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

General Terms Performance, Verification

1. Introduction

SystemC (IEEE Standard 1666-2005) has emerged as a *de facto* standard for modeling of hardware/software systems [4], 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 [3]. 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 [3, 8].

Tabakov et al. [8] 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 Property Specification Language (IEEE Standard 1850-2007) and SystemVerilog Assertions (IEEE Standard 1800-2005), allow the sampling of the execution trace with flexible temporal and transactional resolution. The user-code primitives that have to be exposed according to Tabakov et al. 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 [1]. 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. The framework of Tabakov and Vardi [7] enables full white-box validation, but the user code has to be instrumented

* A full version of this paper is available as technical report at `www.cs.rice.edu/~vardi/papers/`

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

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 [7].

Aspect-oriented programming (AOP) [5], 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++` [6] 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 required to enable full-scale white-box validation seems to require instrumentation that is beyond the current power of AspectC++.

In this work we describe the tool CHIMP (CHIMP Handles Instrumentation and 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.

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 [9], but the framework presented here is applicable also to hand-written monitors.

2. Preliminaries

2.1 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 model under verification (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.

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. 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.

2.2 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 [10], 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. [8] proposes a set of primitives that allow existing specification languages to be extended and applied to SystemC models.

Tabakov and Vardi [7] 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 [7] 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.

Due to space limitations, we invite the reader to consult the full version of this paper online for discussion of related work and additional examples.

3. 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.

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
```

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 [8]. 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 [8], 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.

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.

4. Implementation

Our implementation uses the monitoring framework described in [7] 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 3 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()`.

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. Advice to expose 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`.

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.

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()`.

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() {
  { 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. Exposing the 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.

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 [2]. 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. 3).

```
pointcut reveal() = ‘‘bar’’ || ‘‘bas’’;
advice reveal() : slice class {
  friend class monitor0;
  friend class monitor1;
  ...
};
```

Figure 3. Exposing private and protected members

5. Experimental evaluation

The input to CHIMP[1] 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 [7] and the corresponding AOP advices that trigger the execution of the monitor at the requested locations. We used `AspectC++` to instrument a SystemC model, which was compiled with the generated monitors.

We used version 2.2.0 of the OSCI simulator, which was modified using the framework of [7, 9]. 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 tickets. It approximates actual subsystems currently used in hardware design (see [8] for more details).

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

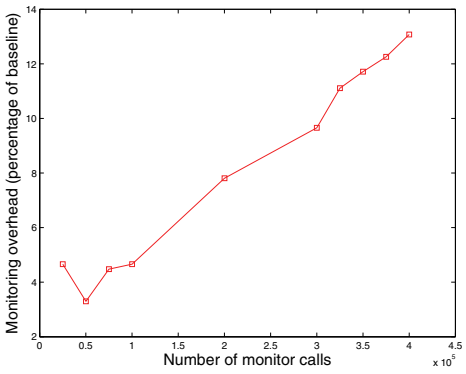

Figure 4. Instrumentation overhead as a percentage of the baseline runtime.

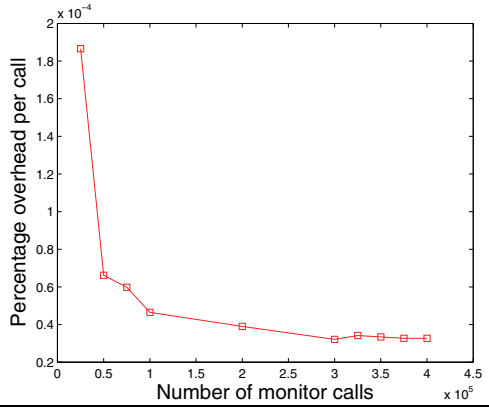

Figure 5. Instrumentation overhead per monitor call as a percentage of the baseline runtime

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 ∼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. 4 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. 5 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 $0.5 \times 10^{-4}\%$.

6. 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 [7].

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] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The BLAST query language for software verification. In *SAS'04*, pages 2–18, 2004.

[2] A. Gal, W. Schröder-Preikschat, and O. Spinczyk. AspectC++: Language proposal and prototype implementation. In *OOPSLA'01*, 2001.

[3] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer, 2002.

[4] 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*, pages 171–178.

[5] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming. In *ECOOP'97*, pages 220–242.

[6] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *CRPIT '02*, pages 53–60.

[7] D. Tabakov and M. Vardi. Monitoring temporal SystemC properties. In *MEMOCODE'10*, pages 123–132.

[8] D. Tabakov, M. Vardi, G. Kamhi, and E. Singerman. A temporal language for SystemC. In *FMCAD'08*, pages 1–9

[9] D. Tabakov and M. Y. Vardi. Optimized temporal monitors for SystemC. In *RV'10*, pages 436–451.

[10] M. Y. Vardi. Formal techniques for SystemC verification. In *DAC '07*, pages 188–192.