MODELING SYSTEMS-ON-A-CHIP AT THE TRANSACTION LEVEL IN SYSTEMC

KAVINDRA KUMAR SINGH*

*Assistant Professor,
School of Computer and System Sciences,
Jaipur National University (JNU),
Jaipur, Rajasthan, India.

ABSTRACT

One of the characteristics of the problem which we are dealing with here is that we control neither the language (SystemC is defined by a consortium on which we have only little influence), nor the execution model (which is part of SystemC). A solution could have been to study the execution model of SystemC, to redefine it in terms of automata or another well-known formalism, and to work directly and only on this formalism. This paper first presents today’s design flow, with the different levels of abstraction used to describe a chip. Also, this paper details the Transaction Level Modeling level of abstraction and present the way it is implemented in SystemC.

KEYWORDS: SystemC, System-on-a-Chip, TLM, ELAB.

INTRODUCTION

A System-on-a-Chip (SoC) integrates many components on the same chip: a processor, several memory components, one or several buses, and specific components like video or audio decoders. A growing part of the functionality is implemented in the software part. The development of the embedded software for such a dedicated hardware platform requires specific methods and tools. The embedded software can, of course, be executed on the physical chip; this approach is fast and perfectly realistic with respect to the final SoC.

However, this approach is not feasible for two main reasons: cost and time-to-market. If executing the software reveals a bug in the hardware, then it is prohibitively costly to correct it. In addition, the execution of software on real hardware offers no detailed debugging capabilities. Finally, because of time-to-market constraints, the embedded software should be ready and tested before the physical chip is delivered, to shorten the integration phase.

The solution is to develop abstract models of the real system, with just enough details to be able to simulate the embedded software. This level of abstraction is called Transaction level modeling (TLM). Since the TL model of a system is less detailed than the RTL description, the TL model can be available earlier. TL simulations are much faster than RTL simulations: the decoding of an image takes only a few seconds. These advantages are obtained at the price of precision loss. The most abstract TL models do not allow timing performance evaluation, in particular because
the non-functional features, like pipelines, are not modeled. The level of abstraction of TL models implies that automatic synthesis of RTL descriptions from TL models is not possible. Consequently, TL models do not replace RTL descriptions; they are used earlier in the design cycle.

Quality and productivity constraints in the development tools for the design of embedded systems are increasing quickly. The physical capacity of chips can usually grow fast enough to satisfy those needs: The evolution of the number of transistors on a chip has been following Moore’s law (growth of 50% per year) for decades, and should keep on following it for at least 10 years. But one of the design flow bottlenecks is the design productivity: with traditional techniques, it grows only by around 30% per year, leaving a gap, increasing by 20% per year between the capacity of the chips, and the amount of code the designers are able to write. This problem is often referred to as the design gap. New techniques have to be settled continuously to be able to fill in this gap.

THE SYSTEMC

SystemC is a C++ library used for the description of SoCs at different levels of abstraction, from cycle accurate to pure functional models. A TL model written in SystemC is based on an architecture, i.e., a set of parallel components and connections between them. Each component has typed connection ports, and its behavior is given by a set of communicating processes that can be programmed in full C++. For managing processes, SystemC provides a scheduler, and several synchronization mechanisms: the low-level events, the synchronous signals that trigger an event when their value changes, and higher level mechanisms.

When a SystemC model is simulated, first the static architecture is built by executing the so-called elaboration phase (ELAB), which creates components and connections. Then the scheduler starts running the processes of the components, according to the informal automaton of Fig. 1(a). A simulation of a SystemC model looks like a sequence of evaluation phases (EV). Signal update phases (UP) and time elapse phases (TE) separate them (see Fig. 1(b)).

**FIGURE 1(A) AUTOMATION OF THE SYSTEMC SCHEDULER; (B) DIAGRAM OF AN EXECUTION**
THE SYSTEMC SCHEDULER

The SystemC Language describes the scheduler algorithm. At the end of the elaboration phase ELAB, some processes are eligible, some others are waiting. During the evaluation phase EV, eligible processes are run in a non-preemptive order, and explicitly suspend themselves when reaching a wait function. A process may wait for some time to elapse, or for an event to occur. While running, it may access shared variables and signals enable other processes by notifying events, or program delayed notifications.

