A Temporal Language for SystemC

Deian Tabakov
Rice University Computer Science
6100 Main Str. MS-132
Houston, TX 77005-1892
dtabakov@cs.rice.edu

Moshe Y. Vardi
Rice University Computer Science
6100 Main Str. MS-132
Houston, TX 77005-1892
vardi@cs.rice.edu

Gila Kamhi
Design Technology and Solutions Division
Intel Corporation
Haifa, Israel
gila.kamhi@intel.com

Eli Singerman
Design Technology and Solutions Division
Intel Corporation
Haifa, Israel
eli.singerman@intel.com

***Abstract*—We describe a general approach for defining new temporal specification languages, and adopting existing languages, for SystemC. We define the concept of "underlying trace" describing the execution of a SystemC model, and then define a set of important primitive assertions about the states in the trace. Our framework not only provides additional expressive power for making atomic assertions, but also provides very fine control over the temporal resolution of the language. Using the primitives defined here as clock expression allows sampling at different levels, from transaction-level to the level of individual statements. The advantage of our approach is that it defines important SystemC properties that have been overlooked previously, and also provides a uniform mechanism for specifying the sampling rate of temporal languages.**

I. Introduction

The increasing complexity of current hardware designs and the need for efficient development of systems-on-chip has motivated a gradual migration away from RTL design and toward more abstract approaches. SystemC[1] has become a successful modeling language partly because it allows designers to model systems at several abstraction levels, from the most concrete (gate level) through the most abstract (system level) [16], as well as to interconnect components from different abstraction levels. In addition to being a modeling language, SystemC is also a simulation environment, driven by a *simulation kernel*. The kernel schedules processes and updates the values of signals and channels in a fashion that mimics concurrent execution, even though in reality the processes are run sequentially. This makes SystemC a particularly useful platform for prototyping and testing hardware and hybrid systems early in the design process [8].

SystemC is built as a library extending C++. The core language provides macros for modeling the fundamental components of the system, for example, modules and channels. The object-oriented encapsulation of C++ and its inheritance capabilities help make designs modular, which, in turn. facilitates reuse and makes IP transfer possible [6]. Various libraries provide further functionality. For example, a popular library called TLM (short for Transaction-Level Modeling) defines channels, interfaces, and protocols that streamline and standardize the development of high-level models in which complex communication and protocols are reduced to a single "transaction". These factors have helped propel SystemC as a *de facto* industry-wide standard modeling language, less than a decade after its first release.

The growing popularity of SystemC has motivated an explosion of research efforts aimed at the verification of SystemC models. Most verification efforts for SystemC so far have been focused on *dynamic verification* (also called *testing* and *simulation*). This approach involves executing the model under verification (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 the expected behavior or output. The complementary approach of *formal verification* either produces a mathematical proof that the MUV correctly implements a specified property, or returns a counterexample, which is a trace that violates the property. Formal verification approaches have received less attention [30], mostly because they are currently applicable only to rather small designs [22], [24], [25] or simple gate-level models [11], [15].

Assertion-based verification (ABV) has recently been gaining acceptance as an essential method for validation of hardware and hybrid models [10]. With ABV, the designer writes properties that capture the design intent in a formal language, e.g., PSL[2] [14] or SVA[3] [31]. The design 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 transforming the specifications into monitors [10]. Many approaches to SystemC specification are limited to simple invariance assertions (using, for example,

Work partially supported by a gift from Intel.

[1]IEEE Standard 1666-2005

[2]IEEE Standard 1850-2007

[3]IEEE Standard 1800-2005

C++'s `assert`) [16], but the industry has recently recognized the need for temporal languages that can express properties related to ongoing behavior, such as p must hold until q is true [3].

There have been a few attempts to adapt temporal languages to SystemC. Ecker et al. [13] describe an implementation of a SystemC Assertion Library inspired by Accellera's Open Verification Library. The library defines 11 properties, most of which are invariance properties and a few are simple temporal templates. The library does not provide a mechanism for defining new temporal templates. Große and Drechsler [11], [15] use a C++ representation of bounded LTL formulas, which are then compiled together with the SystemC model. Their approach is limited to gate-level models. Traulsen et al. [29] translate SystemC models into Promela models, which enables them to use the model checker Spin to verify LTL properties. Habibi, Gawanmeh, and Tahar advocate using PSL [17] and SVA [18] for SystemC, but do not propose an adaptation.

Karlsson et al. [20] use Petri-nets [26] to create a formal representation of the SystemC model at the statement level (i.e., each statement is represented by one place and one transition). Ecker et al. [12] propose an extension of PSL and SVA to express properties of `sc_event`s. A disadvantage for both of these two approaches is that the formal model has one level of abstraction. Pierre and Ferro's framework [27] samples at the statement level, and then only considers those states which are relevant to the property, thus providing a somewhat more flexible temporal resolution. This approach shares the same drawback as all existing approaches: the state of the library code and the state of the simulation kernel are not taken into consideration.

