Partial Order Reduction for Scalable Testing of SystemC TLM Designs

Sudipta Kundu *
UC San Diego
skundu@cs.ucsd.edu

Malay Ganai
NEC Labs America
malay@nec-labs.com

Rajesh Gupta
UC San Diego
rgupta@cs.ucsd.edu

ABSTRACT

A SystemC simulation kernel consists of a deterministic implementation of the scheduler, whose specification is nondeterministic. To leverage testing of a SystemC TLM design, we focus on automatically exploring all possible behaviors of the design for a given data input. We combine static and dynamic partial order reduction techniques with SystemC semantics to intelligently explore a subset of the possible traces, while still being provably sufficient for detecting deadlocks and safety property violations. We have implemented our exploration algorithm in a framework called Satya and have applied it to a variety of examples including the TAC benchmark. Using Satya, we automatically found an assertion violation in a benchmark distributed as a part of the OSCI repository.

Categories and Subject Descriptors

I.6.4 [**Computing Methodologies**]: Simulation and Modeling—*Model Validation and Analysis*

General Terms

Algorithms, Design and Verification

Keywords

Partial-Order Reduction, Verification, Simulation, Testing.

1. INTRODUCTION

The growing complexity of systems and their implementation into silicon encourages designers to look for ways to model designs at higher levels of abstraction and then incrementally build portions of these designs – automatically or manually – while ensuring system-level functional correctness. System description languages such as SystemC and SystemVerilog enable designers to describe designs at various levels of abstraction. These are particularly useful in behavioral/algorithmic and transaction level modeling (TLM) [21, 12, 5]. The idea of SystemC TLM is to provide a golden reference of the system in an early phase of the development process and allow fast simulation. This design abstraction supports new synchronization procedures which make current techniques for RTL validation inapplicable. SystemC is a set of library routines and macros implemented in C++, which makes it possible to simulate concurrent processes, each described by ordinary C++ syntax. SystemC is both a description language and a simulation kernel.

Problem Statement: Dynamic validation has so far been the "workhorse" for validating SystemC designs. As pointed out in [22], adapting software formal verification techniques to SystemC has been formidable task, mainly due to its object-oriented nature and its support for both synchronous and asynchronous semantics with a notion of time. In the absence of accepted formal semantics, SystemC models and methods attempt to speed up simulation. However, simulation does not guarantee completeness in verification or validation as it cannot expose all possible ordering of events. The need is therefore to apply formal verification techniques to improve system level simulation coverage.

Related Work: Prior work on SystemC focuses mainly on improving simulation performance [17] and generating representative inputs for the design [11] and formalizing the semantics of SystemC [18, 16, 13, 20], while ignoring the problem of generating all possible behaviors of the design. However, recently researchers address the above problem by automatically generating all valid scheduling of the design [14]. Their work used dynamic partial-order reduction (POR) techniques [7] to avoid generation of two schedulings that have the same effect on the system's behavior. POR techniques are extensively used by *software model checkers* for reducing the size of the state space of concurrent system at the implementation level [10, 15]. Other state space reduction techniques, such as slicing [6, 19] and abstraction [4], are orthogonal and can be used in conjunction with POR. The POR techniques can be divided in two main categories: *static* [9] and *dynamic* [7].

The main static POR techniques are *persistent/stubborn* sets and *sleep* sets [9]. Intuitively, the persistent/stubborn set techniques compute a provably sufficient subset of the enabled transitions in each visited states such that if a selective search is done using only the transitions from these subsets the detection of all the deadlocks and safety property violations is guaranteed. All these algorithms infer the persistent sets from the static structure (code) of the system being verified. On the other hand, the sleep set techniques exploits independencies between the transitions in the persistent sets to reduce interleavings. Both these techniques are orthogonal and can be applied simultaneously [9]. In contrast, the dynamic POR technique evaluates the dependency relation dynamically between the enabled and executed transitions for a given execution.

Overview: In this paper, we present a practical technique for checking all possible execution traces of a SystemC design. We focus on using formal verification techniques developed for software to extend dynamic validation of SystemC

*The author worked on this project as an intern at NECLA, USA.

DAC 2008, June 8–13, 2008, Anaheim, California, USA.

TLM designs. In what follows, we assume the representative inputs are already provided, possibly using techniques presented in [11] and the execution terminates. Thus, we focus our discussion mainly on detecting deadlocks, write-conflicts and safety property violations such as assertion violations. Note that termination can be guaranteed in SystemC by bounding the execution length during simulation.

