A fast SystemC simulation methodology for multilevel IP/SOC design

by Samy Meftali, Joel Vennin and Jean-Luc Dekeyser, LIFL
{meftali, vennin, dekeyser}@lifl.fr
Lille, France

Abstract:

In this paper, we present a fast method which allows connecting together SystemC modules. These modules may be specified at different abstraction levels, and we obtain an executable simulation model of the whole system. The originality of our work is the fact that it does not require SystemC external libraries. Indeed, the simulation model generation is based only on: modules specifications, a SystemC added library and a simple rules description language. Thus with our methodology, we generate simulation module adapters having a simple structure, in a fast and automatic way, by instantiating a generic configurable SystemC class.

1. INTRODUCTION

Generally before obtaining a SoC on silicium, systems are specified at several abstraction levels (functional, logical, RT,..etc), each specification is called a model. Any system design flow consists in refining, more or less automatically, each model to obtain another, starting from a functional model until Register Transfer level (RTL) model.
So each design tool must contain simulation engines in order to validate each model before refining it [Mef02]. But, as complex systems are mainly designed by assembling existing components (IPs), designers want sometimes to just add some functionality to a system constituted by a set of already existing IPs to obtain scalable SoSs. In the case that this initial system is available only at RT level for example, it can be very benefic, in terms of design time, to design only the part to add to the system from the functional level, and not to specify again all the application at high abstraction levels. However, a multilevel simulation platform will be needed to validate the whole system (the part to add at a high abstraction level, and the old application at a low level). This constitutes the objective of our work.

This paper is organized as follow: in section 2, we present some necessary basics to understand this work. The related works are presented in the section 3. Section 4 presents the different steps of our multi level simulation model generation. Our approach is applied to a system containing an SDRAM described at RT level and a processor module described at UTF level in the section 5. We conclude this paper in the section 6.

2. BASICS

The multi level simulation methodology that we present in this paper concerns exclusively systems described entirely in SystemC. We present in this section some basics necessary to understand how our approach works.

2.1. Abstraction levels

SystemC design flow allows description of system modules (IPs) mainly at four abstraction levels [Gro02]. The general characteristics of each abstraction level are described in the following paragraphs.

2.1.1. Untimed Functional Level (UTF)

At this level a model is similar to an executable specification, but no time delays at all are present in the model. Shared communication links (as buses) are not modeled at UTF level. The communication between modules is point-to-point, and usually modeled using FIFOs with blocking write and read methods.

2.1.2. Timed Functional Level (TF)

A TF model is similar to UTF one in that communication between modules is still point-to-point, and there is no shared communication links. However, at this abstraction level timing delays are added to processes within the design to reflect timing constraints of the specification and also processing delays of target architecture.

2.1.3. Transaction Level

In a transaction level specification, communication between modules is modeled using function calls. At this level the communication model is accurate in terms of functionality and often in terms of timing. For example in a SoC transaction level specification, we may model the different types of transactions that the on-chip bus supports, as burst read/write transactions, but we don't model the pins of the modules that connect to the bus.

2.1.4. Register Transfer Level

It is the lowest abstraction level in SystemC systems design flow. The internal structure of an RT level model accurately reflects the registers and combinatorial logic of the target architecture. The communication between modules is described in details in terms of used protocols and timing. Each module's behavior corresponds exactly to a physical component behavior. The data types used at RT level are mainly bits (or bit-vectors).

2.2. Multi level simulation

A multi level simulation model is an executable specification containing a set of modules described at different abstraction levels. The problem in such a model is how to connect automatically and with low-cost two (or many) modules communicating each one using different data types via different communication protocols. Figure 1 shows an example of a system containing two modules: module 1 and module 2. The first one is specified at UTF level and communicates throw 3 ports using blocking FIFOs with integer data, the second one communicates throw 11 ports using several protocols (Master-Slave, Hand-shack, ..etc) with bit and bit vector types. Thus the problem is how to convert data types and protocols to allow the communication between these two modules.

