Compiled Symbolic Simulation for SystemC*

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

***Abstract*—Ensuring the correctness of SystemC virtual prototypes is indispensable. For such models, existing symbolic simulation approaches are based on interpreting their behavior. In this paper we propose a major enhancement called Compiled Symbolic Simulation (CSS). For more scalable state space exploration, CSS augments the DUV to integrate the symbolic execution engine and the Partial Order Reduction based scheduler. Then, a standard C++ compiler is used to generate a native binary, whose execution performs exhaustive verification of the DUV. An extensive experimental evaluation demonstrates the potential of our approach.**

I. Introduction

In todays *Electronic System Level* (ESL) [1] design flow, *Virtual Prototypes* (VPs) play a very important role. They are essentially an abstract representation of the entire hardware platform and serve as reference for embedded software development and hardware verification. For the creation of VPs, which are composed of abstract models of the individual IP blocks, the C++-based language SystemC [2], [3] has been established as a standard. By virtue of their impact on the design flow, these abstract SystemC models should be thoroughly verified.

For this verification task, simulation-based approaches are still the main work horse due to their scalability and ease-of-use. However, in contrast to formal verification, they cannot prove the absence of errors and often miss corner-case bugs. Recently, symbolic simulation approaches [4], [5], [6] have been shown to be very effective for formally verifying SystemC. Such an approach essentially combines symbolic execution [7] with *Partial Order Reduction* (POR) [8], [9]. This combination enables complete exploration of the state space of a *Design-Under-Verification* (DUV), which consists of all possible values of its inputs and all possible schedules (i.e. orders of execution) of its (typically) asynchronous SystemC processes.

* 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 and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

ICCAD '16, November 07-10, 2016, Austin, TX, USA
 ISBN 978-1-4503-4466-1/16/11...$15.00
DOI: http://dx.doi.org/10.1145/2966986.2967016

The existing symbolic simulators are commonly interpreter-based, i.e. they interpret the behavior of the DUV statement by statement. An interpreter-based symbolic simulator manages directly the execution state (i.e. the DUV state and the scheduler state) and update it according to the semantics of the to-be-interpreted statement. The main advantage of this approach is the ease of implementation. Especially, when a conditional branch is encountered or multiple orders of execution of runnable processes are possible during the symbolic simulation, cloning the current execution state for further exploration is required and can be straightforwardly implemented. The trade-off for this convenience is the relatively low performance of the overall verification, which becomes clearer when symbolic simulation is applied to large models, e.g. the barebone of a VP.

In this paper, we propose *Compiled Symbolic Simulation* (CSS) to tackle this performance issue and make the following contributions:

1) We describe a source code instrumentation technique – the main novelty of CSS – that augments the DUV to integrate the symbolic execution engine and the POR based scheduler. The augmented DUV can then be compiled using any standard C++ compiler, leveraging its efficient code optimizations, to create native binary code. This binary, when executed, can achieve the complete state space exploration much faster than interpreter-based techniques.
2) We incorporate a set of optimization techniques (e.g. path merging) tailored for CSS to improve the performance further.

An extensive set of experiments including comparisons with existing approaches demonstrates the benefits of CSS.

II. Related Work

SystemC formal verification has been an active research area in the last few years. In addition to symbolic simulation, there are also a handful of other formal verification approaches for SystemC. We refer to the Related Work section of [10] for a detailed review of these works. Notable approaches include KRATOS [10] and SCIVER [11]. The main model checking algorithm in KRATOS is performed on an intermediate representation called threaded C. The algorithm combines an explicit scheduler and symbolic lazy abstraction. POR is also integrated into the explicit scheduler to prune redundant schedules. Furthermore, KRATOS supports the transformation of a SystemC DUV into sequential C code. SCIVER also performs a similar transformation in the first step, then for

verification, it applies high-level induction on top of existing C model checkers.

Although the C code produced by KRATOS and SCIVER is executable, its execution only explores a randomly chosen path in the whole verification state space. The code is rather meant to be processed by a C model checker, which will explore all feasible execution paths exhaustively. To the best of our knowledge, SPIN [12] is the only verification tool that currently also supports compiled verification. SPIN translates a model in its input language Promela into an executable C program. Translation from SystemC to Promela has also been considered [13], [14]. In contrast to our symbolic approach, the program generated by these approaches, when compiled and executed, performs explicit model checking on the original Promela model.

III. Preliminaries

A. Intermediate Representation for SystemC

SystemC is essentially a C++ class library that includes an event-driven simulation kernel with non-preemptive semantics. The structure of a SystemC design is described with ports and modules, whereas the behavior is described in processes which are triggered by events. 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 formal verification, before dealing with the state space of a SystemC DUV, a front-end is required to translate it to a formal model. If one only wants to focus on efficient state space exploration, a popular approach to avoid the need to handle the full complexity of C++ is to employ an intermediate representation, such as the threaded C subset [10] or the *Intermediate Verification Language* (IVL) for SystemC [6].

