Automatic Aspectization of SystemC

Deian Tabakov
Schlumberger Information Solutions
5599 San Felipe Str.
Houston, TX, USA
dtabakov@slb.com

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

Abstract

A successful monitoring framework for SystemC requires access to internal variables of modules and channels, and the ability to trace the execution of threads and methods. We propose a framework for automatically instrumenting user code and exposing its state and syntax via automatically generated Aspect-Oriented Programming code and direct instrumentation. This allows monitoring the execution with a fine-grained temporal resolution. Our tool, CHIMP, allows the users to declare specification primitives referring to the values of internal variables, the values of parameters passed to function calls, and function return values. Tracing execution of processes is enabled by allowing statements' execution or function calls to be used as atomic propositions. The correct behavior of the model can then be specified by forming temporal formulas and clock expressions using these primitives, without requiring manual instrumentation of the user code.

Categories and Subject Descriptors B.6.2 [*Logic Design*]: Reliability and Testing; D.2.4 [*Software Engineering*]: Software/Program Verification—Assertion checkers, Formal methods

1. Introduction

SystemC (IEEE Standard 1666-2005) has emerged as a *de facto* standard for modeling of hardware/software systems [21], in part because it allows modeling at high levels of abstraction, gradual refinement of the model, and execution of the model during each design stage. SystemC is a library of classes and macros extending `C++`. Hardware components like *modules*, *channels*, *signals*, *ports* and *interfaces*, have corresponding SystemC objects. Each module can have any number of internal variables, representing local memory; threads and methods, defining the functionality of the module; and other sub-modules, allowing for complex hierarchical models. SystemC *events* trigger the execution (or resumption) of processes, and their effect can be immediate or delayed, depending on the event's *notification* type. SystemC channels and signals carry out communication between modules, and can have their own internal variables, processes, and sub-modules [19]. All modules, channels, and ports are implemented internally as `C++` classes. This provides natural object-oriented encapsulation, data hiding, and well-defined inheritance mechanisms.

In addition to being a modeling language, SystemC is also a simulation framework, allowing efficient execution of the model to be simulated. The SystemC *kernel* keeps track of event notifications, maintains a set of triggered processes that are *eligible* to run, and schedules the order of their execution. The kernel also keeps track of, and advances, the execution time, and triggers events that are set to be notified after some delay. For a detailed discussion of the SystemC kernel we direct the reader to [19, 33].

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 [9]. With ABV, the designer asserts properties that capture the design intent in a formal language, e.g., PSL[1] [12] or SVA[2] [37]. 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 monitors in parallel with the model. The monitors typically observe the inputs to the MUV and ensure that the behavior or the output is consistent with asserted properties [19]. The complementary approach of *formal verification* produces a

MISS'12, March 27, 2012, Potsdam, Germany.

[1] IEEE Standard 1850-2007

[2] IEEE Standard 1800-2005

mathematical proof that the MUV correctly implements the asserted properties. In case a violation of the property is detected, both methods are able to return a counterexample, which is a trace that violates 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 [33] for a detailed discussion). Tabakov et al. [33] pointed 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. proposed 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 monitor module in parallel with the MUV for each property to be checked; see, e.g., [3, 19]. 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 monitor 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 [33] is that certain nominal information about the kernel, specifically, kernel phases and event notification, has to be exposed to the monitors. In [32], Tabakov and Vardi showed how to expose the semantics of the SystemC kernel in a modular way without a serious performance hit. Their work makes minor additions to the existing code of the kernel, and is easy to adapt to existing installations of the reference SystemC simulation kernel provided by the Open SystemC Initiative (OSCI), and to standard-compliant proprietary implementations. Tabakov and Vardi [32] tested their implementation using manually generated monitors and manual instrumentation of the user code and demonstrate a nominal performance overhead.

The user-code primitives that have to be exposed according to Tabakov et al. [33] are meant to enable *white-box validation*, which means that the C++ code of all processes in the model, the values of user-define variables, location counter, and the call stack are first-class members of the property specification language [4]. This allows a very flexible temporal resolution of the *execution trace*. In particular, it enables specification of properties across a wide spectrum of temporal granularities, from cycle level to transaction level.

To enable such white-box validation requires exposing to the monitoring framework the execution flow, the C++ syntax, and the state of the user code. The difficulty is that the very philosophy of object orientation is an obstacle to white-box validation. The benefits of object-oriented encapsulation and data hiding of modules, ports, and channels are deterrents for effective monitoring, because access to internal variables is limited to the objects' own processes. Good software-engineering practices suggest keeping internal data private and exposing it only via dedicated function calls or ports. Because of this, most other works on monitoring SystemC models (see discussion later) focus on using only the publicly available data members, exposing the execution of user code only at the level of function calls and returns. The framework of Tabakov and Vardi [32] enables full white-box validation, but the user code has to be instrumented extensively. References to user-defined variables, location counter, and the call stack require adding function calls at the appropriate locations, which would invoke the monitoring processes at the appropriate points in the execution [32].

Exposing the state of the model and the flow of its execution can be achieved by instrumenting the model's code. Manual instrumentation, however, is prohibitively expensive and error prone. For example, the specification of a low-level model of a driver may assert that $x \neq 0$ in all statements where x is a divisor, which may require instrumentation of many statements. As another example, when using a transaction-level model (TLM) of the system, it may be important to monitor the start and end of each transaction. This requires the monitor to be aware of each call to the function initiating the transaction and of the function's return value; potentially, hundreds of call sites may need to be instrumented.

Requiring extensive code instrumentation is also highly problematic from a software-engineering perspective. The goal of high-level modeling languages such as SystemC is to push design and validation to an earlier point in the development flow. This means that one would expect SystemC models to represent early view of the system under development, being subject to extensive modifications as the design matures. Extensive code instrumentation does not mesh well with code that is subject continually to extensive modifications. In addition, code development and validation are often carried out by separate teams, but instrumentation would require verification engineers to "mess up" code developed by designers.

