A Methodology for Interfacing Open Source SystemC with a Third Party Software

Luc Charest Michel Reid E.Mostapha Aboulhamid
Université de Montréal
{chareslu, reid, aboulham}@iro.umontreal.ca

Guy Bois
Ecole Polytechnique de Montréal
bois@vlsi.polymtl.ca

Abstract

SystemC is a new open source library in C++ for developing cycle-accurate or more abstract models of software algorithms, hardware architecture and system-level designs. SystemC is meant to be an interoperable, modeling platform allowing seamless tool integration. Our objective is to evaluate the feasibility of linking a third party software to SystemC without modifying the SystemC source. We chose the development of a GUI as such an application. This application illustrates a set of applications following the observer pattern defined recently in software engineering. This class of applications can be loosely coupled to a platform designed following specific rules of software reuse.

1. Introduction

In 1999, systems and embedded software companies announced the "Open SystemC Initiative" (OSCI) and availability of a C++ modeling platform called SystemC for free distribution [3][6]. SystemC enables system-level intellectual property model exchange and co-design using a common C++ modeling environment containing a class library and a standard ANSI C++ compiler. Electronic design automation (EDA) vendors have access to the SystemC modeling platform necessary to build interoperable tools.

Our goal is to explore how easy it is to create such tools with as little as possible modification to SystemC. Since SystemC has presently no GUI (Graphical User Interface), we chose such an application as a candidate. Presently, it is possible to generate a waveform trace available only after the simulation has ended and must be viewed by, usually, a tool similar to Synopsys Waveform Viewer [3]. For feedback during the course of the simulation, the user has to incorporate calls to `printf` or `cout` in the design itself. If the user decides to view additional results, he must change his model and recompile the code. It is also possible to use a debugger like GDB, but, although it is a very powerful solution, it might not offer a very intuitive way of viewing the results. One would appreciate an interactive visualization tool with a GUI rather than an aftermath solution.

Our objectives are:

- Since SystemC is constantly evolving, it is desirable to develop a loosely coupled GUI to SystemC so that both can evolve independently but cooperate afterwards.
- Use open source libraries to develop the GUI. This will shorten the design time and allow the GUI to evolve in an open environment too.
- Seamless use of the GUI by the designer meaning that the designer can use the present SystemC models without any new syntax. For example, we could have introduced a new class of signals aware of the interface, but this will change the modeling style of the designer and the resulting model could not be exchanged easily.
- Use a Software Engineering methodology enabling other tools to be coupled to SystemC without hampering its own evolution, and providing a standard framework for such integration.

Section 2 describes the concept of software patterns. Section 3 introduces SystemC architecture. In section 4, implementation issues are discussed. The GUI prototype is described in section 5. A generalization of the use of design patterns is the topic of section 6. Section 7 concludes this paper.

2. Software Patterns and O.O. Paradigms

The software engineering and object-oriented programming communities are advocating the use of software design patterns, which have received a lot of attention since the publication of the book [2] late 1994. Design patterns contribute to the process of software development by providing a common design vocabulary, documentation and learning aids, support for the process of converting an analysis model into an implementation model, and encouragement for reuse of already designed code[4][7][8][12].

These patterns identify, document, and catalog successful solutions to common software problems. Patterns aid the development of reusable software by expressing the structure and collaboration of components to developers at a level higher than source code or object-oriented design models that focus on individual objects and classes.

This work is supported by a GRANT from NSERC Micronet Centre of Excellence

23 preliminary design patterns are catalogued in [2]. Each pattern is described by its intent, motivation, applicability, structure (in an UML-like format[1]), its implementation, a sample code and its known uses. We will concentrate on the *observer pattern*, which fits most our application. Its intent is to define a one-to-many dependency between objects so that when one object (subject) changes state, all its dependents (observers) are notified. The number of observers may vary. All the objects are loosely coupled by using abstraction as described later, allowing evolution of the implementation of each object without affecting the other participants.

Design pattern methodology relies heavily on two object-oriented concepts: abstraction and templates.

To implement abstraction, we usually define a base class, which contains certain methods common to all or some children. We then redefine these methods in children that need to override the common behavior given by the base class method. Such an overriding is called *polymorphism*.

This is done via the keyword `virtual` in C++. If a method is not declared virtual, a call to this method, using pointers, may lead to the base class method, even if the child class overrides it.

