Comparisons of Different Approaches of Realizing IP Block Configuration in SystemC

Luc Charest and Philippe Marquet
Université des Sciences et Technologies de Lille, France
{luc.charest, philippe.marquet}@lifl.fr

***Abstract*— SystemC is a quasi open source C++ event driven HDL (Hardware Description Language) reference simulator which was introduced in September 1999 from the OSCI [1]. At first, it meant to be a replacement for VHDL, and, although SystemC can be use for RTL modeling, it is now envisioned by the community as a high level system simulator.**

SystemC inherits all the properties, methodologies and mechanics of its bases (C and C++) which can be seen as macro-assemblers. This has the positive effect of having a lot of freedom in manners of doing things.

This freedom can be beneficial because a given methodology can be chosen accordingly and more appropriately to the situation. On the other hand, freedom has a cost and a designer or IP (Intellectual Property) provider can lose a lot of time trying to figure out which methodology best fit his needs. In this paper, we establish a comprehensive list of all the different mechanisms for configuring an IP in SystemC. We then compare the different methods and highlight the ones which would best suite, following our opinions, the IP development process and publishing cases.

I. Introduction

In order to help an IP (Intellectual Property) publisher to select appropriate configuration methods for their IP, we present in this paper different mechanisms for configuring an IP in SystemC and then compare them.

Most of the methodologies presented in this paper are not SystemC specific. They are derived from C and C++ methodologies and can therefore be directly applied to development of other non-SystemC projects. Some methodologies such as templates, polymorphism and inheritance are C++ specific.

Choosing an appropriate combination of configuration mechanisms, along with their methodology is crucial. Even though it is not implemented with SystemC, one of the best example is SimpleScalar [2], a generic core which uses in one of its implementation the threaded code technique [3] along with a well constructed macro configuration scheme for their machine description, and GNU GCC jump table to speedup the interpretation of the emulated instructions. For their slower but more elaborated simulator, dynamic configurations are used, such as parameters passed through command line.

We have used in [4] command line parameters in order to configure the number of DLX cores inside the same simulation. A simple repetitive control statement (`for`) was used to instantiate and then configure the core, its memory, bridge and interconnect.

There is an ongoing standardization effort; SPIRIT [5] (Structure for Packaging, Integrating and Re-using IP within Tool flows). Its main goal is to achieve automatic configuration and integration of IP through plug-in tools. The specification is loose about the configuration method, it only offers a standard interface between the configuration metadata, the IP configurators and generators supplied by the IP vendor.

In the remaining of this paper, we first explain in depth the compilation flow applied to SystemC in Section II. Then in Section III we present a comprehensive list of configuration mechanisms for a SystemC IP, along with the description of each method. We discuss about the consequences of the methods in more details in Section IV. We present in Section V some indications on when to use each mechanism appropriately. Finally, Section VI conclude this work.

II. The C/C++/SystemC Configuration and Compilation Flow

The major stages in the making of a system simulation are shown in Figure 1. The first step is to create the source files (0.), then to compile the whole project (1.), should it succeed, the generated executable can be launched (2.) and the simulation can then start (3.).

The very first step is (0.0) which is required in some projects and where code gets generated by a tool external to the usual tool chain. It can be anything (Perl, `sed`, Java...). Step (0.1) is when the programmer adds its final touch to the program. The next step (0.2) (that can actually be done in parallel with the previous) is to build a makefile to establish the configuration. Once the makefile has been created, it can be configured and then the compilation process (1.{0-4}) can be launched.

The first part of the compilation phase (1.0) is the preprocessor which resolve the preprocessor directives (macros and every directives that begins with `#`) to produce pure C/C++ code. Typically verbatim replacement and code selection are made. Compilation stage (1.1) is the heart of the compilation process where high level statements are translated into low level assembly code. The assembly code is then translated (1.2) into object code (machine language code). The object code can be put into

libraries (1.3) to be fetched and reused by others. The next step (1.4) is to take all the object code and bind them together into a whole executable.