To cope with the state-space-explosion problem associated with any concurrent system we use a combination of static and dynamic POR techniques. In particular, we first use static analysis techniques to compute if two atomic blocks are *independent*, meaning that their execution does not interfere with each other, and changing their order of execution will not alter their combined effect. Next, we start by executing one random trace of the program until completion, and then dynamically compute backtracking points along the trace that identify alternative transitions that need to be explored because they may lead to different final states. However, unlike dynamic techniques [14, 7] we use the information obtained by static analysis in a *query-based* framework, rather than dynamically collecting the information and analyzing it during runtime. Using static information we tradeoff precision for performance. We chose performance since for most SystemC designs we can find the dependency relation quite precisely by using static analysis only. Intuitively, our approach infers the persistent sets dynamically using information obtained by static analysis. To further reduce the number of explored traces we use the *sleep sets* in conjunction with the above technique. Our algorithm is *stateless* [10], i.e., it stores no state representations in memory but only information about which transitions and traces have been executed so far.

Moreover, we adapt the POR techniques to further improve the efficiency of the algorithms by using SystemC specific semantics. Adaptations are needed because in SystemC: processes are co-operatively multitasking; supports the concept of δ-cycle, which reduces the analysis of backtracking points immensely; supports signal variables that do not change values until an update phase; synchronization is done using events instead of locks; and enabled processes cannot be disabled by another one.

Contributions:

1. We propose a novel *query-based* framework that combines static and dynamic POR techniques to cover all possible executions of a SystemC design. We reduce the runtime overhead by computing the dependency information statically, and use it during runtime, without much loss of precision.

2. We use SystemC specific semantics to further improve the efficiency of the POR techniques. In SystemC, processes are co-operatively multitasking and supports the concept of δ-cycle. This synchronous semantics of SystemC reduces the size of persistent set and consequently reduces the analysis of backtracking points immensely.

3. We use the Open SystemC Initiative's (OSCI) SystemC simulator [2] to implement our algorithm of exploring all possible behaviors of a SystemC design in a validating system called Satya. We use Satya to check the correctness of a variety of small examples and two benchmark designs. In particular, we were able to automatically find an assertion violation in the FIFO benchmark (distributed as a part of OSCI repository), which may not have been found by simulation. We also applied our tool on an industrial benchmark namely the TAC platform [3].

2. EXAMPLE

Let us start by examining the salient features of SystemC using a simple producer-consumer example shown in Figure 1. A SystemC program is a set of interconnected modules communicating through channels using *transactions*, events and shared variables collectively called *communication objects*. A module comprises of a set of ports, variables, processes and methods. Processes are small pieces of code that run concurrently with other processes and are managed by a *non-preemptive scheduler*. The semantics of concurrency is *cooperatively multitasking*: a type of multitasking in which the process currently executing must offer control to other processes. As such, a wait-to-wait block in a process is atomic. The processes exchange data between themselves using shared variables (signals and non-signals). During the execution of a SystemC design, all signal values are stable until all processes reach the waiting state. When all processes are waiting, signals are updated with the new values (see Update Phase in §3). In contrast, the non-signal variables are standard C++ variables which are updated immediately during execution.

For clarity the syntactic details of SystemC are not shown in Figure 1. It has three processes namely P_1 (lines 5-10), P_2 (lines 11-19) and C_1 (lines 20-31). The global variables of the program are shown in lines 1-4. The program uses a shared *data* array as a buffer, and an integer *num*, which indicates the total number of elements in the buffer. The producer P_1 in a loop writes to the buffer and then syncronizes by waiting (or blocking) (line 9) on time for 4 nanoseconds (SC_NS). Similarly, producer P_2 writes to the buffer and if *timer* is set then notifies the event e and then synchronizes using time. The consumer C_1 on the other hand, waits (or blocks) (line 25) on the event e when the buffer is empty, until the notify (line 17) on e is invoked in the P_2 process. If there are elements in the buffer then C_1 consumes it and synchronizes on time like the other processes. For synchronization SystemC uses wait-notify on events and time. In what follows, we will use this example to guide our discussion.

3. SYSTEMC SIMULATION KERNEL

Simulation involves the execution of a discrete event scheduler, which in turn triggers or resumes the execution of processes within the application. The functionality of the scheduler (as per IEEE std. [2]) can be summarized as follows:

1. Initialization Phase: Initialize every eligible method and thread process instance in the object hierarchy to the set of runnable processes.

2. Evaluation Phase: From the set of runnable processes, select a process instance in an unspecified order and execute it non-preemptively. This can, in turn, notify other events, which can result in new processes being ready to run. Continue this step till there are processes to run.