The base class can be defined as *abstract* if one or more of its method is declared as pure virtual. A pure virtual method is a method for which no implementation is given and must then be overridden in all children classes. An abstract class cannot be instantiated.

The purpose of templates is to allow programmers to generalize the behavior of a method or a class [5]. When using templates, the programmer does not have to implement the same kind of methods for different types. Only a general statement is given and the compiler analyses the program to know which type is needed. Only needed methods are generated by the compiler following the programmer's template.

3. SystemC Architecture

Understanding the internals of SystemC class library is important if we want to link it to another tool and propose changes to improve its reuse. Figure 1 illustrate its object structure as well as the relation between objects. This structure has been abbreviated from an UML representation of SystemC. Note that italic means that a class is abstract or a method is virtual. A member preceded by "+", "-", or "#" means the member is public, private, or protected respectively. `sc_object` is the base class of most C++ objects. It contains important basic methods and properties for identifying and classifying SystemC objects. `sc_module` is an organizational SystemC object. An instance of this class contains among other objects a list of ports, although this list is private. `sc_port` is an object, which describes the connections between signals. Ports are important structural information contained usually in modules. `sc_port_manager` is responsible for managing all the ports during simulation. `sc_signal_base` is a general abstract base class for all signals types; it cannot be instantiated but all other signal classes are derived from it. `sc_signal` is a template class which enables the definition of signals of different types. `sc_object_manager` is responsible for managing different SystemC objects and contains a private list of them (i.e. modules, signals, clocks, etc.). Although this list is private, objects in it are available through public querying methods.

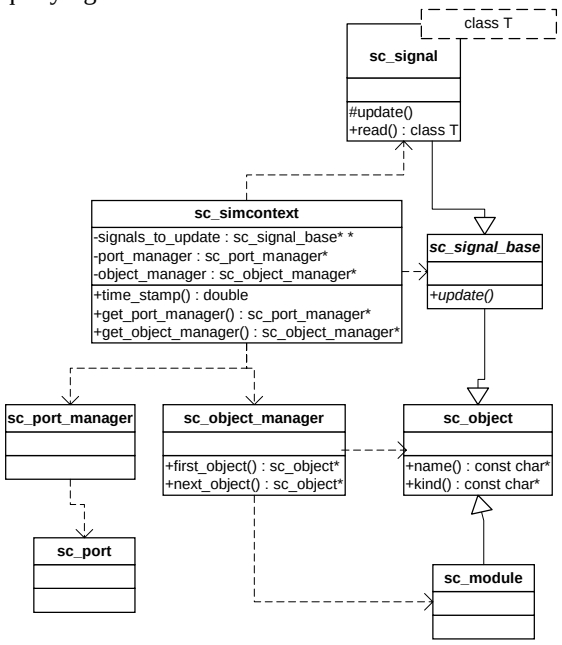

Figure 1: Partial SystemC architecture

4. Implementation Issues

4.1. Prerequisites

Since we want to build a GUI to observe simulation results and control its evolution, we need the signal list so we can insert that list into a menu allowing the user to choose the signal(s) he wants to display. We would also need the value of that signal and the time associated with every signal change. Simulation time is provided by the `sc_time_stamp()` method; however getting signal values is more difficult as it will be seen later.

Obtaining the list of modules and the list of ports would allow us to display the signals according to their associated ports making the output more readable.

We explored the existing SystemC code looking for some of the non-documented methods to query its « internal database » of signals and module structure. The search was unsuccessful since its data is well encapsulated behind protected/private properties. A balance should be stroke between flexibility and reuse on

one hand, and data protection on the other. In our opinion, the implementation we are proposing reaches that balance.

Note that the changes we are proposing is a new class, which is an abstraction, it includes pure virtual methods and some static members.

4.2. Necessary changes to SystemC

In this section we review the changes that we must bring to SystemC in order to allow a loose coupling between SystemC and the GUI. One solution was to implement a method in the `sc_simcontex` class that could fetch the signals and return them as `const` objects. Methods such as `get_first_signal()` and `get_next_signal()` would have accomplished this task. However since `sc_signal` is a C++ template class, it is complicated to keep track of the type of each of its instances. An elegant solution is obtained by abstraction, SystemC designers used an `sc_signal_base` as an abstract class for the `sc_signal` and so they were able to create a list of « `sc_signal_base *` ». When they insert a pointer toward any signal type derived from `sc_signal_base`, its type is cast automatically.