Once the executable has been generated, one can invoke it (2.0) and pass parameters to realize some configuration. Once the executable is invoked, the first thing the OS loader does (3.0) is to realize the dynamic link by fetching the required dynamic libraries. Typically the `main()` is then invoked, but for SystemC simulation, a call (3.1) to `sc_main()` is issued by `main()`. SystemC elaboration phase can then take place (3.2) where we can read (3.2.0) the configuration from certain inputs (command line, file, user input...). The objects of the simulation (modules, ports, signals...) can then be created (3.2.1) and bound (3.2.2). The simulation can then start (3.3) until a stop is requested.

The IP configurations (represented by asterisks in circles in the Figure 1) can occur during all of the source code creation steps (0.). It can also be made at the end of the compilation phase, when the static link occurs (1.4), when the executable is launched (2.0) via command line parameters and just before the linker loads the dynamic library (3.0). Finally, reading of a configuration file and user interaction (3.2.0) can be done during the elaboration step but module creation, port and signal binding must be finished before `sc_start()` is called (3.3).

III. Mechanism Descriptions

As stated before, most of the methodologies presented in this section are mainly inherited from C and C++ and are not SystemC specific. For each methodology, due to space constraints, we can only show a short and simple illustration of the technique. Here are the methodologies descriptions for configuring a SystemC IP:

Code generation. Foreign file(s) contains different versions of the module. An external program (as Perl or the C preprocessor itself) is used to generate the appropriate version of the module. It can be a complex program that generates SystemC code from a specification written in a foreign language. The whole compilation chain must be re-lunched if configuration is used at this level.

Macros. Typically a `#define` statement is assigned in order to configure the module. There are at least two different approaches of realizing the configuration:

1) A `#ifdef` is issued at the beginning of the module's file which activates a first version while it deactivates the second version of the module. 2) Instead of selecting large pieces of code, the processor directives are placed at strategic points, to increase common code reuse.

Although the second solution emphasis on reuse of common behavior and structure of the program, it has the perverse side effect of cluttering the code with directives obfuscating the original code.

```
#define 32_BITS_PROCESSOR
/* (...) */
#ifdef 32_BITS_PROCESSOR
  Processor_32 proc;
```

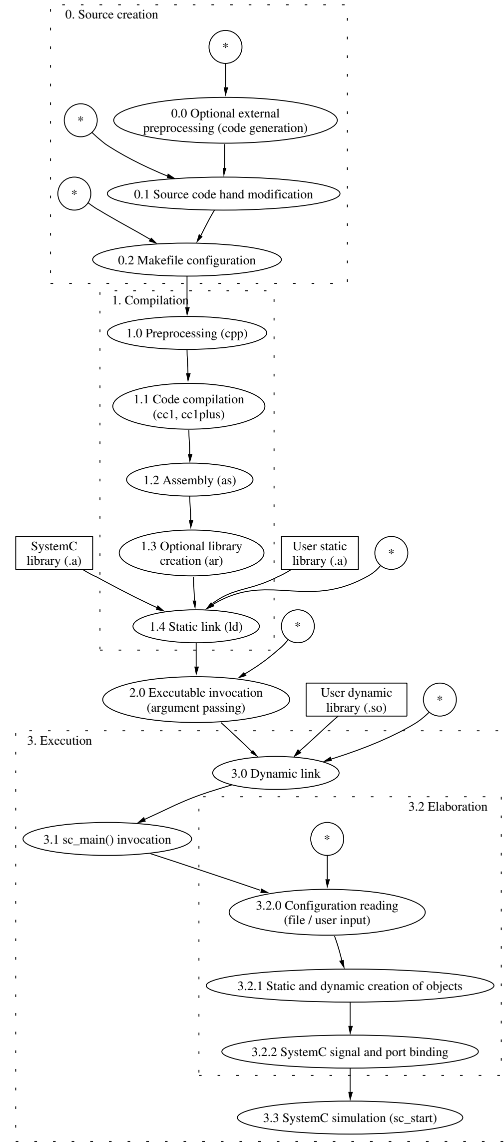

Fig. 1. C/C++/SystemC compilation and execution flow. Steps are in ellipses, configuration points are asterisks in circle, libraries are in rectangles.

```
#else
  Processor_16 proc;
#endif
```

Link. Different versions of the module (or IP) are first compiled using different configurations. When a another configuration of a module is required, since it is already compiled, there is no need to recompile; it is only necessary to link again. As for example, here we invoke the linker (`ld`) via the GNU compiler (no compilation is performed, only the static link) with the precompiled 32 bits version of the processor:

```
g++ -o my_exec proc_32.o main.o -lsystemc
```

Static link is realized during the compilation phase while dynamic link is made when the OS loads the executable.

