Verifying SystemC using an Intermediate Verification Language and Symbolic Simulation*

Hoang M. Le[1] Daniel Große[2] Vladimir Herdt[1] Rolf Drechsler[1,3]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]solvertec GmbH, 28359 Bremen, Germany
[3]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{hle, vherdt, drechsle}@informatik.uni-bremen.de grosse@solvertec.de

ABSTRACT

Formal verification of SystemC is challenging. Before dealing with symbolic inputs and the concurrency semantics, a front-end is required to translate the design to a formal model. The lack of such front-ends has hampered the development of efficient back-ends so far.

In this paper, we propose an isolated approach by using an *Intermediate Verification Language* (IVL). This enables a SystemC-to-IVL translator (frond-end) and an IVL verifier (back-end) to be developed independently. We present a compact but general IVL that together with an extensive benchmark set will facilitate future research.

Furthermore, we propose an efficient symbolic simulator integrating Partial Order Reduction. Experimental comparison with existing approaches has shown its potential.

1. INTRODUCTION

The system modeling language SystemC [16, 12] is being widely adopted to create golden models in the *Electronic System Level* (ESL) design and verification flow [2]. The golden models are developed using a behavioral/algorithmic style in combination with abstract communication based on *Transaction Level Modeling* (TLM) [16]. Ensuring the correctness of these models is of major importance since undetected errors become very expensive in later design steps. To verify the abstract SystemC models, the straight-forward approach is simulation offered already by the free event-driven simulation kernel shipped with the SystemC class library [1]. Substantial improvements have been proposed by supporting the validation of (TLM) assertions, see e.g. [4, 8, 9, 23]. To further enhance simulation coverage, methods based on *Partial Order Reduction* (POR) have been proposed [19, 3]. They allow to explore all possible scheduling sequences of SystemC processes for a given data input. However, representative inputs are still needed. Therefore, formal approaches for SystemC TLM have been devised. But due to the object-oriented nature of SystemC *and* its sophisticated synchronous and asynchronous simulation semantics, formal verification is very challenging [25].

We review existing formal approaches in detail in Section 2. With some exceptions[1], they use freely available SystemC parsers and thus are hampered by their limitations as detailed in [20]. At the same time most of the existing approaches translate the SystemC design into a formal representation. These representations are similar in their expressiveness which motivates the first major contribution of this paper: the *Intermediate Verification Language* (IVL). The properties of IVL and the resulting advantages include:

- Compact, intuitive and readable language: IVL has been designed in such a way that both manual and automatic transformations from SystemC are possible.
- Independent development of front-end and back-end: IVL enables to focus on the problem that the user wants to address.
- Open language and support: IVL is open and a free parser is provided. Moreover, all freely available benchmarks used by existing formal verification approaches for SystemC have been transformed into an extensive IVL benchmark set. This accelerates research in particular with respect to new formal approaches.

Based on IVL the second contribution of this paper is an *efficient symbolic simulator*. The novelty of our simulator is to combine and adapt two efficient verification techniques – POR [11, 10] and symbolic execution [5] – under the simulation semantics of SystemC. While POR prunes redundant process scheduling sequences, symbolic execution efficiently explores all conditional execution paths of each individual process in conjunction with symbolic inputs. Subsequently, the simulator covers all possible inputs and scheduling sequences of the design exhaustively. It supports both static and dynamic POR and is configurable with respect to the search algorithm for state traversal. For the first time we provide a full experimental comparison of all available state-of-the-art formal approaches. This comparison also demonstrates clearly the potential of our proposed approach.

The rest of this paper is organized as follows. In Section 2 we review related work mainly on formal verification of SystemC. Section 3 gives a brief introduction to SystemC. Then, in Section 4 the IVL is introduced. The symbolic simulator is presented in Section 5. Section 6 gives the experimental results. Finally, the paper is concluded in Section 7.

*This work was supported in part the German Federal Ministry of Education and Research (BMBF) within the project SANITAS under contract no. 16M3088 and by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1.

