Open access • Journal Article • DOI:10.1145/2003695.2003697

Concurrency-oriented verification and coverage of system-level designs

Alper Sen

Institutions: Boğaziçi University

Published on: 27 Oct 2011 - ACM Transactions on Design Automation of Electronic Systems (ACM)

Topics: Functional verification, High-level verification, Runtime verification, Software verification and Intelligent verification

Concurrency-Oriented Verification and Coverage of System-Level Designs

ALPER SEN, Bogazici University

Correct concurrent System-on-Chips (SoCs) are very hard to design and reason about. In this work, we develop an automated framework complete with concurrency-oriented verification and coverage techniques for system-level designs. Our techniques are different from traditional simulation-based reliability techniques, since concurrency information is often lost in traditional techniques. We preserve concurrency information to obtain unique verification techniques that allow us to predict potential errors (formulated as transaction-level assertions) from error-free simulations. In order to do this, we exploit the inherent concurrency in the designs to generate and analyze novel partial-order simulation traces. Additionally, to evaluate the confidence on verification results and the gauge progress of verification, we develop novel mutation testing based on concurrent coverage metrics. Mutation testing is a fault insertion-based simulation technique that has been successfully applied in software testing. We present a comprehensive list of mutation operators for SystemC, similar to behavioral fault models, and show the effectiveness of these operators by relating them to actual bug patterns. We have successfully applied our verification and coverage techniques on industrial systems and demonstrated that current verification test suites need to be improved for concurrent designs, and we have found errors in systems that were tested previously.

Categories and Subject Descriptors: J.6 [**Computer Applications**]: Computer-Aided Engineering—*Computer-Aided design* (CAD); D.2.4 [**Software Engineering**]: Software Program Verification

General Terms: Algorithms, Verification

Additional Key Words and Phrases: SystemC, simulation, concurrency, assertion-based verification, predictive verification, coverage, mutation testing, partial-orders

ACM Reference Format:
Sen, A. 2011. Concurrency-oriented verification and coverage of system level designs. ACM Trans. Des. Autom. Electron. Syst. 16, 4, Article 37 (October 2011), 25 pages.
DOI = 10.1145/2003695.2003697 http://doi.acm.org/10.1145/2003695.2003697

1. INTRODUCTION

The complexity of System-on-Chips (SoCs) has been steadily increasing with the emergence of concurrent systems such as multicore, multiprocessor, and multithreaded systems. Concurrent systems have the property that multiple components can execute simultaneously, resulting in a very complex behavior of the system. Various scenarios of component interaction can happen due to concurrency, which are often very difficult to find. Concurrency information is often lost when traditional simulations are used because during execution of a concurrent system, the simulation scheduler (Verilog, VHDL, or SystemC scheduler) puts an arbitrary ordering on executions of concurrent components. Hence, bugs that may reveal themselves on a different execution ordering

This research was supported by a Marie Curie European Reintegration Grant within the 7th European Community Framework Programme, BU Research Fund, and the Turkish Academy of Sciences.
Author's address: A. Sen, Department of Computer Engineering, Bogazici University, 34342, Istanbul, Turkey; email: alper.sen@boun.edu.tr.

DOI 10.1145/2003695.2003697 http://doi.acm.org/10.1145/2003695.2003697

of concurrent components can be missed. Given this increasing complexity of concurrent SoCs, the traditional design entry-level Register Transfer Level (RTL) is no longer scalable for verification. It is easier to diagnose concurrency and protocol problems at abstract system levels, whereas these problems are hidden at the lower implementation levels. Additionally, the increasing design costs are also pushing design entry to system levels.

SystemC is the most popular system-level modeling language used for designing SoCs in the industry. It is a C++ library that contains constructs to represent the concurrent behavior of hardware. It is freely available from OSCI [OSCI 2010] and is an IEEE standard. The SystemC Transaction-Level Modeling (TLM) standard enables verification and debugging, as well as hardware/software co-design, architectural exploration, and power/performance analysis, all at the system level [Ghenassia 2005]. However, the combination of system-level design with concurrent multiprocessors is a challenge to EDA tool providers, and system-level tools are still in their infancy. New concurrency-oriented system-level techniques of analysis are required to improve reliability of concurrent software running on concurrent hardware.

In this article, we present concurrency-oriented verification and coverage techniques. Our techniques are based on simulation of system-level designs, rather than a static analysis of these designs. For verification, we develop techniques based on Assertion-Based Verification (ABV) [Foster et al. 2004], which is a popular simulation-based verification technique, where the assertions (specifications) are monitored during the execution of the system. ABV is a commonly used verification technique, since it does not suffer from the complexities of using formal verification techniques such as model checking. Our ABV technique differs from traditional ABV techniques because ours is a predictive verification technique, where we can predict potential errors using error-free executions. This is a valuable debugging tool that can greatly increase the detection of hard-to-find scheduling dependent errors during simulation. Potential error detection is possible in our approach due to preserving and exploiting concurrency information by tracking dependencies during the execution of a system. Using this information, we obtain a partial-order execution trace, as opposed to a total order trace, which traditional simulation-based approaches generate. In partial-order traces, only causally dependent actions are ordered, hence concurrent actions are not artificially ordered. However, the complexity of verification on partial-order traces may be high. To tackle this complexity, we use an abstraction technique called computation (trace) slicing, where a slice of a trace with respect to an assertion is a subtrace that contains all the states of the trace that satisfy the assertion such that it is computed efficiently (without traversing the state space) and represented concisely (without explicit representation of individual states). In this work, we develop a concurrency-oriented verification technique that is predictive and can be used to verify transaction-level assertions of SystemC designs. In particular, our goal is to efficiently explore a concurrent system-level program using partial-order traces and computation slicing to find schedule-related bugs.

Verification is not complete without coverage metrics [Tasiran and Keutzer 2001] to measure effectiveness, gauge progress, and help determine when the design is robust enough. For coverage, we develop techniques based on mutation testing, which is a fault injection-based simulation technique where faults are inserted at various design points and where it is checked whether the test suites used in predictive verification can propagate these faults to the outputs of the design. Mutation operators are similar to functional fault models for high-level design descriptions. Mutation testing was shown to be an effective technique for improving test suites [Frankl et al. 1997; Walsh 1985; Li et al. 2009]. A verification test suite developed for a sequential program is not adequate for a concurrent program. We need a concurrency-oriented verification test suites for

concurrent programs. In this work, we develop a novel mutation coverage metric for all concurrency constructs in SystemC.

One of our major requirements for implementation is to obtain a scalable and fast solution that can be seamlessly integrated with current design flows. Techniques based on translating SystemC models to an internal formal model are not scalable to complex real designs. Similarly for coverage work, since the number of mutations can be high, performance can be a problem. However, the more abstract the design description, the fewer the number of mutations. Hence, at SystemC TLM-level, we do not suffer from performance problems as seen in lower levels.

We performed experiments with multiple applications including industrial designs to validate the effectiveness of our verification and coverage environment. Our experimental results confirm the inadequacy of current verification test suites for checking concurrent features of SystemC. We also predicted and found errors on earlier tested systems.

The rest of the article is organized as follows. Section 2 gives an overview of previous work in SystemC verification and mutation testing; Section 3 details the background in SystemC. Sections 4, 5, and 6 detail how we have developed concurrency-oriented predictive verification techniques using partial-order traces and transaction-level assertions for SystemC, as well as our modifications to the SystemC kernel to accomplish these. We also describe the abstraction generation mechanism of computation slicing in Section 7. Section 8 demonstrates our predictive verification technique. Section 9 describes our mutation operators for concurrent SystemC functions and relates these operators to actual bugs. Our mutation coverage algorithm is described in Section 10. Experimental results are explained in Section 11. We conclude our article with a plan for future work and conclusions.

2. RELATED WORK

Formal verification of SystemC designs has been studied in Grosse and Drechsler [2003]. Due to the state space search, formal techniques are limited in practice. Furthermore, these techniques can only translate a subset of SystemC into an FSM model. In Vardi [2007], a summary of formal techniques in a SystemC context is presented. Blanc and Kroening [2010] present a static partial-order reduction technique for race analysis in SystemC with the use of a model checker. System Verilog Assertions have been used for SystemC descriptions [Habibi and Tahar 2004]. SystemC-specific transaction-level assertions have also been developed [Tabakov et al. 2008; Kasuya and Tesfaye 2007; Ecker et al. 2007]. We use these transaction level assertions in our work. There is a growing vendor tool support for SystemC modeling and verification.

Our work is similar to dynamic partial-order reduction (DPOR), in that both techniques are simulation-based and apply a single input to the design, whereas in model checking, all possible input combinations are applied. However, our work is also different from DPOR. In DPOR, the order of dependent transitions in the generated simulation trace is changed, leading to the generation of new simulation traces until all possible changes are exhausted. This may lead to state explosion problem for complex designs. However, we generate a partial-order trace from a single simulation trace and do not modify the order of dependent transitions in order to generate new traces. DPOR is orthogonal to our approach and can be used in conjunction, where DPOR can provide all partial-order traces for a given input, and our work can check the temporal properties on each partial-order trace efficiently. DPOR has also been applied to SystemC designs [Kundu et al. 2008; Helmstetter et al. 2009]. In Helmstetter et al. [2009], different SystemC schedules are generated to improve test coverage using DPOR. This is different from our verification work, since our goal is verification of temporal assertions.

Our ABV technique differs from traditional ABV techniques [Pierre and Ferro 2008] because ours is a predictive verification technique, where we can predict potential errors using error-free executions. Partial-order execution traces for multithreaded Java programs have been analyzed in Sen and Garg [2007], whereas we analyze partial-order traces for SystemC. We earlier considered multiple schedules for assertion-based verification of SystemC in Sen et al. [2008]. In this article, we extend this work with transaction-level assertions and add mutation testing-based coverage metrics for SystemC.

