A Heterogeneous and Distributed Co-Simulation Environment

Alexandre Amory, Fernando Moraes, Leandro Oliveira, Ney Calazans, Fabiano Hessel
Pontifícia Universidade Católica do Rio Grande do Sul (FACIN-PUCRS)
Av. Ipiranga, 6681 - Prédio 30 / BLOCO 4 - 90619-900 - Porto Alegre – RS – BRASIL
{amory, moraes, laugusto, calazans, hessel}@inf.pucrs.br

Abstract

This paper presents the implementation and evaluation of a hardware and software co-simulation tool. Different simulators, which can be geographically distributed, compose this environment. The communication between simulators is done using a co-simulation backplane. The co-simulation backplane reads a file describing how the modules are connected, automatically launches the simulators and controls the simulation process. A case study is used as a benchmark to validate the implementation and to evaluate the system performance using different hardware-software partitions.

1. Introduction

Complex embedded systems require the design of hardware, software and, in some situations, analog devices and mechanical parts, too. Different languages and models of computation (MOC) may be used to describe these domains. This kind of system is characterized as a multi-language heterogeneous system [2],[3]. On the other hand, there is the homogeneous approach, where a single language or MOC is used to describe all types of modules [2],[3]. The main limitation of such approach is the absence of a language able to express the semantics of all MOCs related to a generic embedded system. Some approaches propose to add specific libraries to support MOCs not easily modeled with the chosen language constructs (e.g. SystemC). The heterogeneous multi-language model approach, used in this work, keeps the conceptual differences of each domain, describing each module of the system using an appropriate and possibly specific language. The main challenge of this approach is to define a mechanism to control and synchronize the interaction between heterogeneous modules described using distinct formalisms.

The current practice of system design the different modules of a heterogeneous system are usually designed and validated separately. When the overall system functionality has to be validated, this approach may not be possible for a system-on-a-chip design.

Figure 1 presents different levels where co-simulation can be performed. At the system level, the main goal is to characterize the system functionality. At this level, the software part of the system is described using a programming language, such as C or C++. The hardware part is described using a hardware description language, such as VHDL or Verilog. The communication protocol between hardware and software is abstracted at this level. At the architectural level the hardware/software communication is taken into account together with the target device. At the cycle-level simulation, the software components are simulated using the binary code running on a cycle level simulator of the target processor at the host machine. The hardware components are simulated at RTL level, with possibly back-annotated physical parasitic components.

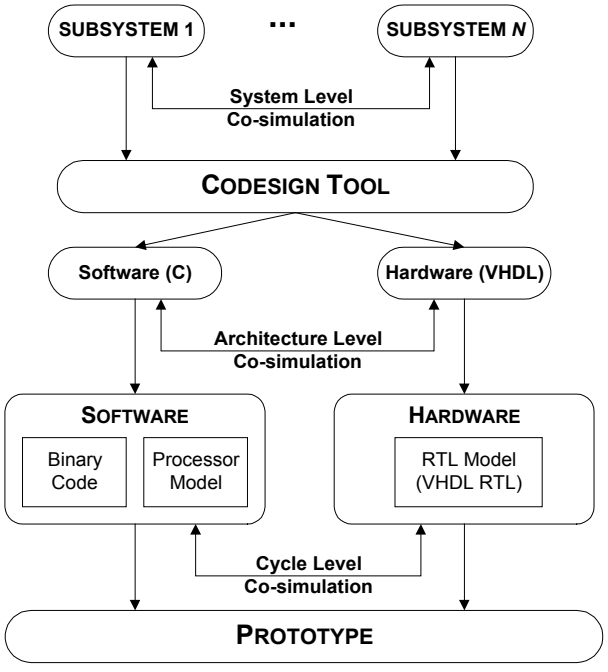

Figure 1. Co-design flow, highlighting co-simulation at different abstraction levels [2].

Proceedings of the 15 th Symposium on Integrated Circuits and Systems Design (SBCCI'02)
0-7695-1807-9/02 $17.00 © 2002 IEEE

Therefore, the task of a co-simulation tool is twofold: (*i*) to validate through simulation each module of the design, according to the language used to describe the module (C, VHDL, JAVA, SDL, …); (*ii*) to manage the communication between different simulators. Formal verification tools are another possibility to validate hardware or software modules, but this approach is still immature [1].