[1] The parser is either undocumented or a proprietary tool.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC '13, May 29 - June 07 2013, Austin, TX, USA.
Copyright 2013 ACM 978-1-4503-2701-9/13/05 ...$15.00.

2. RELATED WORK

A handful of formal verification approaches for SystemC TLM have been proposed. Early efforts, for example [21, 17, 24], have very limited scalability or do not model the SystemC simulation semantics thoroughly [18]. Among the more recent approaches, the following four are the most promising and currently represent the state-of-the-art.

STATE, first proposed in [15], translates SystemC designs to timed automata. With STATE it is not possible to verify properties on SystemC designs directly. Instead, they have to be formulated on the automata and can then be checked using the UPPAAL model checker.

SCIVER [13] translates SystemC designs into sequential C models first. Temporal properties using an extension of PSL [22] can be formulated and integrated into the C model during generation. Then, C model checkers can be applied to check for assertion violations. High-level induction on the generated C model has been proposed to achieve completeness and efficiency. However, no dedicated techniques to prune redundant scheduling sequences are provided.

KRATOS [7] translates SystemC designs into threaded C models. Then, the ESST algorithm is employed, which combines an explicit scheduler and symbolic lazy abstraction. POR techniques are also integrated into the explicit scheduler. For property specification, simple C assertions are supported. The main performance bottleneck of KRATOS is the potentially slow abstraction refinements.

SDSS [6] formalizes the semantics of SystemC designs in terms of Kripke structures. Then, BMC and induction can be applied in a similar manner as SCIVER. The main difference is that the scheduler is not involved in the encoding of SDSS. It is rather explicitly executed to generate an SMT formula that covers the whole state space. Still, no dedicated techniques to handle equivalent scheduling sequences are supported. SDSS allows simple properties reasoning about variable values at the beginning of each evaluation phase.

With respect to our proposed IVL, the threaded C programs used by KRATOS are the most close representation. However, KRATOS just employs this representation as a means to enable explicit process scheduling in its model checking algorithm. It is not designed as an IVL and thus for example it is not documented whether the full language or which subset of C can be used. KRATOS itself appears to support only a very small fraction of C. The parser of KRATOS for this representation is also not available. Furthermore, the SystemC-related constructs cannot be cleanly separated, e.g. processes and channel updates are identified by function name prefixes, events are declared as enum values, etc. The new SystemC constructs for process control such as *suspend* and *resume* (cf. SystemC 2.3) are also not supported.

3. BACKGROUND ON SYSTEMC

In the following only the essential aspects of SystemC are described. SystemC has been implemented as a C++ class library, which includes an event-driven simulation kernel. The structure of the system is described with ports and modules, whereas the behavior is described in processes which are triggered by events and communicate through channels. A process gains the *runnable* status when one or more events of its sensitivity list have been notified. The simulation kernel selects one of the runnable processes and gives this process the control. The execution of a process is non-preemptive, i.e. the kernel receives the control back if the process has finished its execution or suspends itself by calling *wait()*. SystemC provides three types of processes with *SC_THREAD* being the most general type, i.e. the other two can be modeled by using SC_THREAD. For event-based synchronization, SystemC offers many variants of *wait()* and *notify()* such as *wait(time)*, *wait(event)*, *event.notify(delay)*, *event.notify()*, etc.

The simulation semantics of SystemC can be summarized as follows [16]: First, the system is elaborated, i.e. instantiation of modules and binding of channels and ports is carried out. Then, there are the following steps to process:

1. *Initialization*: Processes are made 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 channels are performed. These updates have been requested during the evaluation phase.
4. *Delta notification*: If there are delta notifications, the waiting processes are made runnable, and then the simulation is continued with the Evaluation step.
5. *Timed notification*: If there are timed notifications, the simulation time is advanced to the earliest one, the waiting processes are made runnable, and the simulation is continued with the Evaluation step. Otherwise the simulation is stopped.