Using ABV for partial-order traces may lead to a state-explosion problem. We use computation slicing as an abstraction technique for ABV in this work. Earlier, we provided an application of computation slicing on concurrent and distributed systems and Java programs in Sen and Garg [2007]. As opposed to *program slicing* [Weiser 1982], where the user is interested in generating a projection of a program with respect to a set of interesting variables, computation slicing generates a projection of a partial-order execution trace with respect to temporal assertions.

Traditional coverage metrics have been used for RTL and gate-level designs. These techniques can be summarized as code coverage, structural coverage, functional coverage, and observability-based coverage techniques [Tasiran and Keutzer 2001; Fallah et al. 1998]. Code coverage techniques measure the amount of activation of lines, branches, and expressions in the source codes of designs during simulation. This is a limited approach, since activations cannot be observed at the outputs of the design, hence this approach may not have an impact on the design. Structural coverage techniques extract an abstract state machine from design descriptions and measure the number of states traversed during simulation. This approach is limited due to the growing size of the state machine for complex designs. Functional coverage metrics target design functionalities to cover interesting scenarios. However, these techniques are not automatic. They need to be redeveloped for new designs, since they rely on internal monitors defined and built by engineers having knowledge of both the design specification and implementation. Observability-based coverage metrics observe the impact of errors activated by the verification tests at the outputs of the design. All of the above techniques have been applied at the lower-level designs such as RTL and gate-level designs.

The mutation testing technique is an observability-based coverage technique based on software testing [Budd 1981; Offutt et al. 2006; Offutt and Untch 2001]. It has been applied to programming languages such as Java [Ma et al. 2005; Bradbury et al. 2006] and state machines [Fabbri et al. 1994]. Mutation testing is based on a given fault model. Mutations developed on the basis of this fault model are injected into the design one at a time, and it is checked whether verification tests activate and propagate such mutations to the outputs of the design. Mutation testing helps to strengthen the quality of verification tests iteratively until reaching a given target coverage. The previously developed stuck-at-fault model is similar to mutation testing, and has been very successful for manufacturing faults [Abramovici et al. 1990]. Fault models have also been used for test generation [Campenhout et al. 1998; Hsiao et al. 2000].

Typical mutation operators for a programming language are designed to modify variables and expressions by replacement, insertion, or deletion operators. For example, a simple nonconcurrent arithmetic operation mutation can change an assignment statement like $x = y + z$ into $x = y - z$, $x = y * z$, or $x = y/z$.

Several previous works have shown the effectiveness of mutation testing for assessing the quality of a test suite. Mutation testing is more powerful than statement or branch coverage, and all-use dataflow coverage criteria [Frankl et al. 1997; Walsh 1985; Li et al. 2009]. Similarly, in Andrews et al. [2005], it is shown that generated mutants are similar to real faults.

Mutation operators have been defined for concurrency constructs in Java [Bradbury et al. 2006; Farchi et al. 2003]. We make use of some of these operators in our work and enhance them for SystemC. Mutation testing has recently been applied to Verilog [Hampton and Petithomme 2007] and SystemC TLM 2.0 communication interfaces [Bombieri et al. 2008, 2009]. Our work differs from these in that we are concerned about all concurrency constructs in SystemC, rather than only TLM communication constructs. We do not modify TLM libraries, hence we can provide better integration into already developed industrial frameworks.

Mutation testing has been explored in the context of formal and assertion-based verification [Fummi and Pravadelli 2007; Kupferman et al. 2008; Tong et al. 2010] in reasoning about vacuity and coverage of assertions. In particular, the quality of formal specifications (assertions) is measured for RTL descriptions. Another approach would be to measure the number of assertions that have been executed, or what portions of assertions have been executed. This approach can quickly become infeasible for large designs.

In Helmstetter et al. [2006], the authors generate different SystemC schedules to improve test coverage. In particular, they generate nonequivalent (no two schedules have the same output) SystemC execution schedules using dynamic partial-order reductions. This allows them to explore all possible schedules with the same test suite. Our coverage work is different, in that we provide a coverage metric to determine the quality of the test suite; that is, for each schedule, we check whether the test suite can detect inserted mutations. Hence, their approach is orthogonal to our approach, and we can use it in conjunction with our mutation testing-based approach.

3. SYSTEMC BACKGROUND

SystemC models the concurrent activities of a system using processes. Processes can be combined into modules to create hierarchies. Process registration and module interconnection happen during the elaboration phase. Processes run concurrently, but code inside a process is sequential. There are method processes and thread processes. The SystemC scheduler controls the timing and order of process execution, handles event notifications, and manages updates to channels. It is an event-based simulator similar to VHDL. SystemC processes are nonpreemptive; hence, a process has to voluntarily yield control for another process to be executed. Threads are run exactly once by the kernel, and typically have a loop that keeps the thread alive for the required duration. The program-flow control remains with the thread until it yields explicitly. The thread then stays in a *wait* state until some event *triggers* it, and it resumes execution from the next statement after *wait*. A method executes atomically, and cannot yield to another process. The simulator regains control after the entire method has been executed.

Processes are triggered and synchronized with respect to sensitivity on events. An event keeps the list of processes that are sensitive to it and informs the scheduler which processes to trigger. Concretely, an event is used to represent a condition that may occur during the simulation. A SystemC event is the occurrence of an *sc_event* notification, and happens at a single point in time. An event has no duration or value. There are two types of sensitivity. Static sensitivity is defined before simulation starts, such as sensitivity to a clock signal, and dynamic sensitivity is defined after simulation starts and can be altered during simulation. Events are controlled via *wait*, *next_trigger*, and *notify* functions of the *sc_event* class. A *wait* function changes dynamic sensitivity of a thread process and suspends its execution. For example, $wait(SC\_ZERO\_TIME)$ delays the process by one delta cycle, a process waits on event e with $wait(e)$, and, with $wait(e1|e2|e3)$, a process waits on event $e1$, $e2$, or $e3$. Similarly, a *next_trigger* function changes the dynamic sensitivity of a method process. However, this function returns immediately rather than suspending execution.

Table I. SystemC Concurrency Functions

Construct	Available Functions
Event	*notify*, *wait*, *next_trigger*
Channel	*read*, *write*, *put*, *get*, *peek*, *nb_put*, *nb_get*, *nb_peek*, *b_transport*, *nb_transport_fw*, *nb_transport_bw*
Semaphore	*wait*, *trywait*, *post*
Mutex	*lock*, *trylock*, *unlock*

Events occur explicitly by using the *notify* function, and the scheduler resumes execution of a thread or method process by executing the *trigger* function. For example, *e.notify*() is called an immediate notification, since processes sensitive to event e will run in the current evaluation phase or delta cycle. Using $e.notify(SC\_ZERO\_TIME)$ processes sensitive to event e will run in the evaluation phase of the next delta cycle. Using $e.notify(t)$ processes sensitive to event e will run during the evaluation phase of some future simulation time.

Process synchronization also occurs with the usage of channels, interfaces, and ports. These constructs are the core of the Transaction-Level Model (TLM)-based methodology. Channel functions *read*, *write*, *b_transport*, *nb_transport_fw*, *nb_transport_bw*, *put*, *get*, *peek*, *nb_put*, *nb_get*, and *nb_peek* generate synchronization between processes. Synchronization is also established through instantiating *sc_semaphore* and *sc_mutex* objects, which provide *wait*, *trywait*, *post* and *lock*, *trylock*, *unlock* functions, respectively. Table I summarizes SystemC concurrency functions. Note that, in SystemC, communication between processes is established either by explicit concurrency functions or by shared variables.

The following shows the steps of the simulation scheduler in more detail.

(1) *Initialization.* All processes are made executable in an unspecified order.
(2) *Evaluate.* Select a ready-to-run process and resume its execution. This may result in more processes ready for execution in this same phase due to immediate notification. Signals and channels may invoke a request for update in the update phase.
(3) Repeat step 2 until no more processes are ready-to-run.
(4) *Update.* Execute all pending update requests due to calls made in step 2.
(5) If steps 2 or 4 resulted in delta event notifications, go back to step 2.
(6) If there are no more events, simulation is finished for the current time.
(7) Advance to the next simulation time that has pending events. If none, exit simulation.
(8) Go back to step 2.

Transaction-level models allow to specify a super-set of the realistic behaviors of the hardware using nondeterministic schedulings and loose timings. The set of behaviors of a model can change based on the effect of simulated time elapses. Helmstetter et al. [2009] state that a design can be faithfully modeled in TLM with fixed-time durations, without time durations, or with loose timing annotations. Each such model returns either a subset, a large super-set, or a super-set of realistic behaviors, respectively. Specifically, in modeling with fixed-time durations, SystemC wait functions are used with fixed durations, where fixed durations means that the value is given as a constant in the test scenario, and is the same for all executions. Duration values refer to the SystemC simulated time. In this article we are not concerned in developing faithful models at the SystemC level, rather we assume that the design is already modeled with fixed-time durations.

```
SC_MODULE (M1) {
  sc_event e;
  bool cs1, cs2;

  SC_CTOR(M1) {
    SC_THREAD(T1);
    SC_THREAD(T2);
    cs1 = false;     // internal action
    cs2 = true;      // internal action
  }

  void T1() {
    wait(e);         // receive action
    wait(10,SC_NS); // internal action
    cs1 = true;     // internal action
  }

  void T2() {
    e.notify();     // send action
    wait(10_SC_NS); // internal action
    cs2 = false;    // internal action
  }
};
```

Fig. 1. A SystemC design.

