Open access • Proceedings Article • DOI:10.1109/MEMCOD.2010.5558640

Monitoring temporal SystemC properties — Source link

Deian Tabakov, Moshe Y. Vardi

Published on: 26 Jul 2010 - Formal Methods

Topics: SystemC, Transaction-level modeling, Specification language and Formal verification

Monitoring Temporal SystemC Properties

Deian Tabakov
6100 Main Str. MS-132
Houston, TX 77005
dtabakov@cs.rice.edu

Moshe Y. Vardi
6100 Main Str. MS-132
Houston, TX 77005
vardi@cs.rice.edu

***Abstract*—Monitoring temporal SystemC properties is crucial for the validation of functional and transaction-level models, yet the current SystemC standard provides no support for temporal specifications. In this work we describe a temporal monitoring framework for the SystemC specification language defined by Tabakov et al. at FMCAD'08. Our framework uses a very minimal modification of the SystemC kernel, exposing event notifications and simulation phases. The user code is instrumented to allow observation of the relevant parts of the model state. As proof of concept, we use the framework to specify and check properties of two SystemC models. We show that monitoring SystemC properties using this framework has reasonable overhead (0.01% – 1%) and has decreasing marginal cost. Finally, we demonstrate that monitoring at different levels of abstraction requires very small changes to the specification and the generated monitors. Based on our empirical results we argue that the additional expressive powers and flexibility of the framework does not incur a serious performance hit.**

I. Introduction

SystemC[1] has become a de facto industry-wide standard modeling language less than a decade after its first release. In addition, SystemC is a simulation environment that creates the appearance of concurrent execution, even though in reality the processes are run sequentially. The core language is built as a library extending C++ and provides macros for modeling the fundamental components of hardware and hybrid systems, for example, modules and channels. The object-oriented encapsulation of C++ and its inheritance capabilities help make designs modular, which in turn makes IP transfer and reuse easier [5]. Various libraries provide further functionality, for example, SystemC's Transaction-Level Modeling (TLM) library defines structures and protocols that streamline the development of high-level models.

One of the strengths of SystemC is that it can handle different models of computation and communication, levels of abstraction, and system design methodologies. This is achieved by a layered approach where high-level constructs share an efficient simulation engine [9]. At the base layer, SystemC provides an event-driven *simulation kernel*. User-defined *modules* and *ports* represent structural information, and *interfaces* and *channels* abstract communication. The behavior of a module is specified by defining one or more *processes*. Each process can declare a *sensitivity list*: a number of *events* that trigger its execution. For a detailed description of the semantics of SystemC we direct the reader to [9], [19].

Work partly supported by a gift from Intel.

[1]IEEE Standard 1666-2005

From a high level point of view the execution of a SystemC model alternates between user code and the kernel, and the events provide a bridge between the two. During the execution of the user code a process may request an event to be *notified*, either immediately, at the end of the delta cycle[2], or after some time period. Immediate event notifications take effect immediately, while delta-delayed and time-delayed event notifications take effect during the execution of the kernel code. Event notifications determine which user processes are eligible to run next. A *waiting* process becomes *runnable* when one or more of the events on its sensitivity list has been notified. If there are several processes that are runnable, the kernel arbitrarily selects one of them and gives it execution control. The simulation semantics imposes *non-preemptive execution* of processes, that is, once the kernel gives a process execution control the kernel cannot take it back until the process terminates or explicitly suspends itself by calling `wait()`.

The growing popularity of SystemC has motivated research efforts aimed at the verification of SystemC models using *assertion-based verification* (ABV) – an essential method for validation of hardware and hybrid models [6]. With ABV, the designer asserts properties that capture the design intent in a formal language, e.g., PSL[3] [8] or SVA[4] [20]. The model then can be verified against the properties using dynamic or formal verification techniques.

A successful ABV solution requires two components: a formal declarative language for expressing properties, and a mechanism for checking that the model under verification (MUV) satisfies the properties. Most ABV efforts for SystemC so far have been focused on *dynamic verification* (also called *testing* and *simulation*). This approach involves executing the MUV in some environment, while running checkers in parallel with the model. The checkers typically monitor the inputs to the MUV and ensure that the behavior or the output is consistent with asserted properties [9]. The complementary approach of *formal verification* produces a mathematical proof that the MUV correctly implements the asserted properties. In case a violation of the property is detected, both methods are able to returns a counterexample, which is a trace that violates

[2]In SystemC, like in VHDL and in Verilog, a delta cycle represents an infinitesimal amount of time that separates events occurring in successive simulation cycles but at the same simulation time. This mechanism allows the (sequential) simulation of concurrent execution. See [9] for further details.

[3]IEEE Standard 1850-2007

[4]IEEE Standard 1800-2005

the property. Our focus in this paper is on dynamic verification.

There have been several attempts to develop a formal declarative language for expressing SystemC properties by adapting existing temporal languages (see [19] for a detailed discussion). Tabakov et al. [19] point out three major deficiencies in existing temporal languages for SystemC: 1) lack of definition of an execution trace, 2) lack of flexibility to handle modeling at different abstraction levels, and 3) failure to take advantage of well-known and widely used primitives for software specification. Tabakov et al. propose a precise definition of SystemC traces, which captures the alternation between the user code and the kernel. They also define a systematic way for enriching existing specification languages with a set of Boolean properties, which, together with existing clock-sampling mechanisms in PSL and SVA, allow the sampling of the execution trace with flexible temporal and transactional resolution. Their overall framework enables the specification of properties at different levels of abstraction.

The traditional approaches to dynamic verification involve connecting a separate checker module in parallel with the MUV for each property to be checked; see, e.g., [2], [9]. The difficulty of applying this approach to SystemC is that it only allows us to monitor the state of the model when control is passed to the checker module. Thus, this approach cannot apply to monitor properties that refer to finer temporal resolution, e.g., referring to a particular event notification or the end of a delta cycle. A key conclusion of the proposal in [19] is that certain nominal information about the kernel, specifically, kernel phases and event notification, has to be exposed to the monitors.