In this paper, we also follow this approach and use the *Extended Intermediate Verification Language* (XIVL) proposed in [15]. Essentially, XIVL provides modeling primitives for SC_THREADs (called threads for simplicity), events and corresponding synchronization functions (i.e. *wait* and *notify* in different variants). Boolean and integer data types of C++ together with all arithmetic and logic operators are supported. Conditional goto statements are used to model the control flow of a thread. For verification purposes, the functions *assume* and *assert* as well as the construct $?(type)$ to model symbolic value are provided. Furthermore, XIVL also supports OOP features such as classes, inheritance, virtual methods with overrides and dynamic dispatch. These features allow to capture the established modeling styles of SystemC virtual prototypes more naturally. Fig. 1 shows a simple XIVL program with a single thread that iteratively computes the sum for a given symbolic input value n. This will serve as the running example in the remainder of this paper.

XIVL offers the same simulation semantics as the SystemC kernel [2]. Essentially, one of the runnable XIVL threads will be non-deterministically selected. This thread is then executed non-preemptively until it finishes or suspends itself by calling *wait*. This causes a context switch back to the scheduler, which can again select another runnable thread. If no runnable thread is available, the scheduler performs pending delta or timed notifications accordingly to activate waiting threads.

```
 1 int n;
 2 int sum;
 3
 4 thread A {
 5   int i = 0;
 6   loop:
 7   if (i >= n) goto
         end_loop;
 8     i += 1;
 9     sum = sum + i;
10     wait_time(1);
11     goto loop;
12   end_loop:
13 }
14
15 main {
16   sum = 0;
17   n = ?(int);
18   assume (n <= 9);
19   start;
20   assert (sum <= 45);
21 }
```

Fig. 1. Basic XIVL example.

B. SystemC Symbolic Simulation

SystemC symbolic simulation as proposed in [6], [4], [5] combines symbolic execution with complete exploration of all process schedules.

Symbolic execution analyzes the behavior of each individual SystemC process pathwise by treating inputs as symbolic values. Along an execution path, XIVL statements are interpreted as updates to the current DUV state. This state is represented by a set of symbolic expressions and a path condition PC, which must be satisfied by the expressions. The interpretation of each conditional goto statement is as follows: the execution path s is *forked* into two independent paths s_T and s_F due to two possible evaluations of the condition c. The PC for each path is updated accordingly as $PC(s_T) := PC(s) \wedge c$ and $PC(s_F) := PC(s) \wedge \neg c$. An SMT solver is used to determine if s_T and s_F are feasible, i.e. their PC is satisfiable. Only feasible paths will be explored further. For verification purposes, *assume(c)* adds c to the current PC to prune irrelevant paths and *assert(c)* calls an SMT solver to check for assertion violation, i.e. $PC \wedge \neg c$ is satisfiable.

The scalability of symbolic simulation can be improved if visiting redundant process schedule is avoided. This is accomplished by integrating POR techniques. Essentially, each process is separated into multiple transitions. A transition is a list of statements that is executed non-preemptively following the SystemC semantics. Thus every process has a currently active transition, which is runnable, iff the process is runnable. The first transition begins at the first statement of the thread. Subsequent transitions continue right after the context switch of the previous transition. POR requires a dependency relation between transitions. Intuitively, two transitions t_1 and t_2 are dependent, if the two possible orders of execution t_1t_2 and t_2t_1 lead to different results. In SystemC context, t_1 and t_2 are dependent if one of the following holds: 1) they access the same variable with at least one write access, 2) one immediately notifies an event that the other awaits, 3) a transition is suspended by the other. Transition dependencies are used at runtime to compute a subset of runnable transitions, called a *persistent set* [8], in each state. Exploration of transitions, and hence processes, is limited to the persistent sets.

The basic symbolic simulation approach for SystemC described above can be improved in various ways, for example,

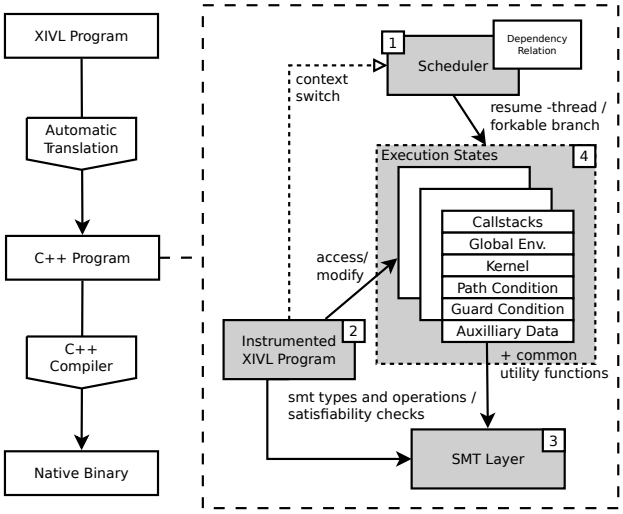

Fig. 2. CSS Overview

```
1  // callframes allow to preserve local execution state
       across context switches
2  struct Callframe {
3    unsigned ip;
4    void *env;
5    Callable fn;
6    void *result;
7  };
8  struct GlobalEnv { // global variables
9    SmtExpr n;
10   SmtExpr sum;
11 };
12 struct LocalEnv_A { // local variables of thread A
13   SmtExpr i;
14 };
15 // specialized callframe for every thread/function
16 struct Callframe_A : public Callframe {
17   LocalEnv_A env_A;
18   Callframe_f(Callable fn) {
19     env = &env_A;
20     result = nullptr; // threads have no return value
21     ip = 0;
22     fn = f;
23   }
24 };
```

Fig. 3. Data structures for the CSS transformation of thread A to store and access local and global program state (located in block 2 of Fig. 2)

[16] uses model checking to compute a stronger dependency relation, [17] applies symbolic subsumption checking to detect cycles in the symbolic state space, etc. In the following, we present another major enhancement, that is also complementary to existing improvements.