3. Update Phase: Update signal values for all processes in step 2, that requested for it.

4. δ-Notification Phase: Trigger all pending δ-delayed notifications, which can wake up new processes. If, at the end of this phase, the set of runnable processes is non-empty, go back to the evaluation phase.

5. Timed-Notification Phase (τ): If there are pending timed notification, advance simulation time to the earliest deadline. Determine the set of runnable processes that can run at this time and go to step 2. Otherwise, end simulation.

To simulate synchronous concurrent reactions on a sequen-

```
1.  int MAX, num = 0, i = 0
2.  char data[2]
3.  sc_event e
4.  bool timer = false

5.  process P1()
6.      while (i < MAX)
7.          data[num] = 'A'
8.          ++num
9.          wait (4, SC_NS)
10.     return

11. process P2()
12.     while (i < MAX)
13.         data[num] = 'B'
14.         ++num
15.         if (timer)
16.             timer = false
17.             notify (e)
18.         wait (4, SC_NS)
19.     return

20. process C1(int x)
21.     while (i < MAX)
22.         if (num == 0)
23.             timer = true
24.             i++
25.             wait (e)
26.         num--
27.         assert (num ≥ 0)
28.         c = data[num]
29.         i++
30.         wait (x, SC_NS)
31.     return
```

Figure 1: (Left) Simple Producer-Consumer Example

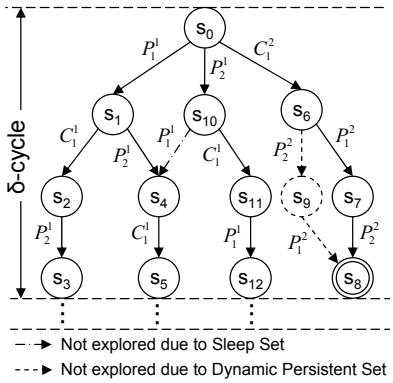

Figure 2: (Right) A partial execution-tree showing only the first δ-cycle

tial computer SystemC supports the concept of δ-cycle. A δ-cycle is an event cycle (consisting of evaluate, update and δ-notification phase) that occurs in 0 simulation time.

Nondeterminism: For a given input, a SystemC program can produce different output behavior due to nondeterministic scheduling. To illustrate this let us consider the processes P_1 and C_1 from the example in §2 with $\mathsf{MAX} = 2$ and $x = 4$ (line 20). It has the following 4 possible executions, where τ denotes a time elapse:

- $\mathsf{P}_1\mathsf{C}_1\tau\mathsf{P}_1\mathsf{C}_1\tau\mathsf{P}_1\mathsf{C}_1$ and $\mathsf{P}_1\mathsf{C}_1\tau\mathsf{P}_1\mathsf{C}_1\tau\mathsf{C}_1\mathsf{P}_1$ leads to a successful termination of the program with 2 A's being produced and consumed.
- $\mathsf{P}_1\mathsf{C}_1\tau\mathsf{C}_1\mathsf{P}_1$ leads to a deadlock situation. As C_1 is waiting for the event e (line 25) and P_1 has terminated.
- $\mathsf{C}_1(\mathsf{P}_1\tau)^*$ leads to an array bound violation as C_1 waits for the event e and P_1 goes on producing in the array *data*.

In general, a simulator will execute only one of the 4 possible executions. For instance with the reference OSCI simulation kernel [2], only the first execution will be scheduled and the other buggy executions will be ignored. Thus, it is important to test all possible execution of a SystemC design.

Now consider the same example with all 3 processes and $\mathsf{MAX} = 8$ and $x = 2$ (line 20). It has 3701 possible executions. A naive algorithm will try to explore all possible executions one by one and will face scalability issues. In the following sections we discuss our approach of exploring these executions and how we adapt POR techniques for SystemC. For this example, our approach will explore only 767 executions and still remain *provably sufficient for detecting deadlocks and assertion violations*.

4. FORMAL SETTING

In this section, we describe some standard definitions used in the context of POR [14, 7, 9], which have been adapted here for SystemC.

DEFINITION 1. Transition: *A transition moves the system from one state to a subsequent state. In SystemC there are three types of transitions:*

1. Immediate-transition *change the state by executing a finite sequence of operations of a chosen process followed by a* wait *operation or termination of the same process.*
2. δ-transition *change the state by updating all the signals, and by triggering all the δ-delayed notification that were requested in the current δ-cycle.*
3. *A* time-transition *change the system state by updating the simulation time.*

Let $\mathcal{T}$ denote the set of all transitions of the system. An i^{th} transition of process P is denoted by P^i. For $t = P^i \in \mathcal{T}$ we denote, $\mathsf{Process}(t)$ as the process P.