Templates. Native types of the C++ language (`int`, `bool`, `float`...) are not common rooted as in Java and

templates were introduced mainly to overcome this problem. Templates can be use to generate diverse versions of a module with different types for ports and attributes. The mechanism even permits to configure a module by passing a constant value via parameter. The compiler generates one version of the template construction for each different invocation.

Because templates are resolved during compilation phase, it brings at least three important consequences:

1) Static optimizations are possible. 2) Source code must be modified; therefore the whole compilation process has to be re-run to achieve the configuration. 3) Since the compiler realize a specific version of the template only upon request, the source code of the implementation of the template, which is usually put in header files, must be supplied.

For example, configuring a template version of the processor instantiated with 32 bits might be as simple as:

```
Processor<32> proc;
```

Control statements. Can be used to select different behaviors of the program. The statements `if/else` must be used with consideration because it has at least two perverse effects:

1) Just as with the macros, it has the tendency to clutter and obfuscate the source code. 2) Both part of the `if/else` statement will be compiled, unless the compiler can prove that one of the `if/else` branch will never be taken, that we are in a rare case where the condition is always true or always false. If this strategy is used unconsciously all over the code, the size of the generated executable will grow uselessly and the next method ("Inheritance / Polymorphic") should be considered.

```
if (proc_size == 32)
  mem_store_32(reg_32[i]);
else
  mem_store_16(reg_16[i]);
```

The good point of this approach is that it is dynamic (after the compilation phase has been done), therefore the user can decide at the last moment of the configuration. Repetitive `for` and `while` can be used to configure an IP with multiple compound units.

Inheritance / Polymorphism. Of all the solutions, this one corresponds more to the modern techniques of software engineering. Typically a software interface of a module is built by creating a base class. Different versions of the module can then be implemented over the base definition, by inheritance. This establishes a kind of contract between the module and all other classes that have to interact with this module, should it be another module or a control class. Polymorphism is the mechanism that permits to treat all the versions of the module in a general manner, according to the contract of the base class. The mechanism then chooses dynamically (during execution phase) which version of a given method to call according to the real underlying nature of the object instance.

Although this is the case for all object oriented language, the major drawback is that since it is a dynamic mechanism as the `if/else` statements, static optimizations are not possible on the polymorphic methods of the class. However, the major benefits of this technique are that, as for the `if/else` statement, it permits a late configuration (during execution phase, preferably before simulation has started) therefore, there is not necessarily the need to recompile everything to change the configuration of the module.

For our example, we have two version, a static one:

```
Processor_32 proc();
```

...and a dynamic one:

```
Processor *proc;
if (proc_size == 32)
  proc = new Processor_32();
else
  proc = new Processor_16();
```

The static one has to be fixed before compilation phase, in the source code. It can not be changed dynamically but this can lead to better optimizations since the compiler is not using the polymorphic mechanism. The dynamic object is determined at execution stage but each call to a `virtual` method will lead to the use of the polymorphic mechanism which cost an indirection.

Combining Mechanisms. All mechanisms presented in this section are far from being mutually exclusive. At the cost of code cluttering, here we show an example where the choice of static or dynamic configuration is made during the compilation phase with a simple definition:

```
#ifdef IP_STATIC_CONF
  #ifdef 32_BITS_PROCESSOR
    Processor_32 proc;
  #else
    Processor_16 proc;
  #endif
#else
  if (proc_size == 32)
    proc = new Processor_32();
  else
    proc = new Processor_16();
#endif
```

If the preprocessor finds a defined `IP_STATIC_CONF`, the appropriate processor is immediately selected during the preprocessor stage. Otherwise, the IP configuration is postponed until the execution.

IV. Properties and Consequences of Methodologies

Based on the compilation flow presented in Section II we can now present the consequences of selecting a methodologies rather than an other (see Table I).

Regeneration of source code. Only a mechanism which generate the configured source code will suffer from this extra step which can not be made in parallel because the compilation phase depend on it.