In the next section, we define the IVL based on this simulation semantics.

4. INTERMEDIATE VERIFICATION LANGUAGE

The IVL is the stepping stone between a front-end and a back-end. Ideally, it should be compact and easily manageable but at the same time powerful enough to allow the translation of SystemC designs. Our view is that a back-end should focus purely on the behavior of the considered SystemC design. This behavior is fully captured by the SystemC processes under the simulation semantics of the SystemC kernel. Therefore, a front-end should first perform the elaboration phase, i.e. determine the binding of ports and channels. Then it should extract and map the design behavior to the IVL, whose elements are detailed in the following.

Based on the simulation semantics described above, we identify the three basic components of the SystemC kernel: *SC_THREAD*, *sc_event* and *channel update*. These are adopted to be *kernel primitives* of the IVL: *thread*, *event* and *update*, respectively. Associated to them are the following primitive functions:

- *suspend* and *resume* to suspend and resume a thread, respectively;
- *wait* and *notify* to wait for and notify an event (the notification can be either immediate or delayed depending on the function arguments);
- *request_update* to request an update to be performed during the update phase.

These primitives form the backbone of the kernel. Other SystemC constructs such as *sc_signal*, *sc_mutex*, static sensitivity, etc. can be modeled using this backbone.

```
1   SC_MODULE(Module) {
2     sc_core::sc_event e;
3     uint x, a, b;
4
5     SC_CTOR(Module)
6       : x(rand()), a(0)
7       , b(0) {
8       SC_THREAD(A);
9       SC_THREAD(B);
10      SC_THREAD(C);
11    }
12
13    void A() {
14      if (x % 2)
15        a = 1;
16      else
17        a = 0;
18    }
19
20    void B() {
21      e.wait();
22      b = x / 2;
23    }
24
25    void C() {
26      e.notify();
27    }
28  };
29
30  int sc_main() {
31    Module m("top");
32    sc_start();
33     assert(2 * m.b + m.a
          == m.x);
34    return 0;
35  }
```

Figure 1: A SystemC example

```
1   event e
2    uint x = ?<uint>
3   uint a = 0
4   uint b = 0
5
6   thread A begin
7     if x % 2 goto elseif
8       a = 0
9       goto endif
10    elseif:
11      a = 1
12    endif:
13  end
14
15  thread B begin
16    wait e
17    b = x / 2
18  end
19
20  thread C begin
21    notify e
22  end
23
24  main begin
25    start
26    assert 2 * b + a == x
27  end
```

Figure 2: The example in IVL

The behavior of a *thread* or an *update* is defined by a function. Functions which are neither *threads* nor *updates* can also be declared. Every function possesses a body which is a list of statements. We allow only assignments, (conditional) goto statements and function calls. Every structural control statement (*if-then-else*, *while-do*, *switch-case*, etc.) can be mapped to conditional *goto* statements (this task should also be performed by the front-end). Therefore, the representation of a function body as a list of statements is general and at the same time much more manageable for a back-end.

As *data primitives* the IVL supports Boolean and integer data types of C++ together with all arithmetic and logic operators. Furthermore, arrays and pointers of primitive types are also supported. Additionally, bit-vectors of finite width can be declared. This enables the modeling of SystemC data types such as *sc_int* or *sc_uint* in the IVL.

For verification purpose, the IVL provides *assert* and *assume*. More expressive temporal properties can be translated to FSMs and embedded into an IVL description by a front-end. Symbolic values of primitive types are also supported.

SystemC Example.

Figure 1 shows a simple SystemC example. The main purpose of the example is to demonstrate some elements of the IVL. The design has one module and three SC_THREADs A, B and C. Thread A sets variable a to 0, if x is divisible by 2, and to 1 otherwise (Line 14-17). Variable x is initialized with a random integer value on Line 6 (i.e. it models an input). Thread B waits for the notification of event e and sets $b = x\ /\ 2$ subsequently (Line 21-22). Thread C performs an immediate notification of event e (Line 26). If thread B is not already waiting for it, the notification is lost. After the simulation the value of variable a and b should be x % 2 and $x\ /\ 2$, respectively. Thus the assertion $(2*b+a == x)$ is expected to hold (Line 33). Nevertheless, there exist counterexamples, for example the scheduling sequence CAB leads to a violation of the assertion. The reason is that b has not been set correctly due to the lost notification.

