# Verifying SystemC using Stateful Symbolic Simulation[*]

Vladimir Herdt[1] 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
{vherdt, hle, drechsle}@informatik.uni-bremen.de

## ABSTRACT

Formal verification of high-level SystemC designs is an important and challenging problem. Recent works have proposed symbolic simulation in combination with *Partial Order Reduction* (POR) as a promising solution and experimentally demonstrated its potential. However, these symbolic simulation approaches have a fundamental limitation in handling cyclic state spaces. The reason is that they are based on stateless model checking and thus unable to avoid revisiting states in a cycle. In this paper, we propose a novel stateful symbolic simulation approach for SystemC. For the efficient detection of revisited symbolic states, we apply symbolic subsumption checking. Furthermore, our implementation integrates a cycle proviso to preserve the soundness of POR in the presence of cycles. We demonstrate the scalability and the efficiency of the proposed approach using an extensive set of experiments.

## 1. INTRODUCTION

The C++-based description language SystemC [11] has become the standard for modeling electronic systems at high levels of abstraction. These abstract SystemC designs serve as an executable specification for subsequent development steps in the design flow. Therefore, ensuring the correctness of high-level SystemC designs is crucial, especially for safety critical systems, as undetected errors will propagate and become very costly.

Due to their scalability and ease-of-use, simulation-based approaches are still prevalent for SystemC verification. However, in contrast to formal verification, they cannot prove the absence of errors and are very susceptible to subtle bugs caused by corner-case scenarios. Unfortunately, formal verification of SystemC is very challenging due to its object-oriented nature and event-driven simulation semantics [15]. The overall challenge in developing a SystemC verifier is threefold. First, it must obviously consider all possible inputs of the *Design-Under-Verification* (DUV). Second, a typical high-level SystemC DUV consists of multiple asynchronous processes, whose different orders of execution (also referred to as *schedules*) can lead to different behaviors, these must also be considered to the full extent by the verifier. Third, the verifier is required to deal with the full complexity of C++ to extract a suitable formal model.

Recently, an *Intermediate Verification Language* (IVL) for SystemC has been proposed [13]. The IVL is compact and easily manageable but at the same time powerful enough to model the behavior of SystemC designs. The IVL allows to separate the development of a SystemC verifier into two components: a *front-end* to translate a DUV into IVL and a *back-end* to verify this IVL description. Consequently, one can focus on addressing the first two challenges to increase the scalability and efficiency of the back-end in handling large state spaces.

As a viable solution for this task, recent works have proposed symbolic simulation [5, 4, 13], which is basically a combination of symbolic execution [12] with complete exploration of all possible process schedules. These approaches allow the verification of safety properties specified in the form of source code assertions. In [4, 13], symbolic simulation is further combined with *Partial Order Reduction* (POR) [9, 8] to avoid visiting redundant schedules. The whole state space of a DUV, which consists of all possible inputs and process schedules, can thus be exhaustively and efficiently explored.

However, these approaches are fundamentally limited in handling cyclic state spaces. Cycles arise naturally in many high-level SystemC designs due to the use of unbounded loops inside SC_THREADs. The limitation is due to the fact that the core algorithm employed by these approaches is based on a stateless search. Essentially, no record of already visited states is kept and thus, re-exploration of states in a cycle cannot be avoided. This might cause the search to be non-terminating if the symbolic simulation is not bounded. Consequently, existing SystemC symbolic simulation approaches are unable to prove properties on cyclic state spaces and thus can be considered *incomplete*.

To overcome the aforementioned limitation, it is necessary to employ a stateful search in symbolic simulation. Two issues must be solved to enable this combination. First, symbolic simulation stores and manipulates symbolic expressions, which represent sets of concrete values. Therefore, the *state matching* process, required by a stateful search to decide whether a state has already been visited, involves non-trivial comparison of complex symbolic expressions. Second,

[*]This work was supported in part 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.

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
DAC '15, June 07 - 11, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3520-1/15/06...$15.00
http://dx.doi.org/10.1145/2744769.2744927

a naive combination of POR with stateful search can potentially lead to unsoundness, i.e. assertion violations can be missed. This is due to the (transition/action) *ignoring problem*, which refers to a situation, where a relevant transition is not explored.