An eligible process cannot become “waiting” without being executed. When there are no more eligible processes, signal values are updated (UP). Since there is no interaction between processes during the update phase, the order of the updates has no consequence. When there is still no eligible process at the end of an update phase, the scheduler lets time elapse (TE), and awakes the processes that have the earliest deadline. A notification of a SystemC event can be immediate, delayed or time-delayed. Processes can thus become eligible at any of the three steps EV, UP or TE.

In SystemC, the modules are instances of the class sc module. Each module may contain one or more processes. Communication internal to a module can be done in several ways (shared variables, events, etc.), but inter-module communication should be expressed with SystemC communication primitives: ports and channels.

A module contains ports, which are the interface to the external world. The ports of different modules are bound together with communication channels to enable communication. SystemC provides a set of communication interfaces such as sc_signal (synchronous signals), and abstract classes to let the user derive his own communication channels. In the pieces of code of Figures 2 and 3 have two definitions of modules, one of which is instantiated twice. The model is represented graphically in Figure 4.

The elaboration phase ends with a call to the function sc_start() that hands the control back to the SystemC kernel (line 70 in example). The last part of the execution is the simulation of the program’s behavior. The SystemC kernel executes the processes one by one, according to the algorithm presented in Figure 5. Initially, all processes are eligible. Processes are ran one by one, non-preemptively, and explicitly suspend themselves when reaching a waiting instruction.

```c
1. #include “systemc.h”
2. #include <iostream>
3. #include <vector>
4. struct module1: public sc_module {
5.    sc_out<bool> port;
```
7. bool m_val;
8. void code1 () {
9.     if (m_val) {
10.         port.write(true);
11.     }
12. }
13. SC_HAS_PROCESS (module1);
14. module1 (sc_module_name name, bool val) :
15.     sc_module (name), m_val (val) {
16. // register “void code1 ()”
17. // as an SC_THREAD (code1);
18.     SC_THREAD (code1);
19. }
20.};
21.
22. struct module2 : public sc_module {
23. sc_in<bool> ports [2];
24. void code2 () {
25.     std::cout<< “module2.code2”
26.         <<std::end1;
27.     int x = ports[1].read();
28.     for (int i=0; i<2; i++) {
29.         sc_in<bool>& port = ports[i];
30.         if (port.read()) {

31. std::cout << "module2.code2: exit"
32.    << std::endl;
33. }
34. wait(); // wait with no argument.
35. // use static sensitivity list.
36. }
37. }
38. SC_HAS_PROCESS (module2);
39. module2 (sc_module_name name)
40.   : sc_module (name) {
41.     // register "void code 2 ()"
42.     // as an SC_METHOD
43.     SC_METHOD (code2);
44.     don't_initialize ();
45.     // static sensitivity list for code2
46.     sensitive << ports [0];
47.     sensitive << ports [1];
48. }
49. }

FIGURE 2: EXAMPLE OF SYSTEMC PROGRAM: DEFINITION OF MODULES

50. int sc_main (int argc, char ** argv) {
51.    bool init1 = true;
52.    bool init2 = true;
53.    if (argc > 2) {
54. init1 = !strcmp (argv[1], "true");
55. init2 = !strcmp (argv[2], "true");
56. }
57. sc_signal<bool> signal1, singal2;
58. // instantiate modules
59. module1 * instance1_1 =
60. new module1 ("instance1_1", init1);
61. module1 * instance1_2 =
62. new module1("instance1_2", init2);
63. module2 * instance2 =
64. new module2("instance2");
65. // connect the modules
66. instance1_1 ->port.bind(signal1);
67. instance1_2 ->port.bind(signal2);
68. instance2 -> ports[0].bind(signal1);
69. instance2 -> ports[1].bind(signal2);
70. sc_start(-1);
71. }

FIGURE 3: EXAMPLE OF SYSTEMC PROGRAM: MAIN FUNCTION
FIGURE 4: GRAPHICAL VIEW OF THE SYSTEMC PROGRAM

Note that this algorithm gives only the simpler cases. Uninitialized processes, instantaneous notification for example would add particular cases.