DEFINITION 2. Runnable: *A transition $t \in \mathcal{T}$ is runnable in state s, written $t \in runnable(s)$ if it can be executed in s.*

If $t \in runnable(s)$, then we say the execution of t from s produces a successor state s', written $s \xrightarrow{t} s'$. We write $s \xRightarrow{w} s'$ to mean that the execution of the finite sequence $w \in \mathcal{T}^*$ leads from s to s'. A state s, where $runnable(s) = \emptyset$ is called a deadlock, or a terminating state.

The behavior of a SystemC program is represented using a transition system $M = (State, s_0, \Delta)$, where $State$ is a finite non-empty set of states, s_0 is the initial state of the system and $\Delta \subseteq State \times State$ is the transition relation defined by $(s, s') \in \Delta$ iff $\exists\, t \in \mathcal{T} : s \xrightarrow{t} s'$.

A transition $t_1 \in \mathcal{T}$ is called *co-runnable* with another transition $t_2 \in \mathcal{T}$, written $\mathsf{CoRunnable}(t_1, t_2)$ if $\exists\, s \in State$ such that both $t_1, t_2 \in runnable(s)$. Note that 2 transitions of the same process cannot be co-runnable in SystemC. An execution of the program is defined by a trace of the system.

DEFINITION 3. Trace: *A trace $\phi \in \mathcal{T}^*$ of M is a finite (possibly empty) sequence of transitions $t_0, \cdots, t_{n-1}$ where there exists states $s_0, \cdots s_n$ such that s_0 is the initial state of M and $s_0 \xrightarrow{t_0} s_1 \cdots \xrightarrow{t_{n-1}} s_n$.*

For a given trace $\phi = t_0, \cdots, t_n$; ϕ_i represents the transition t_i; $\phi_{0\cdots i}$ denotes the trace $t_0, \cdots, t_i$; $\mathsf{Pre}(\phi, i)$ denotes the state s_i and $\mathsf{Post}(\phi, i)$ denotes the state s_{i+1}.

The following definition states the condition when two transitions are independent, meaning that they result in the same state when executed in different orders.

DEFINITION 4. Independence Relation: *A relation $\mathcal{I} \subseteq \mathcal{T} \times \mathcal{T}$ is an independence relation of M if $\mathcal{I}$ is symmetric and irreflexive and the following conditions hold for each $s \in State$ and for each $(t_1, t_2) \in \mathcal{I}$:*

1. *if $t_1, t_2 \in runnable(s)$ and $s \xrightarrow{t_1} s'$ then $t_2 \in runnable(s')$*
2. *if $t_1, t_2 \in runnable(s)$, then there is a unique state s' such that $s \xRightarrow{t_1 t_2} s'$ and $s \xRightarrow{t_2 t_1} s'$*

Transitions $t_1, t_2 \in \mathcal{T}$ are *independent* in M if $(t_1, t_2) \in \mathcal{I}$. Thus, a pair of independent transitions cannot make each other runnable when executed and runnable independent transition commute. The complementary dependence relation $\mathcal{D}$ is given by $(\mathcal{T} \times \mathcal{T}) - \mathcal{I}$.

Two traces are said to be *equivalent* if they can be obtained from each other by successively permuting adjacent independent transitions. Thus, given a valid independence

relation, traces can be grouped together into *equivalence classes*. For a given trace, we define a *happens-before* relation between its transitions as follows:

DEFINITION 5. Happens-before: *Let $\phi = t_0 \cdots t_n$ be a trace in M. A happens-before relation $\prec_\phi$ is the smallest relation on $\{0 \cdots n\}$ such that*

1. *if $i \leq j$ and $(\phi_i, \phi_j) \in \mathcal{D}$ then $i \prec_\phi j$.*
2. *$\prec_\phi$ is transitively closed.*

In our algorithm we use a variant of the above happens-before relation which is defined as follows: for a given trace $\phi = t_0 \cdots t_n$ in M and $i \in \{0 \cdots n\}$, i happens-before process P, written, $i \prec_\phi P$ if either

1. $\mathsf{Process}(\phi_i) = P$ or
2. $\exists k \in \{i+1, \cdots, n\}$ such that $i \prec_\phi k$ and $\mathsf{Process}(\phi_k) = P$.

