An Integrated Hardware-Software Cosimulation Environment with Automated Interface Generation

Kyuseok Kim
LG Electronics Research Center
Information Technology Laboratory
Seoul 137-140, Korea
kskim@erc.goldstar.co.kr

Yongjoo Kim, Youngsoo Shin, and Kiyoung Choi
Seoul National University
Dept. of Electronics Eng.
Seoul 151-742, Korea
kchoi@azalea.snu.ac.kr

Abstract

We present a hardware-software cosimulation environment for heterogeneous systems. To be an efficient system verification environment for the rapid prototyping of heterogeneous systems, the environment provides following features: interface transparency, smooth transition to cosynthesis, simulation acceleration, and integrated user interface and internal representation. Among them, the first two are more important than the others. To support these two features, we have developed automatic interface generation schemes. As demonstrating experiments, two heterogeneous systems performing same function with different target architectures were cosimulated and prototyped successfully in our environment. The experimental results show that our environment can be a useful heterogeneous system specification/verification environment for rapid prototyping.

1. Introduction

Cosimulation refers to the simulation of heterogeneous systems whose hardware and software components are interacting. Traditionally, the task has been performed only after the prototype hardware became available and with the help of in-circuit emulators and/or other techniques [14]. With hardware-software codesign, it is essential to verify functionality even before hardware is built. In contrast to the conventional(or homogeneous) simulation of digital hardware, cosimulation should care for the interactions among hardware and software components. To model the interactions during cosimulation in exact and efficient manner, target architecture and interface among the components must be considered.

The available techniques for hardware-software cosimulation trade off among a number of factors such as performance, timing accuracy, and model availability [14]. The model availability of processor in target architecture dominates the choice of techniques. Becker, et al. [5] performed cosimulation of a network interface unit on a distributed network using UNIX socket. They used C++ and Verilog in describing the software component and the hardware component, respectively. Their cosimulation is a combination of synchronized handshake and cycle accurate processor model. Cosimulation scheme of Thomas, et al. [18] is similar to [5], but their technique is based on synchronized handshake with no processor model. Cosimulation techniques of Poseidon [7] and Ptolemy [8] need pin-level model of processors. Their approaches are most accurate but take much more simulation time. Although above existing approaches vary in model availability, accuracy, and time, they all did not take into account interface model explicitly or efficiently.

In this paper, we present a hardware-software cosimulation environment for heterogeneous systems. To be an efficient system verification environment for the rapid prototyping of a heterogeneous system, the environment provides interface transparency, smooth transition to prototype synthesis from simulation, simulation acceleration, and integrated user interface and internal representation. Among those features, the first two are more important than the others. To support these two features, we have developed automatic interface generation schemes. The resultant benefits of those features are no need of processor models, target architecture independence, and the conceptual simplicity and easiness in establishing and expanding the environment.

The rest of this paper is organized as follows: Section 2 presents the overview of our cosimulation environment. Section 3 describes automatic interface generation schemes. Section 4 describes the application of our cosimulation environment to real systems prototyping as experimental examples. Then Section 5 concludes with some remarks on future work.

0-8186-7603-5/96 $5.00 © 1996 IEEE

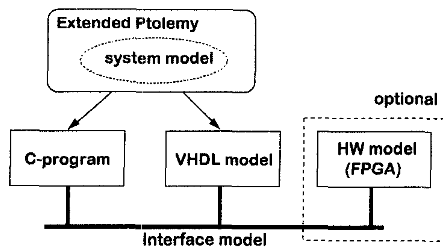

Figure 1. Cosimulation environment.

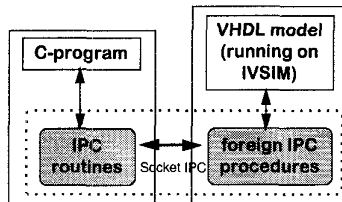

Figure 2. Cosimulation at abstract level.