IV. Compiled Symbolic Simulation

An overview of the proposed approach is shown in Fig. 2. The starting point is an XIVL program, which is an intermediate representation of the SystemC DUV. Please note that the to-be-verified properties are already embedded into this program by means of assertions. Our approach first automatically translates the XIVL description into an executable C++ program, then invokes a C++ compiler to produce a native binary. Executing the binary equates to verifying the properties on the XIVL model with symbolic simulation. To accomplish this, several symbolic simulation components must be integrated into the XIVL-to-C++ translation. In the following, we discuss the main parts of the generated C++ program, which is also shown on the right hand side of Fig. 2.

A. Generated C++ Program Overview

Essentially such a C++ program consists of four parts: 1) The scheduler; 2) The instrumented XIVL program; 3) The SMT layer; 4) The declaration of data structures to store and manipulate execution states. The *symbolic execution engine* (SEE) is located inside the instrumented program. This interacts with the scheduler, the SMT layer and execution states via API functions to perform the state space exploration.

At each point of time during the execution, only one execution state is considered active. The scheduler provides API functions for managing a set of execution states including cloning (implemented as deep-copy) and swapping the currently active execution state. This allows to explore different independent paths, which arise due to process scheduling decisions and forking in symbolic branches. The scheduler selects which runnable thread or which direction of a branch to be executed next. It will also backtrack when execution of a path finishes and thus eventually explore all different alternatives. Furthermore, the (static) dependency relation is computed during translation process and embedded into the scheduler to prune redundant schedules.

The SMT layer provides API functions to create symbolic data types, to manipulate symbolic expressions and to check their satisfiability. Essentially, it is a convenience layer on top of underlying SMT solvers.

The *main* function in the C++ program will first create the scheduler and then setup an initial execution state. Essentially this step registers all threads and the XIVL *main* function so that they can be selected by the scheduler to execute. In the following we will discuss the data structures for execution state and the instrumented XIVL code together with its interactions with other components in more details.

B. Data Structures for Execution State

The first component of an execution state is the state of the simulation kernel (i.e. status of threads and events). The second component is the state of the XIVL model itself. This is further divided into global and local program state.

All global variables are stored in a *GlobalEnv* class. In Fig. 3, Line 8-11 show the definition of this class for the XIVL example. The local program state consists of the local state of all functions declared in the XIVL description. Please note that the local state of a function must be saved when a blocking statement (i.e. *wait*) is executed. This local state is stored in a callframe to preserve it across context switches. A callframe contains the following data: a) the label identifier to resume execution at the interrupted point (IP); b) a local environment (*env*), which stores arguments and local variables; c) a pointer to a callable object, which is either a global function or a pair of class instance and member function, which allows to call the actual function; d) a generic pointer to the result stored in the local environment, so a callee can retrieve it.

```
1  #define GLOBAL(VAR) active_state->global_env->VAR
2  bool dir;
3  void thread_A(Callframe *c) {
4    // retrieve local state and goto beginning or last
         context switch location
5    LocalEnv_A *env = (LocalEnv_A *)c->local_env;
6    switch (c->ip) {
7      case 0: goto A_start;
8      case 1: goto A_branch_1;
9      case 2: goto A_interrupt_1;
10     default: assert (false);
11   }
12   A_start:
13   // create a constant SMT expression (compatible with
         symbolic SMT expressions)
14   env->i = smt_create(0);
15   loop:
16   // store location in case of context switch
17   c->ip = 1;
18   A_branch_1:
19   dir = active_state->on_branch(smt_ge(env->i,
         GLOBAL(n))); // check feasibility, might fork
20   if (dir) goto end_loop;
21     env->i = smt_add(env->i, smt_create(1));
22     GLOBAL(sum) = smt_add(GLOBAL(sum), env->i);
23     c->ip = 2;
24     wait_time (1); // modifies kernel state and
           triggers context switch
25     A_interrupt_1;
26     goto loop;
27   end_loop;
28   A_end:
29   c->ip = -1;
30 }
```