The 1850 PSL Working Group proposed adding a "SystemC flavor" to PSL [2]. Most of the changes in this flavor of PSL are cosmetic; the only significant addition is allowing SystemC's event expressions to be used as clock expressions in PSL. A more serious industrial effort for developing temporal languages for SystemC is by Jeda Technologies [21]. They describe two languages for SystemC inspired by SVA. NSCa is an adaptation of SVA and is aimed at cycle-level verification. NSCv is a variant aimed at transaction-level verification.

In our opinion, current works on temporal languages for SystemC are lacking in three major respects. First, none of these works addresses the most fundamental issue for temporal languages, which is a precise definition of a trace. The notion of a trace is a key notion in any discussion of temporal semantics. Hardware-oriented languages such as PSL or SVA assume an underlying notion of a clock-cycle-level trace, but for higher-level languages such as SystemC the notion of a clock- cycle-level trace may not be appropriate. Second, a temporal language should be adaptable to different levels of abstraction in the design of the model. One of the strengths of SystemC is modeling at different levels of abstraction; during the design process the model typically gets refined, evolving from a system-level model to a gate-level model. Jeda Technologies' solution of different languages for different levels of abstractions is simply not flexible enough. Third, the existing specification languages are approaching the issue mostly from a hardware perspective and are ignoring the fact that a SystemC model is, fundamentally, a C++ program. There is a large body of work on specification and model checking of Java, C++ and C code (e.g., [4], [5], [7], [9], [19]) and the specification primitives used there should be adapted for SystemC.

In this paper we describe a new approach to defining temporal languages for SystemC. Our starting point is a precise definition of a SystemC trace. Intuitively, a trace is a sequence of states in the execution of the model. Defining this notion precisely for SystemC is quite nontrivial. First, we abstract the simulation kernel and define its state with respect to this abstraction. Second, we recognize that one needs to distinguish between the SystemC model developed by the designer and the set of SystemC libraries used by this model. While the state of the model is fully detailed, the libraries are modeled only at the level of their exposed interfaces. Finally, we define the notion of a trace with respect to these abstractions of the kernel and the libraries.

We then argue that SVA and the SystemC flavor of PSL fail to identify important Boolean properties relevant to the execution of SystemC models. We propose enriching the Boolean layer with a new set of atomic properties, thereby making the specification language more expressive. First, we extend PSL and SVA with a set of atomic propositions that are adequate for expressing properties of C++ programs. For example, we add propositions that enable us to express pre- and post-conditions for functions. Second, we extend PSL and SVA with propositions that enable us to refer to the current phase of the simulation execution, corresponding to our abstraction of the simulation kernel. Finally, we observe that the clock-sampling mechanism available in PSL and SVA offers us a way to express temporal properties at different levels of abstractions. A fine sampling would correspond, say, to subcycle-level abstraction, while coarser sampling would correspond, say, to transaction-level abstraction. Since in PSL and SVA any Boolean expression can be used as a clock expression, the additions to the Boolean layer that we propose here also provide us a much finer control over the temporal resolution. We show that this approach enables us to tailor temporal languages to SystemC in a uniform way; all that is needed is to adapt the underlying syntax for state assertions. The main feature of the resulting framework is the ease with which properties can be expressed at different levels of abstractions, without having to use different languages.

II. Semantics of SystemC Simulation

SystemC was originally developed for the specification of circuit models that could be compiled using a regular C++ compiler and simulated efficiently [22]. As the language evolved it changed its primary goal to enable system-level modeling, that is, modeling of systems that might be implemented in software, hardware, or a combination of the two (e.g., system-on-a-chip). 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 [16].

At the base layer SystemC provides an event-driven *simulation kernel*. *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. 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 finishes executing or explicitly suspends itself by calling `wait()`.

Like VHDL and Verilog, the SystemC kernel supports the notion of a delta-cycle. The simulation clock does not advance during a delta-cycle, and as a result all processes that execute during the delta-cycle appear to be executing simultaneously. In order to maintain the appearance of parallel execution it is also necessary to postpone the effect of all channel and signal updates and event notifications. To that end, during a delta-cycle the kernel switches from *evaluation* phase to *update* phase to *delta-notification* phase. During the evaluation phase any values written to a channel or a signal are not immediately available, and the value of the channel or signal is not updated until there are no more runnable processes and the kernel enters the update phase. Likewise, during the evaluation phase an event might be notified but the processes sensitive to that event will not become runnable until the delta notification phase. An exception to this rule are *immediate notifications*, which cause dependent processes to be added to the pool of runnable processes during the evaluation phase, rather than waiting until the delta notification phase.