The SystemC scheduler is not preemptive; that is, a process runs without interruption until it explicitly gives control back with a wait statement. In Helmstetter and Ponsini [2008], the authors show that a nonpreemptive scheduler introduces implicit atomic sections (a wait-to-wait block in a process), hiding most of the issues regarding concurrent accesses to shared resources. Therefore, potential erroneous behaviors of the real system may not be revealed by SystemC simulation and formal methods complying with the SystemC simulation semantics. Hence, they compare nonpreemptive SystemC semantics with a schedulerless SystemC semantics, which corresponds to an asynchronous concurrency where interleaving in wait blocks is allowed.

3.1. SystemC Example

Figure 1 demonstrates a SystemC example with two threads T1 and T2. In this example, thread T1 is waiting for an event from thread T2 to make progress. Variables $cs1$ and $cs2$ are initially $false(f)$ and $true(t)$, respectively. There are three possible execution schedules by the kernel. These are (T1;T2;T1;TE;T2;T1), (T1;T2;T1;TE;T1;T2), and (T2;T1;TE;T2), where TE denotes a time-elapse. In the first and second schedules, threads run until completion and T1 is executed before T2. The choice of thread execution after a time elapse is different in both schedules. This leads to a state in the second schedule where both $cs1$ and $cs2$ can be true at the same time. In the third schedule, T2 is executed before T1 (T2;T1;TE;T2). In this case, the $notify$ message is lost, since T1 is not in a waiting state, and this leads to a special case of deadlock for T1. These three schedules lead to three total order traces, shown in Figure 2.

4. CONCURRENCY-ORIENTED PREDICTIVE ASSERTION VERIFICATION

The SystemC scheduler nondeterministically schedules ready-to-run processes and has an asynchronous interleaving semantics where scheduling of processes is nondeterministic. Scheduling of processes occurs at specific locations such as the *wait* function, or the end of a thread, but not inside the atomic wait-to-wait blocks. The nondeterministic thread scheduling may result in a discrepancy between the simulation and synthesis models. Although scheduling can be restricted with constructs such as

Vector Clock		Action
		Initially:
[1,0]	e_0	T1:cs1 = **false**;
[0,1]	f_0	T2:cs2 = **true**;
		Total Order Trace 1
		Schedule: (T1;T2;T1;TE;T2;T1)
		T1: **wait**(e);
[0,2]	f_1	T2: e.**notify**();
		T2: **wait**(10,SC_NS);
[2,2]	e_1	T1: trigger **wait**(e)
		T1: **wait**(10,SC_NS);
		Time Elapse
[0,3]	f_2	T2: trigger **wait**(10,SC_NS)
[0,4]	f_3	T2: cs2 = **false**;
[3,2]	e_2	T1: trigger **wait**(10,SC_NS)
[4,2]	e_3	T1: cs1 = **true**;
		Total Order Trace 2
		Schedule: (T1;T2;T1;TE;T1;T2)
		T1: **wait**(e);
[0,2]	f_1	T2: e.**notify**();
		T2: **wait**(10,SC_NS);
[2,2]	e_1	T1: trigger **wait**(e)
		T1: **wait**(10,SC_NS);
		Time Elapse
[3,2]	e_2	T1: trigger **wait**(10,SC_NS)
[4,2]	e_3	T1: cs1 = **true**;
[0,3]	f_2	T2: trigger **wait**(10,SC_NS)
[0,4]	f_3	T2: cs2 = **false**;
		Total Order Trace 3
		Schedule: (T2;T1;TE;T2)
[0,2]		T2: e.**notify**();
		T2: **wait**(10,SC_NS);
		T1: **wait**(e);
		Time Elapse
[0,3]		T2: trigger **wait**(10,SC_NS)
[0,4]		T2: cs2 = **false**;

Fig. 2. Different execution schedules corresponding to the design in Figure 1.

explicit events for the lower-level models, for high-level models such as TLM, designers often want the nondeterminism to model nondeterministic choices implicit in the design. Our goal is to explore multiple schedules to find schedule-related bugs. This is often done in practice by manually instrumenting the SystemC model with statements that attempt to randomize the schedule chosen by the SystemC scheduler. Whereas, we develop an automated technique that can efficiently analyze multiple schedules using a single schedule of a SystemC model, we accomplish this with our predictive verification framework.

Algorithm 1 displays an outline of our predictive assertion-based verification algorithm, which consists of tracing, execution, and verification phases. In the tracing phase, the user-given assertion (which includes SystemC variables) is parsed to automatically determine the relevant variables of interest (atomic propositions), so that we can trace changes in those variables of the design. We also add tracing for shared variables in this phase. In the execution phase, we generate a partial-order execution trace using our modified SystemC kernel. In the verification phase, we use

ALGORITHM 1: Predictive Assertion-Based Verification Algorithm

Input: a SystemC program P, an assertion f.
Output: assertion is satisfied or not.
tracing phase
1: read f and determine the relevant variables of interest;
2: automatically add tracing info for relevant variables; execution phase
3: compile and execute with the modified SystemC kernel;
4: generate a partial order trace;
verification phase
5: compute the slice wrt. the assertion using the trace;
recursively process f from inside to outside while applying temporal and boolean operators to compute slices
6: determine whether the assertion is satisfied or not;
if initial states of P and slice are different then return false and a counter-example else return true

computation slicing to determine whether the assertion is satisfied or not. We next describe the details of these phases, starting with the tracing and execution phases. However, we first give the technical background on the partial-order trace model and how it can be generated from SystemC programs.

4.1. Partial-Order Trace Model

We assume a system consisting of processes denoted by $P_1, \ldots, P_n$. Examples of processes are a node sitting on a PCI bus, a cache-in-a-cache coherence protocol. Processes execute actions. Actions on the same process are totally ordered. However, actions on different processes are only partially ordered. In this article, we relax the partial-order restriction on the set of actions and use directed graphs.

Given a directed graph G, let $\mathsf{V}(G)$ and $\mathsf{E}(G)$ denote the set of vertices and edges, respectively. We define a *consistent global state* (reachable global state) on directed graphs as a subset of vertices such that if the subset contains a vertex, then it contains all its incoming neighbors. Formally, C is a consistent global state of G, if $\forall e,\ f \in \mathsf{V}(G)$: $(e,\ f) \in \mathsf{E}(G) \wedge (f \in C) \Rightarrow (e \in C)$. The set of consistent global states of a directed graph forms a distributive lattice under the subset relation [Mittal and Garg 2001]. We say that a state D is *reachable* from a state C if $C \subseteq D$. In the rest of the article, unless otherwise stated, a global state or simply a state refers to a consistent global state. We model an *execution trace* (simulation trace or a *computation*) as a directed graph, denoted by $\langle E, \rightarrow \rangle$, with vertices as the set of actions E and edges as $\rightarrow$. We use action and vertex interchangeably. To limit our attention to only those consistent global states that can actually occur during an execution, we assume that the paths in $\langle E, \rightarrow \rangle$ contain at least the partial-order relation.

A permutation of all the actions $e_1, e_2, \ldots, e_r$ which does not violate the partial-order relation is called a consistent run (interleaving, schedule). Note that an interleaving is similar to a linearization of a partial order, and there are possibly an exponential number of interleavings of a computation.

Figure 3 parts (a) and (b) display a partial-order trace and its set of consistent global states reachable from the initial state $\{e_0,\ f_0\}$, respectively. For example, $V = \{e_3, e_2, e_1, e_0,\ f_1,\ f_0\}$ in Figure 3(b) is a consistent global state. Only the frontier of the global states is displayed in the figures, where the frontier is composed of actions in a global state that occurs last in process order (order of events on a given process/thread). For example, global state $V = \{e_3, e_2, e_1, e_0,\ f_1,\ f_0\}$ is denoted by the frontier $V = \{e_3,\ f_1\}$ instead. Observe that $C = \{e_1, e_0,\ f_0\}$ is not a consistent global state because $(f_1, e_1) \in \mathsf{E}(G) \wedge (e_1 \in C)$ but $(f_1 \notin C)$. Intuitively, this depicts a situation where T_1 has

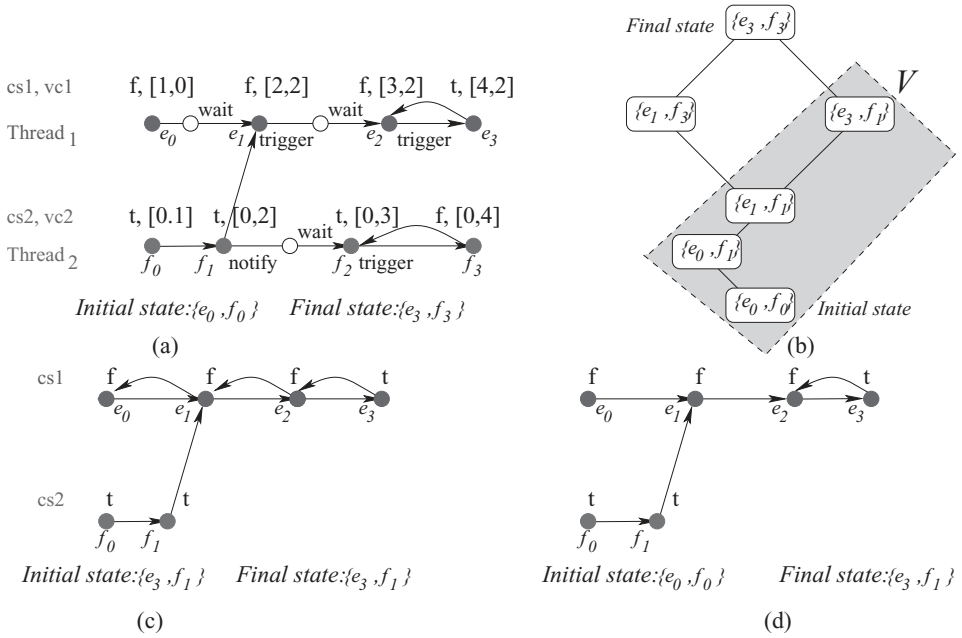

Fig. 3. (a) A partial-order trace corresponding to an execution of Figure 1; (b) its set of all consistent global states with 6 states; (c) its slice with respect to $cs1 = true \wedge cs2 = true$, the slice has state V; (d) its slice with respect to $possibly(cs1 = true \wedge cs2 = true)$, its states are enclosed in the shaded region

been notified by receiving a message from T_2; that is e_1, but T_2 has not yet sent the notification message. Similarly, $\{e_3, e_2, e_1, e_0, f_2, f_1, f_0\}$ is not a consistent global state as well, since f_3 has to be included in the state as well. Schedule (T1;T2;T1;TE;T2;T1) corresponding to Trace 1 in Figure 2 is obtained by the schedule $e_0, f_0, f_1, e_1, f_2, f_3, e_2, e_3$ of actions in the partial order. This schedule generates an interleaving sequence of states $\{e_0, f_0\}$, $\{e_0, f_1\}$, $\{e_1, f_1\}$, $\{e_1, f_3\}$, $\{e_3, f_3\}$ in Figure 3(b). The other interleaving sequence of states $\{e_0, f_0\}$, $\{e_0, f_1\}$, $\{e_1, f_1\}$, $\{e_3, f_1\}$, $\{e_3, f_3\}$ in Figure 3(b) corresponds to the total order Trace 2 in Figure 2. By using a partial-order representation, we are able to capture both of these schedules. We next describe how to generate this partial-order trace from a SystemC program execution.