The problem we are facing is that even though we now have a pointer to an `sc_signal_base`, our GUI is not aware of what type the signal is. If we could know the type of signal, we could type cast the signal to its original form, (e.g.: `(sc_signal<bool> *) signal`) and then use any method of the original signal type.

Because SystemC needed to update all the signals using a general pointers list, they defined a `virtual update(void)` in the `sc_signal_base` class and by redefining a `update(void)` in SystemC `sc_signal` class, when this method is called from a generalized pointer, the appropriate `update()` is called according to the `sc_signal` data type and it is the signal responsibility to update itself.

The problem is that the `read()` method, which returns the actual value of the signal is not virtual and is not in the base class. The reason is that at no point, SystemC needs to read or modify the value of the signal. The signal is read and modified by the user who actually knows which type of signal he is dealing with since he created the signal. We would have liked to implement a new `virtual read()`, but this would require virtual template methods, which are forbidden in C++ [11]. We opted for the declaration of a new pure virtual method `notify_interface()` in the `sc_signal_base class`, which must be redefined in derived classes. When SystemC performs its `crunch()` cycle, we added a call to `notify_interface()` for every modified signal. So from there, it is the responsibility of the signal to notify the GUI of any change using this method. Since the signal knows its data type, it is easy to pass the value of the signal with the appropriate data type to the GUI. Figure 2 illustrates the proposed SystemC architecture.

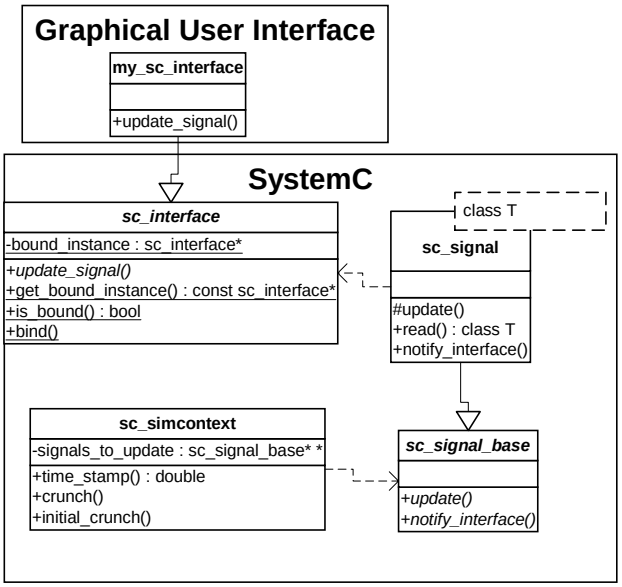

Figure 2: Proposed SystemC architecture

4.3. Constructing the interface

As stated previously, it is the signal responsibility to notify the GUI, but how can it notify the interface? One possibility is sending a message via a method to the GUI. Since SystemC is a standalone library and since we want our GUI to be independent from SystemC, how can we build the SystemC library without having to supply the GUI code to SystemC? Sending a message is usually done by calling a known method of the recipient class. How can we call this method if we do not supply the recipient class?

These questions are answered by following the *observer pattern* defined earlier (Section 2). In fact, we used a special case of this pattern, since our GUI does not modify any signal and gives no feedback to SystemC; furthermore the pattern allows for more than one observer, but we implemented only one. We defined an abstract class called `sc_interface`. We named this class following the general unwritten naming convention of SystemC. This class is abstract because one (in this case all) of its member methods is (are) pure virtual(s), so the class cannot be instantiated. We do not want the class to be instantiated because this class has only one purpose: defining a standard for implementing derived classes. The derived class must implement every method that are pure virtual before it can be instantiated.

Once the interface class is well defined, we can compile SystemC and have it call methods that will be implemented later, in the third party projects. In our case, the `notify_interface()` method of the `sc_signal` class calls the appropriate `update_signal()` of the `sc_interface` class according to the data type. Since we cannot define virtual template methods [11], we were

forced to « unroll » the template by prototyping every method with all possible data types.

4.4. SystemC and the GUI interaction

We have also to define the way the `sc_interface` interacts with its derived class and SystemC. This is due to the fact that we cannot supply a reference of the derived class of `sc_interface` to SystemC since the class is not yet defined when we construct the SystemC library. Therefore we have to provide a pointer to the instance of the derived class, this is accomplished by having a static pointer in the abstract class to point to the instance of the derived class. The static member is common to all instances of the class (and derived classes) and can be used without having any instantiation of the class. We have implemented a method (namely `sc_interface::bind()`) that binds the derived interface to SystemC by setting this static member (`bound_instance`). In our case, the derived class of `sc_interface` is named `my_sc_interface`. So upon instantiation of `my_sc_interface`, the constructor automatically calls the bind method.