Once it becomes clear that the SystemC kernel needs to be exposed, the two key questions are how to do it with small changes to an existing implementation, and how to avoid performance penalties. Optimizing for performance alone would require direct modification of the existing source code to hook the new functionality to the existing data structures. This would require our framework to be rewritten for each SystemC implementation, limiting portability. On the other hand, optimizing for portability would require adding a layer of indirection that abstracts away the concrete implementation of the SystemC kernel, which slows down the execution. Since each optimization affects inversely the other, the challenge is to find a good balance between the two. We found an approach that accomplishes both small change and low performance overhead. We modularized the necessary changes to the SystemC code to make it easy and fast to modify existing installations. Our framework can easily adapt to changes in the SystemC semantics that may be added in future releases.

In this paper we show that monitoring SystemC properties using this framework has reasonable overhead (0.05% – 1% per monitor) and that the marginal cost of additional monitors decreases. As proof of concept, we use the framework to specify and check properties of two SystemC models. We introduce only a few new lines of code to the existing source of the SystemC kernel and all necessary additional functionality is encapsulated in two new objects. The user code is instrumented to allow the observation of the relevant components. Based on our empirical results we argue that the additional expressive powers and flexibility of the framework does not incur a prohibitive performance hit.

II. Motivation

A. Exposing the simulation semantics

Tabakov et al. argue in [19] that the simulation semantics of SystemC should be exposed as a part of the system state. Specifically, they propose exposing the phase of the simulation kernel and event notification. It is not immediately clear why the state of the kernel needs to be exposed. For example, in the work of Kroening and Sharygina [14] the kernel is abstracted away completely. Each process is modeled as a labeled transition system, and the global system is defined as a product of these local transition systems. The transitions of the global system are defined according to the simulation semantics, which requires that "components must synchronize on shared actions and proceed independently on local actions" [14]. Under this model synchronization occurs when a process encounters a `wait()` or a `notify()` instruction. The observable behavior of their abstraction of execution matches the execution of a SystemC model. Thus, on the surface, it may seem that taking into account the state of the kernel would only complicate the semantics.

Similar philosophy has been adopted by Karlsson et al. [12], Ecker et al. [7], and Pierre and Ferro [17], which likewise do not model the kernel. This may sound reasonable at first, but one soon realizes that many important properties require some knowledge of the state of the kernel. A consistency property may be required to hold all the times, at the evaluation-phase boundary, at the delta-cycle boundary, or at a timed-cycle boundary. If the kernel is abstracted away completely, then there is no way to make these distinctions and specify the consistency requirement properly. Tabakov et al. conclude that the state of the kernel must be exposed to a certain extent, in order to enable the user to specify properties at different temporal resolutions. Moy et al. [15], [16] also proposed exposing information from the kernel, but their abstraction is motivated by the types of properties they want to check while Tabakov et al. argue that the abstraction should expose fully SystemC's simulation semantics, as described in [1]. A coarse abstraction might hide details that may be of importance to some users. Thus, an abstraction at the level of the simulation semantics is as generic as possible, enabling further abstraction if required by specific applications.

In addition to proposing the exposure of the kernel phases, in [19] Tabakov et al. argue in favor of exposing the notification of events. SystemC events are objects derived from the pre-defined class `sc_event`. A particular "waiting" process does not becomes "runnable" until the event on which the process is waiting is *notified*. For example, if a TLM channel is full, a thread that wishes to write to the channel may suspend itself by calling `wait(ok_to_put)`. As soon as there is free space on the channel, the channel notifies the `ok_to_put` event, and the waiting thread is moved to the

pool of "runnable" processes among which the kernel selects the next process to run. Most core SystemC objects have an associated event that indicates that some change has occurred. For example, an `sc_signal` has an event that is notified when the signal changes; an `sc_fifo` has an event for writing to and an event for reading from the channel; an `sc_clock`'s positive and negative edges are represented by events. Thus, events are the fundamental synchronization mechanism in SystemC, and keeping track of when a particular event is notified allows us to pinpoint the instant in time when something important happens. In the particular example mentioned before, we might want to specify that every time `ok_to_put` is notified the number of items in the channel is strictly smaller than the capacity of the channel.

Event notifications can be requested in the user model by calling the `notify()` method of class `sc_event`. The actual moment when the event is notified is determined by the kernel depending on the type of each event notification, the status of the other processes in the model, and the kernel phase. There are three types of event notification:

1) `notify()` with no arguments: immediate notification. Notification happens upon execution.
2) `notify(SC_ZERO_TIME)`: delta notification. Notification is postponed until the delta-notification phase.
3) `notify(`*`time`*`)` with a non-zero time argument: timed notification. Notification happens during a subsequent timed-notification phase.

Pending event notifications can be canceled using the `cancel()` method, pending timed notifications are canceled by delta notifications, and pending delta notifications are canceled by immediate notifications.

Tabakov et al. argue that the fundamental role played by events in the execution of SystemC models justifies fully exposing event notification in the simulation state. This means that properties can refer directly to event notification, for example, specifying that `ok_to_put` is notified at least once every clock cycle.

One might argue that keeping track of all phases of the simulation semantics and all event notifications is unnecessary because very few properties relate to those specific phases or notifications. Our guiding philosophy in developing the verification framework described here is to expose all event notifications and all phases of the kernel that are described by the simulation semantics and identified by Tabakov et al. in [19], rather than try to pass judgment on which ones are the most important. The users can use coarser abstractions if needed. Since we need to anticipate all possible uses of SystemC specifications, exposing the semantic fully is the most justifiable approach.

Below we present two SystemC models whose properties cannot be expressed (and monitored) without reference to kernel phases and events. The same two models and the temporal properties described here are also used to evaluate empirically the performance of our proof-of-concept implementation[5].

B. Squaring via addition

The first SystemC model implements a squaring function by using repeated incrementing by 1. The system consists of an Adder module that implements addition $a+b$ by triggering b copies of `add_1()` process, each of which adds 1 to a.

