Design Automation for Embedded Systems 3, 163–186 (1998)

An Integrated Cosimulation Environment for Heterogeneous Systems Prototyping

YONGJOO KIM, KYUSEOK KIM, YOUNGSOO SHIN, TAEKYOON AHN AND KIYOUNG CHOI
School of Electrical Engineering, Seoul National University, Seoul 151-742, Korea

Abstract. In this paper, we present a hardware-software cosimulation environment for heterogeneous systems. To be an efficient and convenient verification environment for the rapid prototyping of heterogeneous systems consisting of hardware and software components, the environment supports (i) modular cosimulation, (ii) cosimulation acceleration, and (iii) integrated user interface and internal representation. For modular cosimulation, we treat software and hardware components as separate processes that communicate with each other only through inter-process communication. We generate interface model automatically and insert between software and hardware components. We can accelerate cosimulation through hardware-emulation using FPGAs which also supports our incremental system prototyping strategy. Finally, to provide an integrated user interface and internal representation consistent with various prototyping tasks, we modified and extended Ptolemy, a cosimulation and cosynthesis environment for heterogeneous systems. The benefits of our cosimulation environment are as follows: expandability of the environment, target architecture and protocol independence, interface transparency, seamless transition to cosynthesis, cosimulation speedup, and convenient cosimulation. As experimental examples, we cosimulated and prototyped several heterogeneous systems successfully, which shows that our environment can be a useful heterogeneous systems specification/verification environment for fast prototyping.

Keywords:

1. Introduction

With ever-increasing complexity along with ever-tightening constraints on cost, performance, and time-to-market of systems to be designed, systematic methodologies for system development is becoming one of hottest issues in system design and design automation areas. Among them, hardware-software codesign and rapid prototyping of heterogeneous systems are the most active research areas.

Codesign is a design methodology for a system, which consists of hardware and software components. Figure 1 represents a typical design flow and the related design data in a codesign process. Inputs to a codesign system are system specification describing system's intended function or behavior along with constraints on cost and performance which the final designed system must satisfy. System description can be done using a single language (e.g., C, VHDL, HardwareC...) or a combination of several languages (e.g., C and VHDL). Or the system description can be done using a user interface such as Ptolemy. The user interface generally provides graphical tools to facilitate system specification compliant with its internal representation whether it is activity-based (e.g., CDFG, flowchart) or state-based (e.g., FSM, Petri-net).

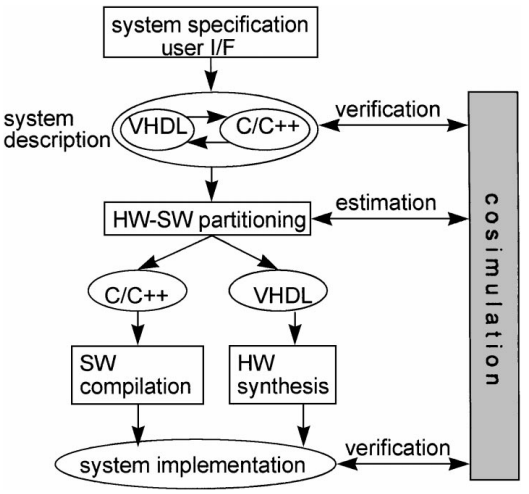


Figure 1. Generic codesign flow of heterogeneous system.

Once the system specification is translated into the internal representation suitable for the remaining codesign steps, hardware-software partitioning is done to find out the optimum solution satisfying the design constraints by exploring the design space. The design space to be searched during partitioning is much larger than for designing hardware-only or software-only homogeneous systems and the abstraction level of partitioning is very high above the physical implementation level of each hardware and software component. So, it is very hard but very important to have an efficient estimator of good quality. After partitioning the system specification into hardware and software components, synthesis for hardware and software components are done concurrently. The combination of hardware-software partitioning and concurrent synthesis of hardware and software components enables the design of heterogeneous systems to be free from many problems such as over-design or under-design related to system integration and thus can save the system development cost and cycle.

Despite that it cannot provide exhaustive verification, simulation is the most efficient and widely used design verification method. This is the same for simulation in codesign of heterogeneous systems. As one of the important steps in codesign and rapid prototyping of heterogeneous systems, cosimulation refers to the simulation of heterogeneous systems whose hardware and software components interact [1].

The typical usage of cosimulation during codesign is as follows:

- *Verification of system specification*: Verify that the behavioral part of system specification is described as the intent of system designer.
- *Verification of system implementation*: Verify that the final system implementation, the result of system integration, functions correctly and satisfies the constraints in the initial system specification.
- *Performance estimation for system partitioning*: This case is not so general as the above cases. When it is hard to obtain analytical or closed mathematical model for performance

estimation during hardware-software partitioning, cosimulation can be used as a tool for estimating performance [2].

Figure 1 shows an example of codesign flow where cosimulation is used for both verification and estimation.

The problems unique in cosimulation must address in comparison with the conventional hardware simulation are as follows:

- 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 [1]. With hardware-software codesign, it is essential to verify correct functionality even before hardware is built.
- In contrast to the conventional (or homogeneous) simulation of digital hardware, cosimulation should care for the interaction among hardware and software components. Hardware components execute concurrently. In contrast, software components execute inherently in sequential manner (assuming single processor). Therefore, for correct cosimulation of system to be codesigned, the problems such as synchronization and communication among hardware and software components must be taken into consideration.
- Trade off between simulation accuracy and speed must be addressed. As in homogeneous digital simulation, accuracy and performance in cosimulation also tend to contradict each other. In other words, the more accurate the cosimulation model is, the slower the cosimulation performance is. For cosimulation of heterogeneous systems, one needs not only the simulation model of hardware components as in homogeneous simulation but also the simulation model of the processor on which software components will execute. Simulation accuracy of systems to be designed will be determined by the accuracy of the processor model. For example, if one models the processor at the register-transfer or lower level, he can even trace the cycle-by-cycle behavior of the processor and consequently, can do even timing simulation of the overall system. But he has to spend much time and effort in preparing the processor model and running the simulation.
- When hardware and software components are represented in different languages or internal representation models, cosimulator must manage the simulation data generated dynamically among those components during the simulation in a consistent and unified manner.
- Minimization of semantic gaps between cosimulation and cosynthesis: This is the problem of not only the simulation, but also the synthesis. It largely depends on the language one uses to describe the system specification. Especially for VHDL, a standard hardware description language, which is a simulation-oriented language rather than a synthesis-oriented language, one has to resolve the gap between simulation semantics and synthesis semantics for the identical language constructs or statements. The situation can be aggravated when multiple languages are used in the system specification and language conversion occurs due to the migration of components of the system as a result of partitioning (e.g., C to VHDL or VHDL to C).

In this paper, we present a hardware-software cosimulation environment for heterogeneous systems. To be an efficient and convenient verification environment for the rapid

prototyping of heterogeneous systems consisting of hardware and software components, the environment supports (i) modularity of cosimulation, (ii) cosimulation acceleration, and (iii) integrated user interface and internal representation.