### *Contributions.*

In this paper, we propose a *novel stateful symbolic simulation approach* for SystemC. For the efficient detection of revisited symbolic states, we employ symbolic subsumption checking, inspired by [1]. If the set of concrete states represented by a symbolic state $s_2$ contains the set of concrete states represented by a symbolic state $s_1$, $s_1$ is *subsumed* by $s_2$ and it is not necessary to explore $s_1$ if $s_2$ has already been explored. Thus, subsumption checking can lead to a reduction of the state space, which we denote as *State Subsumption Reduction* (SSR). We present a powerful exact subsumption checking method which involves solving a quantified SMT formula. As this computation is potentially very expensive, we also propose several optimizations. Furthermore, to preserve the soundness of POR, our implementation integrates a cycle proviso tailored for SSR. We show the potential of our approach using an extensive set of benchmarks.

### *Related Work.*

KRATOS [6] is the state-of-the-art SystemC model checker for handling cyclic state spaces. As input language, KRATOS accepts threaded C, which is similar to the IVL proposed in [13]. The underlying model checking algorithm combines an explicit scheduler and symbolic lazy abstraction. POR is also integrated into the explicit scheduler to prune redundant schedules. For property specification, simple C assertions are supported. Although this approach is complete, its potentially slow abstraction refinements may become a performance bottleneck. SCIVER [10] is another complete model checker for SystemC. It translates a SystemC DUV into a sequential C model, then applies high-level induction on top of existing C model checkers. However, due to the absence of POR, it does not scale well to designs with a large number of processes.

There are also a handful of other formal verification approaches for SystemC which we do not discuss here. They are either incomplete or have very limited scalability. For a detailed review of these works we refer to the Related Work section of [6].

In the context of symbolic model checking for Java, a subsumption checking technique similar to ours has been considered [1]. However, the authors applied this technique to sequential Java programs, while we combine subsumption checking with POR under the concurrency semantics of SystemC.

The *ignoring problem* has first been identified in [14]. Since then, a number of cycle provisos has been proposed as solution in combination with different search strategies, see e.g. [9, 3, 7]. However, in the context of SSR, these provisos are unsuitable, since they are too restrictive. In this paper we use an adapted proviso from [7], for the combination of POR and SSR. To the best of our knowledge, such a proviso has not yet been proposed.

## 2. PRELIMINARIES

### 2.1 SystemC and IVL

SystemC is a C++ class library that includes an event-driven simulation kernel. 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.

The IVL as proposed in [13] provides modeling primitives for SC_THREADs (called threads for simplicity), events and corresponding synchronization functions (i.e. *wait* and *notify* in different variants). The IVL supports Boolean and integer data types of C++ together with all arithmetic and logic operators. The control flow of a thread is modeled using conditional goto statements. For verification purposes in the context of symbolic execution, the functions *assume* and *assert* are provided. An IVL example is shown in Fig. 1 (explanation follows in Section 3.1). To improve the readability, we use high-level control structures instead of conditional gotos.

The simulation semantics of SystemC (see [11]) is also precisely followed. Essentially, if multiple IVL threads are runnable, one of them 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.

### 2.2 SystemC Symbolic Simulation

Symbolic simulation of SystemC designs as proposed in [13, 5, 4] is a combination of symbolic execution and complete exploration of all process schedules.

Symbolic execution analyzes the behavior of each individual SystemC process/IVL thread pathwise by treating inputs as symbolic values. Along an execution path, the design state is represented by a set of symbolic expressions and a path condition $PC$, which must be satisfied by the expressions. At each conditional goto statement, 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 whether $s_T$ 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 violations, i.e. $PC \wedge \neg c$ is satisfiable.

POR is also employed to improve the scalability of symbolic simulation by avoid visiting redundant process schedules. Each process is separated into multiple transitions. A transition corresponds to a list of statements that is executed non-preemptively following the SystemC semantics. Thus every (non-terminated) 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 their execution does not commute, i.e. $t_1 t_2$ and $t_2 t_1$ leads 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*