The simulation semantics of SystemC, which is defined in [1], is presented in pseudo code in Figure 1.

The execution of a SystemC application starts with the *Elaboration phase*, during which all modules are instantiated and channels are bound to ports, and some channels may register a `request_update()` (line 7). Then the kernel enters the *Initialization phase* (lines 8–20). During the Initialization phase all channels with pending updates are updated (lines 8–10), all initializable `SC_THREAD`s and `SC_METHOD`s are made runnable (lines 11–15), and pending delta notifications cause their dependent processes to become runnable (lines 16–20). Next the kernel starts a delta-cycle and runs all runnable processes one at a time (*Evaluation phase*, lines 23–26). During this phase pending channel updates are collected in U, and pending event notifications are collected in D. The evaluation phase is followed by an update phase (lines 27–29) where all collected channel update requests are executed and writes to signals take effect. After that the kernel enters the delta-notification phase (lines 30–34) where notified events trigger their dependent processes. Note that immediate notifications may make new processes runnable during the execution of lines 22-26.

If at this point there are runnable processes the kernel loops back to line 22 and starts another evaluation phase and a new delta-cycle. Alternatively, if there are no more runnable processes, the kernel advances the simulation clock to the earliest timed-delay notification (essentially, a notification that

```
1:  PC ← all primitive channels
2:  P ← all processes
3:  R ← ∅ /* Set of runnable processes */
4:  D ← ∅ /* Set of pending delta notifications */
5:  U ← ∅ /* Set of update requests */
6:  T ← ∅ /* Set of pending timed notifications */
7:  /* Start elaboration; collect all update requests in U */
8:  for all chan ∈ U do
9:    run chan.update()
10: end for
11: for all p ∈ P do
12:   if p is initializable and p is not clocked thread then
13:     R ← R ∪ p /* Make p runnable */
14:   end if
15: end for
16: for all p ∈ P do
17:   if p is triggered by an event in D then
18:     R ← R ∪ p /* Make p runnable */
19:   end if
20: end for /* End of Initialization phase */
21: repeat
22:   while R ≠ ∅ do /* New delta-cycle begins */
23:     for all r ∈ R do /* Evaluation phase */
24:       R ← R \ r
25:       run r until it invokes wait() or returns
26:     end for
27:     for all chan ∈ U do /* Update phase */
28:       run chan.update()
29:     end for
30:     for all p ∈ P do /* Delta notification phase */
31:       if p is triggered by an event in D then
32:         R ← R ∪ p /* p is now runnable */
33:       end if
34:     end for /* End of delta-cycle */
35:   end while
36:   if T ≠ ∅ then
37:     Advance the clock to the earliest timed delay t.
38:     T ← T \ t
39:     for all p ∈ P do /* Timed notification phase */
40:       if t triggers p then
41:         R ← R ∪ p /* p is now runnable */
42:       end if
43:     end for
44:   end if
45: until end of simulation
```

Fig. 1. Simulation Semantics of SystemC

is explicitly set to be notified after some delay). All processes sensitive to this event are triggered (lines 39–43) and the kernel loops back to line 21 and starts a new delta-cycle. This process is repeated indefinitely, unless the designer has specified a fixed simulation time or all processes have terminated.

III. Definition of Execution Trace

All (linear) temporal languages are interpreted over execution traces, therefore before we can define the semantics of temporal properties for SystemC we need a precise definition of an execution trace. Traditionally, a trace has been defined as a sequence of states in the execution of the model, but there has been remarkably little discussion in the literature about the definition of SystemC traces.

Some existing approaches adopt a clock-cycle-level temporal resolution, so implicitly a state is a valuation of all variables at the boundaries of clock cycles, and the trace consists of the sequence of such valuations [21]. We believe that such an approach 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 [16], [24]. 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. In this section we give a more refined definition of SystemC traces that accounts for SystemC's simulation semantics. Our definition starts with a precise definition of a system state, which encompasses the state of the kernel, the state of the user model, and the state of the external libraries.

A. Kernel State

1) Kernel Phases: It is not immediately clear why the state of the kernel needs to be included in the execution trace. For example, in the work of Kroening and Sharygina [22] 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" [22]. Under this model synchronization occurs when a process encounters a `wait()` or a `notify()` instruction. The observable behavior of their abstraction of execution matches well 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. [20], Ecker et al. [12], and Pierre and Ferro [27], 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

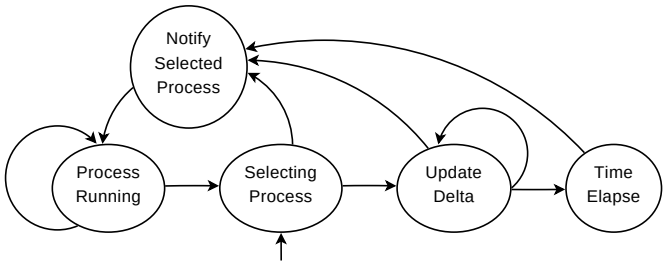

Fig. 2. Kernel states proposed by Moy et al.