2. Cosimulation environment overview

As shown in Fig. 1, the environment [11] has three major elements for the execution of cosimulation: a software process running C program, a simulation process executing hardware model in VHDL, and interface model. A custom board is optional for simulation acceleration by emulating whole or part of hardware component. Interface model is based on inter-process communication (IPC) routines which connect the two processes through UNIX Socket [17] on a single SUN Sparc CPU. Ptolemy [6], which is a framework for simulation and prototyping of heterogeneous systems, is extended to provide user interface and internal representation for system specification and verification. The environment supports cosimulation at any abstraction level. Initially, the specification of a heterogeneous system is given in VHDL and C for hardware core and software components. Then simulation models for the interface are generated automatically and then combined with the cores, thereby allowing cosimulation at very abstract level. Fig. 2 represents the abstract level cosimulation. As shown in the figure, the interface simulation models are mainly IPC routine calls with appropriate parameters. After a target architecture is determined and an interface is synthesized, more detailed simulation models for the interface are generated and inserted, thereby allowing detailed level cosimulation.

Since we use 'synchronized handshake' simulation technique [14], there is no need for processor models. Using this technique, the software can run at the workstation speed even though overall speed will be dominated by the hardware simulator performance.

The simulator, IVSIM (SNU ISRC VHDL SIMulator),

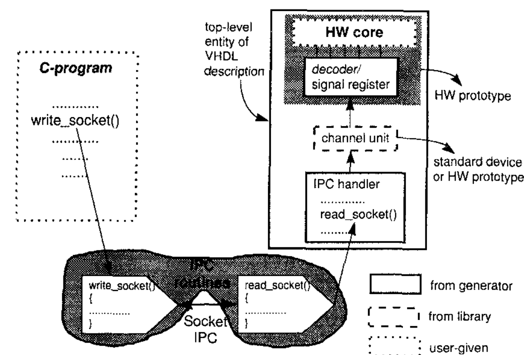

Figure 3. Interface generation or selection from library for detailed level cosimulation.

is a VHDL simulator based on an event-driven compiled code simulation algorithm. It can recognize and support 'foreign' attribute defined in VHDL-93 [4]. The attribute enables a system to be described by not only VHDL but also non-VHDL procedures such as IPC routines in C language.

2.1. Interface transparency

Our cosimulation environment provides the users with transparency about communication interface between software core and hardware core regardless of the type of target architectures and communication protocols. This is especially important for detailed level cosimulation.

Once the user selects the target architecture and communication protocol, he or she can concentrate on the functionality simulation of the hardware and software components without concerning about the details of the interface or communication.

To provide the interface transparency, we implemented automatic interface model generation and instantiation. Appropriate simulation models for the interface between the two components are 'created' by automatic interface model generation with parameters according to the chosen communication protocol and the target architecture as shown in Fig. 2 and Fig. 3. Interface simulation models should also be 'inserted' at appropriate places in the C program and VHDL model in automatic manner. Fig. 3 shows the relevant interface elements for hardware part and software part, and how they are created (or selected) and combined to provide the overall simulation model of interface and enable detailed level cosimulation. If the users give only the hardware and software cores, the interface elements are automatically created by generators or selected from libraries.

The function of each interface element is as follows:
(i) IPC handler takes care of reading/writing data from/to

the software process using foreign IPC routines. It handles IPC jobs by handshaking. It also transfers events from software core to hardware core through channel unit, which consequently activates VHDL simulation. It is created by a generator.
(ii) Channel unit represents the abstract simulation model of a physical channel device. It is selected from interface library.
(iii) Signal register stores data to be transferred between hardware component and software component. Protocol converter makes hardware core connected to system bus of target architecture by interfacing between system bus and signal register.
(iv) Top-level entity acts as a top-level container which gathers all interface elements. Interface elements are interconnected using component instantiations and signals declared in the entity.