We use a delta-delayed notification of a `driver_event` to suspend the driver and allow the `add_1()` processes to initialize. Then the driver uses immediate notification of an `add1_activate_event` to activate the `add_1()` processes, which then proceed to execute sequentially within the same delta cycle. At the end of execution, each `add_1` process (immediate–) notifies an `addition_event`. Thus, the result $a+b$ is calculated within two delta cycles and using b (immediate) notifications of the `addition_event`.

The Adder is embedded inside a Squarer module, which implements c^2 by repeatedly calculating the sum of c and `running_total`. The squarer waits until the next clock cycle before calculating the next addition. It takes c clock cycles to complete the calculation.

This simple model (∼100 lines of code) is intentionally inefficient. Notice that it is driven by all three types of event notifications (immediate, delta-delayed, timed). It also allows us to vary the size of the model by varying the number of processes. A small piece of code illustrating the functionality of the Adder is presented below.

```
void
adder::driver() {
  while(true) {
    //Suspending until values on the inputs change
    wait(input1.value_changed_event() |
                    input2.value_changed_event());
    int i1 = input1.read();
    int i2 = input2.read();
    _a = i1;
    for (int i=0; i < i2; i++) {
      sc_spawn( sc_bind(&adder::do_add1, this) );
    }
    // Allow the do_add1() processes to initialize
    // by suspending until the next delta-cycle
    driver_event.notify(SC_ZERO_TIME);
    wait(driver_event);
    add1_activate_event.notify();  // immed. notifc.
    // Suspend for a delta-cycle to allow all
    // computations to complete
    driver_event.notify(SC_ZERO_TIME);
    wait(driver_event);
    result.write(_a);
  }
}
void
adder::do_add1() {
  wait(add1_activate_event);
  (_a) = (_a) + 1;
  addition_event.notify(); // immediate notification
}
```

A correct implementation of the Adder must satisfy the following property (using the syntax of [19]):

[5]The source code of the models is online at http://www.cs.rice.edu/∼vardi/papers/memocode10.tar.bz2

```
ALWAYS (adder.add1_activate_event.notified &
adder._a > 0) -> ((adder.addition_event.notified ->
(adder._a > 0)) UNTIL ``result.write(_a)'')
```
(1)

i.e., if $a > 0$ at `add1_activate_event`, then $a > 0$ at every instance when the `addition_event` is notified until the result is pushed to the output wire. Sampling at such low temporal resolution is not possible under the current SystemC standard. The best we can do is to check the value of a before the `driver_event` is notified, and to check it again when the result is being written to the wire. There is no mechanism to check an assertion at particular event notifications, so the intermediate steps cannot be verified.

Another property of the Adder is

```
ALWAYS (adder.input.read($1) & adder.input.read($2))
-> ((within [2 MON_DELTA_CYCLE_END]
adder.output.write($3)) & ($3 = $1 + $2))
```
(2)

i.e., the correct result is always returned within 2 delta cycles of receiving the inputs (a and b). This property also cannot be monitored using the current SystemC standard. A monitoring process can count the delta cycles in which it is triggered, but there might be delta cycles when the monitor is not triggered, and the monitor would not be able to count those. Thus, there is no way of determining that precisely two delta cycles have passed.

These problems stem from the way the SystemC kernel is designed. The kernel makes the *effect* of event notifications visible only to the processes waiting for those events. While this is sufficient for simulation purposes, it makes monitoring some important properties impossible. What is missing is a mechanism for alerting monitors immediately after an event and for alerting monitors that a delta cycle is about to start or end. However, event notification (and delta cycle determination) is done by the kernel and is not exposed to the rest of the system. The solution to this issue is to expose some of the internal state of the kernel for *monitor-only* privileged access.

C. Airline reservation system

The second SystemC model implements a system for reserving and purchasing airplane tickets. The users of the system submit requests by specifying the starting and the ending airports of the trip, the dates of travel, and a few other pieces of data. The system uses a randomly generated flight database to find a direct flight or a sequence of up to three connecting flights. Those are returned back to the user for approval. If the user would like to purchase a ticket she submits payment information that is processed and stored, and the individual legs of the trip are booked.

Internally the system uses several modules connected by finite-capacity channels. Each module has a fixed amount of local memory, implemented as bounded queues, to store the requests that are currently pending processing or are waiting to be sent to another module. The modules use events to synchronize reading and writing to the internal memory. All modules except the I/O modules are connected to the same (slow) clock, and the I/O modules are connected to another (faster) clock. This allows us to stress-test the behavior of the system when there are more requests than it can process. This model is intended to run forever. It approximates actual subsystems currently used in hardware design. A piece of code from the `flight_planner` module is presented below.

```
/**
 * Receives requests from the master module. Adds
 * new requests to the new_planning_requests queue
 * if there is space available, otherwise blocks
 * until space becomes available.
 */
void flight_planner::receive_new_requests() {
  while(true) {
    tlm_tag_t t;
    if (! in_from_master->nb_can_get( &t )) {
      tlm_tag_t tag;
      wait(in_from_master->ok_to_get(&tag));
    }
    token_t* req = new token_t();
    in_from_master->nb_get(req);
    if (req->get_payload() == PLAN) {
      unsigned int curr_size =
                        new_planning_requests.size();
      if (curr_size >= queue_size) {
        wait(new_requests_nonfull);
      }
      new_planning_requests.push(req->get_request());
      new_requests_nonempty.notify(SC_ZERO_TIME);
      wait();
    } // it was a new request
    else {
      handle_special_request(req->get_request(),
                                req->get_payload());
    }
  } // receiving loop
} //receive_new_requests()
```

One safety property of the system is that whenever some process notifies the event `new_requests_nonfull`, the corresponding queue (`new_planning_requests`) must have capacity for storing at least one request. Formally,

```
ALWAYS (new_requests_nonfull.notified ->
(new_planning_requests.size() < capacity))
```
(3)

Notice that placing this assertion at the locations where the event notification is requested may lead to false negatives. Even if the assertion fails in that location, a subsequent process may *cancel* the event notification and the property would still be satisfied. This example demonstrates further the need for sampling at event notifications.