5. OUR APPROACH

We obtain partial-order of *runnable* processes statically by identifying the dependent transitions. A transition in SystemC is an atomic block, which in turn is a non-preemptive sequence of operations between *wait* to *wait*. Note, due to branching within an atomic block, such blocks may not be derived statically. An atomic execution is *dependent* on another atomic execution if it is enabled or disabled by the other or there exists read-write conflicts on the shared variable accesses in these blocks. In our approach, we first derive wait-notify control skeleton of the SystemC design, and then enumerate all possible atomic blocks. We then perform dependency analysis on the set of atomic blocks, and represent the information symbolically. These static information are used later, while exploring the different executions of the design. In particular, we *query* to check if a given pair of atomic blocks (corresponding to the runnable processes) need to be interleaved. If not, we do not consider that interleaving of runnable processes. In the following sections we describe our algorithm in more details.

5.1 Static Analysis

Our goal is to execute only one trace from each equivalence class for a given dependence relation. Thus, the first step is to compute this dependence relation. We use static analysis techniques to compute if two transitions are dependent. Intuitively, two transitions are dependent if they operate on some shared communication objects. In particular, we use the following rules to compute the dependence relation $\mathcal{D}$, i.e. $\forall\ t_1, t_2 \in \mathcal{T}, (t_1, t_2) \in \mathcal{D}$ if any of the following holds:

1. a write on a shared *non-signal* variable v in t_1 and a read or a write on the same variable v in t_2.
2. a write on a shared *signal* variable s in t_1 and a write on the same variable s in t_2.
3. a wait on an *event* e in t_1 and an immediate notification on the same event e in t_2.

Note here that the order in which the statements occur within a transition does not matter. For each transition $t \in \mathcal{T}$, we maintain four sets - read and write sets for shared non-signal variables and shared signal variables (written, $R_{t,ns}, W_{t,ns}, R_{t,s}, W_{t,s}$ respectively). Thus, rule 1 can be re-written as, $(W_{t_1,ns} \cap R_{t_2,ns}) \cup (W_{t_1,ns} \cap W_{t_2,ns}) \neq \emptyset$. And rule 2 can be re-written as, $W_{t_1,s} \cap W_{t_2,s} \neq \emptyset$.

In the rules mentioned above, we saw that, in general, two transitions with write operations on a shared variable are dependent. But to exercise more independency we consider special cases of write operations (called *symmetric write*) that can be considered as being independent (applying Definition 4). For instance, two constant addition or constant multiplication with the same variable can be considered as being independent. We also use static slicing techniques to remove irrelevant operations to further extract more independency between the transitions [6]. Intuitively, if a statement does not influence the property that we are checking than that statement can be removed in the sliced program.

To illustrate the above rules, consider the example from Figure 1. Consider the *wait* to *wait* atomic transition consisting of the lines (6-9) in process P_1 and the transition consisting of the lines (12-18) in process P_2. In general, these two transitions are dependent because they both write to the variable *data* and *num*. However, if the property that we are checking is the assertion in line 27 then we can get a sliced program by removing the statements inside the boxes, while still remaining correct for detecting the assertion violation. Now, if we consider only the rules 1, 2 and 3 from above then the two transitions are still dependent in the sliced program because they both write to the variable *num*. But, notice that both the writes to the variable *num* are symmetric (increment). Thus, we have that the two transitions are independent if the property that we are checking is only the assertion ($num \geq 0$) at line 27.

```
1.  type Runnable := list of Transition
2.  type TSet := set of Transition
3.  type State := Runnable × TSet × TSet
4.  type Schedule := sequence of State

5.  function Explore() : void
6.    let sched := Simulate(∅)
7.    let depth := sched.Size − 1
8.    while depth ≥ 0 do
9.      let φ := sched.Trace
10.     let s := sched.At(depth)
11.     Analyze(φ, depth)
12.     if ∃t ∈ s.Todo \ s.Sleep then
13.       s.Runnable.Add(0, s.Runnable.Remove(t))
14.       let newSched := sched.Copy(0, depth)
15.       sched := Simulate(newSched)
16.       depth := sched.Size − 1
17.     else
18.       depth := depth − 1

19. function Analyze(φ : Trace, depth : int) : void
20.   let start := StartOfDeltaCycle(φ_depth)
21.   for each i | start ≤ i < depth do
22.     if Query(φ_i, φ_depth) = Dependent
23.       and CoRunnable(φ_i, φ_depth)) then
24.       let s := Pre(φ, i)
25.       let p := Process(φ_depth)
26.       if Runnable(s, p) then
27.         s.Todo := s.Todo ∪ {Transition(s, p)}
28.       elseif ∃ j > i | Runnable(s, Process(φ_j))
29.                 and j ≺_{φ_0···depth} p then
30.         s.Todo := s.Todo ∪
31.                   {Transition(s, Process(φ_j))}
32.       else
33.         s.Todo := s.Runnable
```