Aspect-oriented programming (AOP) [24], which aims to increase modularity by allowing the separation of cross-cutting concerns, is a natural solution to the instrumentation problem, since monitoring is an obvious example of a cross-cutting concern. Instead of instrumenting the user code directly, verification engineers could be asked to add an appropriate set of AOP directive, and then use `AspectC++` [29] to instrument the model. This approach has two difficulties.

First, it requires verification engineers to develop expertise in aspect-oriented programming and assume responsibility to maintaining a set of AOP directive. Second, some of the exposures rquired to enable full-scale white-box valiation seems to require instrumentation that is beyond the current power of AspectC++.

In this work we describe the tool CHIMP (CHIMP Handles Instrumentation for Monitoring of Properties) that allows the users to specify instrumentation sites using a high-level description language, and then instruments the code by automatically generating AOP advice and then executing `AspectC++`. CHIMP adds a layer of abstraction above manually writing AOP advice, so that users can use a simple declarative language to describe the desired primitives to be exposed, and assign them to variables that hold if and only if the execution pointer is at that location. We show how to expose the values of local module-level data members, even those marked `protected` or `private`, all invocations of a specific function, arguments passed to and returned from user-code functions, and statements being executed, without requiring additional annotations or instrumentation by the user, or requiring the user to generate fairly sophisticated AOP code.

Detecting violations of a property being tested requires constructing a deterministic monitor that uses the exposed values and locations. The user can generate the monitor by hand or using an automated tool. The monitor is integrated with the instrumented code via simple function callbacks implemented in the monitor. In this work we used the automated monitor construction techniques proposed by Tabakov and Vardi [34], but the framework presented here is applicable also to hand-written monitors.

2. Preliminaries

2.1 SystemC

Many contemporary systems consist of application-specific hardware and software, and tight production cycles make it impossible to wait for the hardware to be manufactured before starting to design the software. In a typical system-on-chip [8], for example, a cell phone, there are hardware components that are controlled by software. In addition, many hardware design decisions, for example, numeric precision or the width of communication buses, are determined based on the needs of the software running on them. This has led to a design methodology where hardware and software are co-designed in the same abstract model. The partitioning between what will be implemented in hardware and what will be written as software is intentionally left blurry at the beginning, allowing the designers the ability to consider different configurations before committing a functional block to silicon or C.

SystemC is a system-level design framework that is capable of handling both hardware and software components. It allows a designer to combine complex electronic systems and control units in a single model, to simulate and observe the behavior, and to check if it meets the performance objectives. In the strict sense of the word, SystemC is not a new language. In fact, it is a library of `C++` classes and macros that model hardware components, like modules and channels; provide hardware-specific data types, like 4-valued logic types; and define both abstract and specific communication interfaces, like Boolean input. SystemC is built entirely on standard `C++`, which means that every SystemC model can be compiled with a `C++` compiler. The compiled model has to be linked with a SystemC simulator (for example, the OSCI-provided reference implementation) to produce an executable program.

Software typically executes sequentially, partly because most computer architectures have a single CPU core, and partly because a single thread of execution is easier to manage by the operating system. However, in a hardware system, many components execute simultaneously. For example, when using a cellphone to make a call, we activate simultaneously a radio subsystem that handles two-way communication with the cell tower, a signal processing unit that converts voice to signal and signal to voice, and a display controller that shows details about the conversation on the screen. Simulating such a system in software requires the ability to simulate a large number of tasks executing simultaneously, and is critical for the early stages of the design.

SystemC addresses this issue by providing mechanisms for simulating (in software) parallel execution. This is achieved by a layered approach where high-level constructs share an efficient simulation engine [19]. The base layer of SystemC provides an event-driven simulation kernel that controls the model's processes in an abstract manner. The kernel leverages a concept borrowed from hardware design languages, called *delta cycle*, to give the executing processes the illusion of parallel execution.

SystemC modules are the most fundamental building blocks. Similar to `C++` objects, modules allow related functionality and data to be incorporated into individual entities and to remain inaccessible by the other components of the system unless exposed explicitly. This allows modules to be developed independently and to be reused or sold in commercial libraries [7]. As an example, the skeleton of a SystemC module is presented in Listing 1:

```
SC_MODULE(Nand) {
  // Definitions of processes, internal
      data, etc

  SC_CTOR(Nand) {
    // Body of constructor,
    // Process registration,
    // Definition of sensitivities, etc.
  }
};
```

Listing 1. Skeleton code for defining a SystemC module.

In this code fragment, SC_MODULE is one of SystemC's macros, which declares a C++ class named "Nand". Like any other C++ class, a module can declare local variables and functions. SC_CTOR is another predefined macro that simplifies the definition of a constructor for the module. A constructor of a module serves the same purpose as a constructor of a C++ class (i.e., initializing local variables, executing functions, etc.), but has some additional functionality that is specific to SystemC. For example, the *processes* of the module have to be declared inside the constructor. This is done using pre-defined SystemC macros that specify which class functions should be treated by the SystemC kernel as runnable processes. After declaring each process, the user can optionally specify its *sensitivity list*. The sensitivity list may include a subset of the channels and signals defined in the module, as well as externally defined clock objects or events. Whenever there is a change of value of any of the channels or signals listed in the sensitivity list, the corresponding process is triggered for execution. Listing 2 illustrates these concepts.

```
SC_MODULE(Nand) {
  // Input signal ports
  sc_in <bool> A, B;
  // Output signal port
  sc_out<bool> F;

  // Definitions of processes
  void some_function() {
    F.write(!(A.read() && B.read()));
  }
  SC_CTOR(Nand) {
    // Indicate that this function
    // is a ''method process''
    SC_METHOD(some_function);
    sensitive << A << B;
  }
};
```

Listing 2. A SystemC module of a NAND gate

This code fragment declares two input and one output signals of type bool. The function some_function() implements the expected functionality of the NAND gate. The macro SC_METHOD declares it to be a SystemC process. When triggered, a *method process* executes from start to finish. In particular, a method process cannot suspend while waiting for some resource to become available. In contrast, a *thread process* may suspend its execution by calling wait(). The state of the thread process at the moment of suspension is preserved, and upon subsequent resumption (for example, when the waited-for resource becomes available) the execution continues from the point of suspension. Thread processes are declared using the macro SC_THREAD. Both thread and method processes can define a sensitivity list. Each sensitivity list declaration applies to the process immediately preceding the declaration. The sensitivity declaration at the end of the module indicates that the method process some_function() should be triggered as soon as one of the input signals changes its value.