When SystemC executes `initial_crunch()` and the `crunch()` loop, a call to `sc_interface::is_bound()` is made, the return value indicates whether the user has attached an interface of his own or not. If a custom `sc_interface` is bound, the message `sc_signal::notify_interface()` is generated, and a call to `sc_interface:: get_bound_instace()` is issued to get the bound interface. The method `update_signal()` of this interface is then called (via polymorphism and inheritance) with the proper data type. Figure 3 illustrates this interaction.

4.5. Cost and benefits

The cost in execution time for the modification we propose is negligible if no interface is bound. In that case the only addition to the simulation is two `if` tests during a call to `sc_interface::is_bound()`, which purpose is to return a boolean value. The first test is made once for each signal in the `initial_crunch()` loop. The second test is made in the `crunch()` loop, each time a signal is updated. Of course, if an interface to a third party tool is bound, the simulation time increases when the control is given to it. Cost in time will mainly be in recording or analyzing the data shipped via the notification. Among the benefits of this approach, is to get rid of the `printf` or `cout` statements, which clutter the model and slow the simulation.

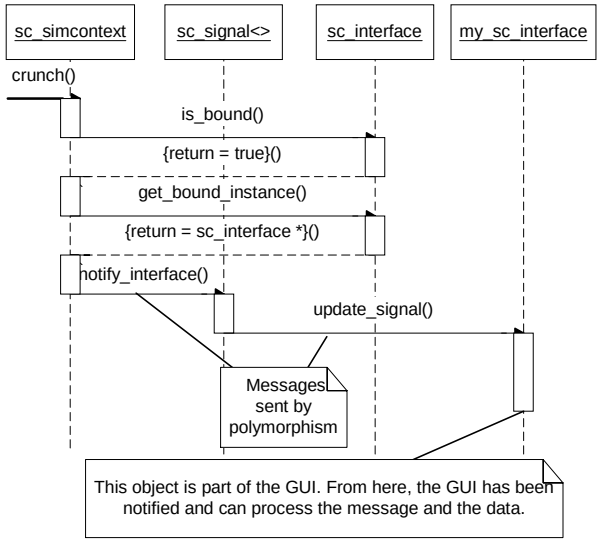

Figure 3: Interaction SystemC - GUI

5. GUI Prototype

In our development, we chose Qt because it is a fully object-oriented, cross-platform C++ GUI application framework providing application developers with all the functionality needed to build GUIs [10]. Qt is available on a wide range of platforms including Linux and it is free for development of free/Open Source software under Unix/X11. Qt offers objects like menus, windows, buttons, etc. We also used the STL library [9] when objects like lists, queues, and vectors were needed. This results in minimizing time for development as well as increasing code efficiency.

The GUI prototype displays Boolean signals traced and listed in windows as in Figure 4. Fundamentals C++ types, such as integers, can be displayed but there is still work to do for the more complex SystemC basic types.

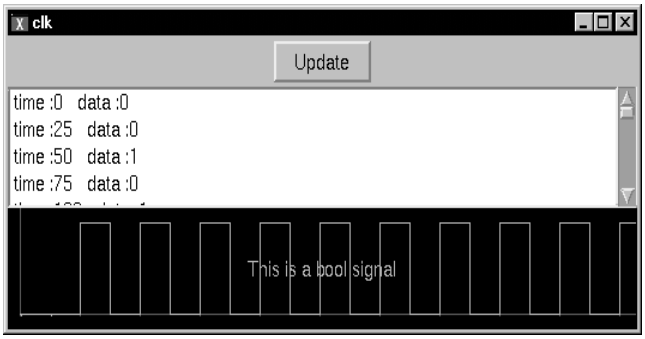

Figure 4: Displaying the signal value

For now, each signal change is memorized only if the user asked to visualize this signal. The simulation speed is not affected by unselected signals. When the user asks for a signal, the recording starts. The user may start, stop or step the simulation at will as shown in Figure 5

The GUI system is compiled independently from the end user's project and is linked as a library. Few modifications must be made to end user's code to create the main window and then pass the execution control to it. If the user wants to be able to debug his signals with meaningful names, he must also attribute a name to each of his signal by modifying his code to use an undocumented SystemC signal constructor.