E: the set of eligible processes

S: a set of tuples (P, e) of sleeping processes

waiting for events

T: a set of tuples (P, t) of processes

associated with time events

F: a set of event or signal consequences:

(EV, e) or (SIG, s, v)

V: a set of tuples (s, v) for signal values

E:= {all processes, except those on which don’t_initialize () has been called.}

loop until the end of simulation

while E ≠ ϕ one execution of this loop body is a δ-cycle

// the so-called evaluation phase:

while E ≠ ϕ

P := one element in E; E := E – {P}
run P, while filling F and reading signal values in V, until it stops;
if P emits an event e: F := F U {(EV, e)}
if P writes a value v on a signal s: F := F- {(s, …)} U {(s, v)}
if P stopped on a wait-time (t) T := T U {(P, t)}
if P stopped on a wait-event S := S U {(P, e)}
end

// the so-called update phase:

For each element f in F

if f = (EV, e) then

if f = (SIG, s, v) then V := V – {(s, …)} U {(s, v)}
F := F – f
end

min = minimum value of the t_i’s in the set T = {(P_i, t_i)} // let time elapse:

for each element x = (P_k, min) in T
T := T – {x}; E := E U {P_k}
end

end loop
FIGURE 5: THE SYSTEMC SCHEDULER ALGORITHM

This algorithm may appear strange to someone used to software scheduling. This idea actually comes from the RTL Hardware Description Languages. In the physical execution, the carry will propagate until all signals are stabilized. If the carry propagation is longer than the clock cycle, then the timing is incorrect. In a digital simulation, we do not have this notion of physical propagation, but the simulation semantics should be as close as possible to the physical behavior. If we execute the processes only once in a clock cycle, in an arbitrary order, then, the result will be dependent on the order of execution. If the least-significant bits are added first, then carry propagation will occur normally. If most-significant bits are added first, then the carry will be ignored.

One solution is to statically check the causal dependencies and compute an order of execution that will respect the causal dependencies (this is the approach followed by synchronous languages such as LUSTRE or ESTEREL. The other approach, followed by the standard HDLs and SystemC, is to reschedule the execution of each process until the signals get stabilized. Most actions (sc_signal value update for example, the notify() function in SystemC being a notable exception) are actually taken into account after all the scheduled process have finished their execution. If those actions wake up other processes, then, those processes will be executed during the next cycle.

THE TLM LIBRARY

In a TL model, all communications between components are done by transactions, which are implemented by function calls. When a process in a component I (initiator) wants to communicate with another component T (target), it calls method of one initiator port of I. The initiator port forwards the function call to its associated target port on T, which is linked to the code that implements this method. The initiator process continues when the function call returns. It does not yield back to the scheduler, allowing for atomic sequences of transactions.

FIGURE 6: COMMUNICATIONS IN TL MODELS
In Figure 6, circles with arrows represent processes; large plain arrows represent the function calls (transactions); dashed arrows represent communications between processes.

THE SYSTEM-ON-A-CHIP DESIGN FLOW

One of the past technological revolutions in the hardware domain has been the introduction of the Register Transfer Level (RTL) to replace the gate-level as an entry point for the design flow. The gate-level description uses only the simple logic operators (and, or, not, . . . ) to describe the design, whereas the RTL level allows the notion of register (one-word memory), and a data-flow description of the transfers between registers at each clock cycle. Since the translation between RTL and gate-level descriptions can be done automatically and efficiently, the gate-level description is today mainly an intermediate representation synthesized from the RTL code, used for the chip manufacturing.

A. HARDWARE – SOFTWARE PARTITIONING

Raising the abstraction level above the gate-level has been a real progress, but is not sufficient to fill in today’s design gap. It is necessary to maximize the reusability of the chip components, usually called Intellectual Properties (IPs). This can be achieved by using software components instead of Application Specific Integrated Circuits (ASIC). Software components can be easily reused, modified at all steps of the design flow. On the other hand, they are much less efficient both in terms of computation time and in terms of power consumption.

Therefore, designers need to find the compromise between software and hardware: implement the performance-critical operations using dedicated hardware components, and the non-critical parts using software. Deciding which feature will be implemented in software and which one will be implemented in hardware is called hardware/software partitioning. The result is a mixture of software and hardware, intermediate between general purpose CPU and ASIC and containing several components executing in parallel. It is called a System-on-a-Chip.

Since one of the main tasks of the embedded software is to program the hardware components, the software, or at least its low-level layers, will be highly hardware-dependent, and will not run unmodified on a standard computer. There are mainly two ways to execute such software: 1) execute it on the physical chip, and 2) execute it on a simulator.

The first option is not acceptable during the development because of time-to-market constraints: the software needs to be finished and validated a few weeks after the chip comes out of the factory. Furthermore, developing embedded software can help in finding bugs in the hardware, and the cost of a bug discovered on the physical chip is several orders of magnitude higher than a bug found before manufacturing the first chip: the first step of the manufacturing is to build the mask that will be used for the lithography of all chips.