2.2 Assertion-based verification (ABV)

Monitors (also called "functional checkers" or just "checkers") are used as aids for run-time verification. Typically, a monitor observes the execution of the MUV and issues a warning or terminates execution if the observed behavior deviates from the expected behavior. In cases when deviation is observed, the problem and its source are easier to identify and debug. Furthermore, using monitors automates the analysis of the tests results and allows a large number of random test vectors to be executed without the need for immediate attention by a verification engineer. The disadvantage is that writing and maintaining monitors manually is an expensive and laborious process, and for intricate specifications it is very easy to make mistakes when constructing the monitor by hand.

Automated monitor generation was first proposed by Abarbanel et al. [1] and its importance was recognized immediately by the industry (see, e.g., [17, 26, 31]. The industry also recognized the need for temporal languages that can express properties related to ongoing behavior, such as p must hold until q is true [2]. Assertions are most commonly used to facilitate verification, hence the name assertion-based *verification*. Some authors even prefer using the term assertion-based *design* to emphasize the idea that assertions should be introduced in the earliest stages of the design process [6, 17].

Among the several advantages of using ABV is the modular nature of assertions: each one is a partial specification of the system, and those specifications can be added incrementally, as time permits. Designs with thousands of assertions are not uncommon [6], and the only practical way to build such a set of specifications is to add them to the design and to debug them one at a time.

A further benefit of using ABV in the initial specification of the design is that the assertions allow the verification and the design teams to base their work on a common set of formal properties. This usage of assertions supplements the natural language description of the design, and is an important part of the documentation of the design.

2.3 ABV framework for SystemC

The first requirement for an assertion-based verification framework for SystemC is a formal specification language that can describe the expected behavior of the model's execution. In the past, design specifications have been given in natural language documents [36], but natural language is inherently ambiguous and it is easy to miscommunicate or misinterpret the intended functionality of the design. The language proposed by Tabakov et al. [33] proposes a set of primitives that allow existing specification languages to be extended and applied to SystemC models.

Tabakov and Vardi [32] define a framework for handling all monitors and for activating them at appropriate sample points. They add an abstract class, `mon_prototype`, to the SystemC kernel; all concrete monitors extend this class. Each monitor that we construct in this work defines callbacks that are called from the instrumented code to communicate with the monitor.

In order to keep track of all monitors we need a centralized list. The framework of [32] adds a new object, `mon_observer`, to the SystemC kernel. At instantiation, each monitor registers with the `mon_observer`, which builds a list of all monitors. `mon_observer` provides a function, `get_monitor_by_index()`, that returns a (generic) `mon_prototype*` pointer to any of the monitors. Our implementation takes advantage of this mechanisms to obtain pointers to monitors from the instrumented code and to call the appropriate callback from the instrumented code.

3. Related Work

AOP has been applied to instrumenting Java programs (see, e.g., [5, 10, 20]). The AOP weaver for Java, `AspectJ`, is very mature and widely used. A lot of ideas from Java-based AOP have been transferred to `AspectC++`, which is a `C++`-based AOP weaver, but less mature than `AspectJ`.

Niemann and Haubelt [27] use AOP to expose function calls in SystemC user code, and then check offline the trace against specification expressed in SVA. They associate with each monitored function an atomic proposition that is true iff the function has been called but has not yet returned. Our solution enable deep exposure, for example, both function calls and returns, function parameters and return value, as well eliminates the need to write AOP directives. Our focus here is on *online* monitoring; as soon as an illegal behavior is detected the execution of the model can be terminated, reset to a known good state, or the error can be logged while the model continues to execute.

Endoh et al. [13, 14] propose using AOP join points directly as Boolean atomic propositions. Intuitively, the atomic proposition associated with a join point holds iff the execution of the model reaches the join point. They do not expose internal variables of the SystemC model, nor do they expose parameters of functions or return values of functions. Their approach requires the user to generate the AOP code manually, and they only consider assertions involving sequences of function calls. The instrumentation provided by our tool exposes a richer set of primitives and uses a higher level of abstraction for declaring primitives, from which the AOP advice code is generated automatically by our pre-processor.

Kallel et al. [22] present an AOP-based framework that targets SystemC transaction-level models. They supply abstract AOP directives that exposes executions of TLM methods in a generic way. The user derives concrete AOP directives from the abstract directives (similar to deriving a concrete class from an abstract class in OOP), indicating the particular interface call that needs to be monitored. The abstract AOP directives provided is tightly coupled with the current TLM framework, and will have to be modified to be adapted to changes in the TLM framework. Furthermore, this approach is applicable only to TLM function calls with specific signatures, and does not allow the user to monitor arbitrary function calls. Both restrictions are removed in CHIMP. In addition, the user is not required to write or even understand the AOP framework; instead, all instrumentation is handled automatically by the tool.

Pierre and Ferro [28] describe a methodology for monitoring communication of SystemC transaction-level models. The specification language is restricted to the simple subset of PSL. The approach involves deriving new classes for each channel whose communication needs to be monitored. The derived classes then override the communication functions (e.g., write(), read(), etc) to add a call to an observer which then notifies the appropriate monitor. A limitation of this approach is that the user has to provide a manually derived subclass of the monitored channel with the appropriate function call to the observer, which requires the user to be intimately familiar with the architecture of the verification tool.

Ferro and Pierre [15, 16] describe a prototype implementation for verifying temporal SystemC properties. Only the simple subset of the Foundation Language class of PSL is supported, and application of the methodology is restricted to TLM models. Furthermore, the approach requires the user to associate *manually* each assertion to each callsite.