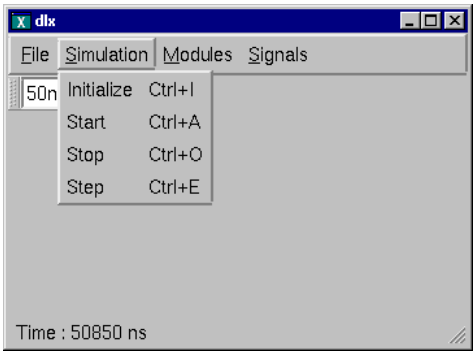

Figure 5: Simulation control from the GUI

6. design patterns to solve more general problems

As an extension of this work, it seems that the `sc_interface` class can be generalized to be used as a facade pattern. A facade is a pattern that provides a unified interface in a subsystem. The facade pattern defines a higher-level interface that makes the subsystem easier to use. The class `sc_simcontext` seems to be a kind of core to the SystemC simulation. We have thought about using the class `sc_simcontext` as a facade by deriving it and using the child class in the same way as `sc_interface` is now used. This implies that `sc_simcontext` would now be playing two roles: a simulation control role and an interface with a third party role. We found it more convenient to separate the responsibilities between two classes, simplifying at the same time the classes structure.

If the `sc_interface` would act as a portal between SystemC and third party applications, this class should be designed carefully to include all the necessary utility methods. This way, a programmer would have a variety of querying, and modifying tools to design his SystemC « plug-in » modules. So our recommendation is to use the `sc_interface` as a kind of two-way facade with SystemC, and provide as many methods as we can. If implementing all the interfacing methods is too overwhelming, « dummy » methods, returning default values, can be defined instead. This would provide a standard on which third party tools may be developed. Programmers of the third party tools could then easily verify that the method as been implemented by checking its returned value. As soon as SystemC implements these methods, all third party tools will have their behavior immediately changed without having to be re-engineered or recompiled.

There is another pattern, « adapter », that enables the communication between two systems, which have different protocols or structures. The `sc_interface` may serve as an adapter, if one desires to adapt an existing tool to SystemC, by deriving and implementing `sc_interface`. If SystemC wishes to modify fundamentally its core, `sc_interface` input could be manipulated to fit the old `sc_interface` standard and a new one could be defined, in parallel, in the same `sc_interface`.

7. Conclusion

SystemC is a step ahead compared to closed systems. It offers a lot of opportunity, as we have shown through this experiment. It is relatively easy to add a class of applications, GUI, testbench development, assertion and verification of properties. All these applications follow the observer software pattern.

If the SystemC community embraces this methodology and the pattern is well defined, well maintained and adequately documented, it could be a powerful framework on which third party software could be built without having to modify the SystemC core or the end user model while keeping the data encapsulated and oblivious to outside manipulation.

8. References

[1] G. Booch, I. Jacobson, and J. Rumbaugh, *The Unified Modeling Language User Guide 1/e*: Addison Wesley, 1999.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison Wesley, 1994.

[3] J. Gerlach and W. Rosenstiel, "System Level Design Using the SystemC Modeling Platform ," Worshop on System Design Automation SDA 2000, pp. 185-189.

[4] S. Krishnamurthi and M. Felleisen, "Toward a formal theory of extensible software," Proceedings of ACM SIGSOFT sixth international symposium on Foundations of software engineering, pp. 88 - 98, 1998.

[5] S. B. Lippman and J. Lajoie, *C++ Primer, 3/e*: Addison Wesley, 1998.

[6] Open SystemC Initiative (OSCI), *SystemC version 1.1 beta documentation*: http://www.systemc.org, 2000.

[7] D. C. Schmidt, "Using design patterns to develop reusable object-oriented communication software", Commun. ACM, vol. 38, 10, pp. 65 - 74, Oct., 1995.

[8] D. Schmidt, "Using design patterns to guide the development of reusable object-oriented software," ACM Comput. Surv., vol. 28, 4, Dec., 1996.

[9] Silicon Graphics Computer Systems, *Standard Template Library Programmer's Guide*: http://www.sgi.com/Technology/STL, 1999.

[10] Trolltech AS, *Qt On-Line Reference Documentation*: http://doc.trolltech.com, 2000.

[11] *Publications by Bjarne Stroustrup*: http://www.research.att.com/~bs/papers.html, 2000.

[12] *Patterns Home Page*, http://hillside.net/patterns/patterns.html, June 1999