In order to meet performance objectives, the system must propagate each request through each channel (or through each module) within 5 cycles of the slow clock. This property is a conjunction of 16 bounded liveness assertions similar to the two shown below.

```
...
// Propagate through module within 5 clock ticks
ALWAYS (io_module_receive_transaction($1) ->
 ( within [5 slow_clock.pos()]
   io_module_send_to_master($2) & ($1 == $2)
 ) AND ...
...
// Propagate through channel within 5 clock ticks
ALWAYS (master_send_to_flight_planner($1) ->
 ( within [5 slow_clock.pos()]
```

```
   flight_planner_receive_from_master($2)
     & ($1 == $2)
 )
) AND ...                                          (4)
```

Monitoring Property (4) requires a process that is aware of the slow clock and can be triggered from multiple processes in a non-deterministic sequence. Implementing a monitor of this type using the existing SystemC kernel would require major instrumentation of the model in order to store and propagate the required information. A more scalable and easier to use approach is to allow the creation of monitors that are accessible by all processes and at the same time have access to the kernel's internal information. The framework presented here solves these and many other problems to allow monitoring of important and previously untestable properties of SystemC models.

III. Related Work

The SystemC verification standard (SCV) [10], proposed by the SystemC Verification Working Group, provides APIs for "data introspection, transaction recording, and randomization." SCV does not address the issue of temporal specifications, which is the focus of this paper. Thus, SCV and our framework are complementary.

Several groups have proposed modifying the standard SystemC kernel in order to expose race conditions that may occur under alternative schedulings, cf. [3], [11], [18].

Braun et al. [4] evaluate different strategies for checking temporal specification properties in a SystemC model. They consider two fundamentally different approaches: 1) an add-on library (a collection of SystemC objects) that implements functions for checking temporal properties, and 2) an interface module that connects the SystemC model with an external test-bench environment (in particular, TestBuilder). The properties are limited to Finite LTL properties and the temporal resolution is fixed to the resolution of the simulation clock.

A number of proprietary specification languages for SystemC come with a monitoring framework. One of the more serious industrial efforts is by Kasuya and Tesfaye (Jeda Technologies) [13]. This work provides a set of primitives to express cycle-accurate and TLM-based temporal primitives, but no mechanism for adapting to different levels of abstraction.

We believe that sampling at the boundaries of clock cycles [4], [13] is inadequate for SystemC, because it fails to take into account the unique simulation semantics of SystemC, which allows for a much finer grained temporal resolution. For example, algorithmic-level SystemC models are often timeless, with the simulation being completely driven by events and the simulation clock making no progress during the whole simulation [9], [15]. In fact, the whole simulation can consist of a single delta cycle, if the simulation is driven solely by immediate event notifications. Thus, clock-cycle-level temporal resolution is clearly inappropriate for such models.

IV. Execution Traces

An execution trace is the sequence of states traversed by the SystemC model. Tabakov et al. propose a precise definition of a state of the model, which encompasses the state of the SystemC kernel, the state of the user model, and the state of the external libraries. Here we summarize their discussion.

Kernel State: Tabakov et al. abstract the simulation semantics of SystemC by a state machine consisting of 15 states, for example, "evaluation phase", "update phase", "running process", "updating a channel", etc. Their abstraction follows precisely the phases of the kernel described in the standard [1], and thus the abstraction is applicable to all standard-compliant SystemC implementations. They introduce a new variable, **kernel_phase**, whose value keeps track of the current kernel phase.

Tabakov et al. propose introducing for each event a Boolean proposition `event_name.`**`notified`**, which is true whenever the kernel carries out the actual notification of event `event_name`. Note that both delta-delayed and time-delayed notification requests can be subsequently canceled, therefore a call to `event_name.notify()` with a non-negative argument does not guarantee that the proposition `event_name.`**`notified`** will be true in the future.

User Model State: Tabakov et al. take the perspective of "white-box validation", which means that the state of the model should be fully exposed. The state of the user model is the full state of the C++ code of all processes in the model, which includes the values of the variables, the location counter, and the call stack. This approach allows the verification engineer to refer explicitly to statements being executed both via their label as well as by their syntax. Furthermore, Tabakov et al. propose exposing the values of the formal parameters of all functions upon invocation and their return values upon return. For each function they provide pre-defined labels (**entry**, **exit**, **call**, and **return**) corresponding to the location immediately before the first statement, the location after the last statement, the location before the function call, and the location immediately after the function call. Finally, Tabakov et al. expose the status of each process (one of *waiting*, *runnable*, or *running*).

Library Code State: Tabakov et al. treat library code as a black box. The state of library objects is exposed without exposing implementation details. Furthermore, the state of a library is exposed only in terms of the API of that library.

Trace: A SystemC trace is a sequence of states corresponding to the execution of the model. Such execution consists of an alternation of control between the kernel on one hand, and the model and the libraries on the other hand. When the kernel is executing, the trace follows the transitions between the kernel phases. When the kernel selects a process to run or a channel to update, control passes to that process, which then runs until it terminates or is suspended via a `wait()` function call. With respect to transitions of processes, the trace follows the "large-step semantics" approach [21]. Under this approach the focus is only on the overall effect of each statement, as opposed to considering the individual subexpressions. For example, `y = x++;` consists of two subexpressions (`y = x;` and `x = x + 1;`), but the trace ignores the valuations of the variables during the execution of the subexpressions.

By following large-step semantics our framework may miss rare cases where a property is violated in a subexpression. For

example, suppose that a program invariant requires that x must always be positive, and suppose that $x = 1$. During the execution of the expression `y = (x--) + (x++);` the value of x is temporarily set to 0 by the `x--` subexpression, but since the value of x is restored back to 1 by the `x++` subexpression, no violation of the property will be reported. Modern design practices discourage the use of complex subexpressions that change the valuation of variables, therefore the choice of large-step semantics over small-step semantics is justified.

Finally, Tabakov et al. consider each invocation of a library method, for example, invoking a channel-interface method, to return in one step. This is consistent with their black-box view of libraries.