Déharbe and Medeiros [11] use `AspectC++` to instrument SystemC for metrics collection, injecting different algorithms into processes (e.g., substituting in different cache policies), fault injection, and hardware-level polymorphism. Kasuya et al. [23] adopt AOP to the Jeda programming language to facilitate adding debugging messages and measuring code coverage. Tartler et al. [35] deal with instrumentation of a running program using the AOP paradigm (i.e., *dynamic weaving*). None of these works applies AOP to monitoring.

Maoz and Harel [25] show how to compile Live Sequence Charts (LSC), a scenario-based visual formalism, into AspectJ, thus allowing the translation of inter-object scenario-based system descriptions to code. Their compilation scheme translates each LSC into a *scenario aspect* that simulates a deterministic finite automaton. An important limitation of their work is the need for central coordination of the different aspects at run time. A similar approach is pursued in [30], where Linear Temporal logic properties are translated into deterministic finite automata coded in AspectJ. In contrast, our focus here is only on exposing user-code state. Thus, our framework can be used with manually constructed monitor as well as with automatically constructed monitors, as in [32].

4. Motivating examples

4.1 Division by zero

Suppose that we have the specification "The value of x is nonzero whenever x is used as a divisor. Notice that this specification consists of two independent components: a condition that is required to hold (the value of x is nonzero) and the code location where the condition is required to hold (whenever x is used as a divisor). While the framework described in [32] allows us to generate automatically a monitor for the condition "the value of x is nonzero", it does not address the issue of "wiring" the monitor to the SystemC model. In the monitor presented in Listing 3 the function `callback_loc1()` is the entry point and must be called from all code locations where we divide by x.

```
class monitor {
monitor() {
  // Initial state id
  next_state = 0;
  current_state = -1;
  status = MON_UNDETERMINED;
  //Count number of steps to failure
  num_steps = 0;
  loc1 = false;
} // Constructor
// Entry point into the monitor
void callback_loc1() {
  loc1 = true;
  step();
  loc1 = false;
}
// Single transition of the monitor
void step() {
 if (status == MON_UNDETERMINED) {
  num_steps++;
  current_state = next_state;
  next_state = -1;

  // Calculate the system state index
  int sys_state_index = 0;
  sys_state_index += (loc1) ? (1 << 1):0;
  sys_state_index += (x!=0) ? (1 << 0):0;

  // The next state depends on the current
  // and on the system's state
  switch (current_state) {
    case 0:
      switch ( sys_state_index ) {
        case 1: next_state = 0; break;
        case 0: next_state = 0; break;
        case 3: next_state = 0; break;
      }
    break;
  }
  if (next_state == -1) {
    // Handle user notification, etc.
    property_failed();
  }
 }
} // step()
...
}; // class
```

Listing 3. Automatically generated monitor for the property "$x \neq 0$.

The correct wiring of the monitor is done with the help of CHIMP. The verification engineer creates a configuration file in which she defines all code locations where some property must be checked, together with the conditions that need to be checked at each location (or set of locations). During the tool's execution it parses all declarations in the configuration file and emits automatically generated monitors and automatically generated AOP advices. Next, the verification engineer instruments the source code of the SystemC model using `AspectC++` and the generated advices. Finally, the instrumented model and the monitors are compiled together, producing an executable with embedded run-time monitors.

4.2 Transaction monitoring

Next consider a transaction-level model with the specification "The value of x is nonzero whenever `simple_bus::begin_transaction()` is called". The monitor for this property is substantially similar to the one presented in Listing 3, however in this case the user code needs to be instrumented in a different way: the monitor must be called immediately before all call sites of `simple_bus::begin_transaction()`. The configuration file for CHIMP would need a single line to be changed to switch from monitoring division by zero to monitoring the call sites of the function. CHIMP will emit a different AOP advice without requiring additional information from the user. Notice that the workflow itself remains unchanged and can be built into continuous integration scripts on the build server.

5. User-code primitives

Our approach provides a mechanism for referring to a rich set of user code primitives in property specifications, without requiring the user to instrument the code manually or to write AOP advices. Primitives are declared by the user via a high-level language, and after that they can be used in any of the properties. We use a configuration file to store all primitive declarations, properties, and optional parameters for our pre-processor and the monitor generator. A command-line interface allows the options specified in the configuration file to be overridden. The primitives that can be used are described below.

Exposing function calls Certain assertions need to be checked immediately before a particular function call is made, or immediately after a particular function call returns. The declaration

```
location loc1 ‘‘% bar::foo()’’:call
```

declares a Boolean atomic proposition `loc1` that holds immediately before the execution of the model reaches a call site of a function `foo()` of class `bar`. Similarly, a Boolean `loc2` that holds immediately after the return of the function is declared using

```
location loc2 ‘‘% bar::foo()’’:return
```

EXAMPLE 1. *Suppose that we have a model consisting of two modules: producer and consumer, connected by a channel. The producer defines a blocking call `send()` to push tokens to the channel, and the consumer defines a non-blocking call `receive_nb()` to read from the channel. We want to specify that `send()` remains blocked until `receive_nb()` has returned. We declare the following primitives:*

```
location snd_start ‘‘% producer::send()’’:call
location rcv ‘‘% consumer::receive_nb()’’:return
location snd_done ‘‘% producer::send()’’:return
```

and use them to specify the expected behavior:

```
Always (snd_start -> ( !snd_done Until rcv ))
```

Exposing function execution Exposing the start and end of execution of user-defined functions allows the specification of pre- and post-conditions and is done by the declarations

```
location loc3 ‘‘% bar::foo()’’:entry
location loc4 ‘‘% bar::foo()’’:exit
```

Both the `call` primitive and the `entry` primitive signal that the function `foo()` is about to execute, but they hold in different locations in the user code. The `call` primitive holds at the call site of `foo()`, while the `entry` primitive holds immediately before the execution of the first statement of `foo()`. Similar is the distinction between `return` and `exit`. Another key distinction is that `entry` and `exit` can only be used with user-defined functions. This restriction is motivated by the property language of [33]. To attain generality, library code is treated as a black box, and the state of library objects is allowed to be exposed only through publicly declared interfaces.