For the modularity of cosimulation, we implemented interface transparency through process modularity and automatic interface generation. We treat software and hardware components as separate processes that communicate with each other only through inter-process communication and automatically generated interface models. We simulate software component and hardware component as C-program and VHDL model, respectively. Due to the modularity of cosimulation, system designer can concentrate on hardware and software cores without concerning much about interface while designing a system.

Another benefit of the modularity is in cosynthesis after cosimulation. To synthesize components of the target architecture after cosimulation, invokes to the interface simulation models are replaced with the corresponding device driver calls or I/O function calls for software components. For each hardware component, top-level VHDL entity with foreign interface is stripped off and the corresponding synthesizable interface hardware model is inserted. The process requires minimal modification of hardware and software descriptions, and thus provides a smooth and fast transition from cosimulation to system prototype synthesis.

We can accelerate the cosimulation with hardware-emulation using FPGAs. It also supports incremental system prototyping. We can implement any hardware subcomponent whose function was already verified with FPGAs and thus can hardware-emulate to accelerate the overall cosimulation process.

Finally, to provide a user interface and internal representations consistent with various prototyping tasks, we modified and extended Ptolemy, a cosimulation and cosynthesis environment for heterogeneous systems.

The overall benefits of our cosimulation environment are as follows: target architecture and protocol independence, interface transparency, smooth transition to cosynthesis, fast cosimulation, and convenient cosimulation.

As experimental examples, we cosimulated and prototyped several heterogeneous systems successfully, which shows that our environment can be a useful heterogeneous system specification/verification environment for fast prototyping.

The organization of the rest of this paper is as follows: Section 2 gives an overview of the related work on cosimulation. Section 3 presents an overview of our cosimulation environment. Sections 4–6 describe implementation details in relation to the modularity of cosimulation—interface transparency, automatic interface generation, and transition to cosynthesis after cosimulation. Sections 7 and 8 present about cosimulation acceleration and integrated user interface and internal representation, respectively. After describing experimental applications of our cosimulation environment to real system prototyping in Section 9, we conclude with some remarks on future work in Section 10.

2. Related Work

The techniques available for hardware-software cosimulation trade off among a number of factors such as performance, timing accuracy, model availability and visibility of internal state for debugging [1]. In [1], Rowson classified cosimulation techniques into 8 classes

(nano-second accurate, cycle accurate, instruction level, synchronized handshake, virtual hardware, bus functional, hardware modeler, emulation) according to the above factors and compared them. Among the factors, the processor model availability dominates the choice of the techniques.

Poseidon, an event-driven cosimulator developed by Gupta et al. [3] at Stanford University performs concurrent simulation of multiple functional units implemented as software or application-specific hardware. Software components are compiled into assembly codes for the target processor and then run on the processor simulator. Hardware components are simulated with Mercury, a logic simulator. System graph model, an internal representation of system specification, runs on Ariadne, graph model simulator. Poseidon maintains and manages an event queue which stores events generated from these three simulators. Poseidon also executes interconnections and interface protocols among models given at system specification. Since Poseidon executes simulation of system partitioned and codesigned at operation-level, one can obtain accurate timing information such as the number of clock cycles, but at a cost of long simulation time.

Becker et al. [4] performed cosimulation of a network interface unit (NIU) on a distributed network using Cadence Verilog-XL simulator and Unix sockets. They used C++ and Verilog in describing the software and the hardware components, respectively. Models interact each other by exchanging data using socket. To link between software components and hardware components, interface functions in the software components were rewritten and simulation modules in the hardware components were modified. These parts were connected through simulator extension constructed using programming language interface (PLI) of Verilog HDL. They developed hardware and software of NIU between HIPPI link and PXPL5 ring network using this technique.

Thomas et al. [5] implemented a cosimulation environment using Unix socket and Verilog hardware simulator and PLI in a manner similar to Becker et al. They hid system details such as processor architecture and bus interface by abstraction and simplified interactions among system components as transactions among application program and ASIC. They treated application programs as processes running on operating system and converted socket operations into events which Verilog simulator can recognize using PLI without modeling the processor and the bus interface. Using these schemes, they could accelerate cosimulation, simulate even electronic systems with an operating system, and expand to general-purpose cosimulation environment at a cost of simulation accuracy.

Wenban et al. [6] developed a system that compiles communication protocol described in Promela, a concurrent programming language similar to Occam, into C++ software, hardware, and interface. In the Promela description, several processes can execute concurrently and communicate or synchronize using message channels or global variables. Whether each process is hardware, software, or interface is already fixed at the time of description. Before compilation, Promela file is converted into a Promela executable file (Pex) to confirm the specification. In Pex, a process scheduler, whose role is almost the same as the cosimulation scheduler, controls the execution of each process. Process scheduler has a circular queue to handle active processes and schedules the processes by invoking the body function of the process in the head position of the queue. It also checks interrupts occurring between process invokes. It is only for the verification of the specification, not for the simulation of the synthesized system.

Olukotun et al. [7] partitioned systems to maximize the degree of simulation acceleration, using cosynthesis which consists of performance estimation, logic synthesis, and scheduling. After partitioning, software components and hardware components run concurrently on a compiled-code based software simulator and an FPGA prototype, respectively, which are connected through a CPU cache bus. Since they have excessive communication overhead due to the interface, real degree of simulation acceleration is much smaller than expected.

Buck et al. [9, 21] developed Ptolemy design environment, a unified framework that allows the hardware and software components to be integrated from the specification through the synthesis, simulation, and evaluation phases. It uses an object-oriented software technology to model each subsystem and has mechanisms to integrate heterogeneous subsystems into a whole. Ptolemy accomplishes multi-paradigm simulation by supporting a number of different design styles encapsulated in objects called domains. A domain realizes a computational model appropriate for subsystems of a particular type. For the simulation of a hardware component, Thor domain is defined. Thor, a functional simulator for digital hardware ranging from gate to programmable DSP chips, is used for the Thor domain. For a model of processor, Thor simply establishes a socket connection with Sim56000, a stand-alone pin-level behavioral simulator for Motorola's DSP56000 processor.

Ismail and Jerraya [8] developed Cosmos, an integrated codesign system which generates system simulation model called 'virtual prototype' in the course of codesign. Virtual prototype is a module-oriented system description, which covers communication components as well as hardware and software components. Communication components are synthesized using channel unit libraries through channel binding and mapping steps during codesign.

Becker's cosimulation technique combines synchronized handshaking and instruction-level processor model according to Rowson's classification of cosimulation techniques. Thomas's method is synchronized handshake technique. For the cosimulation using Poseidon or Ptolemy, since pin-level or cycle accurate behavioral models of processors are involved, much more accurate simulation is possible at a cost of simulation time.

