A Hardware Software Cosimulation Backplane with Automatic Interface Generation

Wonyong Sung Soonhoi Ha
Department of Computer Engineering
Seoul National University
Seoul, Korea 151-742
e-mail: \{yong.sha\}@iris.snu.ac.kr

Abstract — A hardware software cosimulation environment is developed using the backplane approach. This paper defines the backplane protocol for communication and synchronization between client simulators to seamlessly integrate a new simulator without modification. Automatic interface generation facility is also devised for more effective cosimulation environment. The environment is implemented based on Ptolemy and validated with QAM example run on different configurations.

I. Introduction

Codesign methodology has gained much attention in designing an embedded system which has heterogeneous configurations of programmable modules and application specific hardware modules. Using a programmable module reduces the design complexity and adds the flexibility of the architecture while hardware components are required to meet constraints. The codesign workflow consists of many stages from system specification to system synthesis. The codesign process presented in this paper is shown in figure 1.

A dataflow graph is chosen as an initial specification for a given application and simulated for algorithm verification. Dataflow models are effective for representing most types of DSP applications[12]. The next step is to partition the initial dataflow graph into two kinds of subgraphs, software graphs and hardware graphs. After partitioning, each subgraph is modified in order to add interface nodes at the graph boundary. From partitioned graphs, C and VHDL codes are generated including interface code. The generated C code is compiled into UNIX process and the generated VHDL code from the hardware graph is passed to the VHDL simulator for hardware simulation. Therefore, several UNIX processes are running concurrently and cooperatively: C processes and VHDL simulators. With simulation results, the evaluation module makes a decision whether the current partition satisfies the system requirements. Unless they are satisfied, the codesign process continues to iterate from the partition stage to the evaluation stage. Otherwise, the software executables and hardware modules are synthesized through compiler and behavioral synthesis tool, respectively. Among the whole codesign procedure in figure 1, this paper covers the topics related with cosimulation within the dashed line box.

Cosimulation is a key facet of a codesign process. Through cosimulation, a designer can check the functional correctness ahead of final synthesis step. Also, cosimulation is used to profile the dynamic behavior. As shown in [3], the profiling results help the designer partition the target system more efficiently in the next iteration. Another role is to identify the performance bottleneck. In spite of many benefits of cosimulation, many difficulties lie in the implementation of an efficient cosimulation environment. They come from the need of connecting heterogeneous distributed concurrent components. In this paper we present the solutions to those difficulties.

To incorporate heterogeneous simulators, we propose a backplane environment instead of pairwise direct coupling among simulators. As a standard backplane interface, the
backplane protocol is defined in section 2 with communication, synchronization and implementation rules.

On each iteration of codesign process, a new partition is made, which requires the rebuilding of interface. Since making a interface is tedious and error-prone work, and making a new interface is frequent due to the changed partition, it is necessary to generate the interface automatically. The automatic interface generation is described in section 3 which is followed by the implementation details in section 4. Section 5 shows an example and we make a conclusion in section 6.

II. Cosimulation backplane

A. Cosimulation configurations in previous works

To build a cosimulation environment is to combine heterogeneous simulators. For hardware (HW) simulation, most people use a hardware simulator such as a VHDL or Verilog simulator. For software (SW) simulation, however, there have been three approaches pursued with their own merits and limitations.

First, a software program is executed with a processor simulator connected to a hardware simulator. [4] used a structural level hardware simulator and a processor simulator (DLX simulator) connected and communicated through the cosimulator, which may achieve the most exact timing estimation. For large systems, however, this approach is prohibitive because of its extremely long simulation time. Second, a software program is run as a module of the hardware simulator via its foreign interface[9]. The VHDL foreign interface is an example. In this approach, software processes, which are fully controlled by the VHDL simulator, are not adequate to model concurrent processes with the hardware.