Exposing function parameters and return values Exposing the return values of functions allows the specification of post-conditions for functions. The variable `ret` in the following declaration is assigned the return value of `foo()`:

```
value ret ‘‘float bar::foo(...)’’:0
```

This primitive is available for both user-defined and library-defined functions.

Exposing the values of function parameters according to their location in the parameter list allows the specification of pre-conditions without requiring the user to know the name of the actual parameters used in the function body declaration. The primitive declaration

```
value int var1 ‘‘float bar::foo(...)’’:2
```

declares an (integer) variable `var1` whose value is equal to the 2-nd parameter of function `foo()` at the time when the function starts executing. Notice that the function may be defined in a library, but the function call is a part of the user code. Following the framework of [33], we would like to expose the function parameters for both user-defined and library-defined functions. However, due to a limitation of `AspectC++`, this primitive is currently available only for user-defined functions.

EXAMPLE 2. *Referring to the producer-consumer model described above, suppose that if the channel is full, `send()` is supposed to block for a few cycles and then timeout and return `NULL`. We want to assert that if the return value is `NULL` then the channel does not have free space. Here we assume that the channel defines function `num_available()`, and the channel instance is called `my_channel`. We declare the following primitives:*

```
location snd_done ‘‘% producer::send()’’:return
value    ret_val  ‘‘STATUS_T producer::send()’’:0
```

and use them in the property:

```
ALWAYS ((snd_done && ‘‘(ret_val == NULL)’’) ->
    ‘‘( my_channel->num_available() == 0 )’’ )
```

Notice that everything appearing inside quotes is treated as an atomic proposition by CHIMP and is passed through unmodified into the generated monitor. Thus, in the example above, the first occurrence of “`->`” is treated as implication, while the second occurrence is treated as an indivisible part of the proposition `(my_channel->num_available() == 0)`.

Exposing syntax Sometimes it may be desirable to assert that a particular `C++` statement (or a set of `C++` statements) is reached during the execution of the model. In other cases, assertions may need to be checked immediately before or immediately after some statements. This requires exposing the syntax of the user code to the monitoring framework. CHIMP allows the use of regular expressions to specify arbitrary locations in the source code. For example, the primitive declaration

```
plocation loc5 ‘‘/ *a’’:before
```

declares a Boolean atomic proposition `loc5` that holds immediately before the execution of all statements that contain the division operator ”`/`” followed by zero or more spaces, followed by the variable ”`a`”, i.e., the locations where we divide by the variable `a`. The dual,

```
plocation loc6 ‘‘balance *= *.*;’’:after
```

holds immediately after all statements matching the regular expression `‘‘balance *= *.*"`.

Exposing private variables Referring to values of private or protected class variables (i.e., local storage) of modules is critical for white-box validation of models. The declaration

```
makevisible my_class
```

declares a SystemC module or a `C++` class `my_class` fully visible to the monitoring framework and enables references to its class variables in all monitors.

6. Implementation

Our implementation uses the monitoring framework described in [32] to obtain references to the monitors from the instrumented user code. The monitors are agnostic about the semantics of the primitive Booleans used in the property: these primitives are treated as Boolean expressions that determine state change in the monitors. The monitors expect these Boolean primitives to be assigned correct values prior to the execution of monitor steps. In this section we show how the primitives described in Section 5 are assigned values.

Exposing function calls Exposing `location` primitives, e.g.,

```
location loc1 ‘‘% bar::foo()’’:call
```

is done by creating a communication interface between the user code and the monitor, and then instrumenting the user code to communicate with the monitor. The monitor defines a callback function `callback_loc1()` and a local Boolean variable `loc1`. The monitor expects that the callback function `callback_loc1()` will be called from the user code as soon as the execution of the user code reaches the function call `bar::foo()`.

During initialization the monitor sets all Boolean variables corresponding to user-defined `locations` to ***false***. The execution of a callback function `callback_loc1()` consists of the following sequence of steps:

1. The associated Boolean variable `loc1` is set to ***true***;
2. The monitor executes one step; and
3. `loc1` is set to ***false***.

The instrumentation of the user code must call the monitor's `callback_loc1()` function immediately before the function call to `bar::foo()`. Our implementation creates an AOP advice that carries out the communication with the monitor from the user code:

```
advice call("% bar::foo()"): before() {
  // Start new inner scope
  {
    extern sc_core::mon_observer* observer;
    mon_prototype* mp = observer->get_monitor_by_index(42);
    my_monitor42* mon42 = (my_monitor42*) mp;

    // This callback implemented only by my_monitor42
    mon42->callback_loc1();
  }
} // advice
```

Figure 1. AOP advice to expose function calls of `bar::foo()`.

The AOP advice in Fig. 1 uses an inner scope to prevent variable name conflicts. This also ensures that no variable declared during the execution of the advice code will remain in scope after the end of the execution of the advice code. A pointer to the `mon_observer` object `observer` is obtained using its external declaration. This example assumes that the 42-nd property uses the `location` declaration `loc1`. The function call `observer->get_monitor_by_index(42)` returns a pointer to the 42-nd monitor as an abstract `mon_prototype` object (see [32]). It is recast to the type `my_monitor42*` so that the callback function defined by the 42-nd monitor (i.e., `mon42->callback_loc1()`) can be used.

Exposing the locations immediately after the return of a function call is done in a similar way as in Fig. 1, but replacing `before` with `after` in the generated AOP advice. The advice is activated upon the function's return and it calls the monitor's callback function corresponding to the `location` primitive.

Exposing function execution Primitives associated with the start and the end of functions are handled by the monitors in the same way as call and return primitives: the monitor declares a Boolean variable corresponding to the `location` primitive, and this variable is set to ***true*** via a callback. In order to instrument the user code we generate an AOP advice that is activated when the monitored function starts or finishes executing.