Table 1: Example data for the IVL example

| | *PC* | *v* | *C*(*v*) | *runnable* |
|---|---|---|---|---|
| $s_0$ | $x_1 \geq 0 \wedge x_1 \leq 2$ | $x_1$ | $\{0, 1, 2\}$ | $\{I_1, G_1\}$ |
| $s_1$ | $x_1 \geq 0 \wedge x_1 \leq 2$ | $x_1$ | $\{0, 1, 2\}$ | $\{G_1\}$ |
| $s_2$ | $x_1 \geq 0 \wedge x_1 \leq 2$ | $x_1$ | $\{0, 1, 2\}$ | $\{I_2, G_2\}$ |
| $s_3$ | $x_1 \geq 0 \wedge x_1 \leq 2$ | $x_1 + 1$ | $\{1, 2, 3\}$ | $\{G_2\}$ |
| $s_4$ | $x_1 \geq 0 \wedge x_1 \leq 2 \wedge x_1 + 1 < 2$ | $x_1 + 1$ | $\{1\}$ | $\{I_2, G_2\}$ |
| $s_5$ | $x_1 \geq 0 \wedge x_1 \leq 2 \wedge x_1 + 1 \geq 2$ | $x_1$ | $\{1, 2\}$ | $\{I_2, G_2\}$ |
| $s_6$ | $x_1 \geq 0 \wedge x_1 \leq 2 \wedge x_1 + 1 \geq 2$ | $x_1 + 1$ | $\{2, 3\}$ | $\{G_2\}$ |
| $s_7$ | $x_1 \geq 0 \wedge x_1 \leq 2 \wedge x_1 + 1 \geq 2$ | $x_1$ | $\{1, 2\}$ | $\{I_2, G_2\}$ |

```
 1  event e;
 2  int v = ?(int);
 3
 4  thread increment {
 5    while (true) {
 6      wait e;
 7      v += 1;
 8    }
 9  }
10
11  thread guard {
12    while (true) {
13      assert (0 <= v && v <= 2);
14      notify e, 0;
15      wait_time 0;
16      if (v >= 2) {
17        v -= 1;
18      }
19    }
20  }
21
22  main {
23    assume (0 <= v && v <= 2);
24    start;
25  }
```

Figure 1: An IVL example

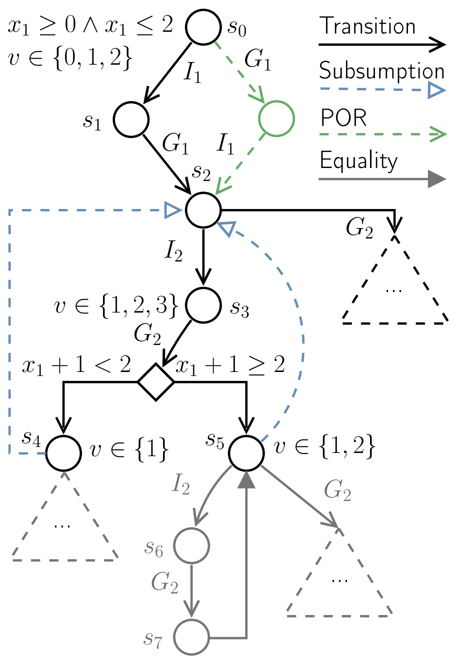

Figure 2: State space for the IVL example

*set* [9], in each state. Exploration of transitions, and hence processes, is limited to the persistent sets.

The above description depicts a basic symbolic simulation approach for SystemC. Many improvements are possible such as path merging [4] or computation of a stronger dependency relation using model checking [2].

## 3. STATE SUBSUMPTION REDUCTION

This section presents the main concepts of SSR in stateful symbolic simulation. We start with a motivating example, that shows the benefits of SSR and demonstrates that SSR and POR are complementary.

### 3.1 Motivating Example

The IVL description in Fig. 1 consists of two threads: *increment* (I) and *guard* (G). They communicate through a global variable $v$ and use the event $e$ for synchronization. The *increment* thread increments $v$ and then blocks until $e$ is notified. The *guard* thread is scheduled to run once in every delta cycle. It ensures that $v$ does not exceed the maximum value and performs a delta notification of the event $e$. In this example the maximum value is 2. Both threads consist of two transitions, denoted as $I_1$,$I_2$ and $G_1$,$G_2$ respectively, separated by the context switches in Line 6 and Line 15. The whole simulation is unbounded and *safe*, i.e. the assertion in Line 13 always holds.

A representative part of the complete state space is shown in Fig. 2. Circles represent states and edges depict transitions between them. A diamond represents a conditional branch, where both branches are feasible. The dashed triangles represent state space parts that are omitted to simplify the description. Initially, before the simulation starts (Line 24), $v$ is assigned a symbolic integer literal $x_1$ in Line 2. Then $v$, and thus $x_1$, is constrained to the values $\{0, 1, 2\}$ in Line 23, resulting in the initial state $s_0$.

Now consider the transition sequence $I_1 G_1 I_2 G_2 I_2 G_2$. The relevant data of the involved states is shown in Table 1. It shows the path condition (PC), the symbolic expression representing variable $v$, the set $C(v)$ of all concrete values of $v$ satisfying the path condition, and the set of runnable transitions.

After $I_1$ is executed and thread *increment* is blocked by event $e$, $s_1$ is reached. Then, $G_1$ performs a delta notification of $e$, and after a delta notification phase, the state $s_2$ is reached, where both $I_2$ and $G_2$ become runnable.