PROPERTIES: Existing property-specification languages like PSL and SVA allow the use of "clock expressions" (CE), which are Boolean expressions that indicate when a state in the execution trace should be sampled. For example, sampling the execution trace at the end of every clock cycle can be done by using the expression (**`kernel_phase`** = `MON_TIMED_NOTIFICATION_END`) as CE. Finer grained resolution is possible by sampling at the end of execution of each process, using (**`kernel_phase`** = `MON_NEXT_PROC_SELECT`) as CE. One can even sample at the boundary of the individual statements in the source code (which is the default sampling rate).

V. MODIFICATIONS OF THE KERNEL

Our first goal is to introduce minimal changes to the reference implementation in order to expose the actions of the kernel in a systematic way[6], while the behavior of the kernel (and thus the simulation semantics) remain unchanged.We expose only those steps described by the SystemC standard. Any implementation that follows the standard can be modified in a similar way.

One way to expose the state of the kernel is to implement an API that returns the current phase of execution of the kernel and relevant data, and another way is to modify the kernel to send updates about its execution. Notice that in the first case it is not clear how the monitors will be alerted when the kernel reaches a particular sample point. Busy waiting of the monitor will not allow other code (including kernel code) to execute, and using multi-threading inside the simulation does not guarantee that the monitor's (OS-level, as opposed to SystemC-level) thread will be active while the kernel is in a particular phase. On the other hand, if the kernel sends updates (via function calls), the monitor will be triggered and will execute as soon as the relevant sample point is reached. This is the mechanism that we chose to implement.

One immediate problem is that this approach requires the kernel to have access to all monitors. While it is conceivable to add the necessary data structures to the existing code, it would require extensive modifications. Our intention was to add as few new lines of code as possible so that our framework can be applied to a wide range of SystemC implementations. To that end, we encapsulated all additional functionality in a new object, `observer`, and connected the existing code to it via callbacks. `observer` stores references to the monitors, receives updates of the kernel state, and then notifies the monitors that need to execute at the current sample point (Fig. 1). The observer implements a callback for each phase of execution of the kernel. A the code below illustrates the main idea.

```
enum sample_point {
  ...
  MON_DELTA_CYCLE_END,
  ...
};
class mon_observer {
 public:

  void delta_cycle_end() {
    unsigned int num_elements =
                  arr_mon_sets[MON_DELTA_CYCLE_END]->size();
    if (num_elements > 0) {
      monitor_set::const_iterator it;
      for (it = arr_mon_sets[MON_DELTA_CYCLE_END]->begin();
           it != arr_mon_sets[MON_DELTA_CYCLE_END]->end();
           it++) {
        mon_prototype* mp = *it;
        mp->callback_delta_cycle_end();
      }
    }
  }
  void register_monitor(mon_prototype* mon, sc_event* eve) {
    if (events_to_monitor_sets[eve] == 0) {
      std::set<mon_prototype*>* n =
                     new std::set<mon_prototype*>();
      eve->register_observer(this);
      n->insert(mon);
      events_to_monitor_sets[eve] = n;
    }
    else {
      (events_to_monitor_sets[eve])->insert(mon);
    }
  }
  void event_notified(sc_event* event) {
    monitor_set::const_iterator it;
    for (it = events_to_monitor_sets[event]->begin();
         it != events_to_monitor_sets[event]->end();
         it++) {
      mon_prototype* mp = *it;
      mp->callback_event_notified(event);
    }
  }
} // class observer
```

The kernel source code is then modified to call the `observer` callback functions at the locations where a change of phase occurs. (In the OSCI reference implementation, the particular file that is modified is `sc_simcontext.cpp`.) For example, here is a snippet of actual code (from `sc_simcontext.cpp`) with our modification:

```
while ( true ) {
  // EVALUATE PHASE
  m_execution_phase = phase_evaluate;

  // New line of code added below
  if (observer != 0) { observer->evaluate_begin(); }

  while( true ) {
    // execute method processes
    sc_method_handle method_h = pop_runnable_method();
    ...
```

The communication between the observer and the monitors

[6]Please find the source code with our modifications at http://www.cs.rice.edu/∼vardi/papers/memocode10.tar.bz2

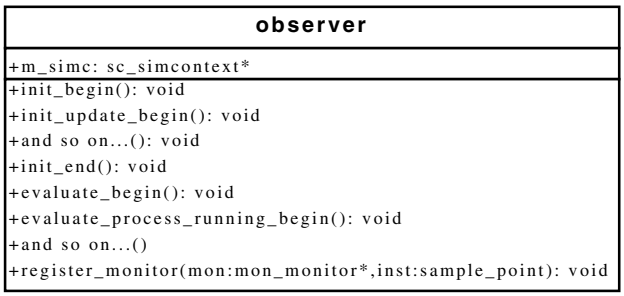

Fig. 1. Partial class diagram for `observer`

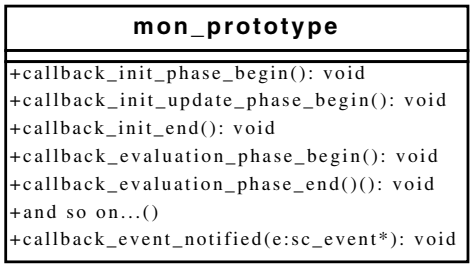

Fig. 2. Partial class diagram for `mon_prototype`

is also via callbacks. However, that also raises another programming issue. The observer needs a guarantee that each monitor implements the appropriate callback function, otherwise we will have a compilation error. To resolve that issue we defined another object, `mon_prototype`, that serves as a base class for all monitors (Fig. 2). This class declares a `virtual` callback function for each type of sample point on the execution trace, for example, `virtual callback_init_phase_begin()` and `virtual callback_evaluate_phase_begin()`. Each monitor implements the callback functions that are relevant to its execution and that implementation overwrites the `virtual` implementation defined in `mon_prototype`.