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. We conclude, therefore, that the state of the kernel must be exposed to a certain extent, in order to enable the user to specify properties at different levels of abstraction. (We do not discuss in detail here how such exposure is to be implemented; that may depend on the context. For example, in the context of dynamic verification such exposure can be implemented by extending the kernel with an API for querying the state of the kernel.)

This approach of exposing the state of the kernel to some extent was taken by Moy et al. [24], [25]. Their work formalizes SystemC models in terms of communicating state machines, where the kernel is modeled as a particular state machine (Figure 2). Thus, the state of the kernel is exposed at an abstraction level corresponding to this specific state machine.

Once one accepts the principle of exposing the kernel state, the question remains at what abstraction level to expose the kernel. Moy et al. offer a specific abstraction, but their choice is open to criticism. Their formalism is somewhat less detailed than the simulation semantics in SystemC's Language Reference Manual (LRM) [1]. One could offer other abstractions of the kernel, but without some guiding principle such abstractions are also open to criticism. Our guiding principle is that the abstraction should abstract away the kernel implementation, but expose fully SystemC's simulation semantics, as described in [1] and Figure 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.

We therefore abstract the simulation semantics, as described in Figure 1, by the state machine described in Figure 3. Our abstraction may seem, at first sight, to be somewhat too detailed. A simpler abstraction of the kernel would consider each phase (*Initialization*, *Evaluation*, *Update*, *Delta notification*, *Timed notification*) as a separate state in a state machine. Why does our model have more states? The answer is that the simpler model does not expose the start and the finish of individual processes (all processes are run while the kernel is in the Evaluation phase, and similarly for the Update phase). Our abstraction exposes the transfer of control

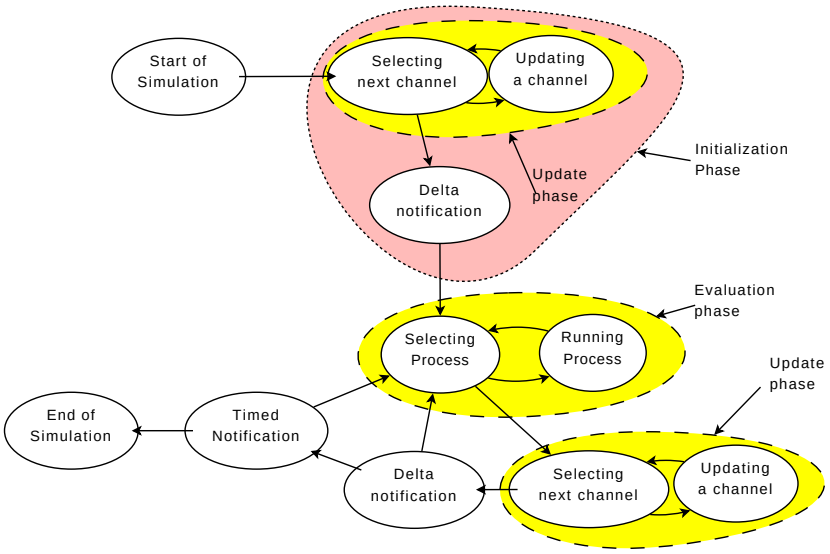

Fig. 3. Proposed Kernel States

between the kernel and the processes by splitting the Update end Evaluation phases into two subphases. Since SystemC uses the interleaving approach to concurrency, we believe that exposing transfer of control is important. Our abstraction of the kernel is more detailed than Moy et al.'s, therefore giving us more expressive power.

One might argue that keeping track of all phases of the simulation semantics is unnecessary because very few properties relate to those specific phases. We decided to take a generic approach and model all phases of the kernel that are described by the simulation semantics rather than try to pass judgment on which ones are the most important. As we show in the sequel, our approach enables users to use coarser abstractions if needed. Since we need to anticipate all possible uses of SystemC, exposing the semantic fully is the most justifiable approach.

2) SystemC 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.

Events are notified by calling the `notify()` method of class `sc_event`. 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 happens during 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.

PSL's and Jeda's treatment of events is to allow them to be used as clock expressions, though the issue when events are actually notified is not discussed explicitly. We believe that the fundamental role played by events in the execution of SystemC models justifies fully exposing event notification in the kernel's state. In essence, we are elevating event notification to the Boolean layer. 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.

B. User Model State

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. Our perspective is that of "white-box validation", which means that the state of the model should be fully exposed. Thus, the property languages should consider all class members to have **public** access, including those that are declared as **private** or **protected**.

