Native ISS-SystemC Integration for the Co-Simulation of Multi-Processor SoC

Franco Fummi[†] Stefano Martini [*] Giovanni Perbellini [*] Massimo Poncino [†]

[†] Università di Verona
Verona, ITALY

[*] Embedded Systems Design Center
Verona, ITALY

Abstract

In a system-level design flow, the transition from a high-level description entry implies the refinement from an untimed, unpartitioned description to a real architecture where applications are executed on a programmable device and interact with ad-hoc hardware components. Simulation of such architectures requires the capability of efficient co-simulation of a model of hardware with a model of the processor.

This paper presents two co-simulation methodologies, based on SystemC as hardware modeling language and on an Instruction Set Simulator (ISS) as a model of the processor. The first one works at the SystemC kernel level and exploits potentialities of the GNU suite, whereas the second uses features offered by the operating system running on the ISS.

The two methodologies improve co-simulation performance with respect to state-of the art methods, and provide different trade-offs between the simplicity of the programming model, the modeling power, and co-simulation performance.

1. Introduction

Historically, one of the most critical issues in system-level design has been the integration of hardware and software components, and the validation of their behavior as interacting entities. In modern embedded systems such hardware-software co-verification amounts to simulate a core embedded within custom hardware specified in some HDL. For efficiency reasons, the two simulations models of the hardware and the software are not at the same abstraction level. Therefore, co-verification has evolved around the idea of *co-simulation*, that is, the possibility of debugging a software program on a version of the hardware executing within a HDL simulator.

Several *co-simulation* platforms have been developed by academic groups as well as by EDA vendors ([8]–[14]). In spite of their variety of architectural targets, software architectures, performance efficiency and languages, most of these frameworks essentially address the same problem, that is, how to efficiently link event-driven hardware simulators and cycle-based instruction set simulators (ISSs). Recently, design flows based on C/C++ [2, 3] have somehow simplified this task, thanks to the possibility of using the same language for describing software and hardware. Such *homogeneous* environments, because of their higher potential efficiency, can be considered as state-of-the-art, in particular those based on SystemC ([12, 13, 14]), that offers both support for hardware modeling in C++ as well as a simulation environment. Such existing co-simulation frameworks have different strengths and limitations, that will be analyzed in depth in Section 2. In this work, we address some of these limitations, by proposing two alternative co-simulation methodologies that allow a SystemC description of hardware and an ISS to co-execute efficiently. The two proposed solutions differ with respect to the simulation kernel (SystemC or the ISS) that drives the co-simulation. However, in both cases the interaction with SystemC simulation sessions is implemented at the kernel level, thus making it transparent to the SystemC code written by the user.

In the first scheme, called *GDB-Kernel* co-simulation, the SystemC simulation kernel works as master simulator, and the communication between simulators is embedded into the SystemC kernel.

In the second scheme, called *Driver-Kernel*, "calls" to the SystemC hardware functionalities are mapped to device drivers calls of the operating system (OS) running on the ISS. The driver communicates via IPC to the SystemC hardware model.

The two approaches provide different tradeoffs between simulation speed and required user-provided support. The GDB-Kernel scheme requires fewer and simpler modifications to the simulator kernels; conversely, in the Driver-Kernel scheme, modifications to the simulator are limited to the ISS (a new driver), but the programming model is slightly more complicated. Experiments on a real-life example show a significant simulation speed-up with respect to state-of-the-art solutions.

1530-1591/04 $20.00 (c) 2004 IEEE

2. Background and Previous Work

Earlier hardware-software co-simulation frameworks ([4, 5, 6]) were mainly focused on multi-language system descriptions, i.e., some HDLs for hardware description, and some programming language for software. All these approaches are quite similar in that, as *heterogeneous* co-simulation solutions, their main effort consisted in solving the issue of controlling and synchronizing two (or more) simulation engines. Such heterogeneous style is sub-optimal in terms of performance and ease of integration; however, it was the only possible choice when VHDL or Verilog simulation was the highest possible level of abstraction for simulating hardware. Some commercial tools such as Mentor Graphics Seamless [10], and Synopsys Eaglei [9] also provide heterogeneous co-simulation capabilities.

Conversely, a *homogeneous* co-simulation environment that uses a single engine for simulation is able overcome these drawbacks. The Ptolemy [7] and Polis [8] environments are pioneering works in that direction; in these approaches, homogeneity is achieved by abstracting away the distinction between hardware and software, and they are thus more suitable in a very initial phase of the design, prior to hardware-software partitioning.