4.2. Generating Partial-Order Traces with Vector Clocks

A physical timestamp cannot be used to generate partial-order traces from executions of concurrent designs, since physical timestamps are totally ordered. We use a vector of logical timestamps for this purpose. These logical timestamps allow us to track the concurrency information and the dependencies among the actions in an execution. A partial-order relation known as Lamport's happened-before relation [Lamport 1978] has been used for capturing dependencies in message-passing systems. A happened-before relation assumes that execution of a process (thread or method) can generate an internal action, a send action, or a receive action. Specifically, Lamport's *happened-before relation* is defined as the smallest transitive relation satisfying the following properties: (a) if actions e and f are generated by the same process, and e occurred before f in real time, then e happened-before f, and (b) if actions e and f correspond to the send and receive, respectively, of a message, then e happened-before f.

We use a mechanism known as *vector clocks* to represent the partial-order relation (the happened-before relation) described above. A vector clock assigns logical

timestamps to actions such that the partial-order relation between actions can be determined by using the timestamps.

Definition 4.1 (*Vector clock*). Given a computation G on n processes, a vector clock vc is a map from $\mathsf{V}(G)$ to $\mathcal{N}^n$ (vectors of natural numbers) such that $\forall e,\ f \in \mathsf{V}(G)\colon (e,\ f) \in \mathsf{E}(G) \iff e.vc \leq f.vc$, where $e.vc$ is the vector assigned to the element e.

Vector clock algorithms for *message-passing* systems exist in Garg [2002]. For example, given an n process program, for every process j, we assign a vector clock of size n denoted by vc_j. Initially, $vc_j[i] = 0$, for $i \neq j$, and $vc_j[j] = 1$. A process increments its own component of the vector clock only after a relevant action. A process includes a copy of its vector clock in every outgoing message. On receiving a message, it updates its vector clock by taking a component-wise maximum with the vector clock included in the message. Finally, if the action is relevant, then the action and its vector clock are output. For message-passing programs, all internal events that assign values to the atomic propositions in the assertion, as well as send and receive actions, are relevant.

Vector clock algorithms for *shared memory* systems also exist [Sen et al. 2003]. In this case, the happened-before relation assumes that execution of a process (thread or method) can also generate read or write actions on shared variables. In this case, for each shared variable x, there are two vector clocks v_x^a and v_x^w, denoted by access and write vector clocks, respectively. These are initially set to 0. A process updates its vector clock on reading a shared variable x by taking a component-wise maximum with the write vector clock of x. Then the access vector clock of x is updated by taking a component-wise maximum with that of the vector clock of the process. On writing a shared variable x, a process updates its vector clock by taking a component-wise maximum with the access vector clock of x. Then, the write and access vector clocks of x are set to the vector clock of the process. Finally, if the action is relevant, then the action and its vector clock pair are output. For shared memory programs, all write actions of shared variables that appear in the assertion to be verified are relevant. Note that this vector clock algorithm allows multiple consecutive reads of the same shared variable. It was proven in Garg [2002] and Sen et al. [2003] that the above-mentioned vector clock algorithms correctly implement causality.

Since a TLM design can use both shared variables and messages, our vector clock algorithm follows both of the above rules. Upon running the instrumented program, a separate log file is created for each process. Each log file consists of a sequence of (action, vector clock) pairs that a process generates. Furthermore, each such pair is also appended by the values of the variables that the action in the pair manipulates. Additionally, in order to capture the nonpreemptive SystemC scheduler with atomic sections, we modify the log files and update vector clocks in atomic sections by adding an edge from the last action in an atomic section to the first action. For example, thread $T1$ in Figure 1 has an atomic section that starts with triggering $wait(10, SC\_NS)$ and ends with $cs1 = true$, with corresponding actions e_2 and e_3, respectively. The corresponding partial-order trace in Figure 3(a) contains an edge from the last action e_3 to e_2. In the absence of such an edge, it would be possible to interleave the actions in an atomic section, leading to global states that are not possible with a nonpreemptive SystemC semantics, such as the global state $\{e_2, e_1, e_0,\ f_1,\ f_0\}$. In other words, if we ignored these edges for atomic sections, we would obtain a trace corresponding to the schedulerless semantics of SystemC, as described earlier. The log files from all processes are then combined to obtain a partial-order representation of the execution trace. Instead of using a log file, if every process sends its trace to a dedicated process which combines them during runtime, we can obtain a simple *online* verification environment. Currently, the instrumentation code for shared variables is added manually to the programs, however the instrumentation of messages is automatic, as described in the

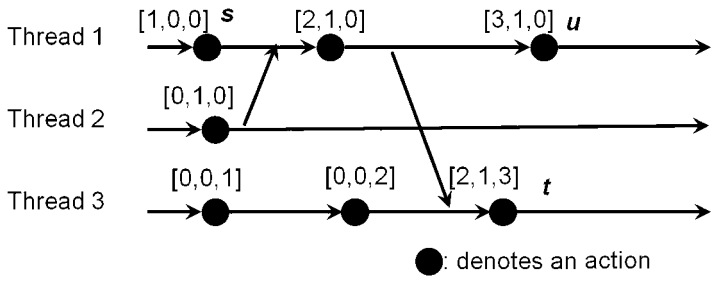

Fig. 4. Partial-order trace with vector clocks.

next section. Note that an approach similar to Helmstetter et al. [2009] can be used to automate instrumentation of shared variables and generation of partial-order traces for a given execution. We do not handle dynamic process creation in this article. However, vector clock algorithms can be extended such that the newly created process starts with the same vector clock of the process that creates it.

A sample execution of the vector clock algorithm is given in Figure 4, where the tuples in brackets represent the vector clocks. In the example, action s happened-before t since $[1, 0, 0] < [2, 1, 3]$, where $vc_i < vc_j$ if all elements of vc_i are less than or equal to the corresponding elements of vc_j and at least one element of vc_i is strictly less than the corresponding element of vc_j. Whereas u is *concurrent* with t, their vector clocks are not comparable, hence the program does not enforce any ordering between them. The reordering of concurrent actions can possibly lead to an exponential number of new schedules, or total orders. However, traditional simulators put an artificial ordering between these concurrent actions and generate only a single schedule. We present verification algorithms that can efficiently check assertions on partial-order traces that may encode an exponential number of schedules or total-order traces.

5. IMPLEMENTING VECTOR CLOCKS IN SYSTEMC KERNEL

Now that we know how to capture concurrency information as partial-order traces using vector clocks, we implement these in the SystemC kernel. Specifically, we add a new class, *sctv_vector_clock*, to implement vector clocks. During the elaboration phase of the simulation, the processes are registered, the total number of processes in the system is determined, and the appropriately-sized vector clocks are instantiated. Every process in the design has the extra data member consisting of the vector clock. Every send and receive is considered to be an action. By default, every evaluate-phase is registered as an internal action. Note that a single evaluate-phase can consist of multiple concurrent actions.

As we discussed above, the communication framework in SystemC is based around events. SystemC events are stateless and extremely lightweight. Execution of a *trigger* function (resulting after a wait function) or a channel read corresponds to a receive action in the partial-order trace. Similarly, execution of a *notify* function or a channel write (which later invokes notify) corresponds to a send action in the trace. It is convenient to have the sender information when a message is received. Hence, we add the sender id to every event. For a trigger action, the sender id is checked. If the sender id is invalid, we conclude that the action is an internal action (e.g., a timed wait). Since an event leads to an action in our partial-order trace model, as per the vector clock algorithm, the vector clock of the sending process is stored with the event. Note that, for channel writes, the events are initialized after the current process has finished executing. Hence, we store the required information in the *sc_prim_channel* class. This works for the primitive channels as well as any user-defined channels. Note

that notify and trigger are two fundamental actions that generate causal dependence in a SystemC model. Therefore we instrument these functions in the SystemC kernel. Blocking channel operations ultimately make use of these fundamental actions. Also, nonblocking operations in TLM 2.0 generate causal dependence when they use these fundamental actions. Finally, accesses to shared variables can also generate causal dependence, but these are invisible from the SystemC kernel, and hence can be dealt with by instrumenting the user code.

The dependencies between processes are dynamically generated as the program is executed. We treat *AND* and *OR* event lists separately. In particular, we receive the vector clock for each notifying process in the *AND* case, whereas the first notifying process's vector clock suffices in the *OR* case.