IVL Example.

Figure 2 depicts the same example in IVL. As can be seen the SystemC module is "unpacked", i.e. variables, functions, and threads of the module are now global declarations. The calls to *wait* and *notify* are directly mapped to statements of the same name. The *if-then-else* block of thread A is converted to a combination of conditional and unconditional goto statements (Line 7-12). Variable x is initialized with a symbolic integer value (Line 2) and can have any value in the range of *unsigned int*. The statement *start* on Line 25 starts the simulation.

In short, the IVL is kept minimal but expressive enough for the purpose of formal verification. It covers all benchmarks used by existing formal verification approaches for SystemC. It would only take little effort to adapt these approaches to support this IVL as their input language. That would lead to the availability of a checker suite for SystemC once a capable front-end is fully developed. A grammar and a parser for the IVL are provided at our website[2].

In the next section, we present a new efficient verifier combining symbolic execution and POR. We also refer to an IVL description as a SystemC design since both define the same behavior.

5. SYMBOLIC SIMULATION

In this section we present a symbolic simulator for the SystemC IVL. The simulator can discover assertion violations and other types of errors such as division by zero or memory access violation. Our approach enables exhaustive exploration of the state space by providing support for symbolic values and taking all possible scheduling sequences into consideration. As a pre-processing step, all non-primitive function calls are inlined.

Figure 3 shows the complete search tree (i.e. state space) for the example from last section. As can be seen, even this very simple design has a total of 14 possible execution paths. Only six of them violate the assertion (paths that end with a filled box). The circles and boxes correspond to *execution states* of the design. An execution state is split at a $\diamond$ node in case of a conditional *goto* statement (explained later in Section 5.1)

Execution State.

An execution state contains values of all variables, states of all threads, a pending event notification list, a pending update list, a *path condition* and a Boolean flag indicating whether this is a split state. The state of a thread consists of a status (either *runnable*, *blocked* or *terminated*) and a *statement pointer* (SP) that determines the next statement to be executed. The path condition describes the constraint on the variables, which must be satisfied to reach this execution state from the initial one. Also note that variables have in general no concrete values, their values are rather expressions of symbolic and concrete values. Take the initial execution state in Figure 3 (the uppermost node) as an example: The variable x is initialized as a symbolic value, while a and b have the initial value of zero. Each thread A, B or C is *runnable* and has a SP pointing to the first statement of the thread body (Line 7, 16 and 21 in Figure 2,