The trivial way to simulate the hardware part of the chip is to use the RTL description. Unfortunately, due to the highly parallel nature of hardware, simulation of a large RTL design is extremely slow. It is possible to replace some components of an RTL simulation by a simulator, typically written in C, to increase the simulation speed. A common technique is to replace the
processor by an instruction set simulator (ISS), and the memory by a simple array. This technique, mixing behavioral and RTL components is called co-simulation.

Figure 7 shows the simulation time we get on the same computation with those different techniques.

**FIGURE 7: SIMULATION TIME FOR THE ENCODING + DECODING OF ONE IMAGE IN A MPEG4 CODEC**

**FIGURE 8: ILLUSTRATION OF THE DIFFERENT LEVELS OF ABSTRACTIONS**
B. DIFFERENT LEVELS OF ABSTRACTION

Ideally, the design flow should start by the highest level, and refine, either automatically or manually to the lowest level. Figure 8 illustrates the relationship between the different levels of abstraction. On the left are the levels of abstractions and on the right are examples of commonly used technologies at this level of abstraction. On this picture, the ideal design flow starts from the top and refines to the bottom.

The distinction between those levels of abstraction is widely (but not quite universally) accepted. Of course, some intermediate levels can be added.

I. ALGORITHM

The highest level (below the specification) is the algorithm. For multimedia devices, at least part of the algorithm is provided by a norm, and reference implementations often exist. Algorithms are usually designed in high level programming languages, such as Matlab™, or in C. At this level, the notion of software or hardware components does not exist, and the notion of parallelism is not yet taken into account.

II. PROGRAMMER VIEW: PV

Then comes the Transaction Level Model, which actually splits into two flavors: the Programmer View (PV), and the Programmer View plus Timing (PVT). At this level, the chip is modeled as a platform made of several modules. Each module shows to the external world all its functionalities, and only its functionalities. The timing aspects, for example, are not yet taken into account.

Communication between modules is done through a model of interconnect (the interconnect itself is the set of channels existing on the chip), made of one or several communication channels, whose role is to route a piece of data from a module to another one. This exchange of data is called a transaction. At this level, the size of a transaction is not necessarily related to the data-width of the bus. For an image processing algorithm, for example, the PV model can decide to transmit the data line by line, block by block, or even image by image.

An important part of the PV level is the system synchronization. At this level of abstraction, we haven real notion of timing, so the system is mostly asynchronous. A set of independent processes could hardly give a consistent behavior. For example, when a process needs a piece of data that is the output of another process, we have to ensure that the value will be produced before being used. The communication channels are not only useful to transmit data from one module to another, but can also be used for the system synchronization. Some channels can also be dedicated to synchronization and transmit no data. This is the case for the model of an interrupt signal. It can be modeled as a Boolean signal (we will observe the edges of the value, but not the value itself), or even a channel without value.

At this level of abstraction, the model contains all the necessary and only the necessary information Torun the embedded software. The organization of the program is completely
different from the one of the algorithmic level. The first partitioning is done, the algorithms are parallelized, even if the decision of implementing some blocks in hardware or software is not necessarily taken. Some tools can help taking partitioning decisions, and some code can be reused, but an automatic translation from the algorithm level to the PV level cannot exist.

III. PROGRAMMER VIEW PLUS TIMING: PVT

While TLM was originally created to ease the development of embedded software, it also proved to be useful for preliminary performance analysis. Since the TLM model contains less details than the RTL, it can be written faster (it is usually admitted that a PV model can be written ten times faster than its RTL equivalent), and should be available earlier than the RTL in the life cycle of a chip. It is therefore reasonable to use it to take decisions about the RTL design (identify the bottlenecks; dimension the modules and communication channels).

Unfortunately, the PV level does not contain the timing information necessary to perform such analysis.

It is therefore necessary to enrich the PV model with timing information, with some constraints: The functionality of the PV model must not be modified, and the effort to write the PVT model based on the PV model must be as low as possible, and in particular, lower than the effort to write a PVT model from scratch.

At this level, the architecture of the model must correspond to the one of the actual platform. Each data-transfer in the RTL model must correspond to a transaction of the same size in the PVT model. By adding timing information on each data treatment or transfer, we get a good approximation of the timing of the platform. Ideally, it should be cycle-count accurate (one knows how long a transaction takes), but not cycle-accurate (one doesn’t know exactly what happens at a given clock cycle).