As argued earlier, we believe that a property specification language for SystemC ought to consider SystemC as a system-level language, rather than a hardware-level language. This requires that the execution of a SystemC model be exposed also at the source-code level [5]. This should enable us to refer explicitly to statements being executed both via their label as well as by their syntax; for example, we should be able to refer to all invocations of a specific method. Formally, we add to the state a variable of type `string` that contains as a string the statement being executed at a given step. Furthermore, as method invocation is central to the execution of object-oriented code, the values of arguments passed to and returned from invoked methods should also be exposed, by exposing the values of the formal parameters of the method upon invocation. In essence, we are requiring traces of SystemsC model to include both state and transition descriptions, in contrast to standard models of temporal logics that are typically state based [28]. By exposing both semantics and syntax of the model, we enable properties that relate source code and execution; for example, we can specify that for every port connected to a specific channel, the `write()` interface method call is passed only positive numbers as arguments. In order to simplify the process of referring to individual statements in what could be many thousands of lines of code, we provide several predefined labels for important locations. This is discussed further in Section IV.

Finally, like Moy et al., we expose for each process its status, as *waiting*, *runnable*, or *running*, corresponding to the simulation semantics, per Figure 3.

C. Library Code State

Most SystemC models rely heavily on external libraries (the TLM library, for example). These libraries encapsulate crucial components of SystemC models, such as signals and channels. When formalizing the notion of SystemC state, we need to decide how to formalize the state of libraries. One approach would be to extend the white-box approach to library code, but users need to be familiar with libraries only at the API level, and not at the implementation level. Furthermore, while in many cases the code of the library is available, in others the library may be supplied as compiled code, thereby hiding the internal state.

Kroening and Sharygina [22] do not discuss how they handle library code. Moy et. al's approach [23], [25] is to provide specific state-machine models reflecting the functionality of TLM constructs. The benefit of this approach is that it preserves important information about the structure and behavior of the design. A major drawback is that this requires manual effort to develop formal models for libraries. These formal models may have to be revised when libraries are revised.

To attain generality, we believe that library code should be treated as a black box. For example, when it comes to `tlm_fifo`, the state of the queue should be exposed without exposing implementation details. Furthermore, the state of a library should be exposed only in terms of the API of that library. Consider, for example, the TLM 1.0 library. Properties should not have access to the state of the `tlm_fifo` other than via side-effect-free function calls, for example, via the `peek` method. Of course, when library source code is available, users can choose to treat it as a part of the user model and view it from a white-box perspective.

D. 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. We have discussed so far how we formalize the state of the kernel, the model, and the libraries. It remains to discuss at what level of granularity we formalize the transition from one state to its successor.

When the kernel is executing, we follow transitions in the state machine described in Figure 3. 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` statement. With respect to transitions of processes, we follow the "large-step semantics" approach [32]. Under this approach we focus 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 we ignore the valuations of the variables during the execution of the subexpressions. We believe that this matches the level at which programmers and verification engineers think about the source code. 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, we consider each invocation of a library method, for example, invoking a channel-interface method, to return in one step. This is consistent with our black-box view of libraries.

IV. Language Definition

After we have given a definition of an execution trace we can define a set of specification primitives for SystemC models. Notice that specification languages like PSL and SVA are already quite expressive temporarily, so no extension is necessary for their temporal layer in SystemC. Thus, our focus in this section is on extending the Boolean layer. The new set of primitives introduced here not only allows the specification of a richer set of properties, but also provides a uniform mechanism for controlling the temporal resolution of the specification language. Unlike Kasuya and Tesfaye's approach [21], which requires a separate language for clock-based and event-based models, our framework allows for greater

flexibility in the temporal granularity of the specification, and is general enough that it can readily be adapted for any temporal language with the notion of clock expressions.

Table I summarizes our proposed specification primitives.

SystemC_expr	::=	*model_expr* | *kernel_expr*
model_expr	::=	*loc_expr* | *arg_expr* | *proc_expr*
loc_expr	::=	[**before** | **after**] {code_label | *syntax_expr* } | *func_name:*{**entry** | **exit** | **call** | **return**}
syntax_expr	::=	$function_{\text{String} \rightarrow \text{Boolean}}$ (**curr_statement, . . .**)
arg_expr	::=	*func_name* **:** non-negative_integer
proc_expr	::=	*proc_name*.**proc_state**
kernel_expr	::=	*phase_expr* | *event_expr*
phase_expr	::=	**kernel_phase**
event_expr	::=	*event_name*.**notified**

TABLE I
PROPOSED SPECIFICATION PRIMITIVES

A. User Model Primitives

A *model_expr* is a Boolean expression about the state of the user model. Under this category we include expressions about the location counter (*loc_expr*), the arguments and return values of functions (*arg_expr*), and expressions about the state of each process (*proc_expr*).