The advent of design flows and HDLs based on C/C++ [2, 3] allowed to use the same language for *describing* software and hardware, thus making co-simulation easier and more efficient, because the system can be simulated within a single simulation engine. Many recent works, in particular, have exploited SystemC as simulation backbone for achieving efficient co-simulation frameworks ([11]–[14]).

All these homogeneous environments are based on two basic building blocks: (i) *interprocess communication* (IPC), used to realize the communication between the ISS and the C/C++ simulator, that run as distinct processes on the host system, and (ii) the concept of *bus wrapper*, that ensures synchronization between the system simulation and the ISS, and translates the information coming from the ISS into cycle-accurate bus transactions.

Most of these approaches [11, 12, 13] define a *custom interface* between the bus wrappers and the ISS; this complicates the integration of new processor cores within the co-simulation framework, because the ISS needs to be modified to support the IPC primitives defined by the co-simulation system. This issues is addressed in [14], where a standardized interface between bus wrapper and ISS is proposed, based on the remote debugging primitives of `gdb` [15]. In this way, any ISS that can communicate with `gdb` (that is, basically any) can also become part of a system-level co-simulation environment. The approach of [14] still suffers from some performance bottlenecks, since the ISS and the SystemC simulators evolve in lock-step, because synchronization is driven by the host operating system via IPC.

The proposed co-simulation schemes build upon the work of Benini et al. [14], yet using two completely different approaches. Before getting into the details of these schemes, we will first outline the basic assumptions on which they rely.

3. GDB-Kernel Co-Simulation

In the following, we assume an architectural template consisting of several processors interacting with hardware blocks, and communicating between them through a common bus. In addition, we consider SystemC as the language for describing HW, and ISSs as a model of software. Finally, we assume that the assignment of tasks to hardware or software has already been decided.

GDB-Kernel co-simulation directly builds upon the solution proposed in [14]. First, it is a fully homogeneous co-simulation scheme: besides modeling hardware, the SystemC simulation kernel serves as a master that drives the overall co-simulation. Second, it is based on the use of the GDB remote debugging interface (RDI) between the ISS and the wrapper used to connect ISS and the SystemC simulator. Third, communication between SystemC and ISS is encapsulated by a *wrapper* that restricts the use of IPC between this wrapper and the ISS. The wrapper can be seen as an extension of SystemC that makes HW/SW communication more efficient. Figure 1 summarizes these features.

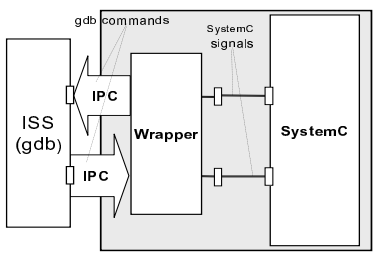

Figure 1. SystemC Wrapper and ISS.

In [14], the HW designer is aware of the existence of wrappers, which must be explicitly instantiated; the wrapper loads the ISS, and establishes IPCs between SystemC and the ISS. This implies a significant modification of the programming model. In addition, the co-simulation performance is limited by the communication control implemented by explicitly writing a `sc_method`.

Conversely, the proposed GDB-kernel approach *embeds the wrapper into the SystemC kernel*. This simplifies the programming model, as well as it makes communication more efficient because the intervention of the host operating system is considerably reduced.

3.1. SystemC Kernel Modifications

The synchronization between ISS and SystemC is realized by modifying the SystemC kernel in such way that it can establish and control the communication, using GDB com-

mands. The required modifications to the SystemC kernel consist essentially in:

- The addition of two type of ports `iss_in` and `iss_out`, that are devoted exclusively to the communication between a SystemC module and an ISS. These are derived from the `sc_in` and `sc_out` SystemC classes, respectively.
- The addition of a special process `iss_process`. Similarly to a `sc_method`, an `iss_process` will start execution when a new data is present on a `iss_in` port to which the process is sensitive.
- The modification of the event scheduling algorithm, in order to handle the presence of special ports and processes.

From the ISS side, the interface between the user program and SystemC is realized through ordinary variables, and does not require any special modifications.

3.2. Programming Model

The basic communication infrastructure (the special ports, the special process, and program variables) is used by the programmer as follows:

1. Set breakpoints on the variables of the application (on the ISS) that are considered as a channel to read or write data from the device described by the SystemC file;
2. Modify the SystemC description by defining `iss_in` and `iss_out` ports and associate them to the breakpoints previously set.