Finally, a software program is executed on the development processor and communicates with the hardware simulator through UNIX IPC. In [5], the HW and SW descriptions are treated as separate UNIX processes which communicate through BSD sockets. Since it is targeting a specific application, it uses a fixed communication model between the HW and SW. A similar approach is described in [6], the tool assists in mapping a specification onto a single processor with multiple ASICs. The software module is executed on the target processor and communicates with a hardware simulator (Verilog) through UNIX IPC mechanism. These works, however, consider only the fixed combination of simulators and fixed topology of connections. There is no standard interface defined so that new communication code needs to be defined to integrate a new simulator. Also, data exchange between the software and the hardware parts is hidden in the simulated program. So, it is not visible to be probed.

For arbitrary connection of various simulators, [8] defines a standard interface through which each simulator can communicate with others. In their approach, they use a C process as a standard interface so that a simulator should send the data to an automatically synthesized C process from which the destination simulator receives it. Using a standard interface between heterogeneous simulators is the main feature of the Ptolemy[1], their cosimulation platform. Even though our research is also based on the Ptolemy environment, we propose another approach; backplane approach. Another limitation of their approach is that it supports only dataflow model of computation, which implies timed cosimulation is not possible.

In our backplane approach, software simulators(processes) are run on the development processor and communicate with the backplane. A hardware simulator also communicates with the backplane. Therefore, the backplane is the master process to manage the interprocess communication between software processes and the hardware simulators. As a standard interface for cosimulation, the backplane defines several rules that the client processes should meet. A new simulator can be seamlessly integrated with the backplane if it satisfies the interface rules.

The backplane approach can be found in [7]. The simulation concept they use is in a standardization process of the CAD Framework Initiative (CFI), where communication for synchronization and data transfer is done by a Procedural Interface which defines a set of callbacks. To ensure an integration of clients to the backplane, it is required for tool manufacturers to provide a set of callback functions in the simulator program. Only based on the callbacks, the interface code between the backplane and the simulator can be made. However, most simulators that we are to use do not support the callbacks, yet. Therefore, we develop a backplane which requires no modification of the simulator program itself.

B. Cosimulation Backplane

Figure 2 shows an architecture of the simulation backplane environment designed and implemented in this paper. The cosimulation environment is composed of three parts: the backplane program, interface link modules (one for each simulator) and the client simulators. To integrate a simulator, one link module needs to be created and placed between the simulator and the backplane. It
acts as an interpreter between the simulator specific communication protocol and the backplane protocol. When a simulator runs with the generated code, the need for interprocess communication between the client simulator and the backplane arises. The backplane protocol is a standard way of communication between them and it consists of three groups of rules. They are communication rules, synchronization rules, and implementation rules.

Communication Rules This group of rules determines how the simulator and the backplane establish, maintain, and terminate the connection. Since the client simulator is invoked by the backplane, the backplane takes the initiative for the establishment of connection by calling a function defined in the link module for the simulator. Currently, we use the Berkeley socket IPC mechanism with only one communication channel. Since the socket mechanism is used, the client process may run on a different machine to make a distributed cosimulation.

The backplane is the scheduling engine of event driven model of computation. By sending a message to a client simulator, the backplane enables the corresponding subgraph. During cosimulation, the exchange of packets is serialized by the backplane to ensure the execution order of communication among multiple simulators. To terminate the connection, a special control packet is used. By receiving the termination packet generated by the backplane, a link module terminates the connection. When the client module terminates the simulation by itself, the simulator should notify the backplane by the sending a termination packet.

Synchronization Rules This group of rules are required to synchronize between hardware and software modules run on their own simulators. Though there is one physical socket connection between the backplane and a client process, there are multiple logical connections in case there is more than one input (or output) connections in the simulated module. Therefore, each packet needs to be identified from which input port it is delivered or to which output port it is delivered. We assign a unique identification number to each input or output port of the partitioned graph as shown in figure 3. When the backplane sends a packet to the client process, this id number is also delivered as a part of the packet header. Since the backplane also inserts a time stamp into a header, a packet from the backplane contains three fields: id, time stamp, and message(data).