Our definition of a trace explicitly keeps track of the location counter during the execution of the user model. With each label `code_label` in the source code of the model we associate a Boolean variable that is true precisely in those states where the statement that is about to be executed is labeled by `code_label`. We introduce two optional modifiers, **before** and **after**, to refer, respectively, to the state immediately before and immediately after that statement is executed (the default behavior corresponds to specifying **before**). Using this primitive allows the specification of forbidden or mandatory paths in the execution of the compiled model, e.g., if execution reaches `lbl1` it should not reach `lbl2`. It also allows the specification of properties that must hold at particular locations in the code.

Inspired by BLAST [5], we also propose adding a primitive **curr_statement** of type `string` that exposes the syntax of the statement that is about to be executed. As mentioned earlier in Section III-B, this mechanism enables properties that relate syntax and semantics. PSL and SVA allow using functions defined in the underlying HDL language, which in the context of SystemC means that we can use a number of C++ functions that operate on strings and return Booleans (*syntax_expr*) and pass **curr_statement** as an argument. Of particular interest are regular expression matching and string comparison functions, because they allow the user to quickly identify a set of "important" locations in the source code without having to introduce labels manually. As an example, one can use this mechanism to identify all locations in the user model where a particular statement is executed.

Example 1 (Matching source code): Suppose that before a function *foo()* is called in any module, some consistency condition *bar* must hold. For this property we first define a function *match(string)* that returns true whenever the string contains "foo(", e.g. via pattern matching. The property then becomes `(before match(curr_statement)) -> bar`.

The specification of pre- and post-conditions requires evaluating assertions at specific locations in the source code that are difficult to identify automatically via the mechanisms described so far. Inspired by SLIC [4], we introduce two additional primitives, **entry** and **exit**, that refer to the location immediately before the first executable statement, and the location immediately after the last executable statement, in a function. In some cases the pre-condition may need to refer to the values of the formal parameters passed on to the function. If the function is a part of the user model, one can use the names of the variables on the parameter list. We also propose an alternative mechanism (previously used in both BLAST [5] and SLIC [4]) to refer to the value of each parameter according to its order. For a function *func(type1 param1, type2 param2, . . .)*, we define implicit variables *func:1*, *func:2*, etc., whose values (and types) are equal to the values (and types) of the formal parameters of the function at the entry point (i.e., the values of the variables before the first statement in the function has been executed).

Example 2 (Precondition): One desirable precondition for a function *long_division(double dividend, double divisor)* is `ALWAYS (long_division:entry -> long_division:2 != 0)`.

This mechanism is inadequate for the specification of pre- and post-conditions of functions defined in a proprietary library because the source code is not exposed in the execution trace. For cases like this we introduce another set of primitives that we adopt from SLIC [4]. For each function call to *func()* we introduce the primitives *func*.**call** and *func*.**return** to refer, respectively, to the location in the source code that contains the function call and to the location immediately after the function call. (Note that here we assume that function calls are not nested.) The values of the arguments can be accessed via implicit variables *func:1*, *func:2*, etc., whose types match the types of the arguments to the function, and whose values are precisely the values of the actual parameters at *func*.**call**. Another implicit variable, *func:0*, is defined as the value returned by the function, and it is only defined at *func*.**return**. This mechanism allows the specification of properties of proprietary functions and objects even if the library does not expose their states directly (e.g., a proprietary channel). For example, we can ensure that a channel contains only positive values by specifying that the arguments to all relevant calls to `write()` are always positive. As a second example, we can express the property that the channel behaves like a queue by using PSL's modeling layer to temporarily remember two values written to the channel, and then verifying that the values are returned in the same order via the channel's `read()` method.

Finally, we propose adding a primitive **proc_state** that for each process name returns a value in the enumerated set { *waiting*, *runnable*, *running* } depending on the status of the process in the kernel. One can use this primitive to specify that a particular process is executed infinitely often, or is executed

at least once during each delta cycle.

B. Kernel Primitives

A *kernel_expr* is an expression about the state of the kernel. We introduce primitives for exposing the current phase (*phase_expr*) and when events are notified (*event_expr*).

When the kernel has the thread of control, the execution trace makes transitions that reflect the changing phases of the kernel. We propose adding a primitive **kernel_phase** that exposes the current phase. The primitive returns a value in an enumerated set { *startsim*, *init_select_channel*, *init_update_channel*, ..., *endsim* }, corresponding to our abstraction of the kernel in Figure 3. **kernel_phase** allows the user to define properties whose evaluation is triggered by different phases of the kernel.