Figure 3: The Explore Algorithm

5.2 The Explore Algorithm

Our Explore algorithm shown in lines 5-18 of Figure 3,

explores a reduced set of possible executions of a SystemC design. Our algorithm presented here is *stateless* [10], i.e., it stores no state representations in memory but only information about which transitions and traces have been executed so far. Although, our approach will be slower than an algorithm that maintains full state information, it requires considerably less amount of memory, especially when the design have large number of variables. It explores each *non-equivalent* trace of the system by re-executing the design from its initial state.

The algorithm maintains a *sched* of type *Schedule*. A *Schedule* is a sequence of *States*. Each *State* s is a 3-tuple $(Runnable, Todo, Sleep)$ where, *Runnable* is a sequence of *Transitions* that are runnable in state s, *Todo* is a set of *Transitions* that needs to be explored from s, and *Sleep* is the set of *Transitions* that are no longer needed to be explored from s. The algorithm also uses a function Simulate (not shown here) that takes as input a prefix schedule and then executes it according to the trace corresponding to the schedule. Once the prefix trace ends, it randomly chooses a runnable transition that is not in the *Sleep Set* of the current state and executes it. The function continues the above step till completion of the simulation and returns the *Schedule* for the current execution. To further reduce the explored transitions, the Simulate function maintains a sleep set for each state in the same way as explained in VeriSoft [9, 10].

The Explore function starts by executing a random schedule (as the prefix trace is $\emptyset$) and returns the schedule in *sched* (line 6). Our algorithm traverses the execution-tree bottom up and *depth* maintains the position in the tree such that the sub-tree below *depth* has been fully explored. Note that by traversing the execution-tree bottom-up we need to maintain very little state information. While we have not traversed the entire execution-tree, let $sched = s_0, \cdots s_{depth}, \cdots s_n$ then $\phi = t_0, \cdots t_i, \cdots t_{n-1}$ (line 9) is the trace corresponding to *sched* such that $\phi_i = s_i.\mathsf{Runnable.At}(0)$ and $s = s_{depth}$ (line 10). Using the computed trace ϕ, Explore then finds out the transitions that can be dependent with the transition ϕ_{depth} using the function Analyze (line 11) and adds those in the *Todo* set of the corresponding state. Next, if there exists any transition $t \in Todo \backslash Sleep$ in the state s (line 12), then the Explore function swap the transition t with the first element of *Runnable* in state s (line 13), copies the prefix schedule (line 14) and simulate it using the Simulate function (line 15). Otherwise, we have explored all required transitions in the sub tree below *depth* and now will explore all the transitions in the sub tree below $depth - 1$ (line 18).

The Analyze function takes as argument a trace ϕ and an integer *depth*. Next, it finds the start of the δ-cycle to which ϕ_{depth} belongs (line 20). Then, for each transition ϕ_i such that $i < depth$ and belongs to the same delta cycle (line 21), we check if ϕ_i and ϕ_{depth} are dependent using a query function (line 22) and may be co-runnable (line 23). If true, then it computes the state $s = \mathsf{Pre}(\phi, i)$ and p as the process to which the transition ϕ_{depth} belongs. Next, if there exists a transition of p that is runnable at s (line 26) then it adds that transition to the *Todo* set of s (line 27). Else, if there exists $j > i$ such that $j \prec_{\phi_{0 \cdots depth}} p$ (see Definition 5) and the runnable set of s contains a transition that belongs to the process to which ϕ_j belongs (line 29) then it adds that transition to the *Todo* set of s (line 30). Otherwise, it adds all runnable transitions to the *Todo* set of s (line 33).

To review our approach, consider the example from Figure 1 with all 3 processes and $\mathsf{MAX} = 1$ and $x = 2$ (line 20). A partial execution-tree for this example consisting of only the first δ-cycle is shown in Figure 2. The j^{th} transition of the process $Proc$ is given by $Proc^j$. In particular, Figure 2 shows the following *wait* to *wait* atomic transitions P_1^1 (lines 6-9), P_1^2 (lines 6, 10), P_2^1 (lines 12-18), P_2^2 (lines 12, 19), C_1^1 (lines 21, 22, 26-30) and C_1^2 (lines 21-25). Using static analysis (as explained in §5.1), we obtain the independence relation $\mathcal{I} = \{(P_2^1, P_1^1), (P_1^1, P_2^1), (P_2^2, P_1^2), (P_1^2, P_2^2)\}$. We use slicing of *data* and symmetric writes on *num* to determine the independency relation.