The hardware core, signal register and protocol converter will be mapped into real hardware prototype. When standard bus architecture is used, a standard channel device such as SBus DMA controller [1] can be used as a physical device for channel unit. If user-defined bus or channel is used, channel unit should be a part of hardware prototype.

2.2. Smooth transition to cosynthesis

After detailed level cosimulation, the system components must be synthesized as physical components on the selected target architecture. For the cosynthesis, the invokes to the IPC routines are replaced with the corresponding device driver calls or I/O function calls - which are called driver functions, collectively - for the software component. For the hardware component, top-level entity with foreign interface procedure declaration is stripped off and the hardware interface model remains to be synthesized together with hardware core. The driver functions and the software core will be compiled and linked together to make executable codes. Since this modification of each component specification for the cosynthesis is very simple and limited to a minimum degree, it is possible to make a smooth and fast transition from cosimulation to the cosynthesis of system prototype in our environment. Fig. 4 depicts the transition from cosimulation to cosynthesis.

2.3. Simulation acceleration

In cosimulation using synchronized handshake technique, hardware simulation time dominates overall cosimulation time [14]. This problem can be alleviated by simulation acceleration. It consists of a CPU(SUN Sparc processor) and a custom board. The CPU is in charge of running VHDL simulation process, C program process for software component, and CAD tools for the synthesis of hardware prototypes. Currently, the custom board consists of an FPGA and bus interface. The FPGA is used to implement a part of hardware model as real hardware. The communication between the CPU and the custom board is done through SBus [3]. To interface between SBus and real hardware, we used a SBus DMA controller IC [1].

Presently, we restrict to the case which CPU is always the bus master of SBus transactions. To send(receive) data to(from) the hardware prototype, software process and VHDL simulation process should write(read) data to(from) the device driver program. The software and VHDL simulation processes communicate each other through socket IPC as mentioned above.

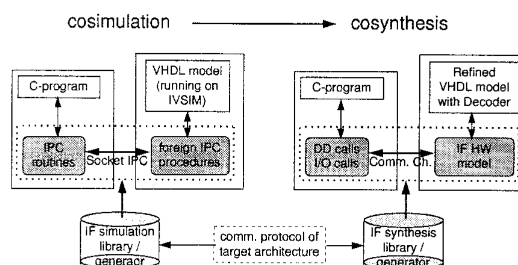

Figure 4. Transition to cosynthesis from cosimulation.

2.4. Integrated user interface and internal representation

To provide an integrated user interface and internal representation, we extended Ptolemy. Ptolemy [6] is a block-diagram oriented environment for simulation and prototyping of heterogeneous systems.

For hardware-software codesign, we are currently developing Hetero Domain where heterogeneous models may coexist in the same representation of Ptolemy.

3. Automatic interface generation

To generate interface models for cosimulation and cosynthesis, we developed an interface generation technique. Interface generation starts from a partitioned control/data flow graph (CDFG). The graph is the internal representation of the systems to be designed in our environment and consists of hardware CDFG and software CDFG. We generate hardware interface and software interface from these graphs as shown in Fig. 5. Fig. 6 shows system CDFG and partitioned CDFG. After partitioning the system CDFG (a) into software CDFG (b) and hardware CDFGs (c), special nodes

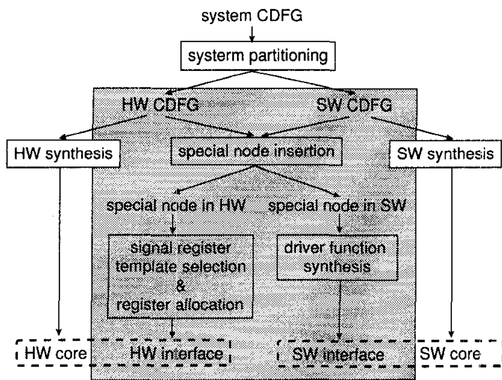

Figure 5. Interface generation flow (shaded area) in heterogeneous systems prototyping.