Next, $I_2$ increments $v$ to $x_1 + 1$ and reaches $s_3$. Then $G_2$ resumes from Line 16. Both, the branch condition $c_T = v \geq 2$ and its negation $c_F = v < 2$ are feasible in Line 16, with the current value of $v = x_1 + 1$ and path condition $x_1 \geq 0 \wedge x_1 \leq 2$. Thus the execution will fork two independent paths. The corresponding path conditions will be extended with $c_T$ and $c_F$, respectively. In the path where $c_T$ holds, $v$ gets decremented in Line 17, whereas nothing happens in the other path. Eventually both paths will reach the context switch in Line 15, resulting in the states $s_5$ and $s_4$, respectively.

Execution of $I_2 G_2$ from $s_5$ will reach the state $s_6$ and then $s_7$. Note that the execution of $G_2$ from $s_6$ does not fork at the conditional branch. The reason is that $v \in \{2, 3\}$ at this point, thus the negated branch condition $v < 2$ is not satisfiable. The state $s_7$ is equal with $s_5$, as shown in Table 1, thus $s_5$,$s_6$ and $s_7 = s_5$ form a cycle.

A *stateless search cannot detect the cycle* and would explore it infinitely unless the search is bounded. Conceptually, a stateful search, that is capable to detect the equality of $C(v)$ and *runnable* in $s_5$ and $s_7$, would solve the problem.

However, *much stronger reduction can be achieved by checking subsumption* of states. For example, the exploration of $I_2 G_2$ from $s_5$ is actually unnecessary. The reason is $C(v)$ of $s_5$ is a subset of $C(v)$ of $s_2$ and the *runnable* sets are identical. Thus, any concrete states that are reachable from $s_5$ can also be reached from $s_2$. We say that $s_5$ is subsumed by $s_2$, and analogously, $s_4$ is also subsumed by $s_2$[1]. Thus the exploration of transitions from both $s_4$ and $s_5$ would be

[1]But neither $s_4$ nor $s_5$ is subsumed by any state from $\{s_0, s_1, s_3\}$ since they have different *runnable* transitions.

prevented by SSR. This is not possible with a simple stateful search based on equality checking. On large cyclic state spaces, SSR is expected to explore significantly less states.

Additionally, POR can be combined with SSR as complementary reduction technique to further reduce the explored state space. In this example the execution of $G_1 I_1$ from the initial state $s_0$ is pruned by POR, since $I_1$ has already been explored from $s_0$ and it is independent with any other transition.

However, as mentioned earlier, care must be taken to avoid unsoundness when applying POR in cyclic state spaces, and especially here in combination with SSR. In the following we introduce the concept of *weak reachability* and a cycle proviso based on this concept to solve the problem. The actual procedure for subsumption checking between two symbolic states is detailed in Section 4.

## 3.2 Weak Reachability

SSR results in the exploration of a reduced state space, denoted as $A_R$. Whenever it is detected, that a state $s_1$ is subsumed by a state $s_2$, denoted as $s_1 \preccurlyeq s_2$, $s_1$ is not further explored. We say that there exists a *weak transition* from $s_1$ to $s_2$ in this case. This concept can be naturally extended to *weak reachability.* A state $s'$ is *weakly reachable* from $s$, if it is reachable from $s$ through a sequence of normal or weak transitions. Intuitively when a state $s'$ is reachable from a state $s$ in the complete state space $A_G$, a state $s''$ will be weakly reachable from $s$ in $A_R$, such that $s' \preccurlyeq s''$ holds. Thus, reachability of concrete states and as a consequence, assertion violations are preserved through SSR.

**Example 1.** In the state space shown in Fig. 2, $s_7$ is reachable from $s_5$ by the sequence of transitions (trace) $I_2 G_2$ in $A_G$. In $A_R$, $s_5$ is weakly reachable from itself by the same trace as follows: first $s_2$ is reached from $s_5$ by a weak transition and than $s_5$ is reached from $s_2$ by the trace $I_2 G_2$. Since $s_5$ and $s_7$ are equivalent, $s_7 \preccurlyeq s_5$ holds.

## 3.3 Cycle Proviso

A cycle proviso is required, to prevent transition ignoring when applying POR in the context of a stateful search and checking properties more elaborate than deadlocks. Otherwise, a relevant transition might be permanently ignored due to a cycle in the reduced state space. We have adapted the proviso $C_2^S$ from [7], to the notion of weak reachability that arises in SSR, resulting in the novel proviso $C_{2W}^S$.