Monitors request to be notified by issuing a call to the `observer`'s `register_monitor()` function. For example, a monitor for the Adder might use `register_monitor(this, MON_DELTA_CYCLE_BEGIN)` in order to be alerted at the start of each delta cycle. For each kernel phase, `observer` maintains a list of monitors that have requested to be alerted when the kernel reaches that particular phase. As soon as the kernel notifies (via a function call) the `observer` that the kernel is entering another phase, the `observer` calls the callback function corresponding to this kernel phase for each monitor that has requested to be notified.

Monitors register with specific events directly and are alerted only when those events are notified. As an example, a monitor for the Adder will use `register_monitor(this, addition_event)` to request to be alerted as soon as the kernel notifies the `addition_event`. The communication mechanism is the same as the communication mechanism for kernel-level sample points, with the only difference that communication is initiated from the `sc_event` object. Implementing this idea requires minimal changes to the code of `sc_event.cpp`.

Notice that the changes to the kernel are intended to be compiled once, together with the rest of the SystemC code, into a static library (for example, `libsystemc.a` in the case of the OSCI reference implementation) and linked with the user code. It is possible for a commercial implementation to adopt all of the changes proposed here and provide the simulator in binary.

The `observer` is instantiated from the user code at the end of elaboration in `sc_main()` and the `observer` instantiates all monitors before the simulation starts. This allows `observer` to pass references to user-code modules to monitors that monitor user-code properties, for example, values of variables. It is not until the end of elaboration that these references become valid, so instantiating the monitors earlier is not possible.

In case the model does not contain any properties to be monitored, there is negligible overhead in the modified kernel. If the user does not instantiate `observer`, the kernel's pointer to `observer` defaults to 0. Before issuing any callback, the kernel checks if the `observer` is non-zero, and only then it issues the callback, for example,

```
if (observer != 0) { observer->update_begin(); }
```

Thus, in a simulation without monitors the kernel's overhead consists of checking a conditional at every sample point and event notification.

One may object to our decision to modify the kernel by arguing that there are several implementations of the kernel. Our response is that the language proposed by Tabakov et al. [19], which enables the expression of rich temporal properties, requires some kernel-level information to be exposed. Our modifications, however, only expose details that are described in the LRM [1] and should be portable to any implementation that follows the standard. Furthermore, our changes of the existing code (of the OSCI implementation) are minimal and localized, and we believe that other implementations would be easily modified.

We want to note that there are many alternative ways of modifying the kernel (see, e.g. [3], [11], [18]) but none of the previous works has achieved the temporal resolution and kernel–monitor communication provided by our modifications. Other reasonable approaches for monitoring temporal SystemC properties have been explored by Broun et al. [4], one of which requires no modifications of the kernel at all (at the cost of 4 times slower execution speed than a comparable approach with a modified kernel). The novelty of our approach is that it introduces a generic monitor object that can be refined to check any safety LTL property [2] and can sample at much finer (e.g., at the boundaries of delta cycles), as well as much coarser temporal resolution (e.g., at the boundaries of timeless transactions), than any existing approach.

In future releases of SystemC, the simulation semantics and the kernel may change, for example, adding new simulation phases. The modular framework that we describe in this work can be modified easily to handle changes in the kernel. For

each new phase in the kernel, the observer will need to be extended to handle one additional callback, and the modified kernel will need to be instrumented with one additional line of code. Similarly, removing a phase from the kernel requires deleting a callback from the observer, and deleting (rather, not adding) a line of code in the instrumented kernel.

VI. Instrumentation of the MUV

Each monitor inherits from the base class `mon_prototype` the declarations of all virtual functions, and overwrites only those callbacks that are relevant to the property that is monitored. Each monitor then *registers* itself with the `observer` indicating which sampling points it is interested in. For example, if the property is

```
(ALWAYS p==0) @ MON_DELTA_CYCLE_BEGIN
```

(i.e., `p==0` must hold at the beginning of each delta cycle) the corresponding monitor overwrites the function `callback_delta_cycle_begin()` to check if p is equal to 0. The monitor then registers itself with the `observer`:

```
observer->register_property(this,
                            MON_DELTA_CYCLE_BEGIN);
```

for the sampling point `MON_DELTA_CYCLE_BEGIN`.

In most cases properties refer to variables that are either `private` or `protected`. We adopt the perspective of "white-box validation", which means that the state of the model should be fully exposed to the monitors. This is consistent with the view in [19], which considers all class members to have public access for verification purposes. We use the `C++ friend` class declaration to give the monitors access to the data members of the monitored modules, while still preserving the encapsulation and the limited access that the designer intended.

Monitoring user-code variables or other user-code data structures is done at the boundaries of statements. This brings up the question of how modules can receive references to the monitors. The difficulty stems from the fact that the monitors are instantiated after the user-code modules. Our solution is to have the `observer` maintain a list of monitors that need to be triggered at user-code sample points. The `observer` does not handle the communication with those monitors. Instead, it defines a `get_mon_by_index()` function that returns a particular monitor by its index in the list. The instrumentation in the MUV then handles the communication with the monitor by issuing pre-defined callbacks. Each callback depends on the variable or data structure being monitored, and must be defined in the monitor.

For example, in the following code snippet, we are monitoring the values of `i1` and `i2`:

```
...
while(true) {
  wait(input1.value_changed_event() |
                    input2.value_changed_event());
  int i1 = input1.read();
  // Callback to pass the value of first input
  observer->get_mon_by_index(42)->val_of_in1(i1);
  int i2 = input2.read();
  // Callback to pass the value of second input
  observer->get_mon_by_index(42)->val_of_in2(i2);
  ...
```

The approach described here may seem cumbersome when the monitored properties are simple assertions. Indeed, in those cases a simple `assert()` statement placed at the relevant location in the code will "monitor" the property more efficiently. Notice, however, that our framework is designed for monitoring temporal properties that `assert()`'s cannot handle. Moreover, we are working on automating the monitor generation and the user code instrumentation, allowing the framework to handle a large number of properties without major manual effort from the user.

[19] allows the state to be sampled at particular locations in the code. There are several pre-defined locations for each function: **entry**, **exit**, **call**, and **return**, We sample the state of the model at these synchronization points by executing a callback. The framework also allows the specification to refer to the arguments and the return value of functions. In order to expose the arguments we evaluate them first and send them to the monitor using a callback. Similarly, we store the return value (in the cases when the value is assigned to a variable, we use that variable) and send it to the monitor using another callback.