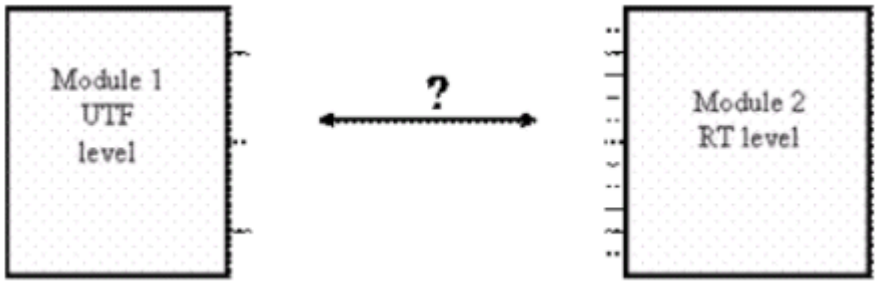

Figure 1: Multilevel simulation problem

3. RELATED WORK

Recently many academic and industrial research teams worked on the multi level simulation problem in embedded systems. This type of simulation implies to execute together models of the system component described at different abstraction levels. Among the proposed solutions, the Bus Functional Model (BFM) [Sem00] constitutes the conventional methodology to interconnect functional simulation models and cycle accurate models, especially to validate software/hardware interfaces. Unfortunately this methodology takes into account only memory accesses, but it doesn't at all allow transformation of high level communication primitives (FIFO for example).

CoWareN2C [Cow02] is an environment which offers a multi level cosimulation solution. It allows the use of two abstraction levels: BCA (Bus Cycle Accurate) which is closely similar to RTL and UT level (without timing references) Thus CoWareN2C presents a concept called BCASH, It is a wrapper of the sub-systems described at UT level allowing the estimation of their sub routines execution time. This wrapper can be automatically generated only in if the sub-system at UT level is targeted in software.(that means that the sub-system will be executed on a processor simulator ISS "Instruction Set Simulator"). This is unfortunately a very strong constraint.

The concept of conversion interface is present in SystemC [Sys]. It may permits the connection of modules communicating by Remote Procedure Calls (RPC) with other module described at RT level. This interface must be entirely hand coded by the designer and this may be of course very time consuming and error prone.

We can find some other multi-level simulation environments as Chinook [Cho95] focusing on the dynamicity while changing modules abstraction levels.

VSI Alliance [Vsi] works on heterogeneous components assembling. It presents a specification and documentation standard, at different abstraction levels and also a bus interface standard. But it doesn't focus yet on the simulation adapters' generation. Thus COSY [Bru00] is a generic model which allows the interconnection of IPs via VCI bus model interfaces, but it still focus on the system level.

Some very recent works, in the literature, treat the problem of automatic generation of multi level simulation models for heterogeneous multiprocessor SoC.

The work described in [Nic02] is one significative example. It permits to generate automatically simulation wrappers to adapt modules abstraction levels to the simulation level. Unfortunately, it doesn't target a specific class of application, and the wrappers are constructed by assembling basic components from external libraries. The structure of these simulation wrappers seems to be complex because of the important number of SystemC components (processes) present in each instance.

Our contribution is the proposal of a new methodology to validate SoCs by simulation. With this approach we can perform a fast and low cost simulation on systems constructed by assembling IPs. The IPs can be described at different abstraction level. The main originality of our approach is that it does not need any external SystemC libraries. It uses only internal SystemC libraries and a small rules description language.

4. METHODOLOGY

Our simulation models generation flow generates a simulation module adapter for each module described at an abstraction level different from those on which we want to perform de simulation of the system. If we have for example a SoC composed by two modules: module 1 specified at UTF level and module 2 specified at RT level, and we want to simulate it at the RT level, our methodology generates automatically a simulation module adapter for the module 1 in order to adapt it's interface with those of module.

4.1. Simulation module adapter's structure

Our simulation module adapter is constituted mainly by three parts: real ports, logical ports and an infinite SystemC process as shown in the Figure 2. These parts are described with details in the remaining of this section.

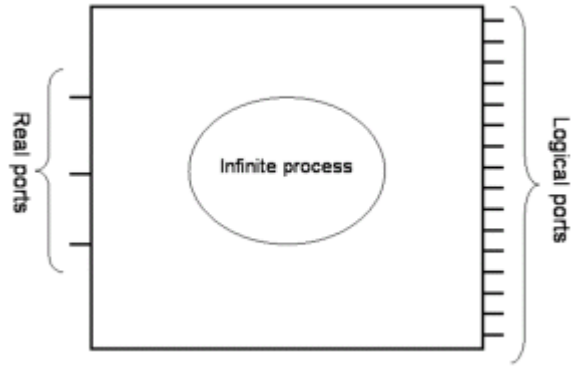

Figure 2: Simulation module adapter's structure

4.1.1. Real ports

They correspond to the initial module ports. They have exactly the same characteristics of the real module's ports (data types, communication protocols,..).

4.1.2. Logical ports

A simulation module adapter must have the same number of logical ports as the number of real ports of the module with which it will be connected. Each one of these logical ports must have identical characteristics as the real port of the module with which it will be connected.