As an example, if the user declares

```
location loc3 ‘‘% bar::foo()’’:entry
```

our implementation generates an AOP advice very similar to the one presented in Fig. 1, but replacing `advice call()` by `advice execution()`. This changes the location where the instrumented code is inserted: instead of instrumenting the call site, the advice is inserted at the beginning of the function body. For the declaration

```
location loc3 ‘‘% bar::foo()’’:exit
```

we generate an advice similar to the one presented in Fig. 1, but changing the first line to

```
advice execution(‘‘% bar::foo()’’):after .
```

Exposing function parameters and return values For each monitored `value` primitive `myval`, e.g.,

```
value int myval ‘‘% bar::foo(...)’’:2
```

the monitor defines a callback function `callback_myval(`T `v)`, where T is the type of `myval`. The monitor also declares a local variable `value_of_myval` of type T. The monitor expects that `callback_myval()` will be called upon execution of the function `bar::foo()`.

The function `callback_myval()` sets the value of `val_of_myval` equal to the value of the parameter `v` and returns without running the monitor. The callback function does not evaluate any assertions, following the principle of separation of concerns; the callback serves only as a communication channel to expose the value of the monitored parameter. If an assertion needs to refer to the value of `myval`, that assertion can be triggered and evaluated upon execution of `bar::foo()` using the mechanisms described earlier.

Instrumenting the user code is done by an automatically generated AOP advice. The advice for

```
value int myval ‘‘% bar::foo(...)’’:2
```

is presented in Fig. 2. The advice uses the built-in AOP function call `tjp->arg(`n`)` that exposes the n-th parameter of the function (counting up from 0). `tjp->arg(`n`)` returns a `void*` pointer that needs to be cast to the type of `myval` before it is passed to the monitor via the callback.

```
advice execution("int driver::foo(...)"): before() {
  // Inner scope
  {
    extern sc_core::mon_observer* observer;
    mon_prototype* mp = observer->get_monitor_by_index(42);
    my_monitor42* mon42 = (my_monitor42*) mp;
    int value_to_send = (int) *(int *)tjp->arg(1);
    mon42->callback_myval(value_to_send);
  }
}
```

Figure 2. AOP advice to expose function parameters of `bar::foo()`.

Exposing syntax Declarations of `plocation` primitives, e.g.,

```
plocation loc6 ‘‘balance *= *.*;’’:after
```

are handled by the monitor as `location` primitives: for each `plocation` the monitor declares a callback function and a local Boolean variable. The value of the variable is set to ***true*** by the callback, the monitor executes a step, and the variable is set to ***false*** before the callback returns. Note that in addition to matching syntax, this mechanism can also be used to match code labels (e.g., `reset:`) and pre-processor directives (e.g., `#ifdef`).

Instrumenting the user code to expose statements that match regular expressions cannot be done using the AOP framework. Thus, our implementation checks all user-code files and identifies locations that need to be instrumented, using pattern matching. At each such location we insert code that obtains a reference to the correct monitor and makes the callback. For example, the injected code corresponding to the `plocation` defined above is presented in Fig. 3:

```
// Inner scope
{
  extern sc_core::mon_observer* observer;
  mon_prototype* mp = observer->get_monitor_by_index(42);
  my_monitor42* mon42 = (my_monitor42*) mp;
  mon42->callback_loc6();
}
```

Figure 3. Code fragment injected in the source code to expose syntax

This code is identical to the AOP advice code presented earlier in Fig. 1. However, when using the AOP-based instrumentation the advice code executes with the support of the AOP framework. In particular, AOP allows `C++` header files to be inserted automatically in the instrumented source code files. The AOP aspect that is generated by our implementation carries an `#include` directive that injects the monitor header file in all instrumented user-code files; casting the `mon_prototype` objects to the derived monitor types (e.g., `my_monitor42`) would otherwise be impossible. When injecting instrumentation code without the help of the AOP framework, our implementation injects the required `#includes` in the user code where `plocation` primitives are exposed.

Exposing private variables All modules and channels in SystemC extend the pre-defined objects `sc_module` and `sc_channel`, which are implemented internally as `C++` classes. To expose their `private` and `protected` data members we use `C++`'s `friend` mechanism. Intuitively, a monitored module declares the monitor class as a `friend` class, which gives the monitor unrestricted access to all internal data members. We show how to do this automatically via an aspect introduction.

AOP introductions allows adding new data members and functions to a class [18]. However, AOP does not restrict the advice code that can be weaved via an introduction. Since introductions extend the static structure of classes, an introduction advice can also be used to declare the monitoring class as a `friend` class. Our implementation generates a named pointcut `reveal()`, in respect to which we define the introduction (Fig. 4).

```
pointcut reveal() = ‘‘bar’’ || ‘‘bas’’;

advice reveal() : slice class {
  friend class monitor0;
  friend class monitor1;
  ...
};
```

Figure 4. AOP advice to expose private and protected data members of modules “bar” and “bas”.

7. Experimental evaluation

The input to CHIMP[3] is a configuration file where the user defines the locations where properties need to be evaluated, and the properties themselves. CHIMP generates a monitor for each property [32] and the corresponding AOP advices that trigger the execution of the monitor at the requested locations. Next, we used `AspectC++` to instrument a SystemC model, which was then compiled together with the generated monitors.

We used version 2.2.0 of the OSCI simulator, which was modified using the framework of [32, 34] and compiled using the default settings. We ran all experiments on Ada, Rice's Cray XD1 compute cluster (`rcsg.rice.edu/ada`). Each of Ada's nodes has two dual core 2.2 GHz AMD Opteron 275 CPUs and 8GB of RAM.

We used a SystemC model with about 3000 LOC implementing a system for reserving and purchasing airplane tick-

[3] CHIMP is available for download from `http://www.cs.rice.edu/CS/Verification/Software/software.html`

ets. The users of the system submit requests and 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 to the user for approval, payment and booking. Internally the system uses modules connected by finite-capacity channels. This model is intended to run forever. It approximates actual subsystems currently used in hardware design (see [33] for more details).