$C_{2W}^S$ For every state $s$ in $A_R$ there exists a *weakly reachable* state $s'$ from $s$ in $A_R$, such that $s'$ is fully expanded, i.e. every runnable transition in $s'$ is explored.

In contrast the $C_2^S$ proviso requires normal reachability of a fully expanded state, which would limit the reduction achieved by SSR. Recall the IVL example presented in Section 3.1. Since e.g. $s_5 \preccurlyeq s_2$ holds, it is not necessary to further explore $s_5$, thus no state is reachable from $s_5$ in $A_R$. However, the $C_2^S$ proviso would enforce that a fully expanded state is reachable from $s_5$, resulting in the exploration of a larger state space.

A search algorithm that explores persistent sets and satisfies the $C_{2W}^S$ proviso preserves assertion violations, as the following theorem states. However, the applicability of SSR and POR is not limited to the verification of safety properties.

**Theorem 1** (Assertion Violation Preserving). *Let $A_R$ be a POR and SSR reduced state space, which satisfies the cycle proviso $C_{2W}^S$. Let $w$ be a sequence of transitions (trace) in $A_G$ leading to an error state from the initial state $s_0$. Then there exists a trace $w_r$ in $A_R$ such that an error state is weakly reachable from $s_0$.*

An error state is reached, when an assertion violation is detected during transition execution. The above theorem can be shown by induction over the length of $w$. However, the proof is omitted due to the page limitation.

## 4. SYMBOLIC SUBSUMPTION CHECKING

Before describing the actual procedure for subsumption checking between a new state $s_1$ and an already visited state $s_2$, we start with the definition of execution states.

### 4.1 Execution State

Essentially, an execution state consists of a path condition $PC$, a name-to-value mapping *vars* of variables and the kernel state $KS$. The kernel state contains the status of each thread, the current simulation time and a list of pending notifications. The kernel state only contains concrete values. It is thus included in the concrete state parts and irrelevant for symbolic subsumption checking. The path condition and variable values, in contrast, can be symbolic expressions or concrete values.

For simplicity of representation, we assume that only global variables are used[2]. Let $V = \{v_1, \dots, v_n\}$ be the set of all variables, the mapping *vars* of a state $s$ can be denoted as $\{(v_1 : e_1^s), \dots, (v_n : e_n^s)\}$, where $e_i^s$ is the value of $v_i$ in $s$. We also refer to $SP(s) = \{(PC : PC(s), v_1 : e_1^s, \dots, v_n : e_2^s)\}$ as the symbolic state parts of $s$.

### 4.2 Exact Symbolic Subsumption

Detecting that $s_1$ is subsumed by $s_2$ requires to show that the set of concrete states represented by a state $s_1$ is a subset of concrete states represented by $s_2$. A necessary condition for subsumption is thus $KS(s_1) = KS(s_2)$. Furthermore, if $e_i^{s_1}$ and $e_i^{s_2}$ are two concrete values, they must also be equal. Therefore, before trying subsumption on symbolic expressions, an equality test for these concrete state parts is performed. As an optimization we abstract away the current simulation time during this test, if the simulation is not bounded by time and the control flow of the program does not depend on the simulation time.

The subset condition for subsumption above can now be rephrased as follows: if a concrete state can be constructed from $s_1$ by assigning valid concrete values to its symbolic literals, then the same concrete state can also be constructed from $s_2$. The *Exact Symbolic Subsumption* (ESS) algorithm generates a quantified formula $F(s_1 \preccurlyeq s_2)$ that naturally encodes this requirement, as shown in the following:

$$\left(\exists x_1..x_p : PC(s_1) \wedge \bigwedge_{i \in \{1,\dots,n\}} e_i^{s_1} = f_i\right) \implies$$
$$\left(\exists y_1..y_q : PC(s_2) \wedge \bigwedge_{i \in \{1,\dots,n\}} e_i^{s_2} = f_i\right)$$

Each term $f_i$ is a fresh symbolic literal corresponding to the type of the variable $v_i$. The symbolic literals $x_1, \dots x_p$ $(y_1, \dots, y_q)$ are all symbolic literals that appear in the variable values or the path condition of $s_1$ $(s_2)$. In order to

[2]The actual implementation supports local variables by matching stack frames.

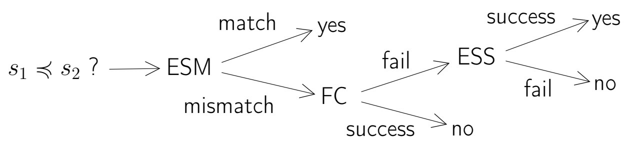

Figure 3: Optimized ESS flow

show that $F$ is valid, we check its negation $\neg F$ for unsatisfiability. Any SMT solver with support for quantifiers can be employed.