There is a small difference in the association of ports to breakpoints. In the case of `iss_in` ports, it is necessary to set the breakpoint to the line that immediately *follows* the target statement (i.e., the line containing the variable to be monitored). This is because, since the ISS stops before executing the target instruction, we want to actually receive the value before stopping. In the case of `iss_out` ports, the breakpoint must be defined on the very same line containing the desired variable.
Figure 2 shows a simple example of how variables and ports are matched to setup communication.
Breakpoints at lines 10 and 12 allow, respectively, to connect the `separator` variable to the SystemC `in_port` and the SystemC `out_port` to the `in_var` variable.
This programming model is considerably simpler than that of [14], and it can be made almost completely automatic, by means of pragmas. A special pragma, containing the name of the variable is inserted before the line where the breakpoint is to be set. A simple filter automatically generates the proper GDB script for execution of the program, and a text file to be used by the SystemC hardware programmer that contains a map of the type $<$ `variable` $>$ $<$ `line` $>$.

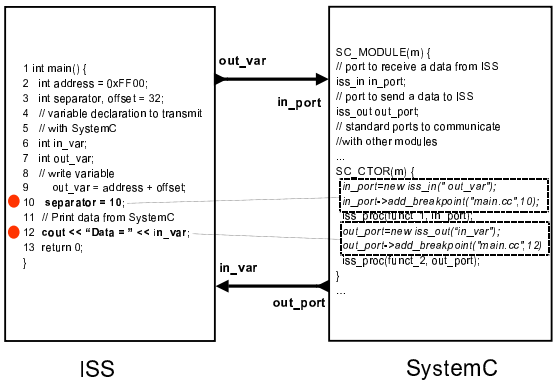

Figure 2. GDB-Kernel Synchronization Example.

In this way, the modifications of the application and of the hardware description is reduced to a minimum.

3.2.1. Scheduling As already mentioned, the complete control of the simulation is handled by the SystemC kernel. Figure 3 shows a high-level flowchart of the modifications of the SystemC scheduler.

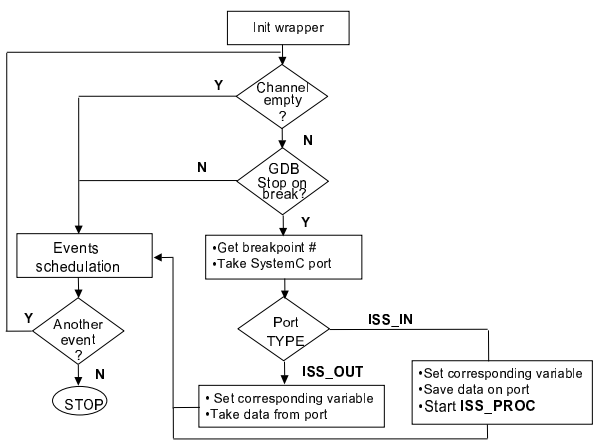

Figure 3. Modified Scheduling Algorithm.

The algorithm, at the beginning of a simulation cycle, checks, through the invocation of special methods of the wrapper class, if the GDB is stopped to a breakpoint. This is done by checking the content of the data structure of the IPC mechanism used to connect the ISS and the wrapper (a pipe, in our implementation).
If not, the kernel does the normal handling of the events in the scheduler queue. Otherwise, the kernel checks at which breakpoint the GDB is stopped. If the breakpoint is associated to an `iss_in` port, a method of the wrapper class is used to get the new value of the variable from the ISS, then the value is stored into corresponding `iss_in` port and any `iss_process`, related to that port, is started.
If the breakpoint is associated to an `iss_out` port, the value stored in this port is copied to the variable by using another method of the wrapper.

Note that the use of the `iss_in`, `iss_out`, and `iss_process` abstractions allows to hide all the communication mechanisms to the designer. Moreover, SystemC processes, related to `iss_in` and `iss_in` ports, are executed only when data are effectively transmitted or received from/to the ISS, thus sensibly reducing co-simulation overhead.