4.1.3. Infinite process

It is an automatically generated SystemC process, by instantiating and configuring a generic SystemC class that we defined. It is sensitive to the signals connecting it to the real ports. This process reads data from the real ports, converts them to be adequate with the logical ports types.

The automatic generation methodology of the simulation module adapters is presented in the remaining of this section.

4.2. Multilevel system simulation model generation flow

Having the interface of each one of the modules to be connected, our flow produces a multi level simulation model as shown in the Figure 3.

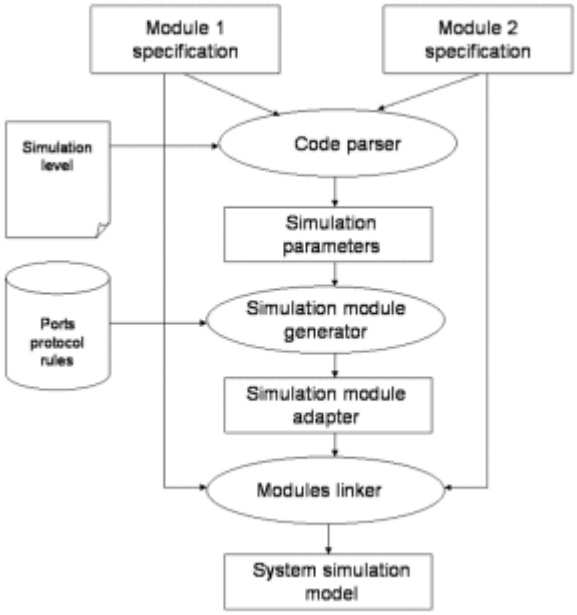

Figure 3: Multi level system simulation model generation flow

Our simulation module adapters are not constructed by assembling basic components for external libraries as in almost literature tools, but it is generated by rules composition. In our simulation models generation flow all simulation module adapters are instances of a SystemC class that we created and called “generic_interface”.

The generic interface is a configurable SystemC class. It is composed, as all SystemC classes, by an interface file (.h) and an implementation/behavior file (.cc). The interface file defining the ports characteristics is automatically configured from the initial system specification, and the behavior file is constructed and configured by the composition of a set of rules. The rules composition is described in details in this section.

We distinguish mainly three engines in our simulation generation flow (Figure 3).

4.2.1. Code parser

The code parser engine takes the SystemC interface files (.h) of the modules to be connected, and the simulation level from the designer. It produces the simulation parameters. These parameters are mainly the number of the logical and real ports, the communication protocol of each one of them and the simulation level at which the designer want perform the system simulation.

4.2.2. Simulation module generator

After obtaining all the simulation parameters (modules interfaces and simulation level), the simulation module generator engine instantiates and configures a generic class to obtain a SystemC module adapting a given module to the simulation level. All necessary simulation adapters are produced by instantiating GenericInterface. This makes the final simulation model’s structure simple and its generation easy.

A module adapter is composed by an interface (describing the ports of the module adapter) and a behavior files (a SystemC infinite process describing the behavior of the simulation module adapter). The first one is produced using association rules and the second one using mainly transformation rules. These two kinds of rules are described in the next paragraphs.

Interface file of the simulation module adapter

The interface file (.h) of a simulation module adapter is generated by instantiating a port

class for each real/logical port.

For instance the following statement in the interface file of the module simulation adapter of the module processor creates an input port of type integer (32 bits):

processor.add_port(create_port<sc_in<sc_int<32 > > >);

The statement creating an output port on the simulation modules interface is similar to the proceeding one. Thus only the direction (output /input) and the data type will be changed. So an output port of type bit is created by the following statement:

processor.add_port (create_port <sc_out<sc_bit > >);

add_port and create_port are two methods that we added to our SystemC library in order to simplify and to automate the code generation.

Association rules

These rules are a set of basic primitives allowing performing read/write operations on the ports. Thus generally when a module is specified at transaction level or at RT level the communication is more or less explicitly modeled, which is not the case for modules described at UTF or TF levels. So when we have for example to simulate at RT level together in a system: a module where the communication is explicitly modeled (RTL) and another module at UTF level, and we want to simulate the system at the RT level, we must use these rules to associate these different ports.