3. Cosimulation Environment

As shown in figure 2, the environment has three major elements for the execution of cosimulation: a software process running C programs, a simulation process executing the hardware

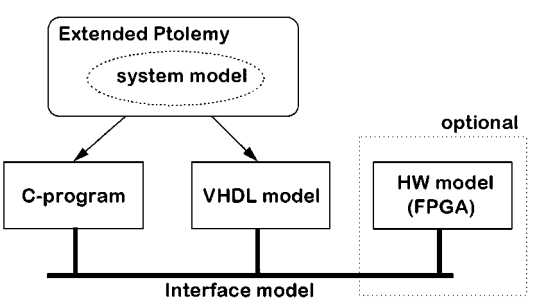


Figure 2. Cosimulation environment.

model in VHDL, and an interface model. An optional custom board is for simulation acceleration by emulating the whole or a part of the hardware components. The interface model is based on inter-process communication (IPC) routines connecting the two processes through Unix socket [10] on a single Sparc CPU. We extended Ptolemy [9, 21], a framework for simulation and prototyping of heterogeneous systems which was developed at University of California, Berkeley, to provide user interface and internal representation for system specification and verification.

3.1. Overall Cosimulation Flow

The overall cosimulation using our environment is done according to following cosimulation flow:

(1) *System description*: System designer describes the specification of a heterogeneous system as a VHDL model for hardware component and a C-program for software components. In describing system specification, it is assumed that system designer already had partitioned the system specification between hardware component and software component manually.
(2) *Abstract-level cosimulation*: System designer inserts calls for IPC routines at appropriate places of C-program and VHDL model. Currently, this is done manually. Those IPC routine calls enable the cosimulation of heterogeneous system at abstract-level (or specification-level) and play the role of interface. At this level, interface has no concrete form of implementation (in hardware or software form), but provides only communication channel between hardware component and software component during cosimulation. Figure 3 represents the abstract-level cosimulation. As shown in the figure, the simulation model of interface mainly consists of IPC routine. All IPC routine calls in a hardware component are grouped as an IPC handler process of VHDL model (as in figure 4) and all IPC routine calls in a software component are distributed in C-program.
(3) *Detailed-level cosimulation*: Figure 4 represents the detailed-level cosimulation. Once system designer selects one target architecture and an interface protocol, he can implement

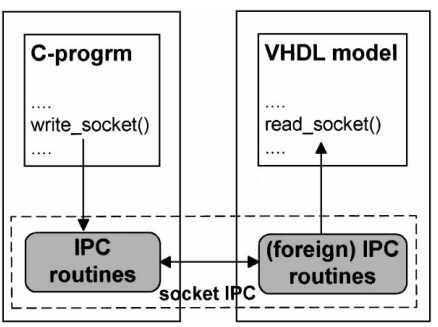


Figure 3. Cosimulation at the abstract level.

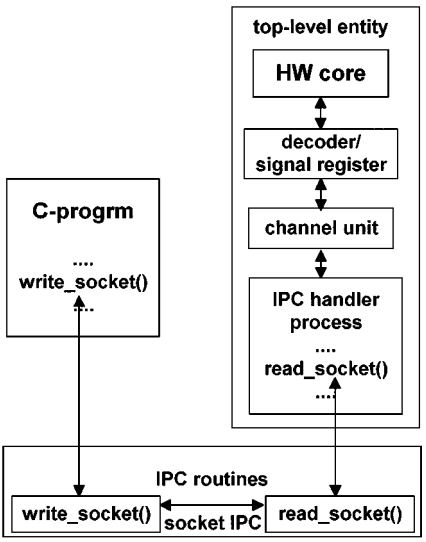


Figure 4. Interface generation or selection from library.

the interface accordingly. The implementation of interface consists of automatic generation of decoder/signal register and manual insertion of channel unit model from device library. IPC handler process remains unchanged. The implementation of interface will be described in detail later.

3.2. Software Process

Software process is a process executing a C-program that is the software component. Since we use "synchronized handshake" simulation technique [1], there is no need of processor models. The communication between hardware and software is done through a synchronized handshake implemented using Unix socket IPC. Using this technique, the software can run at the workstation speed and therefore the hardware simulator performance and IPC overhead would dominate the overall speed. Figure 5 illustrates how the synchronized

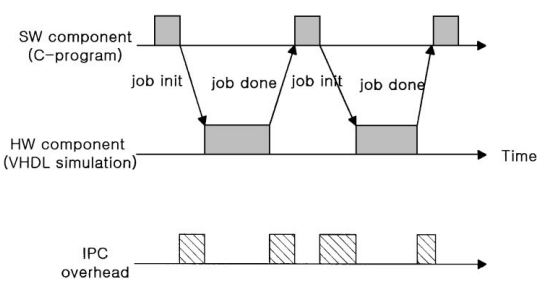


Figure 5. Illustration of synchronized handshaking.

handshake works in the cosimulation. After C-program process sends relevant data and start signal ('job init') to VHDL simulation process, it stops its job and awaits finish signal ('job done') from VHDL simulation process. If it receives the signal, it continues its job.

3.3. Hardware Simulation Process

Hardware simulation process is a process running a VHDL simulator, which exercises a hardware model in VHDL. Our simulator, IVSIM (SNU ISRC VHDL SIMulator), is a VHDL simulator based on an event-driven compiled code simulation algorithm with a concept called 'modified gateways', to improve simulation performance substantially [11]. The simulator is implemented in about 10,000 lines of C++ language (excluding VHDL parser and procedural interface routines of IVDT, SNU ISRC VHDL Development Toolkit [12], which are invoked while data structures are built) and generates C routines for every concurrent statements in VHDL. The generated C routines are linked with C code for the simulation kernel and then executed for a given test vector.

Another important feature of IVSIM is that it can recognize and support 'foreign' attribute defined in VHDL-93 [13]. The attribute enables a system to be described by not only VHDL but also non-VHDL procedures such as IPC routines in C-language. If we declare IPC routines as foreign procedures using the attribute and invoke them at appropriate places in architecture bodies of a VHDL description, they are linked with C codes generated for the hardware description and the simulation kernel mentioned above. The resultant executable code simulates the whole hardware communicating with the software.

3.4. Interface

The interface model for simulation is based on Unix IPC, regardless of the abstraction levels of cosimulation. The IPC routines are implemented using Unix socket. The routines include socket initialization procedure (*init_socket*), socket closing procedure (*close_socket*), socket read procedure (*read_socket*), and socket write procedure (*write_socket*).

4. Interface Transparency

Our cosimulation environment provides users with transparent communication interface between software core and hardware core regardless of target architectures and communication protocols [16, 17]. Hardware core represents the VHDL model that is given at system specification phase. It has no IPC call inside. It communicates with interface modules only through its I/O ports.