[2] www.systemc-verification.org/sissi

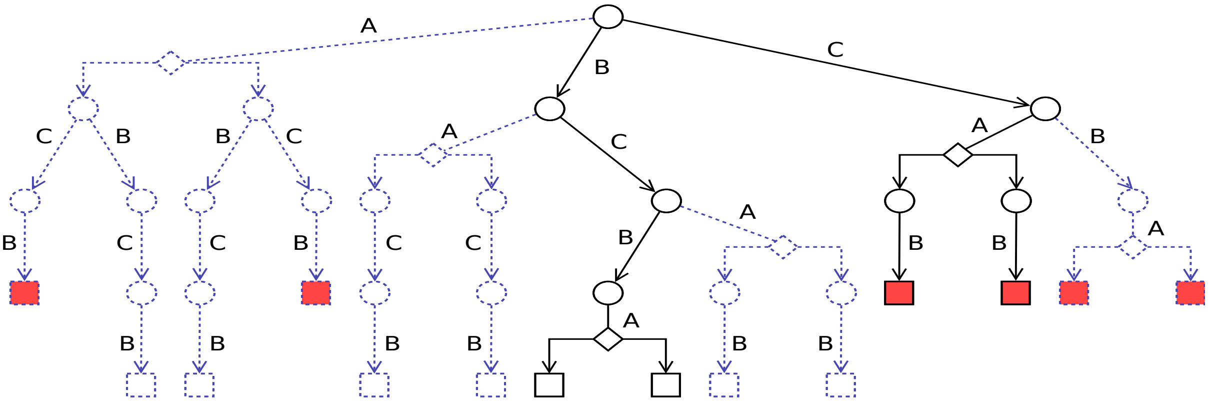

Figure 3: Search tree for the example

respectively). There are no pending event notifications, no pending updates and yet no path condition. Now, the execution of thread B will change its state to *blocked*, add e to the pending notification list, and update its SP to Line 17. All other values remains unchanged.

Every edge depicts a transition between two states and its label shows which thread has been selected and executed in this transition. The execution of B mentioned above thus corresponds to the outgoing edge with label B from the uppermost node. The differences between dashed and solid edges are explained later in Section 5.3. In the following we describe the main components of the simulator: scheduler, interpreter and employed POR techniques.

5.1 Scheduler

The scheduler manages the set of execution states. It selects an unvisited execution state and explores all interleavings in this state. That means for each *runnable* thread, the control is given to the interpreter to execute it. The interpreter implements symbolic execution and is responsible for the handling of symbolic values and the detection of errors during thread execution (see Section 5.2).

The scheduler receives the control back from the interpreter in one of the following two cases:

1. The execution reaches the end of the thread or a *wait* statement. This case results from the simulation semantics of SystemC. The current execution state is marked as visited and a new execution state with the updated status of the thread is added.

2. The thread execution reaches a conditional *goto* statement, whose condition is a symbolic expression. Because the condition can be *true* or *false* depending on the involved symbolic values, the execution state is marked as visited and then split into two new ones for both cases. In the *true* (*false*) case, the condition (its negation) is added to the path condition, the *split* flag is set such that the thread execution can be continued later by the interpreter directly after the conditional goto. Note that, because of this direct continuation (no other runnable thread is executed), the execution of the thread is actually not preempted under the SystemC simulation semantics.

If the selected execution state has no *runnable* threads, that means the evaluation phase is completed. The scheduler performs the other phases (i.e. update, delta and timed notification). If new *runnable* threads arise after that, the exploration is continued, otherwise the end of an execution path is reached.

Now consider the first solid path from the left in Figure 3. From the initial execution state, thread B is executed and then blocked by the *wait* statement. Afterwards, a new execution state is produced and thread C is executed in this new state. This execution notifies event e and terminates. This notification makes B *runnable* again and it is executed in the next execution state. After the termination of B, A is executed. When the execution of A reaches the conditional *goto*, two new execution states are produced because the condition involves the symbolic value of x. There are no *runnable* threads and no pending notifications in both states, therefore both mark the end of an execution path.

For the state space exploration, the scheduler supports depth-first search, breath-first search, and iterative deepening. An interactive mode is also provided to enable the user to select only one *runnable* thread to execute from an execution state. This is useful for example to replay an erroneous execution path.

5.2 Interpreter

The interpreter implements symbolic execution. It executes a thread in an execution state by interpreting the statements of the thread. The execution can either be started or resumed at the statement determined by the thread SP. As mentioned in Section 4, there are three types of statements: assignments, *goto*, and function calls. For an assignment, the interpreter simply replaces the value of the left-hand side in the execution state with the right-hand side expression. In case of an unconditional *goto*, the execution continues at the specified label. Conditional *goto* is handled as follows: if the condition involves a symbolic value, the scheduler takes over as described above, otherwise the condition is evaluated and the execution continues at either the specified label or the next statement.

Since we have inlined non-primitive function calls, only calls to kernel and verification primitive functions are left to process. They are interpreted according to their semantics in SystemC as follows.