Example 3 (Stable states): Variable p in process `proc` in module `mod` must be 0 in all stable states (i.e., states where no process is executing): `ALWAYS (! kernel_phase = eval_run_proc -> mod.proc.p = 0)`. The kernel phase `eval_run_proc` corresponds to the instances where a process is running (line 25 in Figure 1) in the evaluation loop (lines 23–26 on Figure 1).

Event notifications (*event_expr*) allow us to detect when the notifications actually take place. Note that the mechanisms described earlier expose function calls at the source-code level, and event notification requests and cancellations (i.e., calls to `notify()` and `cancel()`) are exposed via the user model primitives. However, these primitives do not expose the particular state when the actual notification is carried out (i.e., when the dependent processes are made *runnable* by the kernel). For each event we propose a primitive **notified** which is true whenever the kernel carries out the actual notification. For immediate notifications this happens concurrently with the function call to `notify()`; for delta-delayed notifications it happens during the earliest delta-notification phase; for time-delayed notifications it happens during the corresponding timed-notification phase. Note that both delta-delayed and time-delayed notification requests can be subsequently canceled, therefore a call to `notify()` with a non-negative argument does not guarantee that **notified** will be true in the future. The role of this primitive is particularly important when referring to events that are notified implicitly, e.g. when an `sc_signal` changes value, a built-in event named `value_changed_event` is notified implicitly by the kernel in the delta notification phase that immediately follows.

Example 4: The requirement that a signal changes in every delta cycle can be expressed as `ALWAYS (kernel_phase = delta_notify -> signal.value_changed_event.notified)`.

Example 5: Variable p in process `proc` in module `mod` must be 0 at the rising edge of clock `cl`: `ALWAYS (cl.posedge.notified -> mod.proc.p = 0)`.

C. Using Primitives as Clock Expressions

So far we have not discussed how the primitives described here can be used to control the temporal resolution of the specification language. Existing 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. Traditionally (e.g., [14], [21], [31]) sampling is done at the boundary of clock cycles. Our framework can easily provide the same functionality by using the event notification primitive described earlier. Note that an `sc_clock` exposes two events, `posedge_event` and `negedge_event`, which are notified every time the value of the clock changes and the new value is, respectively, 1 and 0. Using `posedge_event.notified` and/or `negedge_event.notified` as CEs we can sample at the boundaries of half-clock or clock cycles. Clearly, we are not limited to the simulation clock. If finer grained resolution is required, one can sample at the boundary of delta cycles by using (**kernel_phase** = delta_notification) as a CE (sampling at the delta notification phases of the kernel), or at the end of execution of each process by sampling at (**kernel_phase** = next_proc_select), which corresponds to the phases where the kernel is selecting the next process to be executed. One can even sample at the boundary of the individual statements in the source code (which is the default sampling rate).

Example 6 (Clock expressions): For this example we borrow some of PSL's syntax. The property "A call to function *req()* is followed within 3 clock cycles (of clock `cl`) by a notification of event `ack`" can be expressed as

```
default clock = cl.posedge.notified;
ALWAYS (req:call -> next[3] ack.notified).
```

If the acknowledgment needs to be received within 3 delta cycles instead, all we need to do is change the clock expression:

```
default clock=(kernel_phase=delta_notify);
ALWAYS (req:call -> next[3] ack.notified).
```

The same mechanism allows specifying coarser sampling rates as well. In a transaction-level or system-level model one is typically interested in its behavior at event notification instances. Jeda's framework provides this functionality in a separate language (NSCv), but they require the user to setup callbacks (essentially, function calls) that "report an event occurrence at the point of transaction processing" [21]. In our framework this can be done by using as CEs the event-notification primitives introduced earlier. For example, in a TL model we can sample at the instances when the an `sc_fifo` is written to (by sampling at `data_written_event.notified`), or when a signal changes value (by sampling at `value_changed_event.notified`), etc. The advantage of our framework is that it is using the same language throughout the refinement process as the model is transformed from higher to lower levels of abstraction.

V. Discussion

To the best of our knowledge this is the first work that provides a principled discussion and definition of a SystemC execution trace. Our notion of a state encompasses information about the kernel (current phase and notification of events), as well as statement-level information about the user model, and

publicly exposed state of the libraries. The level of details preserved in the states allows us to define a rich set of new properties about the execution of the SystemC model. Moreover, the user can specify a range of sampling rates, from the most coarse (transaction- and system-level) to the most detailed (statement level) by combining clock expressions with the primitives introduced in this paper. Our framework is general enough that it can be adopted by most existing temporal specification languages by simply enriching the set of allowed atomic expressions.

Bringing techniques from software verification to the SystemC world is our second contribution. The fact that SystemC models should be viewed as software models has been ignored so far. The result is a minimal yet highly expressive extension of PSL/SVA.