**Example 2.** Consider $SP(s_1) = (PC\colon x_1 \neq 0,\ a\colon 2 * x_1,\ b : x_2)$ and $SP(s_2) = (PC\colon True,\ a\colon y_1 + 1,\ b\colon y_2 + y_3)$. All symbolic literals contained in $s_1$ and $s_2$ are $\{x_1, x_2\}$ and $\{y_1, y_2, y_3\}$, respectively. Two fresh symbolic literals $f_1$ and $f_2$ will be introduced with corresponding types for the variables $a$ and $b$. The $\neg F(s_1 \preccurlyeq s_2)$ formula is as follows:

$$\begin{aligned}&[\exists x_1, x_2 : (x_1 \neq 0) \wedge (2 * x_1 = f_1) \wedge (x_2 = f_2)] \wedge \\ &\quad \neg\, [\exists y_1, y_2, y_3 : True \wedge (y_1 + 1 = f_1) \wedge (y_2 + y_3 = f_2)]\end{aligned}$$

## 4.3 Optimizations

The ESS algorithm detects symbolic subsumption precisely, but it can be computationally very expensive due to the use of quantifiers. Therefore, we devise three optimization techniques:

1. *Explicit Structural Matching* (ESM) heuristically detects state equivalence (a special form of subsumption) by matching the structure of symbolic expressions;

2. *Filter Check* (FC) tries to refute subsumption by checking a simpler formula without quantifiers;

3. *Expression Simplification* (SIMP) is a generic technique that reduces the size of symbolic expressions and the path condition, thus also the size of the SMT formulas.

ESM and FC can in many cases avoid expensive ESS queries as shown in Fig. 3.

### 4.3.1 Explicit Structural Matching

The ESM heuristic is based on the simple observation that two symbolic expressions are semantically equal, if they are structurally equal. ESM checks whether every pair $(e_i^{s_1}, e_i^{s_2})$ as well as $(PC(s_1), PC(s_2))$ are equal except for the renaming of symbolic literals. Every symbolic literal has to be mapped to exactly another type-compatible symbolic literal. Matching the path conditions ensures that the mapped symbolic literals have the same constraints on both states. For example, ESM detects equivalence between $SP(s_1) = (pc\colon x_1 > 5,\ x_1 + 1)$ and $SP(s_2) = (pc\colon x_2 > 5,\ x_2 + 1)$ if $x_1$ and $x_2$ have the same type.

### 4.3.2 Filter Check

The FC heuristic constructs a random concrete state $s_C$ from $s_1$ and checks if $s_C$ can be constructed from $s_2$. If the check fails, $s_1$ cannot be subsumed by $s_2$, and thus an ESS query is not necessary. This situation is shown on the left side of Fig. 4. Otherwise, $s_1$ might be subsumed by $s_2$, but not always as depicted on the right side.

The concrete state $s_C$ is obtained by employing an SMT solver to solve $PC(s_1)$ and get a complete model (i.e. every symbolic variable is assigned to a concrete value). Such a model is always available, since the path to $s_1$ has been shown to be feasible before. Now, $s_C$ can be constructed from $s_2$, iff the following formula is satisfiable:

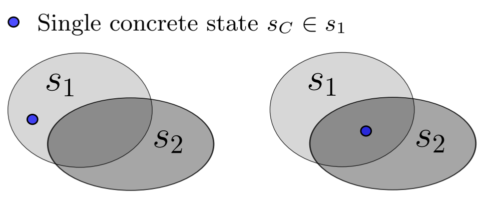

Figure 4: Filter check principle

$$PC(s_2) \wedge \bigwedge_{i \in \{1,\ldots,n\}} e_i^{s_2} = e_i^{s_C}$$

One implementation detail that is crucial to the overall performance of FC is to create and cache $s_C$ when the feasibility of the path to $s_1$ is checked. This avoids an unnecessary solver query by FC to solve $PC(s_1)$ again.

### 4.3.3 Expression Simplification

Symbolic expressions are simplified based on term rewriting, e.g. folding of concrete arguments $(1 + x + 2 \mapsto x + 3)$ or simplification of special cases $(x \vee \neg x \mapsto T)$.