Fig. 4. CSS transformation of thread A (located in block 2 of Fig. 2), which has been defined in Fig. 1

```
1  enum BranchDirection {True, False, None};
2  enum BranchStatus {BothFeasible, TrueOnly, FalseOnly};
3
4  bool ExecutionState::on_branch(const SmtExpr::pointer
       &cond) {
5    // initially *branch_direction* is set to None
6    if (branch_direction == BranchDirection::True) {
7      // remove scheduler annotation, update path
           condition accordingly and return concrete
           scheduler decision
8      branch_direction = BranchDirection::None;
9      PC = smt->bool_and(PC, cond);
10     return true;
11   } else if (branch_direction ==
         BranchDirection::False) {
12     // similar to above branch direction
13     branch_direction = BranchDirection::None;
14     PC = smt->bool_and(PC, smt->bool_not(cond));
15     return false;
16   }
17   assert (branch_direction == BranchDirection::None);
18
19   // let the scheduler decide in case both branches
         are feasible
20   BranchStatus::e b = check_branch_status(cond);
21   if (b == BranchStatus::BothFeasible) {
22     forkable = true;
23     // back to the scheduler, unwind potentially
           nested function
24     throw ContextSwitch(active_state);
25   } else {
26     return b == BranchStatus::TrueOnly ? true : false;
27   }
28 }
```

Fig. 5. Execution of branches with symbolic conditions (located in block 4 of Fig. 2)

For every function a custom local environment and callframe class is statically generated during translation. The definition of the specific callframe class for the thread function A from the example is shown in Line 16-24 in Fig. 3.

C. Instrumented XIVL Code

Essentially, the original XIVL code is instrumented in three steps: 1) Adding context switch logic to interact with the scheduler; 2) Replacing native data types and operations with symbolic types and operations; 3) Transforming branches to support exploration of both paths in case of a symbolic branch condition. The last two points are parts of the integrated SEE. In the following we will describe context switch, symbolic branch and function call execution in more detail.

a) Context Switch Logic: For each defined function, a label is associated with every statement that can interrupt the execution of the function. Every label is assigned a unique identifier within the function. A switch block is added at the beginning of the function to dispatch execution to the correct label. This block for our thread A can be seen in Line 6-11 of Fig. 4 Before the dispatching, the local state of the function is retrieved (Line 5).

The corresponding label is added right after the blocking statement. An assignment of the IP of the callframe to the ID of this label is added right before the blocking statement. This allows to resume it after the context switch. These steps can be seen in Line 23-25.

b) Executing Symbolic Branches: The branch in Line 7 of Fig. 1 is translated into the code block shown in Line 17-20 of Fig. 4. Please note that at this point, native data types and operations have been already replaced with symbolic types and operations (prefixed with *smt* in Fig. 4); The *on_branch* function of the active execution state is called with the symbolic condition and will always return a concrete boolean value. This result is stored in a designated global variable *dir* which is used as concrete condition in the branch.

The *on_branch* function is shown in Fig. 5 and works as follows: First the SMT layer is invoked to check if both paths are feasible in Line 20. If only one path is feasible, then the corresponding concrete boolean value - true for the goto path and false for the fallthrough path - is returned in Line 26 and the execution will continue normally without any interruption. In case both paths are feasible, the execution state will be marked as forkable and a context switch back to the scheduler will be triggered (Line 22-24). The scheduler will then clone the execution state and decide which path to continue. The decision will be annotated on the execution state and execution is resumed by calling the interrupted thread function of A. The execution of thread A will directly continue at the branch statement since the IP of A has been updated to point to the branch label in Line 18 during the first execution. The *on_branch* function is called again, but this time the execution state is marked with a concrete path decision annotated by the scheduler, i.e. either the condition in Line 6 or Line 11 will be true. In this case the function will ignore the symbolic condition, but instead simply return the scheduler decision and update the path condition of the active execution state accordingly. Additionally, it will remove the scheduler annotation from the state to ensure correct handling of the next branch.

```
1  void f_impl(Callframe *c) {
2    LocalEnv_f *env = (LocalEnv_f *)(c->local_env);
3    // ... other instructions
4    env->result = smt_add(env->a, env->b);
5    goto f_end;
6    // ... other instructions
7  }
8  void f_wrapper(SmtExpr a, SmtExpr b) {
9    // wrapper allows correct callframe allocation in
         case of virtual functions
10   Callframe_f *c = new Callframe_f(a, b, f_impl);
11   active_state->push_callframe(c);
12   f_impl(c);
13 }
14
15 // implementation function does not return here when
       interrupted, thus store resumption point
16 c->ip = LABEL_ID(f_call_1);
17 f_wrapper(env->y, env->z);
18 f_call_1:
19 // retrieve the result and cleanup
20 Callframe *c_f = active_state->pop_callframe();
21 env->x = *((SmtExpr *)(c_f->result));
22 delete c_f;
```

Fig. 6. Relevant instructions for a function call (located in block 2 of Fig. 2)

c) Function Calls: Every function is translated into two parts: a wrapper and an implementation. The original function call is replaced by a call to the wrapper function which in turn calls the implementation. An example for the function call *x = f(y,z) = y + z* is shown in Fig. 6. The wrapper (Line 8) has the same signature, except the return type, as the original function. It allocates a callframe based on the given arguments and delegates to the implementation function (Line 12 and Line 1). A wrapper is used because the target function is not statically known when calling a virtual function.

The implementation function contains the logic of the original function, instrumented similarly as the example for thread A has demonstrated (see Fig. 4). Once the implementation function finishes, execution will continue at the callee in Line 18. The callee can retrieve the result value from the callframe and is responsible for cleanup (Line 20-22). A label is placed after each function call (Line 18) and the IP is set to that label right before the function call (Line 16) to resume execution correctly in case the implementation function is interrupted. In this case execution control will return to the scheduler and unwind the native C++ stack. The scheduler is then responsible to call the callee once the implementation function finishes.