5.1. SystemC Partial-Order Trace Example

When the design in Figure 1 is compiled and executed with our modified kernel, a trace is generated that contains internal send and receive actions from every thread together with the values of relevant variables and vector clocks for every action. Trace 1 in Figure 2 corresponds to a generated total-order trace where the scheduling of the threads is (T1;T2;T1:TE;T2;T1). Note that there are three possible schedules (total-order traces) for this program, but the SystemC scheduler will generate only one. The partial order trace of the resulting execution is shown in Figure 3(a). This partial-order trace is obtained from Trace 1 using the vector clock information in Trace 1. In the partial-order trace, the time flows from left to right; the actions of threads are represented by circles. In each thread an action is labeled with an ordered tuple: the value of the respective local variable immediately after the action is executed and a vector clock. For example, the value of $cs1$ immediately after the T1 executing trigger action e_1 is false (f). The first action on each thread initializes the state of the thread. Figure 3(b) contains the set of all consistent global states of the computation reachable from the initial state $\{e_0, f_0\}$. Note that we do not generate actions in the partial-order trace in Figure 3(a) for the nonrelevant actions *wait* and *read*, but only for *notify*, *trigger*, and *write* actions. However, for ease of presentation, we show nonrelevant actions in the traces via empty circles. Also, note that edges e_3 to e_2 and f_3 to f_2 are added in order to preserve atomicity of wait-to-wait blocks.

We can also use shared variables for communication in TLM designs. Figure 5 displays a SystemC design and a partial-order trace where the scheduling of the threads is (T2;T1;T2). Note that edges e_2 to e_1 and f_3 to f_2 are added in order to preserve atomicity of wait-to-wait blocks.

Next, we describe the verification phase of our algorithm. In this phase, we first describe our transaction-level assertion language. Then, we describe how we can check assertions on partial-order traces using the computation slicing technique.

6. TRANSACTION-LEVEL ASSERTION LANGUAGE

For the assertion language, we use a variant of branching temporal logic, CTL, for SystemC and add transaction-level atomic propositions as described in Tabakov et al. [2008]. This logic allows us to express many alternative futures from every instant of time. Many futures are possible, since the state space of a partial-order trace is a lattice structure, as can be seen from the consistent global states in Figure 3(b). We use assertion, property, and specification interchangeably.

Atomic propositions of this logic refer to the variables in a SystemC program. We extend these with the following propositions from Tabakov et al. [2008]. The $func\_name.entry$ is true at the location immediately before the first executable statement in a function. Similarly, $func\_name.exit$ is true at the location immediately after

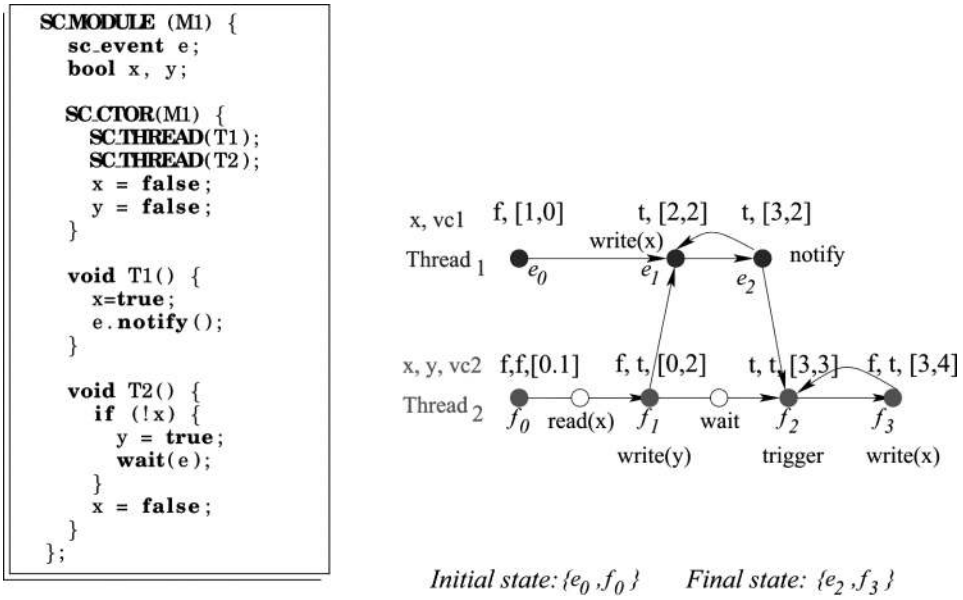

Fig. 5. A SystemC design with shared variable communication and a partial-order trace corresponding to an execution of it.

the last executable statement in a function. The $event\_name.notified$ is true whenever the SystemC kernel carries out the actual notification.

Basic temporal operators of this logic include $possibly$, $eventually$, $possibly-forever$, $always$, $some-nextstate$, and $all-nextstates$. We interpret properties in our logic over the global states of a computation. Given a global state, a property is evaluated with respect to the values of variables resulting after executing all actions in the state, that is, at the frontier of the state. Intuitively, we say that a global state C satisfies $always(p)$ (resp. $possibly-forever(p)$) if every state on all sequences of global states (resp. on some sequence of global states) from C to the final global state satisfies p. Similarly, we say that C satisfies $possibly(p)$ if there exists a global state that satisfies p on some sequence from C to the final state.

Our logic allows us to specify safety, liveness, and transaction properties that a SystemC program must satisfy. A *safety property* specifies that something bad will never happen, whereas a *liveness property* specifies that something good will eventually happen. For example, the safety property in a cache coherence protocol, "no two cache lines are in the modified state at the same time" can be specified as $always(\neg modified_i \vee \neg modified_j)$, where $i \neq j$. The liveness property in a bus protocol, "all requests for the bus are eventually acknowledged" can be specified as $always(request_i \Rightarrow eventually(ack_i))$. A *transaction property* specifies properties about transactions in SystemC. For example, "after an event is notified, a *memory_read* transaction starts" can be specified as $always(e.notified \Rightarrow all-nextstates(memory\_read))$. We can now define assertion-based verification for our partial-order trace-based framework.

Definition 6.1 (*Assertion-based verification*). Given a partial-order execution trace of a SystemC program and an assertion, the assertion-based verification problem is to decide whether the initial global state of the trace satisfies the assertion.

For example, the assertion $possibly(cs1 = true \wedge cs2 = true)$ is true for the trace in Figure 3(a). Since the trace has a global state V that satisfies $(cs1 = true \wedge cs2 = true)$ on some sequence from the initial state to the final state, as seen in Figure 3(b).

7. USING COMPUTATION SLICES FOR VERIFICATION

Verifying properties on a partial-order trace suffers from the state explosion problem. This is because a system with n processes and k actions on each process can have up to n^k number of global states. The concept of slicing is useful because it allows us to reason only on the part of the global state space that could potentially affect the assertion.

A computation slice (or a slice) is a concise representation of all those global states of the computation that satisfy an assertion. Formally the definition is as follows.

Definition 7.1 (*Slice*). A slice of a graph G with respect to an assertion p, denoted by $Slice[p]$, is the directed graph obtained from G by adding edges and removing vertices such that it only contains all the consistent global states of the graph that satisfy the assertion p .

For example, the slice of the trace in Figure 3(a) with respect to the assertion $(cs1 = true \wedge cs2 = true)$ is shown in Figure 3(c). The reader can verify that, indeed, this new trace contains the global state V of the trace in Figure 3(a) that satisfies the assertion. We explain in the next section how we obtained this slice.

Earlier we presented polynomial-time (in the number of processes) computation slicing algorithms for temporal assertions [Sen and Garg 2007]. In the next section, we describe slicing algorithm for temporal assertions of the form $possibly(p)$. In order to compute the slice with respect to an assertion, we recursively process the assertion from inside to outside while applying temporal and Boolean operators at each step to compute slices. Now we are ready to redefine Definition 6.1 in the light of slicing.

Definition 7.2 (*Assertion-based verification using slicing*). Given a partial-order execution trace and an assertion, the assertion-based verification using slicing is to decide whether the initial global state of the trace and the slice with respect to the assertion are the same.

We can use this definition for assertion-based verification because the slice contains all states of the trace that satisfy the assertion including, possibly, the initial state of the trace.

8. SYSTEMC PREDICTIVE ASSERTION VERIFICATION

In this section we describe how to use computation slicing for predictive assertion verification of SystemC designs. In particular, we show that by using partial-order traces and slicing, we can find errors from error-free total-order traces. We demonstrate predictive verification on the design shown in Figure 1. We are interested in checking an error condition where both threads can set their cs variables to true at the same time; that is, $possibly(cs1 = true \wedge cs2 = true)$. This assertion is composed of two local assertions $cs1 = true$ and $cs2 = true$, and we denote their conjunction $(cs1 = true \wedge cs2 = true)$ by p. The relevant variables of this assertion are $cs1$ from thread 1 and $cs2$ from thread 2. Our framework automatically adds tracing for these relevant variables to the design, as explained in Section 5. We earlier showed that executing this program with our modified SystemC kernel generates a total order Trace 1 with vector clock information, which we use to obtain the partial-order trace, as shown in Figure 3(a). Note that there are three possible schedules (total-order traces) for this program, but the SystemC scheduler will generate only one. Note also that the assertion is not satisfied (error is not found) in the total-order of Trace 1. To see this, we observe that the assertion is satisfied only at state $V = \{e_3, f_1\}$ in Figure 3(b). Trace 1

ALGORITHM 2: Algorithm for Generating $Slice[possibly(p)]$

Input: a partial order trace T, an assertion $possibly(p)$.
Output: $Slice[possibly(p)]$.
1: compute $Slice[p]$, where $p = p_1 \wedge p_2 \ldots \wedge p_n$ is a conjunctive assertion;
2: $S1 = T$;
3: add an edge from the successor of an action where $p_1, p_2, \ldots, p_n$ is false back to the action in $S1$;
4: find the largest reachable state V that satisfies p using $Slice[p]$;
5: $S2 = T$;
6: remove vertices and edges after state V in $S2$;
7: output $S2$;

corresponds to the sequence of states $\{e_0, f_0\}$, $\{e_0, f_1\}$, $\{e_0, f_2\}$, $\{e_0, f_3\}$, $\{e_1, f_3\}$, $\{e_2, f_3\}$, $\{e_3, f_3\}$ in Figure 3(b) and does not contain the V state. However, the partial-order trace clearly contains V and satisfies the assertion (finds the error). We now show how we use slicing to automate this.