4. Driver-Kernel Co-Simulation

The *Driver-Kernel* co-simulation scheme is based on establishing the communication by means of a device driver implemented inside the OS running on the ISS. In this case, it is necessary to realize a driver for the hardware device that we want to control. The driver provides the user with a set of APIs that can be called to drive the hardware device described in SystemC. In this way, it is the ISS that masters the co-simulation.
This approach requires a relatively high effort to the designer, that has to write a specific driver for each new (SystemC) device. On the other hand, this approach allows to model *interrupt handling*, thanks to the bi-directional communication between SystemC and the ISS (and unlike the GDB-Kernel paradigm).
Modeling an interrupt in the GDB-Kernel scheme would require to stop GDB execution at any instruction, thus degrading the performance of co-simulation unacceptably. Conversely, with the Driver-Kernel scheme, the user can define an Interrupt Service Routine (ISR) to be executed every time the target device generates an interrupt. Another advantage of this scheme is that co-simulation performance is further improved; in fact, while the IPC communication still occurs at the kernel level from the SystemC side, the GDB interface overhead has been removed from the ISS side.

4.1. Programming Model

In the Kernel-Driver, the most important step consists of the implementation of a driver in the target operating system with the purpose of controlling the SystemC device. The driver consists of (i) the code that handles the interaction with the external device through proper ports, (ii) the ISR to handle interrupts, and (iii) a suitable API that allows to interact with the driver from the application source code. In our case, the driver operates as follows (see Figure 4):

1. It communicates with the SystemC kernel through port 4444 (called the *socket data port*) by sending a proper message, whose format is described in the next section.
2. It then creates a thread that listens to the interrupts generated from the SystemC device; interrupts are received through port 4445 (called the *socket interrupt port*). When an interrupt occurs, the ISR written by the programmer, has to be started to manage the interrupt.

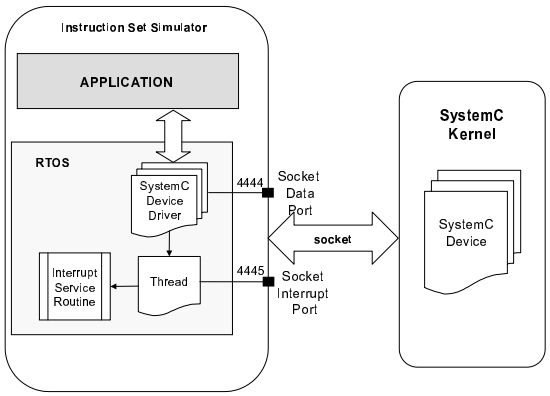

Figure 4. Programming Model Architecture.

The programmer uses the methods of the driver API to build a communication between the application and the hardware device. This programming model is thus very close to the conventional way in which a hardware device is managed by software; the only difference concerns the driver implementation that communicate with the SystemC code instead of controlling a real hardware device. This communication is made possible by the modification of the SystemC scheduling algorithm which is described in the following section.

4.1.1. Scheduling The driver and the SystemC kernel communicate by exchanging messages consisting of the following fields:

- `Packet Size`: size of the whole message.
- `Type`: message type; it can be `READ` or `WRITE`.
- $\texttt{DataSize}_i$: size of the `i`-th data block.
- $\texttt{Data}_i$: data to send to the SystemC port specified in the i-th `SC Port`, for the `WRITE` case
- SC $\texttt{Port}_i$: the name of `iss_in` port to be written, for the `WRITE` case, or the name of the `iss_out` port to be read for the `READ` case.

The adopted protocol specifies the two possible operations (`READ`, when the driver receives data from the SystemC kernel and `WRITE`, when the driver sends data to the SystemC kernel) and the entities involved in the data exchange (e.g., SystemC `iss_in` and `iss_out` ports).
Figure 5 shows the modifications of the SystemC kernel scheduler required to implement the driver-kernel co-simulation mechanism. At the beginning of a new simulation cycle, SystemC kernel checks the content of the message to be possibly exchanged with the driver. If there is no message, the kernel does the normal handling of the events in the scheduler queue.
At the end of the event scheduling, before moving to the following simulation cycle, it verifies if an interrupt has been generated. In this case, the interrupt is notified to the driver by sending a message on the reserved socket interrupt port.

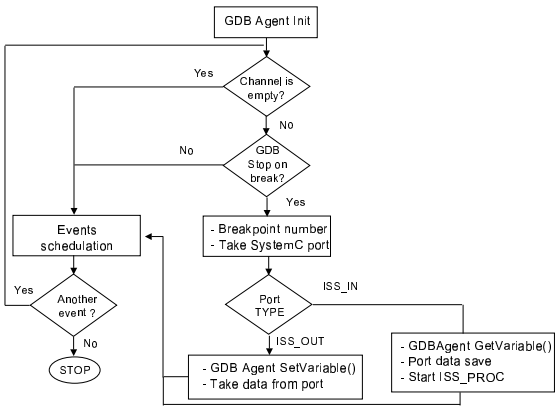

Figure 5. Modified Scheduling Algorithm.