VII. Experimental results

We modified version 2.2.0 of the OSCI simulator and compiled it using the default settings in the Makefile. The empirical results below were measured on a Pentium 4 / 3.20GHz CPU / 1 GB RAM machine running GNU Linux.

First we compiled each of the two models described below, without monitors and without an observer, using both the modified kernel and the original OSCI kernel. We ran a simulation of each version separately and measured a decrease of performance of less than 0.5% when using the modified kernel. We also compiled the models with an observer and no monitors (using the modified kernel) and measured an additional slowdown of the execution time of less than 0.25%. We did not observe any significant memory increase in either of those cases. Thus, our modifications of the kernel do not lead to a significant loss of performance compared to the unmodified kernel.

For the rest of our experiments we measured the effect of running with a different number of monitors and monitoring the properties that we introduced in Section II. Each data point represents the median of 10 measurements. In each case we first ran the model without an observer, monitors, or user code instrumentation to establish the baseline, and then ran several simulations with instrumentation and an increasing number of copies of the same monitor. The results we report are the cost of each copy of the monitor as a percentage of the baseline.

When using the testing methodology described above it is important to consider the possibility of caching effects: if the system were reusing the results of previous computations, averaging the execution time would be meaningless. Our implementation avoids these issues because we create each copy

of the monitor as a separate instance of the same class. Since we are not using `static` variables, each instance contains its own copy of the class data, thus memory use is proportional to the number of copies of the monitors. Furthermore, each instance is handled as a generic `mon_prototype` object, so it is impossible for the SystemC kernel to reuse computations. Finally, the behavior of each monitor is determined by the communication it receives from the SystemC kernel, and the `C++` compiler cannot determine statically that each monitor is doing identical computations, therefore compile-time optimizations will not prevent each monitor from executing.

A. *Squaring via addition*

The first property that we checked was Property (1): the value of the Adder's a variable is always positive, monitored whenever `addition_event` is notified. Since the overhead of checking the property $a > 0$ is minimal, the results mostly expose the overhead of the monitoring framework. For comparison purposes, we measured separately the performance when sampling at the `addition_event` notifications, and when sampling at the end of each delta cycle. The following code snippet shows the key parts of the monitor that checks the safety property at `addition_event`:

```
/**
 * (ALWAYS(adder.a > 0)@adder.driver_event.notified
 * -> ((adder.a > 0)@adder.addition_event.notified
 * UNTIL ``result.write(a)''))
 */

// The constructor
mon1(observer* obs, adder* obj1) : mon_prototype() {
  observer = obs;
  object1 = obj1;
  // Register with the events
  object1->addition_event.register_property(this);
  object1->driver_event.register_property(this);
}
// Overwrite the default virtual void function
virtual void callback_event_notified(sc_event* e) {
  if (e == object1->addition_event) {
    if (state == SECOND_STATE) { // UNTIL clause
      sc_assert((object1->a) >= 0);
    }
  }
  ...
}
```

In each case we instantiated between 10 and 1000 copies of the monitor. The execution time to calculate 1000^2 without monitors is ∼14 seconds. The results are reported as percentage overhead per monitor (Fig. 3).

As the number of monitors increases, the overhead of the framework is amortized over more monitors, thus the average overhead per monitor decreases. Also notice that sampling at event notifications is much more expensive than sampling at the end of delta notifications. Each delta cycle involves (for 1000^2) 1000 event notifications, so the monitor is invoked 1000 times more often. We would also like to point out that while the average overhead per monitor is negligible (∼0.5%–0.003%), the cumulative effect of running 1000 monitors is significant. In the first case (sample at event notification) the execution is slowed down 363% when running with 1000 copies of the monitor (such overhead is not uncommon in industrial applications). The effect is less pronounced when sampling at the end of delta cycles, incurring a total performance penalty of 3.6% when running with 1000 monitors.

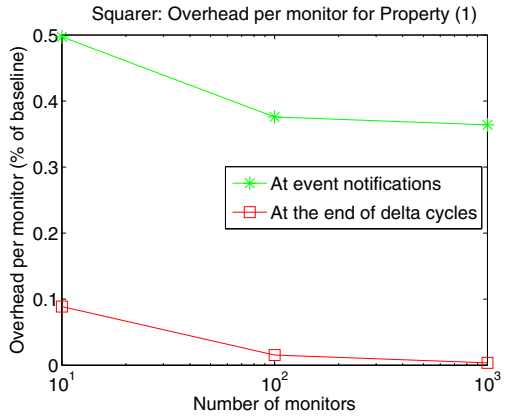

Fig. 3. Monitor overhead as a percentage of baseline (i.e., execution without monitors) for Property (1) using two different temporal resolutions

Fig. 4. Monitor overhead as a percentage of baseline for Property (2)

Property (2) asserts that the adder correctly calculates the sum and returns the result within two delta cycles of reading the input. Notice that this property combines information from the user code (getting the values of the relevant variables) and the kernel (getting information about each delta cycle). We evaluated this property for different sized models – 500, 1000, 1500, and 2000 processes, calculating, respectively, 500^2, 1000^2, 1500^2 and 2000^2. The results are in Figure 4. The behavior we observe is that increasing the number of processes in general reduces the overhead per monitor. The more processes we have, the more work the system needs to do in each delta cycle, thus the effect of the monitoring becomes a smaller fraction of the overall execution. The overhead per monitor averages around 0.01% and the worst cumulative slow-down we observed was by 12.9%, when using 1000 monitors on a 500-process model.