This paper presents the implementation and evaluation of a geographically distributed co-simulation tool. Currently, this co-simulation tool validates heterogeneous designs at the system and RTL levels, with the software part described in C/C++ and the hardware part described in VHDL.

Different hardware/software partitions of a simple benchmark are used to evaluate the tool in terms of CPU time to fulfil the co-simulation in a Local Area Network (LAN) and in a Wide Area Network (WAN). The paper also presents an evaluation of the CPU time spent with simulation and communication. This analysis indicates the overhead induced by the backplane, pointing out where the co-simulation can be improved.

The paper is organized as follows. Main concepts related to co-simulation are presented in Section 2. In Section 3, the architecture of the co-simulation environment is introduced. The benchmark used to validate the tool and preliminary results is presented in Section 4. Finally, Section 5 provides some final remarks and presents possibilities for future work.

2. Co-simulation Classifications

2.1. Criterion: number of simulators

A *homogeneous* co-simulation model translates the description language of the modules composing the system to a single language, called *intermediate format*, able to describe the whole system. In this approach only one simulator is required.

A *heterogeneous* co-simulation model employs a dedicated simulator for each language employed. In this approach, a backplane is required to coordinate the communication between simulators.

2.2. Criterion: simulation execution timing model

A *functional* model is used to validate the system at higher abstraction levels, where temporal and communication details are neglected.

A *temporal* model allows more accurate validation, since timing information, processor architecture and bus model can be taken into account when simulators are interacting. The common approach is to employ the same global time for all simulators. This type of validation is used to characterize the system at the cycle level and it is longer to simulate than the functional model.

2.3. Criterion: distributed and local co-simulation

The *distributed* co-simulation allows parallel execution of simulators in geographically distributed machines over a LAN or over a WAN. The benefits of such approach are: (*i*) project decentralization; (*ii*) design and validation of a system under development by geographically distributed teams; (*iii*) intellectual property management – a core provider may allow the simulation of an IP without giving out its description (source code); (*iv*) simulator's license management, since simulators can be installed only in a few set of machines; (*v*) resource sharing. The main drawback of this approach is the increase in the co-simulation execution time, as a result of the network communication overhead introduced.

The *local* co-simulation has no communication overhead, as a network is not used for message exchange. On the other hand, it does not present the advantages of distributed co-simulation due to its lack of connectivity.

3. Architecture of the Co-simulation Environment

Figure 2 presents the main components of the co-simulation environment (light colored) and the user project (dark colored). To use the environment the user must add the appropriate communication library (*ComLib*) to each module. The description languages, C and VHDL, and the respective simulators, *gcc* and *QuickHDL*, must have their own communication library. This figure will be used in the rest of the paper to describe the environment.

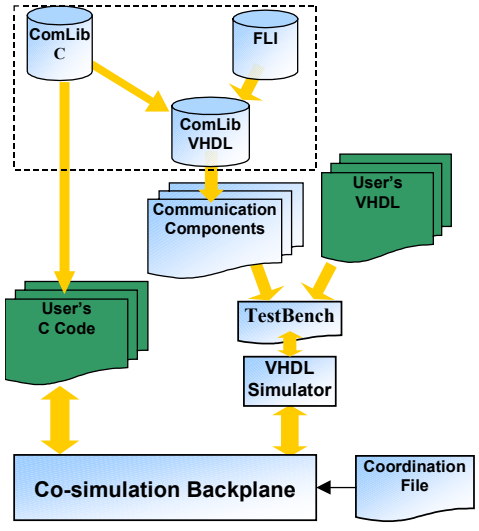

Figure 2. General structure of the co-simulation environment.

The *ComLibC* and *ComLibVHDL* are libraries responsible for encapsulating the communications primitives to the backplane. *FLI* is a VHDL extension to allow interfacing the VHDL simulator to the C language. The *communication components* obtained using the primitives of the ComLibVHDL, and the user modules are instantiated in the *testbench.* The communication between the *testbench* and the C modules is coordinated by the co-simulation backplane.