V. Optimizations

This section presents two optimizations tailored for CSS. First, we show how to efficiently integrate existing path merging methods into our CSS framework. This integration is important, since path merging is a powerful technique to alleviate the path explosion problem during symbolic execution. Second, we show how to generate more efficient code by employing a static analysis to determine which operations can be executed natively. This can further speed-up the execution of concrete code parts significantly.

A. Path Merging

This section describes how to use path merging in combination with CSS. Our algorithm assumes that branches and loops

```
1  env->x = smt_bv(32, true);
2  START_LABEL:
3  if (resume_merging) {
4    resume_merging = false;
5    dir = resume();
6  } else {
7    dir = begin(smt_gt(env->x, smt_create(0)));
8  }
9  if (dir) {
10   env->x = assign(env->x, smt_mul(env->x,
         smt_create(2)));
11 } else {
12   env->x = assign(env->x, smt_create(1));
13 }
14 if (end()) {
15   resume_merging = true;
16   goto START_LABEL;
17 } else {
18   resume_merging = false;
19 }
20 env->x = smt_add(env->x, smt_create(1));
```

Fig. 7. Branch merging example (located in block 2 of Fig. 2)

that should be merged are marked, e.g. by placing *@mergeable* before them in the input code. This is a flexible approach that allows both an automatic analysis, e.g. see [18], and an user to annotate mergeable branches. This allows a fine grained tuning between merging and explicit exploration. Merging can reduces the number of explored paths exponentially at the cost of more complex solver queries. A limitation of our current algorithm is that merging SystemC kernel parts is not yet supported, since the kernel state is available in explicit form. Therefore, loops and branches that update kernel state will not be merged. In the following we will discuss our branch merging approach in more detail. Loop merging in principle works similar to branch merging and thus will not be further discussed.

a) Merging Branches: Consider an example program *x=?(int); if (x > 0) { x = 2*x; } else { x = 1; } x++;*, where the *if-statement* is marked as mergeable. This code is transformed into the code block shown in Fig. 7. The initial assignment of x and the increment at the end are not inside the mergeable block and thus normally translated.

Four functions are involved in the translation of mergeable branches, for convenience we use short names in this example: *begin*, *resume*, *end* and *assign*. They operate on the currently active execution state. The additional global variable *resume_merging* is a simple boolean value, similar to dir, to control execution of the program. It allows to distinguish between the first and second visit of the branch. Initially it is set to *false*. The execution state internally keeps a stack of triples *(GC, status* $\in$ {*none, first, second*}*, cond)* to capture the execution progress of potentially nested mergeable branches. GC is the current guard condition and *cond* the symbolic branch condition passed to the *begin* function. Initially the guard condition is *true*. On every solver query, it is combined with the path condition using a logic *and* operation. There are two choices when *begin* is called:

1) Both paths are feasible. The begin function will store the triple *(GC, first, cond)* and return *true*, an arbitrary choice to start execution with the *if-path*. Backing up the guard condition and branch condition allows to use them for the exploration

of the *else-path*, as they can be modified during execution of the *if-path*. Furthermore, GC is updated as $GC \wedge cond$. The *assign(lhs, rhs)* function will return a guarded expression of the form *GC ? rhs : lhs*, i.e. update lhs to rhs based on GC. Finally the *end* function will notice that *status=first*, i.e. branch merging is active, so it will update status to *second* and return *true*. Therefore, execution will jump to the beginning of the branch again, but this time the *resume* function will be entered. It will retrieve the stored GC and *cond* from the top of stack, update GC as $GC \wedge \neg cond$ and return *false* to execute the *else-path*. Since *status=second*, the *end* function will pop the top of stack and return false, which leaves the mergeable branch.

2) Only one path is feasible. The begin function will push *(GC, none, cond)* on the stack and return the path direction. The *assign* function will either return *GC ? rhs : lhs*, in case this branch is nested within another active mergeable branch, i.e. $GC \neq true$, or just rhs, otherwise. The end function will recognize *status=none*, thus pop the data triple and return *false*.

B. Native Execution

We provide two native execution optimizations to improve the execution performance of instructions and function calls, respectively: 1) A static analysis is employed to determine variables that will never hold a symbolic value. Such variables can keep their native C++ type. Furthermore, native operations can be performed on native datatypes. They are significantly faster than (unnecessarily) manipulating symbolic expressions. 2) To optimize function calls, a static analysis is employed to determine which functions can be interrupted. Essentially, a function is interruptable, iff it contains an interruptable statement, i.e. *wait_time/event* or branch with symbolic condition, or any function it calls is interruptable. Callframe allocation and cleanup is not required for a non-interruptable functions, therefore a function call *x = f(y, z)* is not instrumented but executed unmodified on the native stack. In the following we describe our first static analysis, which determines symbolic variables, in more detail.

a) Computing Maybe-Symbolic Variables: This static analysis starts by computing two pieces of informations: 1) A root set of variables which are symbolic. 2) A dependency graph between variables, where an edge from a to b denotes that b maybe symbolic if a maybe symbolic. Then all variables reachable from the root set following the dependency graph can potentially be symbolic. Such variables are called *maybe-symbolic* The other variables can keep their native data types.