B. *Airline reservation system*

We checked two properties of the system:

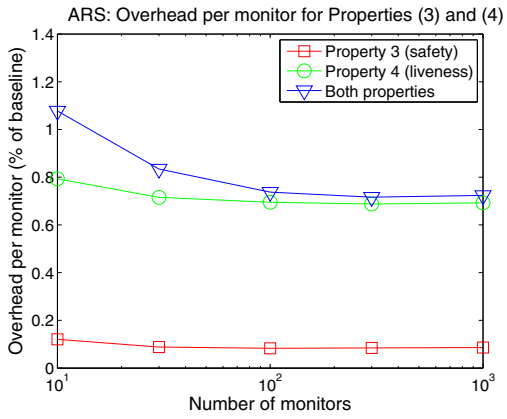

Fig. 5. Experimental results for monitoring Properties (3) and (4) in the airline reservation system

1) The incoming queue has capacity for another request whenever the `incoming_req_nonfull` event is notified (Property (3)), and
2) Every request is propagated through a channel within 5 clock cycles of the slow clock, and it is sent out from each module within 5 slow clock cycles of its receipt by the module (Property (4)).

Since the system is designed to operate indefinitely, we measured the performance by simulating for 1 million (slow-) clock cycles. The wall-clock execution time of the system without monitors is ∼27 seconds. The monitoring overhead is presented as percentage of that baseline (Fig. 5). Checking Property (3) is relatively inexpensive, and the results are consistent with the previous results. Property (4) is much more expensive. The monitor contains a state machine that tracks the arrival and the departure of each request as it travels through the model. This requires a lot of communication from the model to the monitor, as well simulating a state transitions inside the monitor. Running the system with 1000 copies of both monitors slows it down by 715%. Although this is quite significant, it is not unusual. A ten-fold slow-down of the simulation when monitoring complicated properties is often observed in the industry.

VIII. Conclusions

We have described a monitoring framework for the specification language proposed by Tabakov et al. [19] that allows monitoring of temporal properties of SystemC at different levels of resolution, both in clocked and clockless models. Our framework introduces very small changes to the existing code of the kernel and encapsulates the new functionality in two new objects. The changes to the kernel cause a negligible slow-down (less than 0.5%). We implemented and tested several types of properties of two SystemC models at sub-clock-cycle and sub-delta-cycle resolution, involving components from different modules. Our experimental results show that for most properties the overhead is quite small and running hundreds or thousands of monitors does not have a prohibitive cost (usually less than 0.2% per monitor). More complex properties are naturally more expensive, but even in this case the overhead is typically less than 1% per monitor and further optimizations may improve the performance. Our next goal is to automate the process of instrumenting the user code and constructing the monitors.

Acknowledgment: We would like to thank Gila Kamhi and Eli Singerman from Intel's IDC in Haifa, Israel for their insightful comments and many discussions about this work.

References

[1] *IEEE Std 1666 - 2005 IEEE Standard SystemC Language Reference Manual*, 2006.
[2] R. Armoni, D. Korchemny, A. Tiemeyer, M. Y. Vardi, and Y. Zbar. Deterministic dynamic monitors for linear-time assertions. In *Proc. MAMA'2006*, pages 1–20. Springer, 2006.
[3] Nicolas Blanc and Daniel Kroening. Race analysis for SystemC using model checking. In *Proceedings of ICCAD 2008*, pages 356–363. IEEE, 2008.
[4] A. Braun, J. Gerlach, and W. Rosenstiel. Checking temporal properties in SystemC specifications. *High-Level Design Validation and Test Workshop, 2002. 7th IEEE International*, pages 23–27, Oct. 2002.
[5] A. Bunker, G. Gopalakrishnan, and S. A. McKee. Formal Hardware Specification Languages for Protocol Compliance Verification. *ACM Transactions on Design Autom. of Elec. Sys.*, 9(1):1–32, January 2004.
[6] A. Dahan, D. Geist, L. Gluhovsky, D. Pidan, G. Shapir, Y. Wolfsthal, L. Benalycherif, R. Kamdem, and Y. Lahbib. Combining system level modeling with assertion based verification. In *ISQED*, 2005.
[7] W. Ecker, V. Esen, T. Steininger, M. Velten, and M. Hull. Specification language for transaction level assertions. *HLDVT*, pages 77–84, 2006.
[8] C. Eisner and D. Fisman. *A Practical Introduction to PSL (Series on Integrated Circuits and Systems)*. Springer, New York, Inc., Secaucus, NJ, USA, 2006.
[9] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.
[10] SystemC Verification Working Group. SystemC verification standard specification [version 1.0e], 2003.
[11] C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy. Automatic generation of schedulings for improving the test coverage of Systems-on-a-Chip. *FMCAD*, pages 171–178, 2006.
[12] D. Karlsson, P. Eles, and Z. Peng. Formal verification of SystemC designs using a Petri-net based representation. In *DATE '06: Proceedings of the conf. on Design, automation and test in Europe*, pages 1228–1233, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.
[13] A. Kasuya and T. Tesfaye. Verification methodologies in a TLM-to-RTL design flow. In *DAC*, pages 199–204, 2007.
[14] D. Kroening and N. Sharygina. Formal verification of SystemC by automatic hardware/software partitioning. In *MEMOCODE*, pages 101–110, 2005.
[15] M. Moy, F. Maraninchi, and L. Maillet-Contoz. LusSy: an open tool for the analysis of systems-on-a-chip at the transaction level. *Design Automation for Embedded Systems*, 2006.
[16] Matthieu Moy. *Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level*. PhD thesis, INPG, Grenoble, France, December 2005.
[17] L. Pierre and L. Ferro. A tractable and fast method for monitoring systemc TLM specifications. *IEEE Transactions on Computers*, 57:1346–1356, 2008.
[18] A. Sen, V. Ogale, and M. S. Abadir. Predictive runtime verification of multi-processor SoCs in SystemC. In *DAC*, pages 948–953, New York, NY, USA, 2008. ACM.
[19] D. Tabakov, M.Y. Vardi, G. Kamhi, and E. Singerman. A temporal language for SystemC. *Formal Methods in Computer-Aided Design, 2008. FMCAD '08.*, pages 1–9, Nov. 2008.
[20] S. Vijayaraghavan and M. Ramanathan. *A Practical Guide for SystemVerilog Assertions*. Springer, New York, Inc., Secaucus, NJ, USA, 2005.
[21] G. Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.