the presence of three concurrent processes, there are many possible schedules. However, the output is always equal to the input increased by five, i.e. the input-output behavior of the design is deterministic.

Due to a bug in Line 8 (*x += 2* instead of *x += 3*), the first computation path actually adds only four to the input value. This bug is detected by determinism verification *without knowing the functional specification* above since the results differ in two computation paths.

After fixing the bug, we consider the race analysis approach proposed in [7]. This approach computes a very precise race condition using model checking and can also refute the dependency between two SystemC processes reported by conventional read/write analysis. For the example, it reveals races between each pair of processes, e.g. *p1* and *p2* competing for *locked* or when *locked* is not set, the execution order *c* then *p1* would lead to other result than *p1* then *c*. POR techniques

```
1  SC_MODULE(Model) {
2    ...
3    void c() {
4      switch (state) {
5        case 0: x = input; locked = false; break;
6        case 1: x += 2; state = 3; break;
7        case 2: x += 3; state = 4; break;
8        case 3: x += 2; state = 5; break;
9        case 4: x += 2; state = 5; break;
10       case 5: output = x; state = 0; break;
11       default:
12     }
13   }
14   void p1() {
15     if (!locked) { state = 1; locked = true; }
16   }
17   void p2() {
18     if (!locked) { state = 2; locked = true; }
19   }
20   SC_CTOR(Model) {
21     locked = true; state = 0;
22     SC_METHOD(c); sensitive << clk;
23     SC_METHOD(p1); sensitive << clk;
24     SC_METHOD(p2); sensitive << clk;
25   }
26 };
```

Fig. 2. SystemC example

proposed in [7] based on these precise race conditions are unable to reduce the number of process schedules to only one. As a consequence, a simulation over many time steps would be very expensive, while such a reduction is clearly possible with proven determinism.

IV. Benefits of Proven Determinism

In this section, we discuss the usefulness of proving determinism in more detail. First, determinism is a valuable correctness criterion: with the exception of intended non-deterministic outputs, a design should not produce different results for a given input. As motivated by the example, verifying determinism of a design can reveal errors without the need of a functional specification. While the same claim can be made for deadlock detection and race analysis, determinism verification is stronger than both:

- Deadlock-freedom does not ensure determinism. On the other hand, determinism verification can detect deadlocks in a design. Because in deadlocked schedules, the design will produce no output, which is considered to be different than the output from normal schedules.
- Designs with (intended) races can nonetheless be deterministic as can be observed in the SystemC example.

The second benefit is that functional verification can be accelerated in the presence of proven determinism. As can be seen in the example, proven determinism can reduce the number of schedules that need to be considered to only one. Thus, the time of each run for both simulation-based and formal functional verification approaches can be remarkably decreased. The gain is even more significant in consideration of the whole verification process which consists of many runs, since the same schedule space is not explored again and again

```
1  while (runnable_count > 0) { // evaluation loop
2    if (process 1 is runnable) process_1();
3    ...
4    if (process n is runnable) process_n();
5  }
```

Fig. 3. Evaluation loop for one single schedule

in each run. Moreover, if the whole design cannot be proved to be deterministic, proving determinism for many parts of it and combining the results can also achieve strong reduction. For example, consider a fictive big design chaining many instances of the SystemC example. A further application of proven determinism is in equivalence checking of two SystemC designs. With both designs being proved to be deterministic, one only needs to check the behavioral equivalence of them under two arbitrary schedules instead of examining the cross product of two schedule spaces.

V. Preliminary Study

In this section, we propose a solution for determinism verification and describe our first experimental evaluation.

A. First Solution

Since the determinism criterion requires to consider all possible inputs and schedules, the use of formal methods is necessary. For the first step towards efficient solutions, we propose to adapt existing formal verification techniques for SystemC to perform determinism verification. The basic idea is as follows: We calculate the output of the design (symbolically because of the non-deterministic input) under one specific schedule, then execute the design under all possible schedules and require that the output is always equal to the calculated one. As mentioned in Section II, there are two main approaches for formal verification of SystemC: one integrates the scheduler explicitly in the model checking algorithm while the other approach encodes the scheduler (and thus all possible schedules) into the verification instance.

In the following, we focus on adapting the second approach, which has been implemented and evaluated in this paper. To obtain the calculated output under one schedule, one could manipulate the encoding of all schedules to separate that one schedule from the rest, but there is no apparent way to do it. Instead, we use a second encoding which includes only one schedule. Such an encoding can be obtained by slightly modifying the generated scheduler from Fig. 1. The modification applies to the evaluation loop and is depicted in Fig. 3. As can be seen, instead of non-deterministic choices, a process is executed immediately if it is runnable. The encoding generated using this modification is denoted as E_{single}, while the encoding with all schedules is referred to as E_{all}. After their generation, the two encodings are combined to create a harness for determinism verification shown in Fig. 4. First, the single schedule is executed, then the input to E_{all} is constrained to be equal to the input of E_{single}. After the execution of E_{all}, the output of both E_{single} and E_{all} are asserted to be equal. Recall that the encoding is given in C, the

E_{single};
$assume(I_{\text{all}} = I_{\text{single}})$;
E_{all};
$assert(O_{\text{all}} = O_{\text{single}})$;

Fig. 4. Harness for determinism verification

harness is passed to a C model checker as a C program (here we use CBMC [18]). CBMC executes the harness symbolically and thus can either prove or refute the determinism of the considered design. In the latter case, a counter-example is returned. Then, the concrete input values, two conflicting schedules, and the different output values can be extracted, which helps to debug the error. The limitation of the proposed harness is that it requires the number of input and output values to be statically determinable and constant over all possible schedules.