The root set S contains all variables where a symbolic value is directly assigned, e.g. $x = ?(int)$ will add x to S. Similarly $*p = ?(int)$ will add the pointer p to S, since it needs to have *SmtExpr** type.

The dependency graph G essentially records assignments between variables. Here pointer and non-pointer types are treated differently. The assignment $a = b$ will add an edge from b to a. On the other hand, if a is symbolic, then b can still be a native type, since b can simply be wrapped in an *smt_create* call to be compatible with a. This allows native execution of other operations involving b. If a pointer is involved in the assignment, e.g. $p = \&a$, then an edge in each direction is added to G. The reason is that there is no conversion available that allows to assign a *int** to *SmtExpr**. Therefore a must also have a symbolic type. Argument and return value passing during function calls is handled the same way as assignments.

b) Determining Symbolic Operations: Once the set of maybe-symbolic variables is known, it can be used to compute the set of symbolic expressions by walking the expression trees bottom-up. This allows to determine which operations need to be performed by the symbolic execution engine and which can be natively executed. For example consider the expression $a < b + 1$, where a is symbolic and b is not. Then $b + 1$ will first be natively executed and then converted to a symbolic value. Finally the $<$ operation will be executed symbolically.

VI. Experiments

We have implemented the proposed CSS together with the optimizations and evaluated it on an extensive set of benchmarks. The evaluation also includes comparisons to state-of-the-art tools. We employ Z3 v4.4.1 in the SMT Layer and compile the generated C++ programs using Clang 3.8 with $-O3$ optimization. All experiments were performed on an Intel 3.5 GHz machine running Linux. The time and memory limit has been set to 2000s and 6GB, respectively. In the following tables all runtimes are given in seconds. T.O. (M.O.) denotes that the time (memory) limit has been exceeded. The column V (Verdict) denotes if the benchmark is bug-free, i.e. safe (S), or contains bugs (U). Thus, if a runtime can be reported for a tool on a benchmark with (without) bug, it means the tool terminated successfully and detected the bug in the benchmark (confirmed its correctness) as expected.

A. Native Execution Evaluation

Our first experiment demonstrates the benefits of native execution over interpretation in context of symbolic execution. We compared CSS with KLEE [19], the state-of-the-art symbolic executor for C, which includes a highly optimized interpretation engine for the LLVM IR. The results are shown in Table I. For CSS, we show the compilation time (column *Compile*) and include it into the total runtime (column TOTAL) to make the comparison fair. The upper half of the table shows pure C benchmarks, for which KLEE is directly applicable. The *iterative* and *recursive* benchmarks perform some lightweight symbolic computation in 4 (*small*) and 400 (*large*) million iterations/recursive calls, respectively.

The lower half of Table I shows SystemC benchmarks. Since KLEE cannot directly handle SystemC semantics, we applied a sequentialization scheme similar to [10], [11]. For a fair comparison, we limited the state space to a single arbitrary process schedule, otherwise KLEE would perform very poorly because it does not support POR. Both KLEE and CSS are then applied on the same sequentialized programs available in C or XIVL, respectively. These sequentialized

TABLE I
NATIVE EXECUTION EVALUATION (RUNTIMES IN SECONDS)

Benchmark	V	KLEE	CSS	
			TOTAL	Compile
iterative-small	S	4.60	**0.62**	0.61
iterative-large	S	444.64	**0.84**	0.62
recursive-small	S	104.77	**0.65**	0.61
recursive-large	S	M.O.	**0.85**	0.61
mem-slave-tlm-bug.500K*	U	15.16	**0.81**	0.74
mem-slave-tlm-bug.5M*	U	144.56	**0.83**	0.58
mem-slave-tlm-sym.500K*	S	17.17	**10.91**	0.71
mem-slave-tlm-sym.5M*	S	164.65	**113.17**	0.65
pressure.40M*	S	23.03	**0.82**	0.58
pressure.400M*	S	220.74	**3.08**	0.59

benchmarks are marked with *. For scalability investigation, we also varied the size of the benchmarks, indicated by the last number in benchmark name with *K=thousand* and *M=million*. In these cases, that means the number of loop iterations and the maximum simulation time.

Overall, CSS shows significant improvements over KLEE. The *mem-slave-tlm-sym* benchmark performs in every loop iteration heavier symbolic computations, which are not optimized by native execution. Therefore, the benefit of CSS is less pronounced.