This is important especially 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 having to concern about the details of the interface or communication. To provide the interface transparency in a single processor cosimulation environment, we address the following points:

(i) *Process modularity*: We regard hardware and software components as separate processes (VHDL simulation process and C-program process, respectively) running on an identical processor and communicating with each other only through IPC channels during the simulation.
(ii) *Automatic interface generation*: Interface between the two components are generated by invoking automatic interface generator according to the chosen communication protocol and the target architecture as shown in figures 3 and 4. The detailed procedure of interface synthesis will be described in Section 5.
(iii) *Interface simulation model call/instantiation*: To simulate hardware-software interface communication correctly, interface simulation models should be inserted at appropriate places in the C-program and VHDL model. For the hardware part (see figure 4), a top-level VHDL entity is newly defined to accommodate synchronized handshake using IPC routines. In the top-level entity, IPC routines are declared as foreign procedures and then calls to them are placed within a process statement that cares for actual synchronized handshake. The hardware core to be simulated and the relevant interface models are also instantiated as component instances within the top-level entity.

Figure 4 shows the interface elements combined to provide the overall simulation model of the interface, which enables detailed level cosimulation, according to the target architecture and communication protocol. If the user gives only the hardware and software cores, the interface elements are automatically created by interface generators or selected from libraries.

The role of each interface element is as follows:

(i) The 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 interfaces between IPC routines and the channel unit. It is already created manually during abstract-level simulation phase and remain untouched.
(ii) The channel unit represents the abstract simulation model of a physical channel devices (e.g., DMA controller). It is selected from an interface library.
(iii) The decoder/signal register is inserted between the hardware core and the channel unit to overcome the difference in the width of data transfer, the limitation in the number of pins of hardware prototype devices such as FPGAs, and the difference in protocols used. An interface generator automatically creates it.
(iv) The top-level entity acts as a top-level container that gathers all other interface elements. The interface elements are interconnected with an instance of the hardware core using signals declared in the entity.

Among those elements, the hardware core and the decoder/signal register will be mapped into real hardware prototype using FPGAs. When standard bus architecture is used, a standard channel device such as SBus DMA controller [14] can be used as a physical device for the channel unit. If a user-defined bus or channel is used, the channel unit could also be a part of the hardware prototype.

5. Automatic Interface Generation

To generate interface models for both cosimulation and cosynthesis, we developed an interface generation technique. Interface generation starts from control/data-flow graphs (CDFGs) of manually partitioned system specification. The graph is the internal representation of the systems to be designed in our environment and consists of hardware CDFG and software CDFG. Hardware CDFG is obtained by parsing the VHDL model of hardware component. Software CDFG is obtained by parsing the C-program of software component.

We generate hardware interface and software interface from these graphs following the interface generation procedure shown in figure 6. Figure 7 shows partitioned CDFG. After partitioning the system CDFG (figure 7(a)) into software CDFG (figure 7(b)) and hardware CDFGs (figure 7(c)), special nodes ('send' and 'receive' node pairs) for the interface are added to the partitioned CDFG at the partitioning boundary edges. Boundary edges are

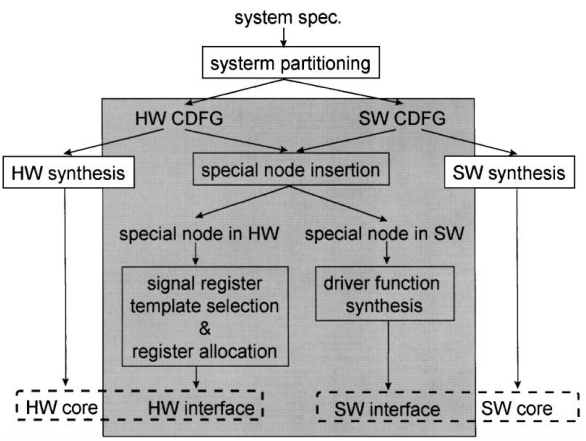


Figure 6. Interface generation flow (shaded area) in heterogeneous systems prototyping.

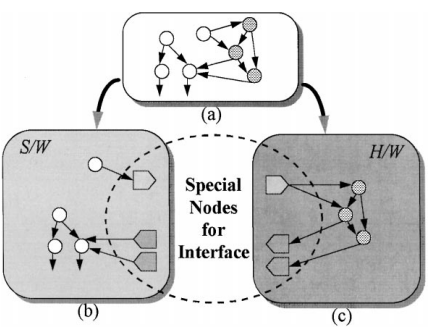


Figure 7. System CDFG and partitioned CDFG.

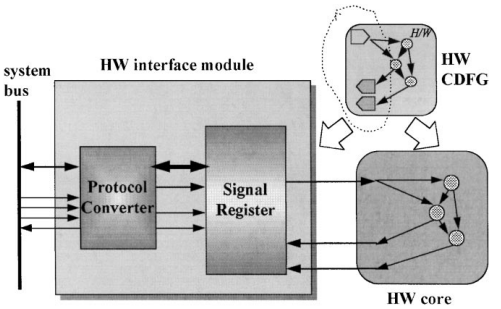


Figure 8. Hardware interface module.

the edges whose one node is to be in hardware CDFG and the other to be in software CDFG. Each special node has its own counter node (for 'send' node, 'receive' node is its counter node, and vise versa).

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 the communication protocol. The software interface module consists of device driver routine calls, I/O function calls, and/or load/store commands to read and/or write data from and/or to system bus. The hardware interface module consists of signal registers and a protocol converter as shown in figure 8. The signal registers store data to be transferred between hardware components and software components. The protocol converter connects the hardware core to the system bus of the target architecture by interfacing between the system bus and the signal registers.

5.1. Hardware Interface Generation

The protocol converter is generated using the algorithm in [23]. Signal registers are generated in two phases—register template selection and signal register allocation. In the register template selection phase, an appropriate template is automatically selected from a template library. Each template differs in the connection of multiplexers, decoders, buffers, and registers. Templates are prepared according to the I/O port characteristics of hardware cores. Currently, four templates are ready for automatic selection [24].

In the signal register allocation phase, inputs and outputs of the selected signal register template are allocated to the ports of hardware core. Using the information of the special nodes such as port name, I/O direction, and bit width, register outputs and multiplexer inputs are bound to the input ports and output ports of the hardware core, respectively. Whenever the bit width of a port to be bound is wider than the bit width of the data transfer through the system bus, data transfer through the bus must be done multiple times. The signal register allocation algorithm is presented in figure 9. P, Q, W, and w_x represent the set of signal registers not yet allocated, the set of allocated registers to each port, the bit width of the system bus, and the bit width of register x, respectively. The algorithm

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

Figure 9. Signal register allocation algorithm.