Without computation slicing, we are forced to examine all global states of the partial-order trace, six in total, to decide whether the trace satisfies the assertion. Instead, we compute the slice of the trace with respect to the assertion $possibly(p)$ and apply Definition 7.2. We show the algorithm that computes $Slice[possibly(p)]$ in Algorithm 2. The algorithm has polynomial-time complexity $O(|p| \cdot n^2|E|)$, where $|p|$ is the number of Boolean and temporal operators in p, n is the number of processes, and $|E|$ is the number of edges. Polynomial-time algorithms for other assertions and their proof of correctness can be found in Sen and Garg [2007].

To obtain the slice with respect to the nontemporal part p, we add an edge from the successor of an action where the local assertion p_1 or p_2 or ..., p_n is false back to the action in the partial-order trace, thereby increasing the number of incoming neighbors of that action. This process eliminates the action from taking part in any consistent global state. For example, action e_0 does not satisfy the local assertion $cs1 = true$, therefore the global states that contain e_0, that is, $\{e_0, f_0\}$, and $\{e_0, f_1\}$ in Figure 3(b), do not satisfy the local assertion. After the addition of the edges (e_1, e_0), (e_2, e_1) in Figure 3(c), none of these states belong to the states of the slice. After addition of these edges, from the definition of a consistent state, any state that includes e_0, e_1, or e_2 must include e_3, where the local assertion is satisfied. Similarly, we remove vertex f_3, since f_3 it does not satisfy local assertion $cs2 = true$. Also, since f_3 and f_2 belong to the same component (atomic section), f_2 has to be removed as well. As a result, from the six consistent global states in Figure 3(b), this exercise eliminates five – retaining state V, which satisfies $cs1 = true \wedge cs2 = true$.

To obtain the slice with respect to the temporal part $possibly(p)$, we use the slice with respect to the nontemporal part p. Specifically, the slice with respect to $possibly(p)$ includes all states up to the largest reachable state that satisfies p. Since, from the definition of $possibly(p)$, every global state of the trace that can reach the largest reachable state that satisfies p also satisfies $possibly(p)$. Using the slice in Figure 3(c), we can obtain the largest reachable state that satisfies p, which is denoted by V. All of the states enclosed in the dashed ellipse in Figure 3(b) can reach V, hence they all satisfy $possibly(p)$ and should be part of the slice. Therefore, in order to generate the slice, we first find the largest reachable state that satisfies p using the slice with respect to p. We have earlier shown that such a state exists. Then, the slice with respect to $possibly(p)$ is the same as the partial-order trace, except none of the vertices and edges after state V is included in the slice. We show the slice with respect to $possibly(p)$ in Figure 3(d).

Finally, we use Definition 7.2 and check whether the initial state of the trace is the same as the initial state of the slice. If the answer is yes, then the assertion is satisfied; otherwise a counter-example is returned. We find that the assertion $possibly(p)$ is satisfied, since the the initial state of the slice with respect to $possibly(p)$, which is $\{e_0, f_0\}$, is the same as the initial state of the partial-order trace. Hence, we can detect this error. However, the assertion is not satisfied (error is not detected) in the total-order Trace 1, which was what the user observed. Indeed, this error could have been found only if the user had observed another total-order trace, Trace 2, which is encoded in the partial-order trace in our case. Hence, our framework is able to find potential errors from error-free total-order traces (from Trace 1).

A strength of the slicing approach is that the slice is computed efficiently by adding edges and removing nodes (without traversing the state space) and represented concisely (without explicit representation of individual states) as a trace. Another strength of our approach is that once the partial-order trace is obtained, the design is never re-executed in order to find other possible traces (Trace 2 and Trace 3); rather our algorithms work on the partial-order trace itself by adding edges or removing vertices. This is different from traditional assertion-based verifiers.

For each simulation of the design in the above predictive verification approach, we need a verification test, be it random or directed or otherwise, that guides the design exploration. Even if predictive assertion verification is successfully completed, there is still a doubt whether verification tests are comprehensive and if they cover all possible behaviors of the system. Increasingly, there is a need to measure the quality of verification efforts. Coverage metrics play an important role in evaluating the confidence in the verification results. In the next section, we develop mutation testing-based coverage metrics in order to determine the quality of the verification tests.

9. MUTATION OPERATORS FOR CONCURRENT SYSTEMC

In this section, we first identify typical bugs that designers make when using concurrency. Then, we develop mutation operators for concurrent SystemC functions, listed in Table I, and relate them to these bugs.

Note that there is work in the literature on bug patterns due to mutation of C++ constructs such as variable mutation, operator mutation, constant mutation, and statement mutation. Though such mutations may also be useful, we are only concerned with the mutations due to SystemC constructs. A tool such as [CERTITUDE 2010] can be used for other cases. It is also important to check the coverage of the assertions used in the verification, since the designer may not have verified all potential behavior of the design with the given set of assertions. There is work on coverage techniques for assertions [Kupferman et al. 2008; Tong et al. 2010] which can be used in conjunction with our work.

The following list of bug patterns is based on resources such as Java concurrency bug patterns [Bradbury et al. 2006; Farchi et al. 2003], patterns for TLM 2.0 communication functions [Bombieri et al. 2008], and our experience.

B1. *Lost notify.* If a *notify* is executed before a corresponding *wait* is executed, the $notify$ has no effect for the process that will start waiting and is lost. As a result, a process executing a *wait* might not be awakened because it is waiting for a *notify* that occurred before the *wait* was executed. This may hold for timed notifications as well as immediate notifications.

B2. *Interference (Data race).* Two or more concurrent threads access a shared variable and at least one access is a write.

B3. *Deadlock.* Two or more processes are unable to proceed due to waiting for one another. Also, we say that a process is deadlocked when it is stuck as in schedule (T2;T1;TE;T2) in Figure 2.
B4. *Starvation.* A process may starve due to actions of other processes. A change in lock acquisition may lead to this.
B5. *Resource exhaustion.* A group of processes holds all of a finite number of resources. One of them needs additional resources but no other process gives up.
B6. *Incorrect count initialization.* This occurs when number of entries in a semaphore initialization is incorrect.
B7. *Nondeterminism.* If an immediate notification is used, this may cause nondeterminism.
B8. *Forgetting functions.* Forgetting to call a *put* before a *get* function.
B9. *Incorrect functions.* Using *read* instead of *write*, or using blocking instead of a nonblocking function.

We now present a set of mutation operators designed to exercise concurrency and synchronization present in SystemC programs. We describe operators in two categories. In the first category, mutations modify parameters of concurrency functions. In the second category, mutations remove, replace, or exchange concurrency functions. We also relate these mutation operators to real bug patterns just described.
Category 1. Modify parameters of concurrency function.

M1. *Modify Function Time.* This operator can be applied to functions with a time parameter such as $wait$, $notify$, and $next\_trigger$. For example, we can modify $wait(time)$ to $wait(time/2)$. This modification may result in an interference or data race bug B2, since a process may access a shared variable when it is not supposed to, due to a potential change in process activation time or order. This mutation can also lead to a lost notify B1 and deadlock B3 if the notification is sent before the corresponding wait.
M2. *Modify Concurrency Construct Count.* This operator can be applied to semaphores that indicate the number of threads that can access a shared resource. This mutation leads to incorrect count initialization B6, and may also lead to resource exhaustion B5 if the count is decremented. For example, modify $sc\_semaphore(num)$ to $sc\_semaphore(num-1)$ or $sc\_semaphore(num+1)$.

Category 2. Remove, replace, and exchange concurrency function.

M3. *Remove Concurrency Construct.* This operator removes calls to concurrency functions described in Table I. This mutation results in a B8 bug. Removing *wait* or *lock* may result in interference B2, since a thread that should have been waiting can potentially access a shared variable. Removing $notify$ may result in lost notify bug B1 and deadlock B3. Removing an *unlock* may result in the process waiting for the lock to be starved, hence B4.
M4. *Replace Timed Construct with Untimed Construct.* This operator replaces timed construct with an untimed construct, and vice versa. For example, we can replace timed notify, with immediate notify which may result in nondeterminism B9. Also, a $wait(e)$ can be replaced by $wait(1, SC\_NS)$, which may result in interference B9, since the process may access a shared variable before waiting for the appropriate event notification.
M5. *Exchange Lock/Permit Acquisition.* This operator exchanges a function of a semaphore or mutex with another one. In a semaphore, *wait* or $trywait$ can be used to acquire permits to a shared resource. Exchanging one function with another may lead to timing changes and starvation. For example, using $trywait$ instead of *wait* may lead to starvation, B4, since in the first case threads do not block,

whereas threads block in the second case. Also, this can result in an incorrect function, B9.

M6. *Exchange Function Call with Another.* This operator exchanges a function in Table I with another appropriate function. For example, a call to semaphore *post* is exchanged with *wait* or *notify* exchanged with *wait*. Also, a call to *get* may be exchanged with an appropriate *peek*. This may result in starvation B4 as in B5; interference B2 since a shared variable may be accessed; deadlock B3 since instead of releasing a lock it may request creating a circular chain of lock requests; exhaustion B5 since resources may have been received from the channel by *get*, lost notify B1, or incorrect function B9.

M7. *Exchange one Concurrency Construct Instance with Another.* When there is more than one lock (mutex or semaphore), we replace a call to a lock with another one. This may lead to a deadlock situation B3, or interference B2, since the correct lock is not used to access critical regions, and may also lead to an incorrect usage of functions B9.