We measured performance by simulating for 1 million clock cycles with focus on the cost of instrumentation. The average wall-clock execution time of the system over 10 runs without instrumentation was $\sim$33 seconds. We call this the "baseline" execution. We next added a simple `assert true` specification that is checked at increasing number of locations in the source code. For each experiment we wrote a configuration file containing the specification and declared the locations at which the specification was to be checked. Our implementation generated the corresponding AOP advice and the monitor. The advice was then weaved into the user source code using `AspectC++`. The instrumented code and the monitor code were compiled and executed using the same input parameters as the baseline execution. At the end of execution the monitor reported how many times it had been called, which corresponds to the number of times the instrumentation had been exercised. Since we are using a very simple monitor, any slow-down of the execution is due to the instrumentation.

Fig. 5 presents the number of times the monitor was called and the corresponding execution overhead of the user-code as a percentage of the baseline execution time. We observe a linear increase in the overhead as we increase the number of calls.

Fig. 6 shows the cost of the instrumentation per monitor call, as a percentage of the baseline execution. Our data suggest that there is a fixed cost of the instrumentation, which, when amortized over more and more calls, leads to lower average cost. The average cost per call stabilizes after 300,000 calls, and is less than $0.5 \times 10^{-4}\%$.

8. Conclusion

In this work we described a framework and a tool called CHIMP for exposing a rich set of user-code primitives via automated source-code instrumentation through tool-generated AOP advice and direct instrumentation of source code. The mechanisms presented here are easy to use and do not require the users to instrument the code manually or to be experts in AOP. The user-code instrumentation techniques have already been integrated successfully with the monitoring framework of [32].

One limitation of the instrumentation approach presented is that arguments of functions calls are not exposed if the called function is not defined in the user code. This affects the monitoring of calls of library functions. We expect that future versions of `AspectC++` will include this functionality, thereby removing the limitation from the instrumentation approach presented here.

Acknowledgments

Work partially done while the first author was at Rice University, supported by a gift from Intel.

References

[1] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal. Focs: Automatic generation of simulation checkers from formal specifications. In *CAV'00: Proc. of the 12th International Conference on Computer Aided Verification*, pages 538–542, 2000.

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

[3] R. Armoni, D. Korchemny, A. Tiemeyer, M. Vardi, and Y. Zbar. Deterministic dynamic monitors for linear-time assertions. In *Proc. Workshop on Formal Approaches to Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*. Springer, 2006.

[4] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The BLAST query language for software verification. In *SAS'04: Static Analysis, 11th International Symposium*, pages 2–18, 2004.

[5] E. Bodden. Efficient and expressive runtime verification for Java. In *Grand Finals of the ACM Student Research Competition 2005*, 2005. Winner paper of the Grand Finals.

[6] M. Boulé and Z. Zilic. *Generating Hardware Assertion Checkers*. Springer Publishing Company, Incorporated, 2008.

[7] 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.

[8] H. Chang, L. Cooke, M. Hunt, G. Martin, A. J. McNelly, and L. Todd. *Surviving the SOC revolution: a guide to platform-based design*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[9] 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.

[10] M. d'Amorim and K. Havelund. Event-based runtime verification of java programs. In *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, pages 1–7, New York, NY, USA, 2005. ACM.

[11] D. Déharbe and S. Medeiros. Aspect-oriented design in SystemC: implementation and applications. In *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*, pages 119–124, New York, NY, USA, 2006. ACM.

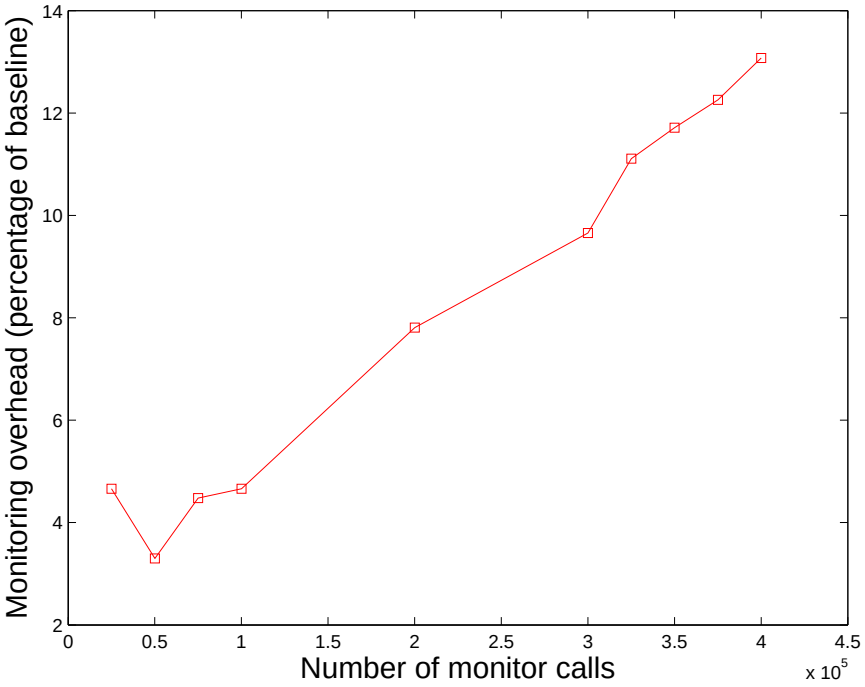

Figure 5. Instrumentation overhead as a percentage of the baseline runtime.