B. Comparison with Existing SystemC Verifiers

a) Freely Available Benchmarks: Table II shows a comparison between CSS with *Interpreted Symbolic Simulation* (ISS), basically a reimplementation of the symbolic simulation technique described in Section III-B, and the state-of-the-art abstraction-based verifier KRATOS [10]. The benchmarks are freely available and commonly used to compare SystemC formal verification tools (see e.g. [10], [6]). The comparison with KRATOS is mainly to confirm that our ISS implementation is reasonably fast. Generally, the obtained results are consistent with the results reported in [6]. Again, we also varied the size of the benchmarks for scalability investigation. The compilation times are already integrated into the CSS runtimes. In general they are negligible for longer running benchmarks. For CSS, improvements of several orders of magnitude can be observed. In one case CSS is unable to verify the up-scaled *pressure* benchmark, since our POR algorithm is unable to limit the exponential growth of thread interleavings due to complex dependencies. This problem can be solved by employing a stronger POR algorithm or combine CSS with a stateful exploration.

b) Real-World Virtual Prototype Models: This second comparison was performed on two larger real-world SystemC VP models: 1) An extended version of the Y86 CPU [20], which implements a subset of the instructions of the IA-32 architecture [21]. 2) A TLM model of the *Interrupt Controller for Multiple Processors* (IRQMP) of the LEON3-based VP SoCRocket, partly developed and used by the European Space Agency [22].

Since both models extensively use object oriented programming features as well as arrays, KRATOS is not applicable and thus omitted from the comparison. The results are shown

TABLE II
COMPARISON WITH EXISTING SYSTEMC VERIFIERS ON PUBLICLY AVAILABLE BENCHMARKS (RUNTIMES IN SECONDS)

Benchmark	V	ISS	KRATOS	CSS
jpeg-p6-bug	U	2.57	T.O.	**1.29**
mem-slave-tlm-bug.50	U	2.74	T.O.	**0.84**
mem-slave-tlm-bug.500K	U	M.O.	T.O.	**19.96**
mem-slave-tlm-bug.5M	U	M.O.	T.O.	**208.81**
mem-slave-tlm-sym.50	S	2.81	T.O.	**0.88**
mem-slave-tlm-sym.500K	S	M.O.	T.O.	**47.96**
mem-slave-tlm-sym.5M	S	M.O.	T.O.	**479.70**
pressure-safe.10	S	7.18	0.41	**0.71**
pressure-safe.15	S	211.36	1.27	**1.42**
pressure-safe.40M	S	M.O.	T.O.	M.O.
pressure-unsafe.25	U	0.83	31.79	**0.67**
pressure-unsafe.50	U	0.82	443.08	**0.68**
simple-fifo-bug-1c2p.20	U	1.33	18.10	**0.92**
simple-fifo-bug-1c2p.50	U	1.94	1806.62	**0.85**
simple-fifo-bug-2c1p.20	U	1.45	14.54	**0.90**
simple-fifo-bug-2c1p.50	U	2.30	434.98	**0.84**
token-ring-bug2.15	U	**1.59**	3.74	1.66
token-ring-bug2.20	U	1.92	M.O	**1.90**
token-ring-bug.20	U	**1.07**	149.26	1.19
token-ring-bug.100	U	4.20	M.O	**3.93**

in Table III. For CSS, the first column shows the total runtime. The next three columns show the detailed breakdown (including percentage) of the total time into native execution time, SMT solving time, and compilation time. In an analogous manner, the ISS total runtime as well as its breakdown into interpretation time and SMT solving time are reported. Also please note that both CSS and ISS explore the state space in the same order, i.e. they follow the same execution paths and solve the same SMT queries.

Benchmarks in the upper half only operate on concrete values (hence the SMT time is not available). The *test-1*, *test-2* and *test-3* are testcases for the IRQMP from the SoCRocket distribution. The *y86-counter* benchmarks executes a computation on the Y86 processor model. As expected, CSS significantly outperforms ISS here.

The lower half shows results for verification of functional properties on the IRQMP model. The *y86-isr* benchmark combines the IRQMP and Y86 processor model. The Y86 model runs a version of the counter program and furthermore an *Interrupt Service Routine* (ISR) is placed in memory. The IRQMP prioritizes a symbolic interrupt received from the test driver and forwards it on a signal line to the Y86 CPU. The CPU stores the received interrupt in its register and triggers the ISR, which processes and acknowledges the interrupt using memory-mapped IO over a bus transaction. The other benchmarks in the lower half verify functional properties of the IRQMP, in particular that broadcasts are send to all CPUs and interrupt prioritization as well as masking works correctly.

For the benchmarks in the lower half, notable improvements in total runtimes can still be observed, although not in the scale of the previous comparison. The reason becomes clear when inspecting the detailed breakdown of runtimes. While significant improvements of native execution over interpretation are still visible, the SMT solving times for CSS are only slightly better. On the other hand, path merging has been shown to be

TABLE III
REAL-WORLD VIRTUAL PROTOTYPE BENCHMARKS (RUNTIMES IN SECONDS)

Benchmark	V	CSS							ISS				
		TOTAL	Execution		SMT		Compilation		TOTAL	Interpretation		SMT	
test-1	S	**4.49**	0.50	11%	-		3.99	89%	79.28	79.28	100%	-	
test-2	S	**4.15**	0.23	5%	-		3.93	95%	44.52	44.52	100%	-	
test-3	S	**3.88**	0.24	6%	-		3.64	94%	35.99	35.99	100%	-	
y86-counter	S	**0.94**	0.12	13%	-		0.82	87%	276.70	276.70	100%	-	
broadcast	S	**10.54**	0.62	6%	6.69	63%	3.23	31%	18.68	10.15	54%	8.53	46%
prioritization-8	S	**34.01**	0.98	3%	30.22	89%	2.81	8%	114.20	70.58	62%	43.62	38%
prioritization-12	S	**497.36**	16.96	3%	477.63	96%	2.77	1%	T.O.	N.A.	N.A.	N.A.	N.A.
masking-regular	U	**5.69**	0.10	2%	2.49	44%	3.10	54%	7.22	4.28	59%	2.94	41%
masking-extended	U	**7.81**	0.36	5%	4.22	54%	3.23	41%	10.69	5.48	51%	5.21	49%
y86-isr	S	**179.83**	30.79	17%	140.14	78%	8.90	5%	T.O.	N.A.	N.A.	N.A.	N.A.