The path condition is a conjunction of terms $PC = c_1 \wedge \ldots \wedge c_n$ representing constraints. During symbolic simulation, symbolic literals go out-of-scope when the variable using them is overwritten. For example, if $x_1$ is only used by the variable $v$, then $x_1$ goes out-of-scope when $v$ is assigned to a new literal $x_2$. If a constraint $c_i$, which contains both out-of-scope and in-scope literals, does not exist, the out-of-scope literals can be safely removed. Constraints, which contains only these literals, also become irrelevant and thus are eliminated. This process is performed using standard garbage collection techniques.

## 5. EXPERIMENTS

We have implemented the proposed approach and evaluated it using an extensive set of benchmarks from the literature on SystemC verification [2, 10, 6, 13] and some new benchmarks. All experiments are performed on a 3.4 GHz AMD machine running Linux. The time and memory limits are set to 1000 seconds and 4GB, respectively. The abbreviations T.O. and M.O. denote that the time and memory limit has been exceeded, respectively. The result tables show the benchmark name in the first column and the verification result, with $S$ for safe and $U$ for unsafe, in the second column. All runtimes are specified in seconds.

First we have performed a comparison of the different optimization techniques which have been discussed in Section 4.3, for our symbolic state matching algorithm ESS. Table 2 shows the results of the base algorithm (ESS), with structural matching (+ESM), with filter check (+FC), with expression simplifications (+SIMP) and a combination of all techniques (+ALL). All configurations use symbolic expression simplifications based on term rewriting. The Z3 solver[3] is used to handle all symbolic queries, since it provides quantifier support as required for the ESS algorithm. It can be observed that every optimization technique results in improvements compared to the base ESS algorithm. The combination of all optimization techniques yields the overall best results. The last column shows the observed factor of improvement (FoI) compared to the base algorithm.

[3]Available at `http://z3.codeplex.com`

Table 2: Comparison of ESS optimizations (runtime in seconds)

| Benchmark | V | ESS | ESS+ESM | ESS+FC | ESS+SIMP | ESS+ALL | FoI |
|---|---|---|---|---|---|---|---|
| buffer.p9 | S | 323.241 | 193.952 | 444.074 | 329.708 | 196.310 | 1.65 |
| mem-slave-tlm-bug2.5 | U | 26.677 | 26.474 | 17.292 | 26.174 | 18.018 | 1.48 |
| pressure-sym.50.5 | S | 8.244 | 6.973 | 8.470 | 8.203 | 7.271 | 1.13 |
| pressure-sym.nb.50.5 | S | 229.240 | 228.437 | 192.680 | 230.193 | 188.766 | 1.21 |
| token-ring2.12 | S | 280.526 | 255.262 | 184.093 | 244.308 | 122.981 | 2.28 |
| token-ring-bug2.20 | U | 8.479 | 8.573 | 7.969 | 8.388 | 7.041 | 1.20 |
| token-ring-bug.20 | U | 7.626 | 7.669 | 6.323 | 7.657 | 6.502 | 1.17 |
| token-ring.40 | S | 223.562 | 216.143 | 192.283 | 226.818 | 167.107 | 1.34 |

Table 3: Comparison with Kratos (runtime in seconds)

| Benchmark | V | ESS+ALL | KRATOS | FoI |
|---|---|---|---|---|
| kundu | S | 5.988 | **1.129** | 0.19 |
| mem-slave-tlm-bug2.1 | U | 4.735 | **1.896** | 0.40 |
| mem-slave-tlm-bug2.5 | U | **18.018** | 30.702 | 1.70 |
| mem-slave-tlm-bug.1 | U | 3.746 | **2.482** | 0.66 |
| mem-slave-tlm-bug.5 | U | **5.230** | 192.143 | 36.74 |
| mem-slave-tlm.1 | S | 5.934 | **3.166** | 0.53 |
| mem-slave-tlm.5 | S | **4.599** | 264.163 | 57.44 |
| pressure-safe.50 | S | **3.884** | 184.896 | 47.60 |
| pressure-safe.100 | S | **6.147** | T.O. | ∞ |
| pressure-sym.50.5 | S | **7.271** | 202.594 | 27.86 |
| buffer.p8 | S | 83.624 | **75.069** | 0.90 |
| buffer.p9 | S | **196.310** | T.O. | ∞ |
| condition-builder.16 | S | 5.494 | 22.156* | - |
| condition-builder.32 | S | 10.682 | 475.397* | - |
| pressure-sym.nb.50.5 | S | 188.766 | 154.878* | - |
| rbuf2-bug1.2 | U | **3.284** | 49.286 | 15.01 |
| rbuf2-bug2.2 | U | **3.052** | 47.801 | 15.66 |
| simple-fifo-1c2p.10 | S | **8.367** | 15.712 | 1.88 |
| simple-fifo-1c2p.20 | S | **14.759** | 106.184 | 7.19 |
| simple-fifo-1c2p.50 | S | **38.957** | T.O. | ∞ |
| simple-fifo-bug-1c2p.20 | U | **7.147** | 54.729 | 7.66 |
| simple-pipeline | S | **3.046** | 7.443 | 2.44 |
| symbolic-counter.1.3 | S | **2.498** | 7.954 | 3.18 |
| symbolic-counter.1.6 | S | **4.304** | 34.579 | 8.03 |
| symbolic-counter.1.12 | S | **8.932** | 767.717 | 85.95 |
| token-ring2.12 | S | 122.981 | **110.560** | 0.90 |
| token-ring2.20 | S | **740.698** | M.O. | ∞ |
| token-ring-bug2.17 | U | **6.089** | 88.252 | 14.49 |
| token-ring-bug2.20 | U | **7.041** | M.O. | ∞ |
| token-ring-bug.10 | U | 4.667 | **0.525** | 0.11 |
| token-ring-bug.20 | U | **6.502** | M.O. | ∞ |
| token-ring.13 | S | 9.096 | **4.279** | 0.47 |
| token-ring.15 | S | **11.039** | 126.983 | 11.50 |
| token-ring.40 | S | **167.107** | M.O. | ∞ |
| toy-sym | S | 3.494 | **3.410** | 0.98 |
| transmitter.90 | U | **16.012** | M.O. | ∞ |
| transmitter.100 | U | **19.590** | 266.907 | 13.62 |