The Association rules are some simple rules describing the ports association between the system modules. Thus, for instance a memory described on RTL level communicates via a certain number of control ports and sets of address and data ports, a processor described at level UTF reaches simply a memory with an address port and a data port both of type integer, and possibly some control ports (WriteEnable for example). To simulate at RT level a system containing these two modules, the association rules allow associating the processor data port (integer) with the set of the memory data ports the memory for instance. Assuming that the processor data port is called PDP and the memory ports are respectively called PMD0, PMD1, ..PMD15, the association rules will be expressed simply by the expression:

PPD – PMD0
– PMD1
– PMD2
– …
– PMD15

This means that the ports PMD0 to PMD15 receive their data from the port PPD.
For instance, the following statement is generated from an association rule. It associates an integer (32) real port with a logical (bit vector (32)) one in the simulation module adapters interface.

m1.add_trans_in (TRANS_GENE (sc_in<sc_int<32 > >, sc_out <sc_bv<32 > >, sc_bv<32 >, m1[0], m1[1]));

TRANS_GENE is a macro defined in file (Association.h). It manipulates the defined association rules.

Behaviors file of the simulation module adapter

The simulation module adapters behavior is an infinite SystemC process sensitive to all (or a set of) its own input ports (real ports). For instance, the following SystemC statement makes m1's behavior sensitive to its input ports:

m1.set_process_in ();

Where m1 is a simulation module adapter, and set_processe_in() is a defined method making a module sensitive to any changes in its input ports.
After that we may specify a specific set of sensitivity ports to the simulation module adapter. For example the statement:

m1.set_event (((sc_in<bool >)m1[2])->default_event ());*

makes the module m1 sensitive to its second input port (m1[2]) which is Boolean. So, the behaviors infinite process of m1 will be executed at each new value on this port. The behavior is then generated using transformation rules.

Transformation rules

As the preceding ones, they are quite simple rules. They mainly define the protocol transformation between the system modules. Thus, in the example of the memory and the processor, a change of value on the processor data port implies a write operation on some memory data ports in a quite precise order (the memory access protocol is provided by the memory vendor).

The following example of a transformation rule means that when the processor writes a data on its data port, three ordered write operations must be performed on the memory ports. Thus, the simulation module adapter must writes data on the memory control ports PMC0 and PMC1 in the same cycle, followed by a write operation on the memory data ports PMD0 to PMD15 in the next cycle and finally a write on the control port PMC3.

PPD – PMC0, PMC1
- *– PMD0,*
- *– PMD1,.., PMD15*
- *– PMC3*

This rule for instance, configures the simulation module interface of the processor module as follow:

```
generic_container * gc = new generic_container;
…
gc.add_trans (TRANS_PUT_VALUE (sc_out<sc_bit>, 0, 1, m1[18]));
gc.add_trans (TRANS_PUT_VALUE (sc_out<sc_bit>, 1, 1, m1[19]));
gc.add_trans (ADD_WAIT (2));
gc.add_trans (TRANS_GENE_SET (sc_in<sc_int<32> >, sc_out<sc_bv<32> >, 0, 15,
m1[0], m1[1]));
gc.add_trans (ADD_WAIT (2));
gc.add_trans (TRANS_PUT_VALUE (sc_out<sc_bit>, 3, 1, m1[21]));
...
m1.add_trans_in (gc);
```

Where *TRANS_PUT_VALUE* is a method allowing to set a giver port, on a fixed value.

Note. For the moment we are specifying the rules simply in a text format, but we are working on a specific formalism and language to define them, in order to make their definition and reuse simpler and easier.

4.2.3. Modules linker

This last stage consists to connect the each generated simulation module adapter with the module with which it is associated, then to connect the different modules together to lead finally to an executable simulation model allowing simulating the whole system.

5. APPLICATION

We want to validate by simulation a SoC containing a transmit module specified at UTF level and an SDRAM (Synchronous Dynamic Random Access Memory) described at RT level. The memory is an existing physical module produced by Micron [Mic] under the reference MT48LC16M16A2. It’s a 256 Mbits memory, containing four bancs of 64 Mbits. Each one of the blocs is characterized by 8192 lines and 512 columns. Memory words are 2 bytes.

5.1. System modules specification

In order to simulate this system, we first specify the two modules at their respective abstraction levels (see table 1).

5.1.1. Functionality of the memory

The SDRAM is an existing module and its functionality and composition are specified by its vendor. It is composed by several parts; the most important ones are: the control logic, lines, columns and banc decoders,

the input/output registers, mask signals, the refresh counter and the clock signal..

5.1.2. Ports types

The communication interface of the SDRAM is defined by the vendor. Thus, in order to connect it with other modules we must respect exactly this interface (number of ports, ports senses and ports types).

Note. The time needed for each operation on the SDRAM is given by the vendor in the form of a low and up bounds for each one. In our implementation of the memory, we choose the average of the two bounds for each operation time.