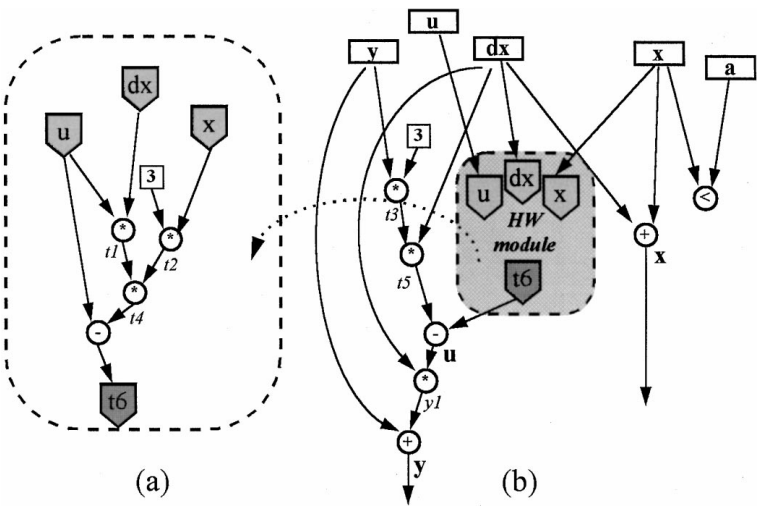


Figure 10. (a) Hardware CDFG, (b) CDFG for the solution of differential equation: $y'' + 3xy' + 3y = 0$.

returns the set of allocated registers for each port whose bit width is ‘width’ (by invoking ‘allocate(width)’).

Example 1. A partitioned CDFG is given as in figure 10. We assume that all edges in the CDFG represent data of 32-bit integer type whereas the bit width of a system bus is 16. The signal register allocation algorithm will allocate signal registers for the CDFG as shown in figure 11. For the send node t6, since the bus in the target architecture is 16 bits wide, a multiplexer with two input m0 and m1 is used.

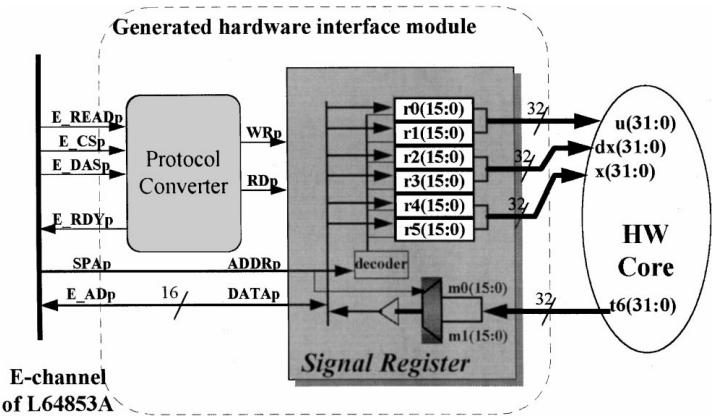


Figure 11. Hardware interface generated for the CDFG in figure 9.

5.2. Software Interface Generation

Using the information of each special node in the software CDFG, driver functions for the corresponding node are synthesized. The information includes function name, function type, argument type, and the result of mapping between the ports of the hardware core and the signal registers. The functions take charge of sending/receiving data to/from the hardware core.

Example 2. For the CDFG in Example 1 (shown in figure 10), the driver function generated for the receive node t6 of the software CDFG is shown in figure 12.

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 the hardware CDFG is mapped to m0 and m1 inputs of the multiplexer in the hardware interface module as shown in figure 11.

The driver functions and the software module synthesized from the software CDFG [25, 26] can be compiled and linked together to make codes executable on the target processor.

```
int receive_t6 ( )
{
   union_type    union_temp;

   ioctl(ddps, RDE_M0, union_data.ch);      /* read from m0 */
   union_temp.ush[0] = union_data.ush[0];  /* store upper half-word */
   ioctl(ddps, RDE_M1, union_data.ch);      /* read from m1 */
   union_temp.ush[1] = union_data.ush[0];  /* store lower half-word */
   return union_temp.in[0];                /* return integer type */
}
```

Figure 12. A driver function generated for the receive node t6 of software CDFG in figure 9.

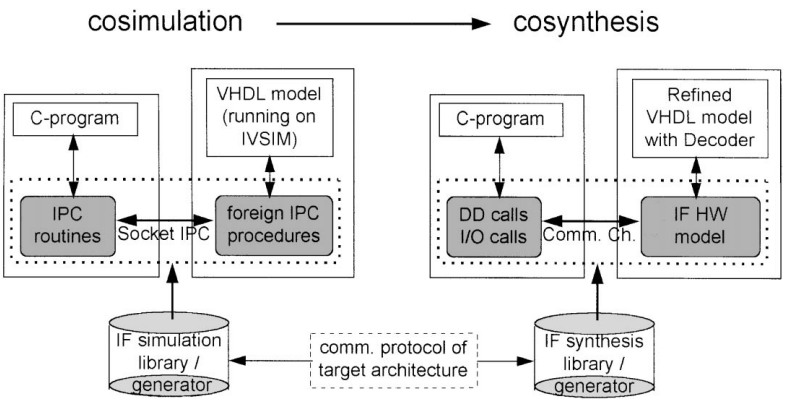


Figure 13. Transition to cosynthesis from cosimulation.

6. Transition to Cosynthesis

After cosimulation, the system components must be synthesized as the physical components on the selected target architecture. For the synthesis on the software side, invokes to the interface simulation models are replaced with the corresponding device driver calls or I/O function calls depending on the target architecture. For the synthesis on the hardware side, the top-level entity with foreign interface procedure declarations is stripped off and the generated and/or selected interface hardware models are inserted. Since this modification of each component specification for the cosynthesis is very simple, it is possible to provide a smooth, fast, and automated transition from cosimulation to cosynthesis of system prototypes. Figure 13 depicts the transition from cosimulation to cosynthesis.

7. Cosimulation Acceleration

Our environment provides a facility to accelerate cosimulation. In cosimulation using synchronized handshake technique, hardware simulation time dominates overall cosimulation time [1]. As hardware subcomponents are added and/or refined incrementally, the whole cosimulation time gets longer. This problem can be alleviated by simulation acceleration through incremental prototyping [15]. At any time during the cosimulation of a heterogeneous system, any hardware subcomponent whose function is already verified through simulation is synthesized and prototyped with FPGAs, thus become a part of the hardware prototype afterward. Newly added or refined hardware subcomponents (incremental part) are described in VHDL and simulated. Because the part already prototyped with FPGAs remains as a hardware prototype during the rest of the cosimulation of the system and we need to simulate only the incremental part, we can reduce the time spent in VHDL simulation considerably and consequently overall cosimulation time. This process which consists of incremental part definition/simulation and incremental prototype synthesis/addition will continue until the whole function of the hardware component is fully verified. When the

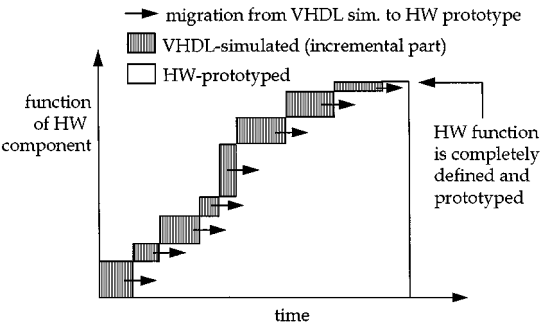


Figure 14. Incremental prototyping process.