IV. CYCLE-ACCURATE: CA

A cycle-accurate model describes what happens at each clock cycle. It can still abstract some details, but needs to be aware of the micro-architecture of each component (for example, a model of a processor needs to model the pipeline tube cycle-accurate).

V. REGISTER TRANSFER LEVEL: RTL

It is the first level of abstraction to be precise enough to be synthesizable automatically and efficiently. It is bit-accurate, cycle-accurate, and describes all the internal details of each component. Designs at this level of abstraction are usually implemented using the VHDL (Very large scale integrated circuits Hardware Description Language) or Verilog language.

VI. GATE LEVEL, BACK-END

The design flow from the RTL design to the factory is well established. The gate-level net list is automatically generated from the RTL, then comes placement and routing to transform the net list into a two-dimensional view, that will be used to draw the lithography mask to be sent to the
factory. Synthesis from RTL to gate level is possible efficiently, and is a well-established methodology.

SYSTEMC AND THE TLM API

A. NEED FOR A NEW “LANGUAGE”

TLM level of abstraction, to be applicable in practice, we need a way to write and execute such models. The main technical requirements are the following:

Efficient simulation: The main reason to write TLM models instead of RTL models is to gain several orders of magnitude in terms of simulation speed.

Modular design and code reuse: Classical software paradigms such as generosity, support for abstract data-type or object oriented programming can be helpful at this level.

Parallel execution semantics: The components must execute their behavior in parallel, synchronize themselves, and communicate together. Additionally, one can require interfacing with standard Hardware Description Language (HDL) languages (VHDL, Verilog), easy debugging, and powerful tools for visualizing execution traces. It is more than desirable to build the TLM technologies on open standards for several reasons. Furthermore, the code reuse objective can only be achieved if the modules to be reused are compatible. This means the technology has to be supported by all the IP providers, which is unlikely to happen for a proprietary technology.

A number of other approaches have been proposed for the description of heterogeneous hardware/software systems with an emphasis on formal analysis. In this type of approach, the definition of the description language is part of the game. The language can be defined formally, and tailored to allow easy connections to validation tools.

The need for efficient simulation and the preference for well-known languages lead to consider a C-based, or C++-based approach. Although oriented towards the promotion of a commercial tool, gives a good overview of the motivations behind this approach. A similar approach is to create a new language, inspired from C and C++, with the additional required features. This is the approach followed bisect. The approach didn’t get a lot of success in the industry, partly because it requires a specific compiler.

System 2.0 has been designed to meet the above requirements. It is open source and relies on an ISOstandard language: C++. This is crucial for two reasons: first, it guarantees a fast learning-curve for the engineers of the domain; second, it guarantees that the models of systems developed in SystemC can beexploited even if the tool that was used to build them is no longer available.

B. THE SYSTEMC LIBRARY

A model written in SystemC is executed in two phases:
ELABORATION: The entry point for the execution of the model is the function sc_main() (the SystemC library itself provides the main() function, which is a very simple wrapper around sc_main()). This starts the elaboration phase during which the modules will be instantiated in the usual C++ way, and connected (bound) together. Then, the program calls the function sc_start that will launch the simulation.

SIMULATION: After the call to sc_start(), the architecture of the model is fixed. No module can be instantiated, and the binding cannot change. The SystemC kernel will call the member functions of the modules that have been registered as processes. This is called the simulation phase.

During the simulation, SystemC distinguishes two kinds of processes: SC THREAD and SC METHOD (plus the semi-deprecated SC CTHREAD). An SC THREAD is an explicit infinite loop. It is a function that never terminates, but that hands back the control to the scheduler using the wait statement or equivalent. An SC METHOD in a process that executes in zero time, implemented by a C++ function that must terminate, and that will be called repeatedly by the scheduler.

CONCLUSION

This paper begins with presentation of SystemC, SystemC Scheduling along with TLM Library. Paper try to presents today’s design flow, with the different levels of abstraction used to describe a chip. Then it details the Transaction Level Modeling level of abstractions that explained in this paper and present the way it is implemented in SystemC and finally, paper ends with TLM API.

REFERENCE


- FeliceBalarin, Luciano Lavagno, Claudio Passerone, Alberto L. Sangiovanni-Vincentelli, Marco Sgroi, and Yosinori Watanabe. Modeling and designing heterogeneous systems.