The backplane is expected to receive the response from the client process at the same simulation time after packets are delivered to the client process. One idea of doing this is to wait after sending input packets until the client process responds with the output packets. If the client process takes long, the backplane is blocked to result in slow simulation especially in case of distributed simulation. To improve the performance, another idea is to poll the socket periodically. If there is no response, the backplane executes other modules until it increments the global time of the simulation. Before it receives the response message from the client process, the backplane can not increment the global time since the response message may generate a past event (a causality error).

Both approaches fail if the client process does not produce any output at the current execution. This situation occurs when the inside graph conditionally produces outputs. Therefore, we enforce that the client process sends a signaling message, DONE signal packet, even though it does not generate a valid message at the current execution. In case multiple messages are transferred from/to the same port, a GO signal packet or a DONE signal packet is appended on each sequence of packets. By receiving this signal packet, the client simulator and the backplane can detect the end of transmission. These control packets are associated with negative id numbers.

Implementation Rules This group of rules define how to implement the link module which plays a role of a driver for its simulator. Like a conventional device driver in operating system, its internal implementation is dependent on the supporting simulator while the backplane calls the same function. There are five call-back functions defined in the current implementation as listed in table I.

In the setup stage, the backplane calls "modifyGalaxy" and "setPTInterface" function once for each link module. In the "modifyGalaxy" function, the link module modifies the partitioned graph by inserting the communication nodes. The "setPTInterface" function makes a socket code with TCP port number given as an input parameter. After the C and VHDL codes are generated, the client simulators are invoked by the backplane. At the beginning of cosimulation the client simulator calls "hostConnect" function to establish a socket connection with backplane. In fact, a socket connection is established only of the simulation.

![Fig. 3. Message format for exchanging data with synchronization](image-url)

Table I Call-back Functions defined in the implementation rules

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>modifyGalaxy</td>
<td>add communication nodes</td>
</tr>
<tr>
<td>setPTInterface</td>
<td>create socket connection code</td>
</tr>
<tr>
<td>hostConnect</td>
<td>make a socket connection</td>
</tr>
<tr>
<td>ReadFromDevice</td>
<td>send a message</td>
</tr>
<tr>
<td>WriteToDevice</td>
<td>receive a message</td>
</tr>
</tbody>
</table>
between the link module and the simulator.

Since the link module is a wrapper module for a client simulator, modification of the simulator source is not required. For example, integration of the VHDL simulator with the backplane is done only through a communication between a link module and VHDL entities which calls the foreign C procedures. Actually the socket connection is established between the link module and the C foreign procedures. The VHDL entities defined in communication nodes which is inserted by the “modifyGalaxy” function. Therefore, any VHDL simulator supporting a foreign interface can be added without any modification of program source. No need for modification of program source is an important requirement for a backplane to support a commercial simulator like Synopsys VSS simulator. Sending and receiving messages from/to the backplane is done by the “ReadFromDevice” and “WriteToDevice” functions. The sending and receiving from/to the simulator is also dependent on the link module implementation.

In figure 4, the implemented rules of the backplane protocol for the VHDL simulator are depicted. The case shown in figure 4 is when the “writeToDevice” is called. When the backplane calls the “writeToDevice” function, a packet is transmitted via a socket connection to the C module which is combined with VHDL simulator via foreign interface. The VHDL simulator periodically polls the “readSocket” function in the foreign interface module to read packets from the socket and store them in the internal buffer. When the receive node(entity) in the VHDL graph is scheduled, it fetches the packet by calling the foreign procedure “readBuffer”. How the partitioned hardware graph is modified and what is in the communication nodes will be explained in the next section.