crucial to avoid path explosion in verifying properties on the IRQMP design. Solving complex SMT queries, as a result of extensive path merging, often dominates the total runtime for these benchmarks. Thus, the overall advantage of CSS over ISS is less pronounced here.

Also please note that while these real-world SystemC VP models are still rather small compared to those that can be verified by simulation-based techniques, we are unaware of any other symbolic verification approach for SystemC, which can scale to models of this complexity (e.g. *y86-isr*).

VII. LIMITATIONS

Even though CSS and further optimizations such as path merging have remarkably improved the scalability of symbolic simulation for SystemC, it is still expectedly subject to state space explosion on large SystemC VP models. The two main limitations of our CSS approach are as follows. First, the native execution selection of CSS is currently based on a static (type) analysis. This prevents optimization in case code parts, e.g. functions, are executed multiple times with concrete and symbolic parameters. Second, CSS is currently stateless, i.e. it does not keep a record of already executed states, and cannot yet be used to verify safe programs with cyclic state spaces. It is still to be investigated, how a stateful symbolic simulation approach, such as described in [17], can be efficiently incorporated into CSS.

VIII. CONCLUSION

In this paper we have proposed Compiled Symbolic Simulation (CSS) for improved SystemC verification. In contrast to existing symbolic simulation approaches CSS is based on compiled execution instead of interpretation. For a scalable exploration, the symbolic execution engine as well as the Partial Order Reduction (POR) based scheduler are integrated into the DUV. Then a standard C++ compiler is used to generate a native binary, which will perform an efficient exhaustive exploration of the DUV. To further improve the efficiency we have proposed two optimizations tailored for CSS: existing path merging methods adapted to CSS in order to mitigate the well-known state explosion problem in symbolic simulation, and native execution, which can further speed-up the execution of concrete code parts significantly. The experiments using an extensive set of freely available SystemC benchmarks as well as larger real-world SystemC TLM models demonstrated the efficiency and applicability of our approach. For future work we plan to investigate the applicability of runtime informations to select code parts for native execution dynamically and integrate a stateful exploration into our CSS framework.

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

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

[4] C.-N. Chou, Y.-S. Ho, C. Hsieh, and C.-Y. Huang, "Symbolic model checking on SystemC designs," in *DAC*, 2012, pp. 327–333.

[5] C.-N. Chou, C.-K. Chu, and C.-Y. R. Huang, "Conquering the scheduling alternative explosion problem of SystemC symbolic simulation," in *ICCAD*, 2013, pp. 685–690.

[6] H. M. Le, D. Große, V. Herdt, and R. Drechsler, "Verifying SystemC using an intermediate verification language and symbolic simulation," in *DAC*, 2013, pp. 116:1–116:6.

[7] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

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

[9] C. Flanagan and P. Godefroid, "Dynamic Partial-Order Reduction for model checking software," in *POPL*, 2005, pp. 110–121.

[10] A. Cimatti, I. Narasamdya, and M. Roveri, "Software model checking SystemC," *TCAD*, vol. 32, no. 5, pp. 774–787, 2013.

[11] 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.

[12] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[13] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi, "A SystemC/TLM semantics in Promela and its possible applications," in *SPIN*, 2007, pp. 204–222.

[14] D. Campana, A. Cimatti, I. Narasamdya, and M. Roveri, "An analytic evaluation of SystemC encodings in Promela," in *SPIN*, 2011, pp. 90–107.

[15] H. M. Le, V. Herdt, D. Große, and R. Drechsler, "Towards formal verification of real-world SystemC TLM peripheral models - a case study," in *DATE*, 2016, pp. 1160–1163.

[16] N. Blanc and D. Kroening, "Race analysis for SystemC using model checking," *TODAES*, vol. 15, no. 3, pp. 21:1–21:32, Jun. 2010.

[17] V. Herdt, H. M. Le, and R. Drechsler, "Verifying SystemC using stateful symbolic simulation," in *DAC*, 2015, pp. 49:1–49:6.

[18] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *PLDI*, 2012, pp. 193–204.

[19] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008, pp. 209–224.

[20] A. Biere, D. Kroening, G. Weissenbacher, and C. Wintersteiger, *Digitaltechnik - eine praxisnahe Einführung*. Springer, 2008.

[21] *IA-32 Architecture Software Developer's Manual*, Intel Corporation, 2003.

[22] T. Schuster, R. Meyer, R. Buchty, L. Fossati, and M. Berekovic, "SoCRocket - A virtual platform for the European Space Agency's SoC development," in *ReCoSoC*, 2014, pp. 1–7.