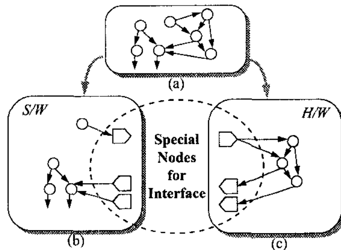

Figure 6. System CDFG and partitioned CDFG.

('send' and 'receive' nodes) for interface is added to the partitioned CDFG at the partitioning boundary. Each special node has its own counter node. Hardware and software interface modules are generated from these special nodes in the hardware CDFG and software CDFG, respectively, according to the selected target architecture and communication protocol. The software interface module combines device driver routine calls, I/O function calls and load/store commands to read/write data from/to system bus. The hardware interface module consists of signal register and protocol converter as shown in Fig. 7. The signal register stores data to be transferred between hardware component and software component. The protocol converter connects the hardware core to the system bus of the target architecture by interfacing between the system bus and the signal register.

3.1. Hardware interface generation

Depending on whether the target architecture is already defined or not, the protocol converter can be instantiated

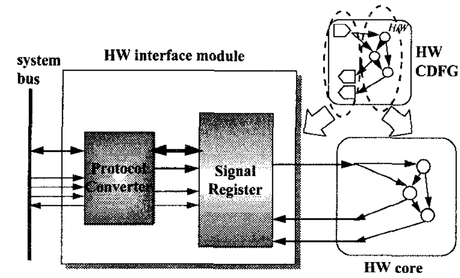

Figure 7. Hardware interface module.

```
allocate (width)
{
Q = ∅;
if (width > W) {
      for (p ∈ P   s.t.   w = W) {
            Q = Q ∪ {p};
            P = P - {p};
            Q = Q + allocate (width - W);
      }
}
else
      for   (p ∈ P   s.t.   w >= width)   {
            P = P - {p | w_p = width} ∪{q | q ∈ P, w_q = w_p - width};
            Q = Q ∪ { p | w_p = width };
      }
}
return Q;
}
```

Figure 8. Signal register allocation algorithm.

or generated. When the architecture is defined before, the protocol converter can be instantiated automatically from interface library since it is already generated before according to the architecture. For newly defined architectures, protocol converter should be generated using the algorithm in [13] and stored in the library for future instantiations.

The signal register is generated in two phases - register template selection and signal register allocation. In the register template selection phase, an appropriate template for signal register is selected from a template library according to the characteristics of I/O port of hardware core. Each template differs in the configuration of components of signal registers: multiplexers, decoders, buffers, and registers. An example of signal register instantiated from template library is shown in Fig. 10. Currently, four templates are supported [9]. In the signal register allocation phase, input and output of selected signal register templates are allocated to each port of the hardware core. Using the information of special nodes such as port name, I/O direction, and bit width, register outputs and multiplexer inputs are bound to the input and output ports of hardware core, respectively. Whenever the bit width of a port to be bound is wider than bit width of single data transfer of system bus, a data transfer through the port must be done by multiple data transfers.

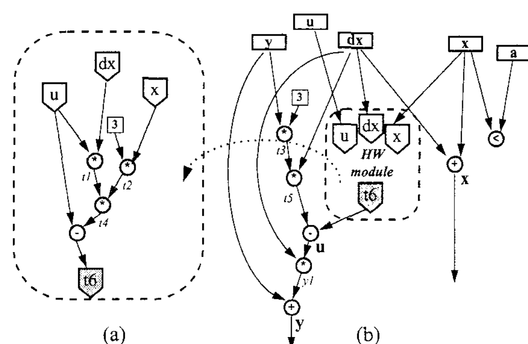

Figure 9. (a) Hardware CDFG, (b) CDFG for the solution of differential equation: $y'' + 3xy' + 3y = 0$.