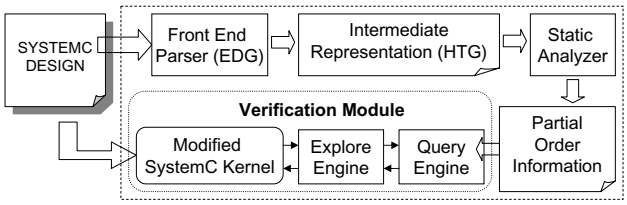

Figure 4: Our Prototype Framework - Satya

For a given data-input, let ϕ be a *trace* $(P_1^1, C_1^1, P_2^1, \cdots)$ and its corresponding state sequence be $(s_0, s_1, s_2, s_3, \cdots)$. Using the trace ϕ, we present an overview of our algorithm to explore all possible behaviours of a design for a given data-input. The state s_0 is the initial state with three runnable processes, i.e., P_1, P_2, C_1. Our Explore algorithm examines the current trace bottom up and restrict its analysis for adding backtracking points within a δ-cycle. Intuitively, for every state s_i, it checks if the transition ϕ_i, which is executed from state s_i is dependent with any other transition ϕ_j for $j < i$, i.e., in its prefix trace, that belongs to the same δ-cycle. If true, then it finds the runnable transition t_k in the pre-state s_j of ϕ_j (see Definition 3), which has a causal order with ϕ_i and adds t_k to the backtracking set of s_j. For example, when the algorithm examines the state s_2, it adds P_2^1 to the backtracking set of s_1 (since, P_2^1 and C_1^1 are dependent). Next, when it analyzes the state s_1 the algorithm adds C_1^2 to the backtracking set of s_0 and then explores the trace $\psi = (P_1^1, P_2^1, C_1^1, \cdots)$ (as P_2^1 was in s_1's backtracking set). Next, it analyzes the new trace ψ in a similar fashion. The algorithm continues in this way, till it reach state s_7, at this point P_2^1 is added to the backtracking set of s_0. The transition and state shown in dashed line is not explored. The state s_8 is a deadlock state. Note that the transition P_1^1 is not explored in the state s_{10} because it is in the *Sleep* set of s_{10} (as P_1^1 and P_2^1 are independent). Our algorithm explores only 4 different traces out of the 8 possible traces for this example.

6. OUR FRAMEWORK: SATYA

We implemented our algorithm to explore all possible valid traces of a SystemC design in a prototype framework called Satya. The implementation of Satya consists of 2 main modules - a static analyzer and a verification module. The Satya software tool is over 18,000 lines of C++ code and uses the EDG C++ front-end parser [1] and the OSCI SystemC simulator [2]. Of those, about 17,500 lines are the intermediate representation (IR) and utility functions needed by the static analyzer and the verification module (explore and query engine) is only about 800 lines.

Figure 4 presents an overview of the Satya framework. It takes a SystemC design as input - currently with the restriction of no dynamic casting and no dynamic process creation. After parsing the design description, we capture the EDG intermediate language into our own IR that consists of basic blocks encapsulated in Hierarchical Task Graphs (HTGs) [8]. The static analyzer work on the HTGs in a compositional

manner to generate the dependency relation, which is then used by the query engine. The implementation of the explore engine follows closely the algorithm described in §5.2.

The SystemC design is compiled with the verification module which contains a modified OSCI's SystemC kernel. The modified kernel implements the Simulate function of our Explore algorithm (Figure 3). It takes as input a prefix schedule and executes it till completion such that the prefix of the executed trace is same as the trace corresponding to the input prefix schedule. The modifications are still in compliance with the SystemC specification [2].

7. EXPERIMENTS AND RESULTS

Using Satya, we experimented on several small examples and two benchmark designs. In this section, we discuss the results for the two benchmarks.

7.1 FIFO Benchmark

The first benchmark is a FIFO channel example obtained from the OSCI's example repository [2]. The example has an hierarchical channel FIFO. To use the FIFO channel it uses a producer-consumer scenario. The example works fine when executed in one producer and one consumer scenario. However, if we use two producers writing to the channel and one consumer reading from that channel then we have an assertion violation. Moreover, since this bug is not visible in every trace of the example, simulation may not find it. Our tool was able to find the bug and consequently we changed the code to correct it. The following results are measured on the corrected example. The example has 3 processes executing concurrently. The total number of possible traces is directly proportional to the number of elements produced by the producers. To quantify the scalability of our tool, we report in Table 1 for different number of elements produced by the two producers, the time required using POR and the number of reduced traces explored by our tool, along with the total number of possible traces and the time required without POR.