5.1.3. SDRAM Commands

To send any command to the SDRAM, the processor must send signal on a set of ports. These signals are specific to each command. The signals associated with each command are given by the vendor. Thus for instance, in order to perform a write operation on the SDRAM, the signal RAS must be set to 1 (H) and the signals CS, CAS and WE to 0 (L).

5.1.3. Transmitter module specification

The transmitter module is a high level module dedicated to communicate with the SDRAM in order to verify its functionality. The transmitter's interface is composed by 5 ports as shown below:

```
sc_out<sc_uint<addr_size>> addressTrans;
sc_in<bool> clock;
 sc_out<bool> enable;
 sc_out<bool> readwr;
sc_out<int> command;
sc_inout<sc_uint<data_size>> dataTrans;
```

The command port is used to specify which operation the transmitter wants to perform on the SDRAM. An integer value is associated to each command of the memory.

5.2. Simulation model generation

We choose to simulate our system at RT level. So we generate a simulation module adapter to encapsulate the processor module and allow it to communicate with the SDRAM.

5.2.1. Rules definition

The association rules consist to associate the simulation module adapter's real ports with its logical ports. Thus, all the memory ports are associated with at least on transmitter port. For instance, the following two rules:

ADDRESSTrans – *ADDR*
- *CAS*
- *RAS*
- *DMQH*
- *DMQL*

DATATrans – *DATA*

means that de real port of the transmitter's simulation module adapter *ADDRESSTrans* will be associated with the logical ports ADDR, CAS, RAS, DMQH and DMQL. Thus, these five logical ports will take their values from the specified real port. So for this application we defined one association rule for each transmitter port.

The transformation rules are all those defining the access protocol to the SDRAM as defined by the vendor. Thus, we defined 11 transformation rules (one for each memory command). Each one of these rules defines how and in which order the value(s) on the real ports are transformed to be written on the logical ports.

5.2.2. Simulation module generator

The generated module contains 22 lines in its interface file (.h) and 90 lines describing the infinite process "behavior" (.cc). As shown in the table 1, the generated code has a very acceptable code compared to the code necessary to specify the SDRAM at the RT

level.

	SDRAM	Processor	Sim. Mod.
Interface	42	60	22
Behavior	1200	170	90

Table 1: Simulation models modules code size.

5.2.3. Modules linker

During this step, we connect the generated simulation module adapter with the processor module, and then we connect it with the SDRAM module. This consist simply to link point-to-point the simulation adapter ports with those of the two system modules. Thus the modules linker instantiates 3 signals to connect to processor with its simulation module adapter's real ports and 10 signals to connect the SDRAM with the simulation module adapter's logical ports.

5.3. Conclusion and analyses

This application shows that our mythology permits to simulate together an existing industrial module described at RT level (SDRAM) and another module specified at a high abstraction level in a fast and simple way. The generated multi level simulation model has an acceptable complexity compared with the initial specification of the system to be simulated.

6. CONCLUSION AND PERSPECTIVES

In this paper we presented an automatic multilevel simulation methodology for SoC design. Our approach consists to instantiate and to configure a generic C++ class to obtain one simulation module adapter for each module, described at an abstraction level different from the simulation level, in the system. The effectiveness of this approach has been shown on an example composed by an SDRAM and a processor described respectively at RT and system levels.

REFERENCES

[Bru00] J-Y Brunel, W. M. Kruijtzer and al, "COSY Communication IP's", Proc. of the DAC, Los Angeles, CA, June 200.

[Cho95] P. H. Chou, R. B. Ortega and al. "The Chinook Hardware/Software Co-Synthesis System", Proc. of the ISSS, 1995.

[Cow02] Coware. Inc., "N2C" available at **http://www.coware.com/cowareN2C.html**

[Gro02] T. Grotker, S. Liao and al, "System Design with SystemC", Kluwer Academic Publishers, USA 2002.

[ITR00] International Technology Roadmap for Semiconductors.

[Mic] MICRON, **http://www.micron.com**

[Mef02] S. Meftali, "Architectures exploration and memory allocation/assignment for multiprocessor SoC", PhD thesis, University Joseph Fourier, TIMA laboratory, France, 2002.

[Nic02] G. Nicolescu, "Specification and validation of heterogeneous embedded systems", PhD thesis, INPG, TIMA laboratory, France, 2002.

[Sem00] L. Semeria and A. Ghosh, "Methodology for Hardware/Software Co-verification in C/C++", Proc. of the ASPDAC, Jan. 2002.

[Sys] SystemC language and user guide, available at **http://www.systemc.org**

[Vsi] **http://www.vsi.org/**