We have described a complete set of mutation operators and how they relate to real bug patterns. Next, we are going to present our mutation coverage algorithm.

10. MUTATION COVERAGE ALGORITHM

We define a *mutant* P' as the introduction of a mutation operator into a program P. We then define what it means to kill a mutant.

Definition 10.1 (*Killing mutant*). Given a mutant P' for a program P and a test t, t is said to *kill* P' if and only if the output of t on P' is different from the output of t on P.

A verification test is expected to kill each mutant with at least one test case. In case a mutant cannot be killed, the tester needs to show that (1) output of t on P' is the same as the output of t on P; or (2) update tests by adding a test case that kills the mutant. We define the ratio of the number of mutants killed to the number of all mutants as *mutation coverage*. This allows us to determine the coverage of verification tests.

Note that in case both the mutant and the original program can generate multiple schedules, a concurrent mutation coverage definition can be given, where we can check output differences of all possible schedules. However, generating multiple schedules of a design can result in huge overheads in execution. Similarly, a SystemC program may not be terminating, leading to many schedules. Hence, such a concurrent mutation coverage may not be useful.

Algorithm 3 displays our automatic mutation coverage algorithm for SystemC. In line 1, for every concurrency function in the original program, we insert a mutation operator, as described above, and obtain a *metaprogram*. The metaprogram uses a mutant schema where each inserted mutation is guarded by a conditional statement that can be switched on and off at runtime. This metaprogram is more efficient than generating a new version of the program for each mutation. Note that not all mutation operators are applicable to every program. In line 3, we enable the inserted mutation operators one by one, obtaining a mutant for each enabling. Then, in line 4, we choose every test in the test suite one by one for simulation. It is possible that some tests may not execute the inserted mutation, so we do not consider these tests during simulation. This check can be done by an initial execution of all tests on the metaprogram. In line 5, we simulate the metaprogram. In line 6, we check if the test kills the mutant using Definition 10.1. Finally, in line 8, we generate mutation coverage.

The complexity of algorithm can reach $M \times V \times T$, where M is the number of inserted mutations, V is the number of verification tests and T is the cost of simulation of

Table II. Outputs of Mutant Example in Figure 6

Test1 Outputs(cs1cs2)		Test2 Outputs(cs1cs2)	
Original (P)	Mutant (P')	Original (P)	Mutant (P')
tf	tf	ft, ff, tf	ft, tt, tf

ALGORITHM 3: Mutation Coverage Algorithm

Input: a SystemC program P, a set of verification tests V.
Output: mutation coverage.
1: insert all "relevant" mutation operators into P;
2: **for** each inserted mutation operator M in P **do**
3: enable M and obtain a mutant P';
4: **for** each verification test $v \in V$ **do**
5: simulate P' with v;
6: check if v kills P';
7: **end for**
8: **end for**
9: generate mutation coverage;

one test. Since the number of mutations can be high, performance can be a problem. However, the more abstract the design description, the fewer the number of mutations. Hence, at SystemC TLM level, we have relatively fewer numbers of mutations as, can be seen from experimental results.

10.1. SystemC Mutation Coverage Example

We apply a mutation operator that removes the concurrency construct *wait* and obtain the mutant in Figure 6. Assume that the test suite only contained Test1, which generates only the final values of $cs1, cs2$, as can be seen from the Test1 columns in Table II. We denote values that are generated before the thread ends or deadlocks as final values. Consider schedule (T1;T2;T1;TE;T2:T1) for the original program and schedule (T1;T2;TE;T1:T2) for the mutant. This mutation is not killed by Test1 because the output of the original program (tf) is the same as the output of the mutant (tf). The designer needs to improve the coverage by adding a new test.

One possible way to improve the quality of the test suite (and kill the mutant) is to generate values of $cs1, cs2$ at all value changes, rather than just at the final state, starting from their initial values ft. We call this test Test2. From Table II, it is clear that the mutant output ft, tt, tf is different from the output of the original program ft, ff, tf, hence the mutant is killed. Another possible way to improve the quality of the test suite would be to use an assertion to check whether $cs1, cs2$ can be true at the same time.

11. EXPERIMENTS

We have performed experiments on concurrent SystemC designs using our predictive verification and mutation testing-based coverage frameworks to validate the effectiveness of our concurrency-oriented approaches. As experimental testbeds, we chose five designs from OSCI SystemC and TLM 2.0 library distributions [OSCI 2010]; two designs from SystemC Runtime Verification Toolbox (SCRV) [SCRV 2009]; and two industrial designs described below. Each design contains its own verification tests.

11.1. SystemC Verification Experiments

In verification of each design, we executed Algorithm 1.

```
SC_MODULE (M1) {
  sc_event e;
  bool cs1, cs2;

  SC_CTOR(M1) {
    SC_THREAD(T1);
    SC_THREAD(T2);
    cs1 = false;     // internal action
    cs2 = true;      // internal action
  }

  void T1() {
    // removed wait
    // wait(e);      // receive action
    wait(10,SC_NS); // internal action
    cs1 = true;   // internal action
  }

  void T2() {
    e.notify();   // send action
    wait(10,SC_NS); // internal action
    cs2 = false;  // internal action
  }
};
```

Fig. 6. A SystemC mutant example.

The first industrial design framework ($ind1$) is a modeling library based on SystemC, used for architectural exploration, RTL development, constrained random verification, and early software development. It includes a complete set of BFM and monitor components for several bus protocols, including proprietary TLM compliant bus protocols. Also included is a new testbench environment built on the existing hardware modeling library, that includes controllers through which all interaction with the device under verification takes place. Some of the controllers are data stimulus, clock, signal, bus, fifo, and memory. The framework consists of around 38,000 lines of SystemC code. The experiments were executed with up to 12 threads. We used 21 testbenches that were used during the original validation. We used several transaction-level assertions (34 in total) to describe high-level communication and synchronization between the controllers and peripherals. Several assertions were violated. One such violation was for the property "After a notification of a start event by the bus controller, the next notification is a test event, - $always(start-event.notified \Rightarrow all-nextstates(test-event.notified))$". This property is useful to demonstrate the correct ordering of communication events. Another property that was violated stated that "Every access request by a core for the bus is eventually granted". This is useful for checking starvation freedom. We checked the complement of this property, which is $\bigvee_{i\in 0\ldots(n-1)}(possibly(request_i \wedge possibly-forever(\neg granted_i)))$, for each core i. The errors could only be generated for an alternative schedule of threads rather than the observed schedule, hence a traditional assertion-based verifier that checks only the observed simulation schedule could not detect this error. The average slowdown on simulation using our kernel versus the unmodified kernel was 7%.

The second industrial framework ($ind2$) is an SOC with DSP cores, cryptographic accelerators, and crossbar. The code base was around 100,000 lines of SystemC code. There were 57 threads maximum during the experiments. We used 6 testbenches that were used during the original validation of the design. We also manually inserted errors and checked several temporal assertions. One of these errors resulted in cache coherency violation, where a cache state was in modified state in more than one cache. The property is $possibly(modified_i \wedge modified_j)$, where i and j are cache identifiers. We caught these errors via our predictive verification framework, whereas errors were not

Table III. Mutation Testing-based Coverage Metric Experiments

Design	Lines	# Mutants	Time (ms)	Coverage (%)
pv_example	2449	18	200	66
p2p_pipe_thread	926	36	240	55
scx_mutex_w_policy	161	30	280	76
*at*1*_phase*	2350	34	240	56
*ind*1	38,080	146	440240	52
pvt_put_example	1161	55	400	41
bozo	139	16	170	56
sirac	253	28	230	60

found by a traditional assertion-based verifier, since the observed simulation did not have the error. Our initial experiments demonstrate a 12% slowdown on simulation versus the unmodified kernel. Designs with a higher number of processes have a higher slowdown, as expected.

We also performed experiments for designs *pvt_put_example*, *bozo*, and *sirac*, and the design in Figure 1, where we added simple temporal assertions to check for mutual exclusion of shared resources. All designs had schedules that violated the assertions. For the example in Figure 1, assertion $possibly(cs1 = true \wedge cs2 = true)$ was violated. Similarly, for the example in Figure 5, we checked the assertion $always((y = false \wedge some - nextstate(y = true)) \Rightarrow (x = false \wedge some - nextstate(x = false))$, which states that "when y becomes true, x must stay false". This assertion was not violated. Since these designs are relatively small with a fewer number of threads, the average slowdown on simulation using our kernel versus the unmodified kernel was around 3%.

11.2. SystemC Coverage Experiments

For each design, we executed Algorithm 3 and displayed mutation coverage together with the mutations killed and not killed as feedback.

Table III shows our results. In the table, we denote the number of lines in the design in the column *Lines*, the time it took to complete the experiment in column $Time$, and the number of generated mutants in column *Mutants*. Finally, the column denoted $Coverage$ represents the mutation coverage percentage. Our experimental results can be summarized as follows.

(1) Low mutation coverage percentages (less than 60% for most designs) confirm the inadequacy of test suites to find many possible design errors, since mutations are closely related to actual errors, as we described.
(2) For each design, execution of our mutation algorithm for all mutants took less than 400ms to complete, except for ind1, where it took 440240ms (close to 5x execution time slowdown), since it is a large design and has a high number of testbenches. However, our approach lends itself to simple parallelization, and we expect to reduce these execution times substantially for large designs when we implement parallelism.
(3) The relatively few number of generated mutants shows that by focusing solely on concurrency functions at TLM, we do not suffer from an explosion in the number of mutants. The industrial design especially used very few concurrency constructs.