Whenever a message arrives, the kernel checks the type of the message. In case of a `WRITE` message, the data value is stored into the corresponding `iss_in` port specified into the $\texttt{SCPort}_i$ field of the message and all `iss_process`, associated to that port, are started; In the case of a `READ` message, the value stored in port specified by the $\texttt{SCPort}_i$ field is sent to the driver.

5. Case Study

The two proposed co-simulation schemes have been applied to design a router that manages simple packets. The forwarding process is based on a static routing table embedded into the router. This case study is an extension of the Multicast Helix Packet Switch example distributed with SystemC 2.0.1.

This router has 4 input ports and 4 output ports. All packets coming into the router are buffered into a FIFO queue. The packet consists of the following fields: *Source address*, *Destination address*, *Packet identifier* (used for debugging purposes), *Data field*, and *Checksum*. The routing table is used to determine the output port for a given packet; each entry matches a destination address and an output port.

The main process of the router takes the first packet in the queue and reads its destination address. By looking in the routing table the correct output port is used to send out the packet. Before sending the packet, the checksum is computed on the packet to detect possible errors. In our testcase, the checksum calculation is performed by an application executed by a CPU, as commonly done in embedded routers.

The overall scenario is shown in Figure 6; it consists of the following objects:

- SystemC model of the router;
- SystemC model of the packet generator (producer);
- SystemC model of the packet destination (consumer);
- C/C++ application computing the checksum.

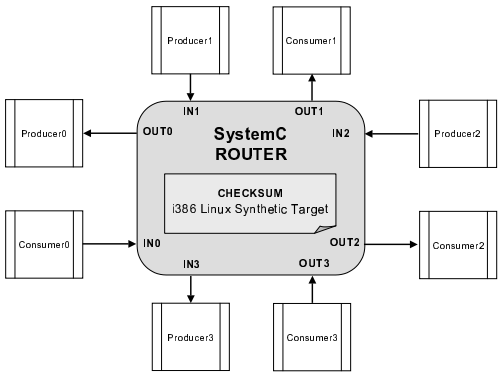

Figure 6. System Architecture.

The checksum algorithm is a C/C++ program executed by the ISS of the i386 Linux synthetic target running eCos as real-time operating system. The communication between the C/C++ program and the SystemC model of the router is realized using both co-simulation schemes.

In the GDB-Kernel methodology, breakpoints are defined in the source code in correspondence of the variables representing data on which the checksum algorithm must be computed. The proper `iss_in` and `iss_out` ports are added to the SystemC router description to establish the communication.

For the Driver-Kernel scheme, a special device driver is created and embedded into eCos, so that the C/C++ source code calls the appropriate function to communicate with the SystemC module.

The producer is a SystemC module attached to an input port of the router. It generates packets with a random destination address. The consumer is also a SystemC module attached to an output port of the router, that analyzes the integrity of the received packet.

	Simulated Times [ms]		
	1000	10000	100000
`GDB-Wrapper`	14760	142170	1482710
`GDB-Kernel`	9710	92579	919860
`Driver-Kernel`	5330	50030	493012

Table 1. Simulation Performance Results.

Table 1 shows the performance of the two proposed co-simulation schemes, compared to the case of a co-simulation scheme based on the instantiation of wrappers in SystemC [14], (called `GDB-Wrapper`). The three columns are relative to different amounts of simulated times (1000, 10000, and 100000 seconds). The table clearly show how both proposed solutions outperform the wrapper-based solution: the GDB-Kernel scheme is about 30% faster, while the Driver-Kernel one improves speed of a factor 3. Speedups are consistently preserved for the various simulation lengths.

Concerning software complexity, the Driver-Kernel requires an overhead (measured in lines of code) of about 40% on the SystemC side, and of a factor 9x on the C++ side (due to the writing of a new driver). with respect to the GDB-Kernel scheme.

5.1. Performance Analysis

The proposed schemes are not just two alternative solutions that offer different tradeoffs between speed and programming effort. They can also be regarded as two solutions that allow to model systems with different degrees of complexity of the software. In the GDB-kernel scheme, hardware interaction is managed by the application itself, and the possibility of supporting an operating system is not considered. This scenario is suitable for medium-to-low complexity systems. Conversely, the Driver-kernel scheme explicitly assumes the presence of an OS, and the interface to the hardware is managed through the API of the (custom) driver.