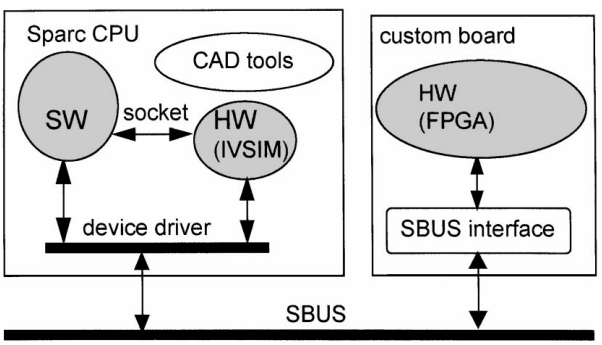


Figure 15. Execution environment of accelerated cosimulation.

cosimulation of the whole system is finished, a full-scale prototype of the hardware part is already obtained. Figure 14 depicts the concept of incremental prototyping process.

Figure 15 shows the execution environment of the accelerated cosimulation. It consists of a general purpose CPU (Sparc processor in a Sparc Classic workstation) and a custom board. The CPU is in charge of running VHDL simulation process for incremental hardware subcomponents, C-program process for software components, and CAD tools for the synthesis of hardware prototypes. Currently, the custom board consists of an FPGA (Xilinx 4010 [18]) and a bus interface. The FPGA is used to implement the hardware prototype. The communication between the CPU and the custom board is done through SBus [19]. To interface between SBus and the hardware prototype, we used a SBus DMA Controller chip (LSI Logic L64853A [14]) and some control logic. They are provided as a part of the SBus-based prototype development board (Dawn VME DPS-1 [20]) which we use for preliminary experiments. 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 the VHDL simulation processes communicate with each other through socket IPC as mentioned above.

Current implementation of cosimulation acceleration facility is based on the following assumptions:

(i) The hardware is implemented as a synchronous circuit.
(ii) The clock signal is applied to the VHDL simulator as an input vector and then fed to the hardware prototype by the simulator via SBus.
(iii) The hardware prototype is fast enough that before the end of the current VHDL simulation cycle which consists of event handling and updating values, etc., the computation by the hardware prototype for that cycle is finished.

8. Integrated User Interface and Internal Representation

To provide an integrated user interface and internal representation, we are under extending Ptolemy. Ptolemy [9, 21] is being developed at University of California, Berkeley as a block-diagram-oriented environment for simulation and prototyping of heterogeneous systems. Instead of trying to capture all possible models of computations into one all-encompassing model, the Ptolemy kernel implements an object-oriented open architecture that enables any extensible model to be defined and added seamlessly. Thus, heterogeneous systems can be specified at different levels of abstraction and semantics for the various subcomponents.

For hardware-software codesign, we are developing Hetero Domain where heterogeneous models may coexist at the same level of hierarchical representation. Figure 16 shows a 'conceptual' design flow under the 'future' extended Ptolemy environment. Initially, the user or system designer represents the abstract system model (or 'universe' in the Ptolemy terminology) which consists of hierarchical blocks ("galaxies") or atomic blocks ('stars') with only data I/O ports. The internal representation of each atomic block may be either C or VHDL model, which does not imply implementation at this level of abstraction.

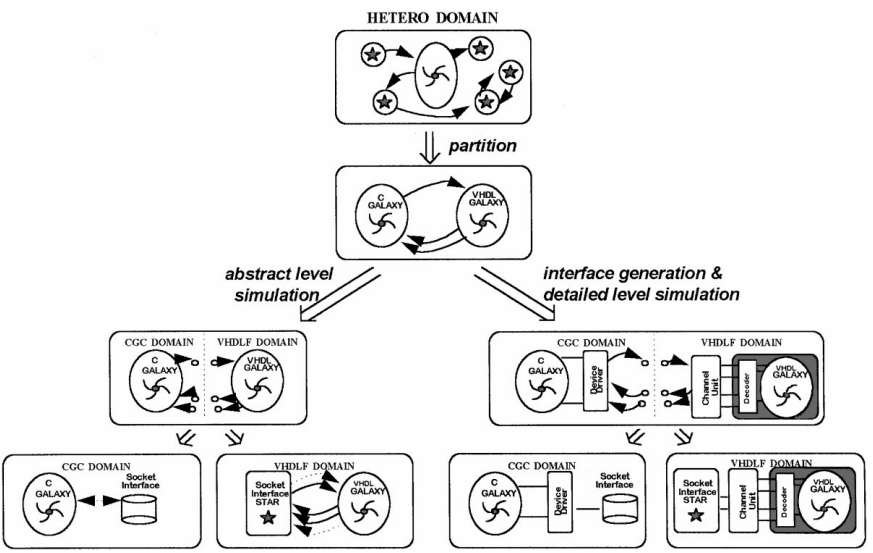


Figure 16. Conceptual design flow under extended Ptolemy.

According to the user's 'manual' partitioning, it is then partitioned into CGC Domain (C Code Generation Domain) and VHDLF Domain (Functional VHDL Code Generation Domain). The partitioned graph is as shown in the second graph of figure 16 which consists of C galaxy and VHDL galaxy. The next action that the user has to do is just to "run" (or cosimulate) the system after selecting some architecture options to be explained below.

The Ptolemy partitions the universe into two separate model-specific universes after inserting the appropriate communication blocks (send and receive stars) at the boundary of these universes. The communication blocks are selected according to the user-specified architecture options such as the level of abstraction or the communication protocol. First, assume that we choose to cosimulate at the abstract level. In our example, two universes to generate C code (for software components) and VHDL code (for hardware components) are created respectively and the communication stars are added to use the UNIX socket for cosimulation. Note that the send and receive stars do not imply that the communication protocol is a message-passing type. Instead, they do imply where communication between two different models arises. In Ptolemy, the kernel object to generate the code is called "target". We can use not only CGC Target which generates C code as a default target, but also VHDLF Target which we have developed to generate a VHDL code for abstract-level cosimulation. The VHDLF Target replaces the communication stars with a Socket Interface Star and adds protocol-related signal so that the modified model can be cosimulated. On the other hand, if we select the option for the detailed level cosimulation, not only communication stars but also appropriate communication models for emulating communication channel are inserted from the interface library.

9. Experimental Results

As experimental examples, several systems were cosimulated and prototyped using our environment and approach.

Experiment 1

We designed, cosimulated, and cosynthesized 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 (called LZ77 algorithm) [22]. Figure 17 represents the skeleton of the program.