Similar to other coverage measures, a target coverage percentage can be provided by the user. Empirical industrial data has shown that mutation coverage over 80% could be the initial target [Bailey 2009]. The user can iteratively improve the test suite by adding new test cases until the target coverage is reached. For example, in the case of the industrial design, although the coverage was 52%, we increased it to 83% after the addition of a new test that activates concurrency constructs by bus transactions.

12. CONCLUSIONS AND FUTURE WORK

The emergence of concurrent multicore designs requires concurrency-oriented design automation techniques. We have presented a concurrency-oriented approach for verification and coverage of system-level designs using SystemC. Unlike formal verification techniques, our solutions are scalable and fast, and can be seamlessly integrated with current design flows. In verification work, we have used a combination of partial-order traces, computation slicing, and transaction-level assertions to develop a concurrency-oriented predictive verification technique. We have demonstrated that our technique can predict potential errors from error-free executions. We not only analyze more schedules than a traditional assertion-based verifier, but we can also do this with very little cost, thanks to our efficient slicing algorithms. We have extended SystemC kernel and added transaction-level assertion support for this purpose. Our experimental results on industrial designs validate the effectiveness of predictive verification.

In coverage work, we developed a comprehensive list of the mutation operators for concurrency and synchronization-related features of SystemC and showed the effectiveness of these operators by relating them to actual bug patterns. Such a comprehensive list had not been developed before. Our coverage metric allows us to adequately generate coverage for concurrent SystemC programs and, ultimately, improve the quality of test suites. Our experimental results confirm the inadequacy of current verification test suites for checking concurrent features of SystemC and demonstrate the effectiveness of mutation testing-based coverage metrics.

In the future, we are planning to extend our verification framework to predict common error types, such as race conditions, and investigate efficient vector clock algorithms to further reduce overhead. Our mutation framework can also be used to optimize the test suite by removing redundant test cases that kill the same set of mutants. Also, a parallel version of our work will substantially improve performance for large designs.

ACKNOWLEDGMENTS

We thank the anonymous referees, whose comments helped us improve the article.

REFERENCES

ABRAMOVICI, M., BREUER, M. A., AND FRIEDMAN, A. D. 1990. *Digital Systems Testing and Testable Design*. Computer Science Press, New York.

ANDREWS, J. H., BRIAND, L. C., AND LABICHE, Y. 2005. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. 402– 411.

BAILEY, B. 2009. Can mutation analysis help fix our broken coverage metrics? In *Proceedings of the 4th International Haifa Verification Conference (HVC '08)*. 5–5.

BLANC, N. AND KROENING, D. 2010. Race analysis for systemc using model checking. *ACM Trans. Des. Autom, Electron. Syst. 15,* 3, 1–32.

BOMBIERI, N., FUMMI, F., AND PRAVADELLI, G. 2008. A mutation model for the SystemC TLM2.0 communication interfaces. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE)*. ACM, New York, 396–401.

BOMBIERI, N., FUMMI, F., PRAVADELLI, G., HAMPTON, M., AND LETOMBE, F. 2009. Functional qualification of TLM verification. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE)*. ACM, New York, 190–195.

BRADBURY, J., CORDY, J., AND DINGEL, J. 2006. Mutation operators for Concurrent Java (J2SE 5.0). In *Workshop on Mutation Analysis, 2006*. 11.

BUDD, T. A. 1981. Mutation analysis: Ideas, examples, problems and prospects. In *Computer Program Testing*, North-Holland, Amsterdam, 129–148.

CAMPENHOUT, D. V., AL-ASAAD, H., HAYES, J. P., MUDGE, T., AND BROWN, R. B. 1998. High-level design verification of microprocessors via error modeling. *ACM Trans. Des. Autom. Electron. Syst, 3,* 4, 581–599.

CERTITUDE. 2010. Springsoft, Certitude website. http://www.springsoft.com/.

ECKER, W., ESEN, V., STEININGER, T., VELTEN, M., AND HULL, M. 2007. Implementation of a transaction level assertion framework in SystemC. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE)*. ACM, New York, 1–6.

FALLAH, F., DEVADAS, S., AND KEUTZER, K. 1998. OCCOM: Efficient computation of observability-based code coverage metrics for functional verification. In *Proceedings of the Design Automation Conference (DAC)*. ACM, New York,152–157.

FARCHI, E., NIR, Y., AND UR, S. 2003. Concurrent bug patterns and how to test them. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*.

FOSTER, H. D., KROLNIK, A. C., AND LACEY, D. J. 2004. *Assertion-Based Design* 2nd Ed., Springer, Berlin.

FRANKL, P. G., WEISS, S. N., AND HU, C. 1997. All-uses vs mutation testing: An experimental comparison of effectiveness. *J. Syst. Softw. 38,* 3, 235–253.

FUMMI, F. AND PRAVADELLI, G. 2007. Too few or too many properties? Measure it by ATPG! *J. Electron. Testing 23,* 5, 373–388.

GARG, V. K. 2002. *Elements of Distributed Computing*. Wiley, New York.

GHENASSIA, F. 2005. *Transaction-Level Modeling with SystemC*. Springer, Berlin.

GROSSE, D. AND DRECHSLER, R. 2003. Formal verification of LTL formulas for SystemC designs. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*. 245–248.

HABIBI, A. AND TAHAR, S. 2004. On the extension of SystemC by SystemVerilog assertions. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*. 1869–1872.

HAMPTON, M. AND PETITHOMME, S. 2007. Leveraging a commercial mutation analysis tool for research. In *Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques (MUTATION'07)*. 203–209.

HELMSTETTER, C., MARANINCHI, F., AND MAILLET-CONTOZ, L. 2009. Full simulation coverage for SystemC transaction-level models of systems-on-a-chip. *Formal Methods Syst, Des. 35,* 2, 152–189.

HELMSTETTER, C., MARANINCHI, F., MAILLET-CONTOZ, L., AND MOY, M. 2006. Automatic generation of schedulings for improving the test coverage of systems-on-a-chip. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. 171–178.

HELMSTETTER, C. AND PONSINI, O. 2008. A comparison of two SystemC/TLM semantics for formal verification. In *Proceedings of the International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. 59–68.

HSIAO, M. S., RUDNICK, E. M., AND PATEL, J. H. 2000. Dynamic state traversal for sequential circuit test generation. *ACM Trans. Des. Autom. Electron. Syst. 5,* 3, 548–565.

KASUYA, A. AND TESFAYE, T. 2007. Verification methodologies in a TLM-to-RTL design flow. In *Proceedings of the Design Automation Conference (DAC)*. 199–204.

KUNDU, S., GANAI, M., AND GUPTA, R. 2008. Partial order reduction for scalable testing of SystemC TLM designs. In *Proceedings of the 45th Annual Design Automation Conference (DAC'08)*. 936–941.

KUPFERMAN, O., LI, W., AND SESHIA, S. A. 2008. A theory of mutations with applications to vacuity, coverage, and fault tolerance. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. 1–9.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM 21,* 7, 558–565.

LI, N., PRAPHAMONTRIPONG, U., AND OFFUTT, J. 2009. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *Proceedings of the IEEE International Conference on Software Testing Verification and Validation Workshop*. IEEE, Los Alamitos, CA, 220–229.

MA, Y.-S., OFFUTT, J., AND KWON, Y. R. 2005. MuJava: An automated class mutation system: Research articles. *Softw. Test. Verification Reliability 15,* 2, 97–133.

MITTAL, N. AND GARG, V. K. 2001. Computation slicing: Techniques and theory. In *Proceedings of the Symposium on Distributed Computing (DISC)*. 78–92.

OFFUTT, J., AMMANN, P., AND LIU, L. 2006. Mutation testing implements grammar-based testing. In *Proceedings of the 2nd Workshop on Mutation Analysis*. 12.

OFFUTT, J. AND UNTCH, R. H. 2001. *Mutation 2000: Uniting the Orthogonal*. Kluwer, Amsterdam.

OSCI 2010. Open SystemC initiative, http://www.systemc.org/.

PINTO FERRAZ FABBRI, S. C., DELAMARO, M., MALDONADO, J., AND MASIERO, P. 1994. Mutation analysis testing for finite state machines. In *Proceedings of the 5th International Symposium on Software Reliability Engineering*. 220–229.

PIERRE, L. AND FERRO, L. 2008. A tractable and fast method for monitoring SystemC TLM specifications. *IEEE Trans. Computers 57,* 10, 1346–1356.

SCRV 2009. SystemC runtime verification toolbox (SCRV). http://mytrac.assembla.com/scrv/wiki.

SEN, A. AND GARG, V. K. 2007. Formal verification of simulation traces using computation slicing. *IEEE Trans. Computers 56,* 4, 511–527.

SEN, A., OGALE, V., AND ABADIR, M. S. 2008. Predictive runtime verification of multi-processor SoCs in SystemC. In *Proceedings of the Design Automation Conference (DAC)*. 948–953.

SEN, K., ROSU, G., AND AGHA, G. 2003. Runtime safety analysis of multithreaded programs. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE).*

TABAKOV, D., VARDI, M. Y., KAMHI, G., AND SINGERMAN, E. 2008. A temporal language for SystemC. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. 1–9.

TASIRAN, S. AND KEUTZER, K. 2001. Coverage metrics for functional validation of hardware designs. *IEEE Des. Test Comput. 18,* 4, 36 –45.

TONG, J. G., BOULE, M., AND ZILIC, Z. 2010. Defining and providing coverage for assertion-based dynamic verification. *J. Electron. Testing 26,* 2, 211–225.

VARDI, M. Y. 2007. Formal techniques for SystemC verification; position paper. In *Proceedings of the Design Automation Conference (DAC)*. 188–192.

WALSH, P. J. 1985. A measure of test case completeness (software, engineering). Ph.D. dissertation, State University of New York at Binghamton.

WEISER, M. 1982. Programmers use slices when debugging. *Comm. ACM 25,* 7, 446–452.

Received July 2010; revised January 2011; accepted February 2011