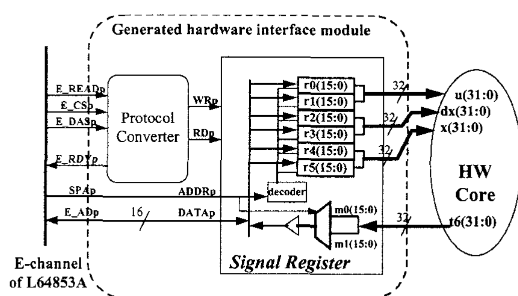

Figure 10. Hardware interface generated for the CDFG in Fig. 9.

Signal register allocation algorithm is presented in Fig. 8. The algorithm returns the set of allocated registers for each ports. P, Q, W, w, and wx represent the set of registers not yet allocated, the set of allocated registers for each port, the bit width of system bus, the bit width of register, and the bit width of register x, respectively.

Example 1. A partitioned CDFG is given as in Fig. 9. We assume that all edges in the CDFG are 32-bit integer type and the bit width of a system bus is 16 bits. The signal register allocation algorithm will generate a signal register for the CDFG as shown in Fig. 10. Since bus width is 16 bits in the target architecture, inputs of a multiplexer (m0, m1) are allocated to a send node t6. Register allocation will be done in the same way for other special nodes.

3.2. Software interface generation

Driver functions are synthesized for each special node using the information of special node in software CDFG such as function name, function type, parameter type, and mapping results between the ports of hardware core and signal registers. The functions take charge of sending/receiving data to/from hardware core.

Example 2. For the CDFG in Example 1 (shown in Fig. 9), the driver function generated for the receive node t6 of the software CDFG will be as follows:

```
int receive_t6 ( )
{
union_type      union_temp;

ioctl(ddps, RDE_M0, union_data.ch);
union_temp.ush[0] = union_data.ush[0];
ioctl(ddps, RDE_M1, union_data.ch);
union_temp.ush[1] = union_data.ush[0];
return union_temp.in[0];
}
```

The driver function calls device driver (here, 'ddps') two times and combines the two received data (16 bits each) into single integer data (32 bits). This is because the send node t6 of hardware CDFG is mapped to m0 and m1 inputs of the multiplexer in the hardware interface module as shown in Fig. 10.

Driver functions and the core software module which is synthesized from software CDFG [16] can be compiled and linked together to make executable codes.

Table 1. Target architectures and communication protocols used in the experiments

	Experiment 1	*Experiment 2*
CPU	Sun Sparc	Intel 80486DX-33
Operating system	Sun OS 4.1.3	DOS 3.0
HW device	Xilinx XC4010	Xilinx XC4025
System bus	L64853A SBus DMA controller E-channel	ISA bus

4. Experimental results

4.1. Experiment 1

A lossless data compression system was cosimulated and cosynthesized using our environment. Initially, the system was only a C program implementing Lempel-Ziv lossless data compression algorithm [19].

The system was manually partitioned into software and hardware components resulting in a mixture of a hardware

component implementing parsing step and a software component implementing the remaining steps - initialization, coding, buffer updating, and file I/O. After inserting IPC routine calls in the components, we performed abstract level cosimulation. Once the target architecture and communication protocol were determined as shown in Table 1, hardware and software interface elements were generated or selected from library and added to the hardware component and software component for detailed level cosimulation. Combining all simulation models, the detailed level cosimulation was done successfully and the result was identical to that of the abstract level cosimulation.

Then the hardware component of the system with buffer size of 16 was prototyped with an FPGA [2] using the architecture in [10]. The resultant hardware has taken 645 CLBs including interface. With the clock frequency of 6.25 MHz for FPGA, the prototyped heterogeneous system shows speedup of 1.7 over the implementation using only software component.