- *wait*: The thread execution is blocked and the control is given back to the scheduler as described above.

- *notify*: In case of an immediate notification, all waiting threads in the execution state are made *runnable*. Otherwise, the pending notification list is updated accordingly.

- *suspend*: The status of the to-be-suspended thread is changed to *blocked*.
- *resume*: The status of the to-be-resumed thread is changed to *runnable*.
- *request_update*: The requested update is added to the pending update list.
- *assert*: The conjunction of the path condition and the negation of the asserted expression is given to an SMT solver. If a solution can be found, the assertion is violated, and a counter-example is created from this solution and reported.
- *assume*: The given expression is added to the path condition.

In addition to assertions, the interpreter also checks for other types of errors such as division by zero or memory access violation.

5.3 Partial Order Reduction

The above described scheduler explores all possible thread scheduling sequences explicitly. Many of them can be reduced without affecting the verification result by applying POR techniques.

The basic idea is to divide a thread into several *transitions*. A transition is a list of statements that can be executed without interruption by the interpreter before the control is given back to the scheduler. A transition begins either at the start of the thread or after a *wait* statement. It ends either at the end of the thread or before another *wait* statement. Thus, every time a thread is executed, actually one of its transitions is being executed. A thread is *runnable* if one of its transitions is *runnable*.

After all transitions are identified, we establish a *dependency relation* between every pair of them. Intuitively, two transitions X and Y are dependent if two execution orders XY and YX lead to different results. We have the following cases of dependency:

- Both transitions access the same memory location identified by a variable, a pointer or an array, with at least one write access;
- A transition notifies an event immediately which the other transition waits for (e.g. thread B and C from Figure 2);
- A transition suspends the other transition by calling *suspend*.

Based on this dependency relation, the persistent set and sleep set techniques [11] are employed to identify equivalent thread scheduling sequences. These two techniques are orthogonal and can be combined to achieve better results. Basically, for each visited execution state, both techniques derive a subset of *runnable* transitions. Future exploration from this state is restricted to this transition subset.

The dependency relation can be either statically or dynamically determined. Dynamic dependency is calculated during the simulation and thus more precise than static dependency which is often an over-approximation. However, it produces a much bigger overhead in comparison to static dependency calculation. For a more detailed formal treatment of static and dynamic POR we refer to [11, 10, 19].

The dashed execution paths in Figure 3 are pruned by using static POR and must not be traversed. This reduction can be intuitively explained as follows. Because A is independent of B and C, it is unimportant when A is executed, e.g. CAB, CBA and ACB are equivalent. In contrast, the order between B and C is important due to their dependency.

5.4 Limitations

Currently, loop detection is not implemented in our symbolic simulator. For models without symbolic inputs, the symbolic simulation becomes explicit model checking and thus well-known loop detection algorithms for example in SPIN can be used. But the general case with symbolic inputs is much more interesting and challenging. Therefore, loop detection is left for future work and consequently our simulator can only be applied to models that either terminate or contain bugs.

Symbolic execution can run into the path explosion problem in some cases. For software verification, advanced techniques for path merging and redundant path elimination have been proposed and implemented in modern tools such as KLEE [5]. Our simulator does not integrate such techniques yet. Nevertheless, its potential is demonstrated by the experiments in the next section.

6. EXPERIMENTAL RESULTS

We have implemented the proposed approach in a prototype called SISSI (SystemC IVL Symbolic Simulator) using Python (version 2.7.3rc2). Our implementation also uses an intermediate SMT layer [14] that allows to switch between different solvers or also run several solvers in parallel. However, for the experiments here we just employ Boolector. Furthermore, we use two variants SISSI-S and SISSI-D which perform static and dynamic POR, respectively.