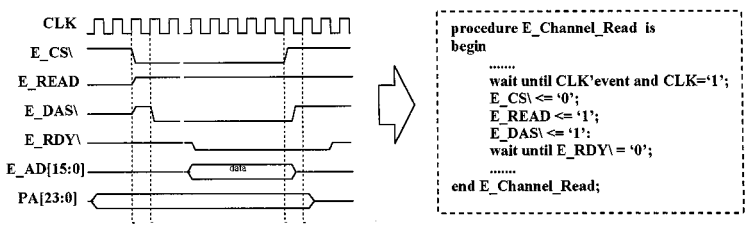
The system was manually partitioned into software and hardware components resulting in a mixture of a hardware component implementing parsing step and a software component implementing the remaining steps—initialization, coding, buffer updating, and file I/O. After inserting IPC routine calls in the components, we performed abstract-level cosimulation. After the target architecture and the communication protocol were determined (Table 1), hardware and software interface elements were generated and selected from the library and added to the hardware component and the software component for detailed level cosimulation. Figure 18 represents the simulation models of the IPC handler and the channel unit (E-channel of SBus DMA controller [14]). After combining all simulation

```
lz77_compression( )
{
   /* fields of code word - maxlength, pointer, last symbol */
   int  maxlen, ptr;
   char  lsym;

   initialize( );                    /* initialization */
   for ( ; ; )  {
      shift_and_feed( );             /* file in & buffer update */
      parse(&maxlen, &ptr);          /* parsing */
      lsym = buf[bhalf + maxlen];
      put_code(ptr, maxlen, lsym);   /* coding & file out */
   }
}
```

Figure 17. The skeleton of the C-program of LZ77 algorithm.

```
entity IPC_handler is
port (maxlen : in bit_vector(3 downto 0);
      pointer : in bit_vector(3 downto 0);
      data : out bit_vector(6 downto 0);
      ..... );
end IPC_handler;

architecture behave_IPC_handler of IPC_handler is
procedure init_socket;     -- foreign procedure declaration
procedure read_socket;     -- foreign procedure declaration
procedure write_socket;    -- foreign procedure declaration
procedure close_socket;    -- foreign procedure declaration
signal  signal_id, input_data, output_data : integer;
begin
init_socket;
process
   .....
   read_socket(signal_id);
   read_socket(input_data);
   ....
   read_socket(signal_id);
   ....
   write_socket(output_data);
   ....
end process;
end behave_IPC_handler;
```

(a)



(b)

Figure 18. Simulation models of (a) IPC handler and (b) channel unit (E-channel read cycle protocol of SBus DMA controller).

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
Hardware device (FPGA)	Xilinx XC4010	Xilinx XC4025 + XC4010
System bus	L64853A SBus DMA controller's E-channel	ISA bus
Computer system	SUN SparcClassic	IBM PC486 compatible

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 $n = 16$ was prototyped with an FPGA [18] using the architecture in [27, 28]. The resultant hardware used 645 CLBs including the interface. With the FPGA clock of 6.25 MHz, the prototyped heterogeneous system shows speedup of 1.7 over the implementation using software only.

Experiment 2

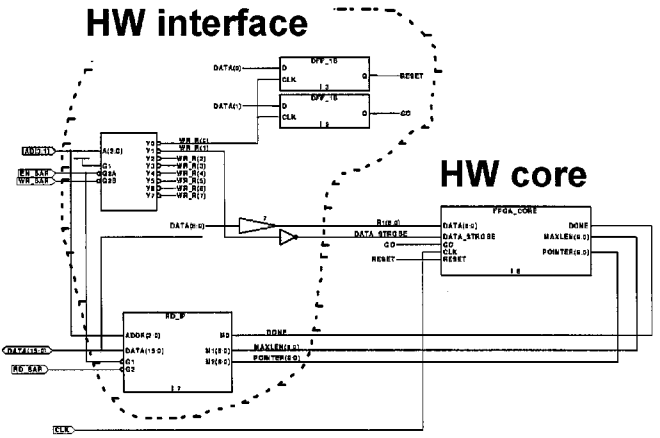
We cosimulated and cosynthesized the same system as Experiment 1 but with different target architecture, communication protocol, and buffer size as shown in Table 1. Signal registers which matched with I/O ports of the hardware core have been generated using the interface module library for the PC system bus (ISA Bus [29]) as shown in figure 19. Figure 19 also shows the generated software driver function. With the FPGA clock of 8.33 MHz, we obtained speedup of 2.5 over the implementation using software only.

Experiment 3

To confirm the feasibility of our prototyping system and methodologies, we did some experiments. The circuit used here is DCT (discrete cosine transform) core for image processing. It consists of four functional modules: 16-bit shift and adder (M1), ROM as look-up table (M2), 16×14-bit parallel multiplier (M3), and 22-bit adder (M4). We conducted two experiments.

Case 1. To confirm the feasibility of incremental design, we partitioned unfinished DCT circuit which consists of M1, M2, and M3 into two parts: one (partition P1) containing M1 and M2, and the other (partition P2) containing M3. Assuming that the function of P2 is designed and verified before P1, P2 is prototyped into hardware and P1 is software-simulated.

Case 2. Just to see the effect of hardware prototype on simulation speed, we arbitrarily partition the system into P1 with M1 and M3, and P2 with M2 and M4. P1 is




```
int reg0=0;
void send_RESET(bit value) {
  reg0 &=  0xFFFE;
  if (value) reg0  |=   0x0001;
  outport(R0, reg0);
}
void send_GO(bit value) {
  reg0 &= 0xFFFD;
  if (value) reg0  |= 0x0002;
  outport(R0, reg0);
}
void send_DATA(int value) {
  outport(R1, value);
}
```

Figure 19. Generated hardware interface and core (upper) and a part of generated software driver function (lower).

hardware-prototyped and P2 is software- simulated. Table 2 presents the simulation result. The experiments show that prototyping some part of system can achieve considerable improvement in simulation speed over all-software simulation, which makes our prototyping methodologies efficient.

Experiment 4

For a more challenging application of our environment, we chose MPEG-2 video decoder [30]. The block diagram is shown in figure 20. We started from a C-program that models the whole function. We selected IDCT block [31] as a hardware module. Other blocks remain as software blocks. With the target architecture shown in Table 1 (Experiment 2),

Table 2. Experimental results: simulation results, hardware size, and VHDL code size (all SW: all modules are simulated, mixed: one partition is simulated, the other is prototyped).

	Case 1		Case 2	
	All SW	Mixed	All SW	Mixed
Sim. time (s)	191	19	196	19
No. of signas	5747	1690	6372	1254
No. of events	239185	29593	251611	30325
No. of gates		3725		5080
No. of flip-flops		79		201
No. of I/O pads		19		18
No. of VHDL lines (SW simulated)	2921	1463	2921	1281

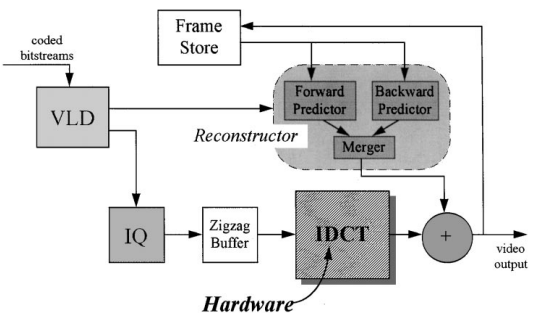


Figure 20. Block diagram of MPEG-2 decoder.

we partitioned the system, generated, inserted appropriate interface simulation models, and cosimulated using our environment. After the cosimulation, prototyping was done successfully. In the prototype, the size of hardware interface module was about 700 gates and 2D-IDCT for each 8×8 pixel block took 305 μs at clock frequency of 8.33 MHz [32].