3.1. Describing Modules for Co-Simulation

- **Software Modules**

Software modules are processes executing C code. In order to be used in the co-simulation environment, the C source code requires just some simple modifications. As shown in Figure 3, the first modification in the C code is the use of the *ComLib* header, on line 1, and the addition of the initialization function (*csiInitialize()*), on line 5,as the first valid command. The *csiSend*() function, used on lines 8, must be added when data are required to be sent to the backplane. The parameters for this function are the port name and the output value. The *csiReceive*() function, used on line 9, must be included to allow the module to receive data from other modules. The parameters are the receiving port for the incoming data, and the variable to hold received values.

Figure 3a shows the C code for the *pong* module, and Figure 3b. the *ping* module VHDL. The connection is done by a *coordination* file explained later. The *ComLib* header is included on line 1, Figure 3a. The initialization function is on line 5, the receive (line 8) and send (line 9) functions are associated to ports B and A respectively.

```
1.  #include "csi.h"
2.  #include <stdio.h>
3.  void main(int argc, char
    *argv[])
4.  {   int x;
5.      CSI csiInitialize( );
6.
7.    while(1) {
8.      csiReceive("B",x);
9.      csiSend("A",x+1);
10.
11.     printf("x  = %d\n",x);
12.   }
13.   getchar();
14. }
```

(a) C pong code

```
1.  architecture tb_arch of ping_tb is
2.  --  component ComLib
3.  --  component ping
4.  begin
5.
6.    -- Unit Under Test Port Map
7.    UUT : ping port map(A=>A,B=>B );
8.
9.    -- Comunication Components
10.   ComLib1 : ComLib port map
            ( input=>A, output=>open);
11.
12.   ComLib2 : ComLib port map
            (input=>open, output=>B);
13.
14.   -- Clock And Reset Generation
15.
16. end tb_arch;
```

(b) VHDL testbench for the ping module

Figure 3. Simple example: *ping-pong* source code, ready for co-simulation.

- **Hardware Modules**

Hardware modules are described in VHDL. The interface between the hardware description language and the outside world is simulator dependent. In this work the QuickHDL simulator is used in conjunction with the FLI library [4]. For Verilog descriptions, the PLI library should be used instead [4]. The user does not need to change the hardware modules. However, a testbench must be created, instantiating the hardware and the communication components, explained later in Section 3.3. The testbench box in Figure 2 represents the instantiations. A *communication component* is instantiated for each module port. A skeleton of the testbench describing the *ping* module, is presented in Figure 3b. Line 7 and lines 10-12 instantiate the *ping* module, the *ComLib* module for port A and the *ComLib* module for port B, respectively.

3.2. Co-simulation Backplane

As stated before, the main function of the backplane is to coordinate the communication between different simulators. The connections between modules are described in a *coordination file*. This file has two parts: the module part and the connection part. The module part describes each module of the system being simulated. Each module has a unique name, a language in which it is described, a simulator used to validate it, the CPU where the module will be simulated, and the name and direction of each external module pin. The connection part is a simple netlist describing the intermodule connection for each net. Each net has a unique name, one sender port and one or more receiver ports.

The *coordination file* for the *ping-pong* example is presented in Figure 4.

```
1.   module ping {
2.       language("VHDL");
3.       simulator("QuickHDL");
4.       machine(name="hios.inf.pucrs.br",alias="hios");
5.       port {      int A : in;   int B : out; }
6.   }
7.   module pong {
8.       language("C");
9.       simulator("ddd");
10.      machine(name="wacken.inf.pucrs.br",alias="wacken");
11.      port {    int A : out;  int B : in;}
12.  }
13.  net N1 { A(ping),A(pong) }
14.  net N2 { B(ping),B(pong) }
```

Figure 4. Coordination file example.

The project has two modules, named *ping* and *pong*. The module *ping* is described in VHDL, simulated with QuickHDL, in the *hios.inf.pucrs.br* machine, and ports A and B are declared as input and output ports respectively, of integer type. The module *pong* is described in the same manner. The net definition follows the modules' specification. In this example we have nets N1 and N2. Net N1 is connected to port A of module *pong* (sender) and to port A of module *ping* (receiver).