KRATOS is called with the options: `-opt_cfa 2 -inline_threaded_function 1 -dfs_complete_tree=false -thread_expand=DFS -node_expand=BFS -po_reduce -po_reduce_sleep -arf_refinement=RestartForest`

Consequently, this optimized version of ESS is compared with KRATOS in its most recent version [6], which is the state of the art SystemC model checker for handling cyclic state spaces as mentioned earlier. The results are presented in Table 3. The table is divided into two halves by a double line. The upper half shows benchmarks with acyclic state spaces, whereas the more important cyclic state spaces are shown in the lower half.

Our approach shows very competitive results compared to KRATOS. Improvements up to two orders of magnitude can be observed, as shown in the last column (FoI). This can especially be observed with up-scaled benchmarks, e.g. the *token-ring*, *symbolic-counter*, *mem-slave-tlm* and *pressure* benchmarks. On some benchmarks, KRATOS shows better results but the runtime differences are not significant. Furthermore, on some safe benchmarks, KRATOS reports spurious counterexamples. These benchmarks are marked with * and not considered in the comparison.

## 6. CONCLUSION

A *stateful symbolic simulation* approach has been proposed in this paper for the efficient and complete verification of safety properties in high-level SystemC designs. With the proposed approach, safety properties can be completely proven on cyclic state spaces, which has not been possible before with symbolic simulation. The well-known state explosion problem is alleviated by integrating two complementary reduction techniques, namely POR and SSR. The former allows to prune redundant schedules, whereas the latter can increase the effectiveness of symbolic state matching significantly. In addition to an exact algorithm for subsumption detection, we proposed several optimizations to improve its efficiency. The experiments using an extensive set of benchmarks demonstrated the efficiency of these optimizations and the competitiveness of the stateful symbolic simulation with the state of the art.

## 7. REFERENCES

[1] S. Anand, C. S. Păsăreanu, and W. Visser. Symbolic execution with abstract subsumption checking. In *SPIN*, pages 163–181, 2006.

[2] N. Blanc and D. Kroening. Race analysis for SystemC using model checking. *ACM Trans. Des. Autom. Electron. Syst.*, 15(3):21:1–21:32, June 2010.

[3] D. Bosnacki, S. Leue, and A. Lafuente. Partial-order reduction for general state exploring algorithms. In A. Valmari, editor, *Model Checking Software*, volume 3925 of *Lecture Notes in Computer Science*, pages 271–287. Springer, 2006.

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

[5] 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.

[6] A. Cimatti, I. Narasamdya, and M. Roveri. Software model checking SystemC. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 32(5):774–787, 2013.

[7] S. Evangelista and C. Pajault. Solving the ignoring problem for partial order reduction. *Int. J. Softw. Tools Technol. Transf.*, 12(2):155–170, May 2010.

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

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

[10] 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.

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

[12] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

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

[14] A. Valmari. Stubborn sets for reduced state space generation. In *Int'l Conf. on Application and Theory of Petri Nets*, pages 1–22, 1989.

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