10. Conclusion

In this paper, we present a hardware-software cosimulation environment for heterogeneous systems. To be an efficient and convenient verification environment for the rapid prototyping of heterogeneous systems consisting of hardware and software components, the environment supports (i) modular cosimulation, (ii) cosimulation acceleration, and (iii) integrated user interface and internal representation. For modular cosimulation, software and hardware components are regarded as separate processes and can communicate with each other only through inter-process communication. Cosimulation can be accelerated with hardware-emulation using FPGAs and our incremental system prototyping strategy. Finally, to provide an integrated user interface and internal representation consistent with various prototyping

tasks, we modified and extended Ptolemy, a cosimulation and cosynthesis environment for heterogeneous systems. The benefits of our cosimulation environment are as follows: target architecture and protocol independence, interface transparency, smooth transition to cosynthesis, fast cosimulation, and convenient cosimulation. As experimental examples, several heterogeneous systems have been cosimulated and prototyped successfully, which shows that our environment can be a useful heterogeneous system specification/verification environment for fast prototyping.

On-going and future works are as follows:

(i) The system has not been completed yet. Complete the implementation of the environment by finishing system partitioning and extending interface model generator and library for various target architecture and protocols.
(ii) Generalize the environment for various target architectures including general-purpose microprocessors or microcontrollers, DSPs, and ASICs. Currently, the system works only for Sparc + SBus + ASIC architecture and PC + ISA bus + ASIC architecture. Synchronization is done by handshaking signals between hardware components and software components. Timed cosimulation must also be implemented so that general interface protocol can be simulated.
(iii) Apply our approach to various system prototyping examples.

References

1. J.A. Rowson, "Hardware/software co-simulation," in *Proceedings of 31st ACM/IEEE Design Automation Conference*, pp. 439–440, June 1994.
2. W. Ye, R. Ernst, Th. Benner, and J. Henkel, "Fast timing analysis for hardware-software co-synthesis," in *Proc. ICCD'93*, October 1993.
3. R.K. Gupta, C. Coelho, and G. De Micheli, "Synthesis and simulation of digital systems containing interacting hardware and software components," in *Proceedings of 29th ACM/IEEE Design Automation Conference*, pp. 129–134, June 1992.
4. D. Becker, R.K. Singh, and S.G. Tell, "An engineering environment for hardware/software co-simulation," in *Proceedings of 29th ACM/IEEE Design Automation Conference*, pp. 129–134, June 1992.
5. D.E. Thomas, J.K. Adams, and H. Schmit, "A model and methodology for hardware-software codesign," *IEEE Design and Test of Computers*, 10(3): 6–15, September 1993.
6. A.S. Wenban, J.W. O'Leary, and G.M. Brown, "Codesign of communication protocols," *IEEE Computer*, 26(2): 46–52, December 1993.
7. K.A. Olukotun, R. Helaihel, J. Levitt, and R. Ramirez, "A software-hardware cosynthesis approach to digital system simulation," *IEEE Micro*, 14(4): 48–58, August 1994.
8. T.B. Ismail and A.A. Jerraya, "Synthesis steps and design models for codesign," *IEEE Computer*, 28(2): 44–52, February 1995.
9. A. Kalavade and E.A. Lee, "A hardware-software codesign methodology for DSP applications," *IEEE Design and Test of Computers*, 10(3): 16–28, September 1993.
10. W.R. Stevens, *UNIX Network Programming*, Prentice-Hall, 1991.
11. Y.S. Lee and P.M. Maurer, "Two new techniques for compiled multi-delay logic simulation," in *Proceedings of 29th ACM/IEEE Design Automation Conference*, pp. 420–423, June 1992.
12. D.H. Ko and K. Choi, "IVDT: A VHDL developer's toolkit," *KITE Journal of Electronics Engineering*, 5(2): 56–63, December 1994.
13. The Institute of Electrical and Electronics Engineers, Inc., New York, *IEEE Standard VHDL Language Reference Manual, ANSI/IEEE Std 1076-1993*, 1994.
14. *L64853A SBus DMA Controller Technical Manual*, LSI Logic, 1991.

15. Y. Kim, Y. Shin, K. Kim, J. Won, and K. Choi, "Efficient prototyping system based on incremental design and module-by- module verification," in *Proceedings of IEEE ISCAS 95*, pp. 924–927, May 1995.
16. K. Kim, Y. Kim, Y. Shin, and K. Choi, "An integrated hardware-software cosimulation environment with automated interface generation," in *Proc. 7th IEEE Int'l Workshop on Rapid System Prototyping*, pp. 66–71, June 1996.
17. 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," in *Proc. of Asia and South Pacific Design Automation Conference*, pp. 101–106, August 1995.
18. *The Programmable Logic Data Book*, Xilinx, 1993.
19. *Standard for a Chip and Module Interconnect Bus: SBus* (*P1496/Draft 2.3*), IEEE Standard Department, 1993.
20. *User Guide for DAWN VME Products DPS-1: Development Platform SBus Version 1.0 Revision B*, DAWN VME Products, April 1991.
21. J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *International Journal of Computer Simulation*, 4: 155–182, April 1994.
22. J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, IT-23(3): 337–343, 1977.
23. S. Narayan and D. Gajski, "Interfacing incompatible protocols using interface process generation," in *Proc. of 32nd ACM/IEEE Design Automation Conference*, June 1995.
24. K. Kim, "Generation of interface module in hardware-software co-design," M.S. Thesis, Dept. of Electronic Eng., Seoul Nat'l Univ., December 1995 (in Korean).
25. Y. Shin and K. Choi, "Thread-based software synthesis for embedded system design," in *Proc. of European Design and Test Conference*, March 1996, to be published.
26. Y. Shin and K. Choi, "Software synthesis through task decomposition by dependency analysis," in *Proc. of Int'l Conf. on Computer Aided Design*, pp. 98–102, November 1996.
27. Y. Kim, K. Kim, and K. Choi, "Efficient VLSI architecture for lossless data compression," *IEE Electronics Letters*, 31(13): 1053–1054, June 1995.
28. Y. Kim, K. Kim, K. Choi, and I. Park, "A scalable VLSI architecture for Lempel-Ziv-based data compression," in *Proc. 5th Int'l Conf. on VLSI and CAD*, pp. 355–357, October 1997.
29. T. Shanley and D. Anderson, *ISA System Architecture*, 3rd edition, MindShare, Inc., 1995.
30. ISO/IEC JTC1/SC29/WG11, *Coding of Moving Pictures and Associated Audio*, Recommendation H.262, (ISO/IECJTC1/SC29/WG11 N0702rev), March 1994.
31. S. Uramoto et al., "A 100-MHz 2-D discrete cosine transform core processor," *IEEE Journal of Solid-State Circuits*, 27(4): 492–499, April 1992.
32. J. Chang, "Interface synthesis using bank switching," M.S. Thesis, Dept. of Electronic Eng., Seoul Nat'l Univ., December 1997 (in Korean).