Using the coordination file the co-simulation backplane builds the internal data structure to control the simulation, enabling the routing of messages between

modules. The integer data type is used by the backplane. The communication libraries implement the necessary data conversion, as *std_logic_vetcor* to integer and vice-versa.

3.3. Communication Library

The communication library, *ComLib*, integrates a simulator to the backplane through UNIX sockets [7]. This library has three functions: initialization (*csiInitialize()*), send data (*csiSend()*) and receive data (*csiReceive()*). The function *csiInitialize()* connects a module to the backplane. It receives as parameters the name of the module to be simulated, and the backplane IP address. The sending and receiving functions receive two parameters: the name of the port and the data to be sent or received.

At present, the environment *supports only C/C++ and VHDL languages*. It is important to add here that the C library can also support C language variations as, for instance, SystemC [5]. Libraries to support Java and SDL will be implemented in a near future.

Figure 2 shows the structure of the C and VHDL libraries. The C library, *ComLib*, could be seen as a simple overloading of standard functions to handle sockets in C. The VHDL simulator communicates with the backplane using the *communication components*. The components are instantiated in the *testbench* to connect each VHDL port to the backplane, as shown in Figure 3b, lines 10 and 12. The *communication components* are implemented using the *ComLib* and a proprietary library called FLI. This library allows implementing VHDL entities or functions in the C language.

3.4. Using the Co-simulation Environment

The first step to use the co-simulation environment is to change manually the user's project adding the *ComLib* primitives for initialization and send/receive. The next step is to describe the coordination file, including the module's name, the language, the machine where it will run, the ports and the connection between modules. Now the backplane can be run. It launches automatically the modules and wait for connection of all the modules. After this, the backplane starts the simulation.

4. Results

Validation was conducted using a simple case study. The case study is an algorithm to fill non-concave polygons [6]. This algorithm has been chosen since it allows an easy description of different hardware-software partitions and depending on the partitions the communication between blocks is very intensive (see field #msgs in Table 2), allowing the evaluation of the communication overhead. This example may not be a good representative of the complexity found in generic embedded systems, but it was chosen as the objective here is to show co-simulation problems.

The algorithm receives a set of (X,Y) coordinates representing a polygon, and it generates the horizontal lines to fill this polygon. Figure 5 illustrates a polygon and a filling table data structure that stores the horizontal lines. The Y coordinate is used as an index for the filling table. For example, row Y_4 of the filling table has the X_3 (begin) and X_4 (end) coordinates of a horizontal line used to fill the Y_4 line of the polygon, as illustrated in Figure 5. The input image can have thousands of polygons, and consequently, the same number of filling tables is created. This behavior involves a large amount of memory accesses. Thus, depending on the hardware/software partitioning, there can be a heavy traffic of messages among modules.

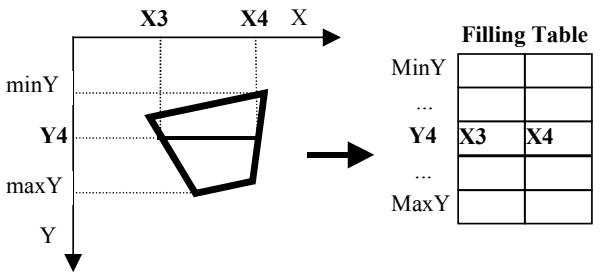

Figure 5. Example polygon and filling table data structure for the polygon-filling algorithm.

4.1. Evaluation Environment

Three different scenarios are evaluated, always using the polygon-filling algorithm.

a. Performance comparison between C Compiled Code (CCC), single simulator VHDL simulation (SSVS) and a hardware-software partition (CO-SIM). The CCC case represents the pure software implementation and the SSVS case the pure hardware implementation. These scenarios put in evidence *the impact of the message traffic in a co-simulation environment*;
b. Performance comparison between different hardware-software partitions modeled in different abstraction levels. In this case, the impact of *the abstraction level* is evaluated;
c. Performance comparison between a LAN and a WAN. This scenario allows the *performance evaluation of a geographically distributed co-simulation*.