This difference in software complexity must be reflected by the co-simulation engine; that is, the results of different co-simulation schemes on a given system should yield different results.

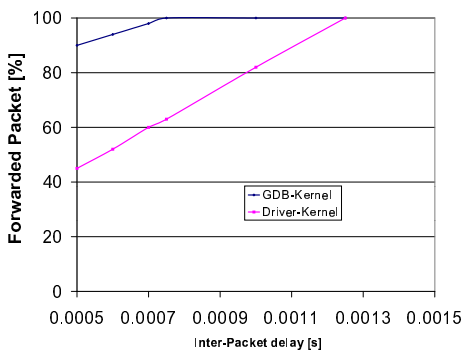

Figure 7. Performance analysis.

An example of this sensitivity to software complexity is shown in Figure 7 for the router example; the plot shows the percentage of packets forwarded by the router vs. the inter-packet delay. The percentage of packets forwarded is a measure of the efficiency of the router and it cannot decrease below a threshold to guarantee a required level of service. Obviously, increasing the inter-packet delay tends to yield a 100% of forwarded packets. The way the rate of forwarded packets varies, however, is different for the two schemes. The difference is a measure of the overhead imposed by the OS; in the Driver-Kernel scheme, this overhead slows down the execution of the application, which manages to forward a smaller number of packets with respect to the GDB-Kernel scheme. Alternatively, the plot can provide the minimum inter-packet delay (maximum frequency) for a given forwarding percentage.

6. Conclusions

Integration of a hardware simulator and an ISS is the core of state-of-the-art co-simulation approaches. The efficiency of the co-simulation is strongly related to the ability of making this integration as tight as possible.

In this work, we have proposed two co-simulation schemes, that differ in how the integration issue is solved, and provide different tradeoffs between simulation speed and the effort required by the programmer to make this interaction possible.

Acknowledgments

We wish to thank the SOC group of Telecom Italia Lab for helping us in the tuning of the case study and the analysis of the simulation results.

References

[1] M. Keating and P. Bricaud, *Reuse Methodology Manual*, Kluwer Academic Publishers, 1999.

[2] G. De Micheli, “Hardware Synthesis from C/C++ Models,” *DATE’99*, pp. 382–383, March 1999.

[3] Synopsys, Inc., “SystemC, Version 2.0”, *http://www.systemc.org*.

[4] C. Liem, F. Nacabal, C. Valderrama, P. Paulin, A. Jerraya, “System-on-Chip Co-Simulation and Compilation,” *IEEE Design and Test*, Vol. 14, No. 2, pp. 16-25, Apr.–Jun. 1997.

[5] C. Valderrama, F. Nacabal, P. Paulin, A. Jerraya, “Automatic VHDL-C Interface Generation for Distributed Co-Simulation: Application to Large Design Examples”, *Design Automation for Embedded Systems*, Vol. 3, No. 2/3, pp. 199–217, March 1998.

[6] P. Coste, F. Hessel, Ph. Le Marrec, Z. Sugar, M. Romdhani, R. Suescun, N. Zergainoh, A. Jerraya, “Multilanguage Design of Heterogeneous Systems”, *International Workshop on Hardware-Software Codesign*, pp. 54–58, May 1999.

[7] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems,” *International Journal of Computer Simulation*, Vol. 4, pp. 155–182, April 1994.

[8] F. Balarin et al., *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*, Kluwer Academic Press, 1997.

[9] Synopsys, Inc., “Eaglei”, *http://www.synopsys.com/products*.

[10] Mentor Graphics Inc., “Seamless CVE”, *http://www.mentor.org/seamless*.

[11] J. Liu, M. Lajolo, A. Sangiovanni-Vincentelli, “Software Timing Analysis Using HW/SW Co-Simulation and Instruction Set Simulator,” *CODES’98*, pp. 65–69, March 1998.

[12] L. Semeria, A. Ghosh, “Methodology for Hardware/Software Co-verification in C/C++”, *ASPDAC’00*, pp. 405–408, January 2000.

[13] K. Lahiri, A. Raghunathan, G. Lakshminarayana, S. Dey, “Communication Architecture Tuners: a Methodology for the Design of High-Performance Communication Architectures for System-on-Chips,” *DAC-37*, pp. 513-518, June 2000.

[14] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, M. Poncino, “SystemC Co-simulation and Emulation of Multi-Processor SoC Designs,” *IEEE Computer*, Vol. 36, No. 4, April 2003, pp. 53–59.

[15] GNU Project Web server, `http://www.gnu.org/software`.