Elements produced	Reduced #traces	Time (POR) sec:msec	Total #traces	Time (no-POR) sec:msec
14	6	00:032	8	00:046
28	42	00:265	80	00:469
44	318	02:313	992	06:344
62	2514	19:031	13376	93:563

Table 1: Results for the FIFO Benchmark

7.2 TAC Benchmark

The second benchmark is the industrial Transaction Accurate Communication (TAC) example [3] developed by ST Microelectronics, which includes a platform composed of the following 6 modules: two traffic generators, two memories, a timer and a router to connect them. These modules are based on the TAC protocol built on top of OSCI's TLM standard. This benchmark is over 12,000 lines of SystemC code and consists of 349 functions. The example can be executed for certain number of transactions. A transaction is a read or write by the masters, namely the two traffic generators. When executed for 80,000 transactions, there are 12032 total possible traces. It took 89.47 mins to explore all these traces, whereas while checking for deadlocks in the program, our tool found all these traces to be equivalent to only one trace, which was executed in 1.3 secs. Although, for this example simulation has same coverage as our tool, simulation cannot provide correctness guarantee that is provided by our tool.

8. CONCLUSION AND FUTURE WORK

We have presented a scalable approach for testing SystemC in a query-based framework, Satya. Our approach combines static and dynamic POR techniques to reduce the number of interleavings required to expose all behaviors of SystemC. Our approach exploits SystemC specific semantics to reduce the number of backtracking points. Our experiments on a set of examples show the efficacy of our approach. Our ongoing work seeks to combine symbolic execution and property specific slicing approaches.

9. REFERENCES

[1] Edison Design Group (EDG) C/C++ Front End, 1992. `www.edg.com`.

[2] IEEE Standard 1666 SystemC Language Reference Manual, 2005. `www.systemc.org`.

[3] ST Microelectronics Transaction Accurate Communication (TAC) platform, 2005. `www.greensocs.com/TACPackage`.

[4] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on PLDI*, 2001.

[5] L. Cai and D. Gajski. Transaction Level Modeling: an overview. In *Proceedings of CODES+ISSS*, 2003.

[6] M. Dwyer and J. Hatcliff. Slicing software for model construction. In ACM Workshop on PEPM, 1999.

[7] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of POPL*, 2005.

[8] M. Girkar and C. D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. Parallel Distrib. Syst.*, 1992.

[9] P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. PhD thesis, Univerite De Liege, 1995.

[10] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of POPL*, 1997.

[11] D. Grobe, R. Ebendt, and R. Drechsler. Improvements for constraint solving in the SystemC verification library. In *Proceedings of GLSVLSI*, 2007.

[12] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[13] A. Habibi and S. Tahar. Design for verification of SystemC Transaction Level Models. In *Proceedings of DATE*, 2005.

[14] C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy. Automatic generation of schedulings for improving the test coverage of Systems-on-a-Chip. In *Proceedings of FMCAD*, 2006.

[15] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 1997.

[16] D. Kroening and N. Sharygina. Formal verification of SystemC by automatic hardware/software partitioning. In *Proceedings of MEMOCODE*, 2005.

[17] S. Meftali, J. Vennin, and J.-L. Dekeyser. A fast SystemC simulation methodology fo Multi-Level IP/SoC design. In *IFIP Intl. Workshop on IP Based SoC Design*, 2003.

[18] M. Moy, Maraninchi, and Maillet-Contoz. Lussy: A toolbox for the analysis of Systems-on-a-Chip at the Transactional Level. In *Proceedings of ACSD*, 2005.

[19] A. Sen and V. K. Garg. Formal verification of simulation traces using computation slicing. *IEEE Transactions on Computers*, 2007.

[20] R. Shyamasundar, F. Doucet, R. Gupta, and I. Kruger. Compositional reactive semantics of SystemC and verification with RuleBase. In *Proceedings of the GM R&D Workshop*, 2007.

[21] S. Swan. SystemC Transaction Level Models and RTL verification. In *Proceedings of DAC*, 2006.

[22] M. Y. Vardi. Formal techniques for SystemC verification. In *Proceedings of DAC*, 2007.