For all scenarios, the algorithm was evaluated with 50, 100, 500, 1000 and 3000 polygons. The first and the second scenarios run on WS1, detailed below.

For the third scenario, a set of 6 workstations was used to validate the geographically distributed co-simulation: (1) WS1: Sun Ultra10, 333 MHz, 256 MB RAM; (2) WS2: Sun Ultra1, 167 MHz, 192 MB RAM; (3) WS3: Sun SPARC 4, 110 MHz, 72 MB RAM; (4) WS4: Sun SPARC 4, 110 MHz, 72 MB RAM; (5) WS5: Sun SPARC 4, 110 MHz, 64 MB RAM; (6) WS6: Sun Ultra2, 296 MHz, 256 MB RAM.

The WS1 to WS5 boxes are connected on a LAN Ethernet 10Mbps. The WS6 is on a WAN 5 hops way. All C programs were compiled using GNU GCC compiler version 2.95.3 without optimization. The VHDL simulator used in all experiments was Mentor Graphics QuickHdl (qhsim v8.5_4.6i). All data presented in the tables represent average values, obtained through several executions.

4.2. The Partitions

Two hardware/software partitions were used, both with five blocks, where four of them are written in C and one of them is written in VHDL. Figure 6 shows the communication mechanism between blocks, with the arrows representing nets. The numbers associated with the arrows represent the number of nets between the blocks connected by the arrow. When two numbers are associated to each arrow, the numbers in parenthesis represent the number of nets in the first partition. When there is only one number in the arrow, it represents the number of nets in both partitions.

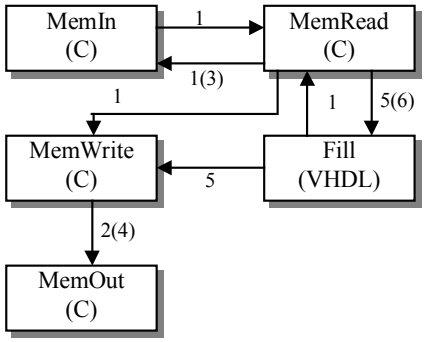

Figure 6. Inter-module connections and the number of nets in each hardware/software partition.

The main difference between these partitions is the number of nets involved in the co-simulation. The number of nets represents the abstraction level employed to describe the design. The first partition, called "C + VHDL", has 20 nets. This partition contains data and control nets, representing an architectural view of the system. On the other hand, the partition called "C + VHDL (optim)" has 15 nets, representing a higher abstraction level description of the design. It is composed of only data nets, without control, being a functional description of the system.

4.3. Results and Discussion

Table 1 presents the CPU time to simulate the software version (CCC), the hardware version (SSVS) and a hardware-software partition (CO-SIM). As expected, CCC and SSVS, consumes little CPU time, since there is no message traffic. On the other hand, the CO-SIM is slower because it has thousands of inter-process communication[1] messages, as shown in Table 2. The advantages of co-simulation, when compared to the high CPU usage to co-simulate the benchmark, have already been depicted in Section 2.

The bottleneck of the benchmark, related to the communication inter-modules, is located at the memory blocks interface. The control nets, as CE (*chip enable*) and RW (*read write*) were removed, reducing significantly the total number of messages. These data are presented in Table 2. An important reduction on the running time was observed. This fact confirms that the co-simulation running time is dominated by the messages being routed by the co-simulation backplane.

# Polygons	CCC	SSVS	CO-SIM
50	0.002	4.4	34.8
100	0.003	4.9	69.4
500	0.017	9.4	352.8
1000	0.033	15.4	703.3
3000	0.100	40.0	2109.2

Table 1. Execution time, in seconds, for different validation mechanisms.

# Polygons	C + VHDL		C + VHDL (optim)	
	time	# msgs	time	# msgs
50	113.7	90495	34.8	56342
100	224.2	183527	69.4	114407
500	1120.9	924745	352.8	576312
1000	2160.3	1840089	703.3	1250308
3000	9608.2	5522387	2109.2	3755731

Table 2. Co-simulation time, in seconds, for two partitions.