The framework that we propose is equally applicable to dynamic verification and formal verification. Enabling a dynamic verification path would require a minimal "one-time" addition to SystemC's simulation kernel source code to expose a part of SystemC kernel's internal state and data structures. The user code will have to be instrumented to allow the monitors to observe the behavior of the relevant components, and the monitors will be compiled and executed together with the model. We are currently working on an implementation that automates the process.

Applying formal methods to SystemC is an active area of research with several different approaches (e.g. using communicating state machines [24], [25], Petri-nets [20], or leveraging Promela/SPIN [29]). All of these works propose some FSM-like abstraction of the SystemC kernel, and no two abstractions are the same. The model presented in this paper corresponds directly to the simulation semantics as described in the SystemC LRM [1] and is the most detailed model without making any assumptions about the particular kernel implementation. The FSM in Figure 3 can easily be adopted by existing and future formal verification approaches. Exposing the syntax further allows the analysis of the model from a purely software point of view. The techniques used in SLIC [4] and BLAST [5] can and should be applied to formal verification of SystemC.

REFERENCES

[1] *IEEE Std 1666 - 2005 IEEE Standard SystemC Language Reference Manual*, 2006.

[2] Standard for property specification language (PSL). *IEC 62531:2007 (E)*, pages 1–156, 2007.

[3] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification logic. In *Proc. 8th ICTACAS*, volume 2280 of *LNCS*, pages 296–211, Grenoble, France, April 2002. Springer-Verlag.

[4] T. Ball and S. Rajamani. SLIC: A specification language for interface checking. Technical report, Microsoft Research, January 2002.

[5] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The BLAST query language for software verification. In *SAS*, pages 2–18, 2004.

[6] 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.

[7] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[8] L. Charest and E. M. Aboulhamid. A VHDL/SystemC comparison in handling design reuse. In *Proceedings of 2002 International Workshop on System-on-Chip for Real-Time Applications*, pages 79–85, Banff, Canada, July 2002.

[9] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. Expressing checkable properties of dynamic systems: the Bandera Specification Language. *STTT*, 4(1):34–56, 2002.

[10] 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*, pages 310–315, 2005.

[11] R. Drechsler and D. Große. Reachability analysis for formal verification of SystemC. In *DSD*, pages 337–340, 2002.

[12] W. Ecker, V. Esen, T. Steininger, M. Velten, and M. Hull. Specification language for transaction level assertions. *HLDVT*, pages 77–84, 2006.

[13] W Ecker, V Esen, T Steininger, M Velten, and J Smit. Implementation of a SystemC assertion library, 2005.

[14] C. Eisner and D. Fisman. *A Practical Introduction to PSL (Series on Integrated Circuits and Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[15] D. Große and R. Drechsler. Formal verification of ltl formulas for SystemC designs. In *ISCAS (5)*, pages 245–248, 2003.

[16] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[17] A. Habibi, A. Gawanmeh, and S. Tahar. Assertion based verification of PSL for SystemC designs. In *International Symposium on System-on-Chip*, pages 177–180, 2004.

[18] A. Habibi and S. Tahar. On the extension of SystemC by SystemVerilog assertions. *Electrical and Computer Engineering, 2004. Canadian Conference on*, 4:1869–1872 Vol.4, 2-5 May 2004.

[19] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Tenth International Workshop on Model Checking of Software (SPIN)*, volume LNCS 2648, 2003.

[20] D. Karlsson, P. Eles, and Z. Peng. Formal verification of SystemC designs using a Petri-net based representation. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1228–1233, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[21] A. Kasuya and T. Tesfaye. Verification methodologies in a tlm-to-rtl design flow. In *DAC*, pages 199–204, 2007.

[22] D. Kroening and N. Sharygina. Formal verification of SystemC by automatic hardware/software partitioning. In *MEMOCODE*, pages 101–110, 2005.

[23] M. Moy, F. Maraninchi, and L. Maillet-Contoz. LusSy: A toolbox for the analysis of systems-on-a-chip at the transactional level. In *International Conference on Application of Concurrency to System Design*, June 2005.

[24] 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.

[25] Matthieu Moy. *Techniques and Tools for the Verifi cation of Systems-on-a-Chip at the Transaction Level*. PhD thesis, INPG, Grenoble, France, December 2005.

[26] James L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[27] L. Pierre and L. Ferro. A tractable and fast method for monitoring SystemC TLM specifications. *IEEE Transactions on Computers*, 2008.

[28] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundation of Computer Science*, pages 46–57, 1977.

[29] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi. A SystemC/TLM semantics in Promela and its possible applications. In *14th Workshop on Model Checking Software SPIN*, July 2007.

[30] M. Y. Vardi. Formal techniques for SystemC verification. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 188–192, New York, NY, USA, 2007. ACM.

[31] S. Vijayaraghavan and M. Ramanathan. *A Practical Guide for SystemVerilog Assertions*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[32] G. Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.