Recompilation. It is necessary only for those configuration mechanisms which modifies the source code (as source generators and hand modifications), the makefile or its configuration.

TABLE I

Properties and Consequences of Methodologies

Methodology	Solution Nature	Optimization on configuration	Common code reuse	Code cluttering	Required in package
Code generation	Static	High	(dependently of the external tool, likely)		External tool + foreign source code
Macros	Static	High	(if code reused, directives clutters the code)		Source code + makefile
Static link	Static	High	(not implicit, typically no)	Typically no	Object file (`.o`) or library (`.a` or `.lib`) + header files (`.h`) + makefile
Templates	Static	High	Yes	No	Implementation code (`.h`)
Inheritance	Static	High	Yes	No	Source code + makefile
Dynamic link	Dynamic	Moderate	(not implicit, typically no)	Typically no	Library (`.so` or `.dll`)
Control sturcture	Dynamic	Low	(if code reused, directives clutters the code)		Executable
Inheritance / polymorphism	Dynamic	Low	Yes	No	Executable

Relink. It has to be done again when the source code has been recompiled or if precompiled object files where switched.

Static and dynamic solutions. Satic are everything that take place before execution (regeneration, hand modifications, makefile reconfiguration…), while dynamic take place during execution (file reading, user input, dynamic instantiation…).

Optimization on mechanism. Usually the more the compiler "knows", the more it can optimize. This means that static solutions have a better chance to be optimized than a dynamic one. It is avantageous to seek optimizations if compilation time is negligible in relation with simulation time.

Code cluttering *vs* reuse. Macros and `if`/`else` statements usually offers alternatives at some point between different solutions. When using large piece of code inside these statements, it prevent code cluttering but it also prevent code reuse.

V. Utilization

A. IP Development

When working at development stage, IP configuration can become a bottleneck of development. An automatic configuration process is better than one which requires the programmer's intervention for debugging. The developer will often have to recompile because he made heavy modifications on the source code. Therefore, reconfiguration at this stage or the previous stages is not really a problem. Typically, the ideal situation is when a script can be made which can test some or all possible configurations and then return a fail/success status recorded inside a single log file.

B. IP Deployment

At the deployment stage, there is often not the need to recompile anything. A static or dynamic library can be given to the end user which then either configure the IP with the makefile, during execution with an external configuration file or with keyboard interaction. Interaction with the end user can be more appropriate in this situation because the end user could want to do manual configuration. Several configuration system can be supplied and enabled or disabled via macros or argument passed via the command line.

C. Source Distribution and Pseudo Security

If the IP provider does not want to give away the source code of his IP, he should select a method which is more dynamic, one which is done during execution and which does not imply compilation which often requires some or all of the source code. It is clear that a program in its compiled form will, at least, not reveal as much information as the source code.

D. Architectural Exploration

When the IP is mature enough, one can consider doing architectural exploration, be it the IP conceptor or the IP user. A dynamic configuration is often prefered because it saves the compilation phase.

VI. Conclusion

We presented an exhaustive list of mechanisms for configuring an IP in SystemC. We highlighted that during the development phase, the configuration is not really a matter since recompilation is made often. However, it should be made as automatic as possible to reduce the time spent in the debugging process. For IP deployment, static libraries and dynamics mechanisms are preferred because they permit a lot of flexibility while restraining the IP leak. As for future works, we will follow SPIRIT's evolution.

References

[1] Open SystemC Initiative (OSCI), "SystemC website," 1999, http://www.systemc.org.

[2] T. Austin, "SimpleScalar hacker's guide," http://www.simplescalar.com/docs/hack_guide_v2.pdf.

[3] J. R. Bell, "Threaded code," *Commun. ACM*, vol. 16, no. 6, pp. 370–372, 1973.

[4] L. Charest, E. M. Aboulhamid, C. Pilkington, and P. Paulin, "Systemc performance evaluation using a pipelined DLX multiprocessor," in *Design Automation and Test in Europe Designers' Forum*. Paris, France: IEEE Computer Society, March 2002, pp. 8–12.

[5] SPIRIT Schema Working Group Membership, "SPIRIT-user guide v1.0," December 2004, http://www.spiritconsortium.com.