4.2. Experiment 2

We cosimulated and cosynthesized the same system as Experiment 1 but with different target architecture, communication protocol, and buffer size n = 32 as shown in Table 1. Signal register which matched with I/O ports of hardware core has been generated using interface module library for system bus (ISA Bus [15]). With the clock frequency of 8.33 MHz for FPGA, we obtained speedup of 2.5 over the implementation using only software component.

5. Conclusions

In this paper, we present a hardware-software cosimulation environment for heterogeneous systems prototyping. To be an efficient system verification environment for the rapid prototyping of heterogeneous systems which consist of hardware and software components, the environment supports special features: interface transparency, cosimulation acceleration, smooth transition to system prototype synthesis, and integrated user interface and internal representations. The resultant benefits of those features are the modularity of cosimulation components, no need of processor models, target architecture independence, and the conceptual simplicity and easiness in establishing and expanding the environment.

On-going and future works are as follows:
(i) Complete the implementation of the environment.
(ii) Extend the environment to more general target architectures including microprocessors or micro-controllers, digital signal processors, and ASICs.
(iii) Apply our approach to various system prototyping examples such as MPEG-2.

References

[1] *L64853A SBus DMA Controller Technical Manual.* LSI Logic, 1991.

[2] *The Programmable Logic Data Book.* Xilinx, 1993.

[3] *Standard for a Chip and Module Interconnect Bus: SBus (P1496/Draft 2.3).* IEEE Standard Department, 1993.

[4] *IEEE Standard VHDL Language Reference Manual, ANSI/IEEE Std 1076-1993.* IEEE, New York, NY, 1994.

[5] D. Becker, R. K. Singh, and S. G. Tell. An engineering environment for hardware/software co-simulation. *Proc. 29th Des. Auto. Conf.*, pages 129–134, June 1992.

[6] J. Buck, S. Ha, and E. A. Lee. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int'l Jour. of Comp. Simulation*, pages 155–182, April 1994.

[7] R. K. Gupta, C. Coelho, and G. De Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. *Proc. 29th Des. Auto. Conf.*, pages 129–134, June 1992.

[8] A. Kalavade and E. A. Lee. A hardware-software codesign methodology for DSP applications. *IEEE Design and Test of Computers*, pages 16–28, September 1993.

[9] K. Kim. Generation of interface module in hardware-software co-design. *MS Thesis, Dept. Electronics, Seoul Nat'l Univ.*, December 1995.

[10] Y. Kim, K. Kim, and K. Choi. Efficient VLSI architecture for lossless data compression. *IEE Electronics Letters*, 31(13):1053–1054, June 1995.

[11] Y. Kim, K. Kim, Y. Shin, T. Ahn, W. Sung, K. Choi, and S. Ha. An integrated hardware-software cosimulation environment for heterogeneous systems prototyping. *Proc. of Asia and South Pac. Des. Auto. Conf.*, pages 101–106, August 1995.

[12] Y. Kim, Y. Shin, K. Kim, J. Won, and K. Choi. Efficient prototyping system based on incremental design and module-by-module verification. *Proc. of Int'l Symp. Circ. and Syst. 95*, pages 924–927, May 1995.

[13] S. Narayan and D. Gajski. Interfacing incompatible protocols using interface process generation. *Proc. 32nd Des. Auto. Conf.*, June 1995.

[14] J. A. Rowson. Hardware/software co-simulation. *Proc. 31th Des. Auto. Conf.*, pages 439–440, June 1994.

[15] T. Shanley and D. Anderson. *ISA System Architecture (3rd ed.).* Mindshare, 1995.

[16] Y. Shin and K. Choi. Thread-based software synthesis for embedded system design. *Proc. Euro. Des. and Test Conf.*, March 1996.

[17] W. R. Stevens. *UNIX Network Programming.* Prentice-Hall, 1991.

[18] D. E. Thomas, J. K. Adams, and H. Schmit. A model and methodology for hardware-software codesign. *IEEE Design and Test of Computers*, pages 6–15, September 1993.

[19] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory*, IT-23(3):337–343, 1977.