All experiments have been conducted on an AMD Phenom 3.4 GHz machine with 8 GB RAM running Linux. Time limit for each run is set to 1200 seconds. Among the four state-of-the-art approaches mentioned in Section 2, SDSS and its benchmarks are to the best of our knowledge not available. We use benchmarks taken from the websites of KRATOS[3], SCIVER[4] and STATE[5], and develop some new benchmarks as well. For each benchmark, three equivalent models are needed (in IVL for SISSI, in threaded C for KRATOS, in SystemC for SCIVER and STATE). The checked properties are source code assertions which are supported by all approaches but STATE. Hence, we need to change the models slightly before giving them to STATE.

Table 1 shows a representative excerpt of the results. The first column gives the name of the benchmark. The next columns present for each benchmark the lines of code in IVL, the verification result (Safe or Unsafe), and the verification time needed by SISSI-S, SISSI-D, KRATOS, SCIVER and STATE, respectively. Furthermore, Table 1 is divided by the dashed line into two halves. The upper half shows results for benchmarks without symbolic inputs. We believe symbolic approaches are not the right tool for these models. But since they have been used in the past for the evaluation of such approaches, we still include them in the comparison. As can be seen, SCIVER and KRATOS do not perform well on these benchmarks. SISSI and STATE resort to explicit model checking and therefore are much faster in general.

The more important results for benchmarks with symbolic inputs are presented in the lower half of Table 1. Note that for these models, STATE explores a much smaller state space

[3] `es.fbk.eu/tools/kratos`

[4] `www.systemc-verification.org/sciver`

[5] `www.pes.tu-berlin.de/state_project`

Table 1: Comparison with state-of-the-art approaches (runtime in seconds)

Benchmark	LoC	Result	SISSI-S	SISSI-D	KRATOS [7]	SCIVER [13]	STATE [15]
kundu	54	S	9.25	12.15	1.07	9.70	0.04
transmitter.10	81	U	0.05	0.34	0.07	18.63	0.03
transmitter.50	361	U	0.24	4.70	304.54	time-out	0.22
transmitter.200	1411	U	1.38	106.30	mem-out	mem-out	12.80
mem-slave-tlm.4	207	S	0.18	0.20	140.24	13.38	0.03
mem-slave-tlm.5	218	S	0.20	0.23	223.78	20.18	0.04
token-ring-bug.10	94	U	0.05	0.11	0.74	6.31	0.53
token-ring-bug.50	414	U	0.25	2.58	mem-out	time-out	mem-out
token-ring-bug.200	1614	U	1.88	57.01	mem-out	mem-out	mem-out
mem-slave-tlm-sym.4	208	S	0.18	0.23	time-out	28.33	50.42
mem-slave-tlm-sym.5	219	S	0.29	0.29	time-out	56.29	62.77
simple-fifo-1c-2p	73	U	0.13	0.10	65.22	1.65	error
simple-fifo-2c-1p	72	U	0.08	0.12	39.26	1.26	error
jpeg	230	U	0.58	0.49	time-out	22.84	error
buffer-ws-p3	51	S	0.25	0.05	2.01	0.48	mem-out
buffer-ws-p4	55	S	2.62	0.06	14.33	3.13	mem-out
buffer-ws-p5	60	S	70.80	0.07	210.82	1.95	mem-out

in comparison to the other methods since its back-end does not support the full range of C++ *int*. Still, STATE does not perform/scale well in the presence of symbolic inputs and also it does not accept some of the models (indicated as *error*). The *token-ring-bug* and *mem-slave-tlm-sym* benchmarks are basically *transmitter* and *mem-slave-tlm*, respectively, with symbolic inputs. With the exception of SISSI, these are notably harder for the model checkers. Overall, SISSI delivers clearly the best performance by far on benchmarks with symbolic inputs.

The trade-off between static and dynamic POR discussed in Section 5.3 can also be observed in the *transmitter*, *token-ring-bug* and *buffer-ws* benchmarks.