B. Experiments

The above solution has been evaluated on the example. As mention in Section III, the example is based on the benchmark B1 from [7], which is a difficult instance for exhaustive simulation with POR according to this paper[1]. In addition, determinism verification must also cover the whole input space of 2^{32} possible values (the range of an *int*). To test the scalability of the solution, we also enlarge the example by adding more inputs and more processes. The design *example1* has two more inputs resulting in a input space of 2^{96} possible values. The design *example2* has been obtained by duplicating the example, i.e. it has six processes and two inputs. The results obtained on an AMD Phenom 3.4 GHz Linux machine are presented in Table I. The first three columns describe the name of the design, the number of processes and the number of *int*-inputs, respectively. Designs with the suffix *bug* in their name contain the error described in Section III. The fourth column shows the depth of the design, i.e. the number of iterations of the evaluation loop (see Fig. 1) required to produce an output. The last two columns present the result and verification time, respectively. As can be seen, determinism of the (faulty) example has been quickly proved or refuted, respectively. The verification time increases slightly with the tripling of the number of inputs for *example1*. However, the determinism of *example2* could not be proved within one hour, indicating that the increase of the number of processes is much more costly. Nevertheless, this design can be divided into two independent parts, each of which is equivalent to the example. A compositional approach for determinism verification would be of help here.

TABLE I
RESULTS OF DETERMINISM VERIFICATION

Design	Process	*int*-Input	Depth	Result	Time
$example\_bug$	3	1	23	failed	3.36s
$example$	3	1	23	ok	9.54s
$example1\_bug$	3	3	23	failed	6.71s
$example1$	3	3	23	ok	33.72s
$example2\_bug$	6	2	41	failed	20.30s
$example2$	6	2	41	?	>3600.00s

VI. CONCLUSIONS

The paper identified input-output determinism as a new verification problem for SystemC designs and discussed its benefits as correctness criterion and performance enhancement for functional verification. In the presented preliminary study, we implemented a first determinism verification solution and evaluated it on several designs. Our experiments showed that for determinism verification, adapting existing formal verification techniques for SystemC is a feasible approach. Our work-in-progress currently involves the adaption of the other formal verification approaches for SystemC. Then, we would like to evaluate these solutions more thoroughly and quantify the discussed benefits of proven determinism on real-life designs. To develop more efficient solutions, we also might need to adapt determinism verification techniques for multi-threaded software to the context of SystemC as well as investigate compositional approaches.

[1]The simulation takes nearly thirty minutes to complete fifteen steps on a 3 GHz Linux machine.

ACKNOWLEDGEMENTS

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project EffektiV under contract no. 01IS13022E and by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1. The authors would like to thank Mathias Soeken for taking the time to review an early version of this paper. We are also grateful for the valuable comments and suggestions from the anonymous reviewers.

REFERENCES

[1] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann/Elsevier, 2007.
[2] Accellera Systems Initiative. (2013) SystemC 2.3 (includes TLM). [Online]. Available: www.accellera.org
[3] F. Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer, 2006.
[4] L. Ferro and L. Pierre, "ISIS: Runtime verification of TLM platforms," in *FDL*, 2009, pp. 1–6.
[5] D. Tabakov and M. Y. Vardi, "Monitoring temporal SystemC properties," in *MEMOCODE*, 2010, pp. 123–132.
[6] S. Kundu, M. Ganai, and R. Gupta, "Partial order reduction for scalable testing of SystemC TLM designs," in *DAC*, 2008, pp. 936–941.
[7] N. Blanc and D. Kroening, "Race analysis for SystemC using model checking," *ACM Trans. on Design Automation of Electronic Systems*, vol. 15, pp. 21:1–21:32, 2010.
[8] D. Große, H. M. Le, and R. Drechsler, "Proving transaction and system-level properties of untimed SystemC TLM designs," in *MEMOCODE*, 2010, pp. 113–122.
[9] C.-N. Chou, Y.-S. Ho, C. Hsieh, and C.-Y. R. Huang, "Symbolic model checking on SystemC designs," in *DAC*, 2012, pp. 327–333.
[10] H. M. Le, D. Große, V. Herdt, and R. Drechsler, "Verifying SystemC using an intermediate verification language and symbolic simulation," in *DAC*, 2013, pp. 116:1–6.
[11] A. Cimatti, I. Narasamdya, and M. Roveri, "Software model checking SystemC," *IEEE Trans. on CAD*, vol. 32, no. 5, pp. 774–787, 2013.
[12] A. Sen, V. Ogale, and M. S. Abadir, "Predictive runtime verification of multi-processor SoCs in SystemC," in *DAC*, 2008, pp. 948–953.
[13] C.-N. Chou, C.-H. Hsu, Y.-T. Chao, and S.-L. Huang, "Formal deadlock checking on high-level SystemC designs," in *ICCAD*, 2010, pp. 794–799.
[14] M. Moiseev, M. Glukhikh, A. Zakharov, and H. Richter, "A static analysis approach to data race detection in systemc designs," in *DDECS*, 2013, pp. 54–59.
[15] J. Burnim and K. Sen, "Asserting and checking determinism for multithreaded programs," in *ESEC/FSE*, 2009, pp. 3–12.
[16] M. Vechev, E. Yahav, R. Raman, and V. Sarkar, "Automatic verification of determinism for structured parallel programs," in *SAS*, 2010, pp. 455–471.
[17] *IEEE Standard SystemC LRM*, IEEE Std. 1666, 2011.
[18] CPROVER. (2013) CBMC. [Online]. Available: www.cprover.org/cbmc