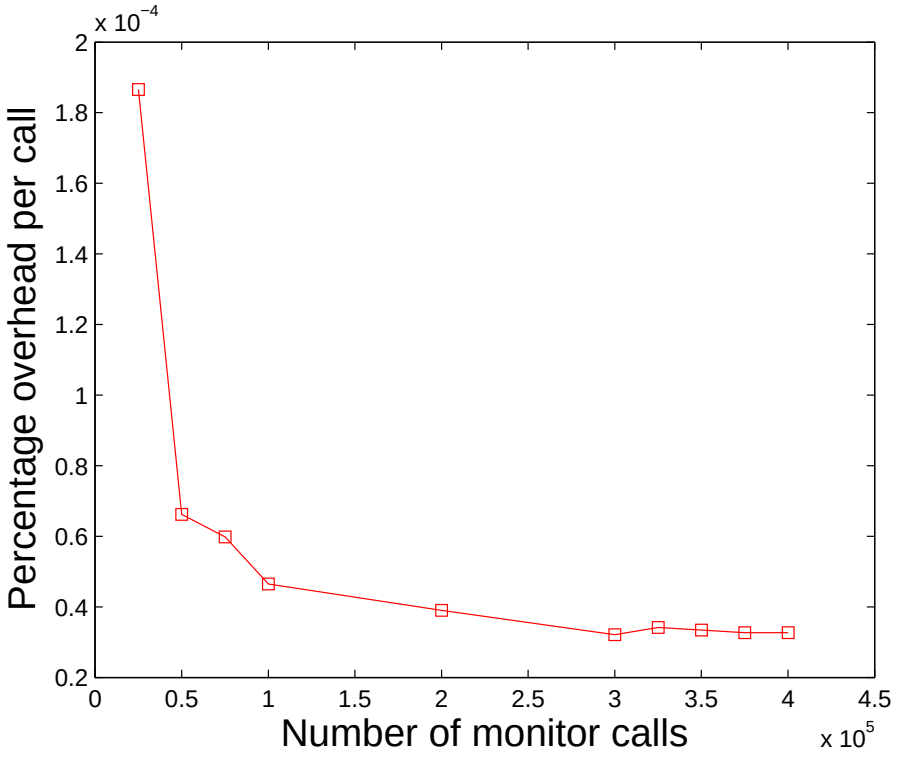

Figure 6. Instrumentation overhead per monitor call as a percentage of the baseline runtime

[12] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer, New York, Inc., Secaucus, NJ, USA, 2006.

[13] Y. Endoh. ASystemC: an AOP extension for hardware description language. In *Companion Volume of the 10th Int'l Conf. on Aspect-Oriented Software Development*, pages 19–28. ACM, 2011.

[14] Y. Endoh, T. Imai, M. Iwamasa, and Y. Kataoka. A pointcut-based assertion for high-level hardware design. In *ACP4IS '08: Proc. AOSD workshop on Aspects, components, and patterns for infrastructure software*, pages 1–6, New York, NY, USA, 2008. ACM.

[15] L. Ferro and L. Pierre. ISIS: Runtime verification of TLM platforms. In *Forum on specification and Design Languages, FDL '09*, pages 1–6, 2009.

[16] L. Ferro and L. Pierre. Formal semantics for PSL modeling layer and application to the verification of transactional models. In *DATE*, pages 1207–1212, 2010.

[17] H. Foster, A. Krolnik, and D. Lacey. *Assertion-Based Design*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[18] A. Gal, W. Schröder-Preikschat, and O. Spinczyk. AspectC++: Language proposal and prototype implementation. In *OOPSLA'01: Object-Oriented Programming, Systems, Languages and Applications*, 2001.

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

[20] K. W. Hamlen and M. Jones. Aspect-oriented in-lined reference monitors. In *PLAS '08: Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 11–20, New York, NY, USA, 2008. ACM.

[21] C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy. Automatic generation of schedulings for improving the test coverage of Systems-on-a-Chip. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, pages 171–178, Washington, DC, USA, 2006. IEEE Computer Society.

[22] M. Kallel, Y. Lahbib, R. Tourki, and A. Baganne. Verification of systemc transaction level models using an aspect-oriented and generic approach. In *5th Intern. Conf. on Design and Technology of Integrated Systems in Nanoscale Era (DTIS), 2010*, pages 1–6, March 2010.

[23] A. Kasuya, E. Hawk, and T. Tesfaye. Verification applications of aspect-oriented-programming (AOP). In *DvCON'04: Design and Verification Conference*, 2004.

[24] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming. In *ECOOP'97: European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[25] S. Maoz and D. Harel. From multi-modal scenarios to code: compiling LSCs into AspectJ. In *Proc. 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 219–230, New York, NY, USA, 2006. ACM.

[26] J. A. Nacif, F. M. de Paula, H. D. Foster, C. J. N. Coelho Jr., F. C. Sica, D. C. da Silva Jr., and A. O. Fernandes. An assertion library for on-chip white-box verification at runtime. In *Proceedings of the 4th IEEE Latin-American Test Workshop (LATW'03)*, Natal, RN, Brazil, February 2003.

[27] B. Niemann and C. Haubelt. Assertion-based verification of transaction level models. In *ITG/GI/GMM Workshop*, pages 232–236, 2006.

[28] L. Pierre and L. Ferro. A tractable and fast method for monitoring SystemC TLM specifications. *IEEE Transactions on Computers*, 57:1346–1356, 2008.

[29] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[30] V. Stolz and E. Bodden. Temporal assertions using AspectJ. *Electron. Notes Theor. Comput. Sci.*, 144(4):109–124, 2006.

[31] Synopsis Inc. Assertion-based verification. White Paper, May 2002. Available online.

[32] D. Tabakov and M. Vardi. Monitoring temporal SystemC properties. In *Proc. 8th Int'l Conf. on Formal Methods and Models for Codesign*, pages 123–132. IEEE, July 2010.

[33] D. Tabakov, M. Vardi, G. Kamhi, and E. Singerman. A temporal language for SystemC. In *FMCAD '08: Proc. Int. Conf. on Formal Methods in Computer-Aided Design*, pages 1–9. IEEE Press, 2008.

[34] D. Tabakov and M. Y. Vardi. Optimized temporal monitors for SystemC. In *Proceedings of the First international conference on Runtime verification*, RV'10, pages 436–451, Berlin, Heidelberg, 2010. Springer-Verlag.

[35] R. Tartler, D. Lohmann, F. Scheler, and O. Spinczyk. AspectC++: An integrated approach for static and dynamic adaptation of system software. *Knowledge-Based Systems*, 2010(in press), 2010.

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

[37] S. Vijayaraghavan and M. Ramanathan. *A Practical Guide for SystemVerilog Assertions*. Springer, New York, NY, USA, 2005.