7. CONCLUSIONS

This paper makes two contributions to the formal verification of SystemC TLM. First, we present a compact, intuitive and readable *Intermediate Verification Language* (IVL) for SystemC that enables the independent development of front-ends and back-ends. With the availability of the IVL, a free parser and an extensive benchmark set, research in particular with respect to new back-ends can be accelerated. Second, we propose a new efficient symbolic simulator integrating Partial Order Reduction and symbolic execution. This combination enables the effective exploration of all possible inputs and process scheduling sequences. The experimental comparison confirms the potential of our symbolic simulator. This is also to the best of our knowledge the most comprehensive comparison of available state-of-the-art approaches.

8. REFERENCES

[1] Accellera Systems Initiative. SystemC, 2012. Available at http://www.systemc.org.

[2] B. Bailey, G. Martin, and A. Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology.* Morgan Kaufmann/Elsevier, 2007.

[3] N. Blanc and D. Kroening. Race analysis for SystemC using model checking. *ACM Trans. on Design Automation of Electronic Systems*, 15:21:1–21:32, 2010.

[4] N. Bombieri, F. Fummi, and G. Pravadelli. Incremental ABV for functional validation of TL-to-RTL design refinement. In *DATE*, pages 882–887, 2007.

[5] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[6] C.-N. Chou, Y.-S. Ho, C. Hsieh, and C.-Y. R. Huang. Symbolic model checking on SystemC designs. In *DAC*, pages 327–333, 2012.

[7] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri. Kratos - a software model checker for SystemC. In *CAV*, pages 310–316, 2011.

[8] W. Ecker, V. Esen, T. Steininger, M. Velten, and M. Hull. Implementation of a transaction level assertion framework in SystemC. In *DATE*, pages 894–899, 2007.

[9] L. Ferro and L. Pierre. ISIS: Runtime verification of TLM platforms. In *FDL*, pages 1–6, 2009.

[10] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.

[11] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer, 1996.

[12] D. Große and R. Drechsler. *Quality-Driven SystemC Design.* Springer, 2010.

[13] D. Große, H. M. Le, and R. Drechsler. Proving transaction and system-level properties of untimed SystemC TLM designs. In *MEMOCODE*, pages 113–122, 2010.

[14] F. Haedicke, S. Frehse, G. Fey, D. Große, and R. Drechsler. metaSMT: Focus on your application not on solver integration. In *DIFTS*, pages 22–29, 2011.

[15] P. Herber, J. Fellmuth, and S. Glesner. Model checking SystemC designs using timed automata. In *CODES+ISSS*, pages 131–136, 2008.

[16] IEEE Std. 1666. *IEEE Standard SystemC Language Reference Manual*, 2011.

[17] D. Karlsson, P. Eles, and Z. Peng. Formal verification of SystemC designs using a petri-net based representation. In *DATE*, pages 1228–1233, 2006.

[18] D. Kroening and N. Sharygina. Formal verification of SystemC by automatic hardware/software partitioning. In *MEMOCODE*, pages 101–110, 2005.

[19] S. Kundu, M. Ganai, and R. Gupta. Partial order reduction for scalable testing of SystemC TLM designs. In *DAC*, pages 936–941, 2008.

[20] K. Marquet, B. Karkare, and M. Moy. A theoretical and experimental review of SystemC front-ends. In *FDL*, pages 124–129, 2010.

[21] M. Moy, F. Maraninchi, and L. Maillet-Contoz. LusSy: an open tool for the analysis of systems-on-a-chip at the transaction level. *Design Automation for Embedded Systems*, pages 73–104, 2006.

[22] D. Tabakov, M. Vardi, G. Kamhi, and E. Singerman. A temporal language for SystemC. In *FMCAD*, pages 1–9, 2008.

[23] D. Tabakov and M. Y. Vardi. Monitoring temporal SystemC properties. In *MEMOCODE*, pages 123–132, 2010.

[24] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi. A SystemC/TLM semantics in promela and its possible applications. In *SPIN*, pages 204–222, 2007.

[25] M. Y. Vardi. Formal techniques for SystemC verification. In *DAC*, pages 188–192, 2007.