These preliminary results can be improved by modifying the routing procedures, e.g., using UDP instead of TCP and message buffering.

The next scenario evaluates the communication overhead of the network in order to complete the co-

[1] There is no network communication since it runs only on WS1

simulation. Table 3 presents the results for three configurations. In the first one, all modules are executed in a single CPU, WS1. In the second case, the co-simulation is executed in 5 different CPUs (WS1 to WS5), belonging to the same LAN. In the third case, one external CPU is added, WS6, replacing the WS5 of the previous case. The CPUs are characterized at Section 4.2.

The total co-simulation time is almost the same in the three cases: single machine, LAN and WAN. We could expect an improvement of the running time, since simulators are been executed in parallel. This is not observed, because individual blocks are very simple, being quickly executed. So, the running time is again a function of the number of messages being routed.

# Polygons	Single Machine	Distributed only LAN	Distributed LAN + WAN
50	34.8	41.0	36.3
100	69.4	76.4	72.5
500	352.8	423.1	368.1
1000	703.3	725.5	751.9
3000	2109.2	2177.4	2238.8

Table 3. Co-simulation time, in seconds, for single machine, LAN and WAN Co-simulation.

To put in evidence the reduction on the running time the benchmarks must be CPU dominated, instead traffic dominated.

These results show us we can have distributed modules over the network, with little impact on the overall running time, corroborating the advantages claimed for geographically distributed simulation.

5. Conclusions And Future Work

The main contribution of this paper is the development and the evaluation of a co-simulation tool, by means a *co-simulation backplane*, responsible for the integration of simulators. The main features of the co-simulation tool are: (*i*) the *co-simulation backplane* automatically launch simulators with no manual intervention; (*ii*) geographically distributed co-simulation is allowed; (*iii*) VHDL and C languages (as well as C variations) are supported; (*iv*) functional and RTL architectural models can be used. Cycle level co-simulation is yet not supported, since the backplane does not have a global synchronization mechanism.

One weakness of the currently implemented co-simulation environment is related to its lack of portability. The C/VHDL communication interface is not a standard, changing from simulator to simulator. The QuickHDL simulator, used in this work, provides the FLI library. The Aldec simulator provides another library, called VHPI. So, if the VHDL simulator changes, some interfaces of the system must be re-written.

It is important to emphasize the backplane's flexibility. The backplane is independent from simulators. New languages, such as Java and SDL, can easily be integrated in the co-simulation environment. The user only needs to implement in these new languages the communication library.

As expected, the communication overhead was very important. We believe this overhead can be reduced using UDP sockets and message buffering. This optimization is under development. Integration to Java and SDL and more detailed performance evaluation benchmarks are examples of future work.

As an academic work this environment can be freely distributed to other research groups.

6. References

[1] Edwards, S., Lavagno, L., Lee, E., and Sangiovanni-vincentelli, A, "Design of Embedded Systems: Formal Models, Validation and Synthesis", Proceedings of the IEEE, 1997.

[2] Hessel, F., "Concepção de Sistemas Heterogêneos Multi-Linguagens", Jornada de Atualização em Informática – JAI, XXI Congresso da Sociedade Brasileira de Computação, 2001.

[3] Liem, C., Nacabal, F., Valderrama, C., Paulin, P., Jerraya, A., "System-on-a-chip Cosimulation and Compilation", IEEE Design & Test of Computers, Vol. 14, Issue 2, 1997, pp. 16 –25.

[4] Modeltech inc., "ModelSim SE/EE User's Manual", Version 5.4. April, 2000.

[5] Swan, S., "An Introduction to System Level Modeling in SystemC 2.0", white paper, Available at homepage: http://www.systemc.org. 2001

[6] Foley, J.D., et al., *Computer Graphics: Principles and Practice*. 2 edn, Addison-Wesley, 1997.

[7] Walton,S., *LINUX Socket Programming,* SAMS, 2001.

Acknowledgements: Fernando Moraes gratefully acknowledges the support of the CNPq through research grant number 522939/96-1.