III. Automatic interface generation for cosimulation

As shown in figure 1, the proposed codesign environment in this paper generates code from the partitioned graph. When we generate codes from a partitioned graph, we modify the graph to insert communication codes. At the input connection, we replace a connection arc with a receive node which is again connected to the master entity. Similarly, we replace an output connection with a send node. Then, the partitioned graph becomes a graph without any disconnected arc and this subgraph is passed to the code generation facility. This modification is done automatically without user's intervention.

Interface generation is very error-prone even when the graph topology is fixed. In [4] and [9], interface descriptions are manually made and provided for cosimulation. [11] made and used a fixed interface using Verilog PPI. A study on automatic interface synthesis for cosimulation is found in [10]. In this work, the VCI tool are developed in order to generate VHDL entities from the interface description called VCI specification. However, even in this approach, designer should make the I/O descriptions whenever the graph topology is modified. The most automated approach of interface generation is studied in [8]. In their work, however, the application specification semantics are limited to a specific class of dataflow known as SDF[12]. Within the limitation of SDF semantics, they are able to statically schedule the graph and guarantee a deadlock free execution. We aim to develop a technique of automatic interface generation from the partitioned dataflow graph which has more generality.

As depicted in figure 1 and 4, automatic interface generation is simply realized by insertion of communication nodes and generation of C or VHDL codes from them. As an example, for VHDL simulator, we append three types of interface nodes: master node, send node, and receive node. The architecture of VHDL simulation interface is as depicted in figure 4. In the initialization stage, the master node establishes the socket connection and setups the input and output queues: one for each partitioned link. When communication occurs, the master nodes serialize the firings of communication nodes to avoid deadlock. See [13] for detailed explanation.

IV. Implementation

This section describes implementation issues encountered in building our backplane approach. Our cosimulation environment is implemented based on Ptolemy[1]. Ptolemy is a framework developed at U.C. Berkeley for heterogeneous system specification, simulation and synthesis. Ptolemy is designed and implemented under object-oriented paradigm. In Ptolemy, design models are expressed with block diagrams. Each atomic block is called a star and an entire application is called a universe. To make a block diagram hierarchical, a subgraph of blocks can be encapsulated in a macro block, called galaxy. A block diagram is assigned a specific syntax and semantics tailored to the computational model of interest called a domain. Heterogeneity in Ptolemy is realized with a wormhole. A wormhole is a special macro block that is treated as an atomic star from the outside domain as a wormhole, but contains a galaxy of different domain
inside. When a wormhole is fired, a token is delivered from the input arc to the inside domain via a special wormhole interface mechanism, called “event-horizon”. The outside and the inside domains are synchronized at the time of message passing between two domains. Heterogeneous simulations using wormhole mechanism have been successfully exploited for various applications.

To implement the backplane approach, we developed the backplane (BP) domain and two target objects of the code generation wormholes (link module described in section 2), one for the CGC domain and the other for the VHDL domain. The five call-back functions described in implementation rules are implemented with virtual functions.

At first, we experimented with IVSIM VHDL simulator. Later, we simulated the same example with Synopsys VSS VHDL simulator. For the programmer who already knows the Synopsys CLI (C Level Interface) mechanism, it requires a couple of days to make new interface module. Table II shows the difference of implementation results.

<table>
<thead>
<tr>
<th></th>
<th>C bytes</th>
<th>VHDL lines</th>
<th>VHDL/R lines</th>
<th>VHDL/S lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSS</td>
<td>17688</td>
<td>341</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>IVSIM</td>
<td>5628</td>
<td>25</td>
<td>27</td>
<td>17</td>
</tr>
</tbody>
</table>

The top view windows also has two kinds of simulation support nodes; a test pattern generation node (a clock node), and a visualization node (a Xgraph node). The input messages are issued to the software block from the clock node. The value of this message is ignored, but the time stamp indicates when to invoke C process. Instead of the clock node, we can use the various source nodes in Ptolemy to generate test patterns for developing system. The simulation output is displayed with the visualization node. We can probe any arc of the program graph for the token transfer by forking the tokens to a monitor node. Since every inter-client token exchange goes through the backplane, we can display the contents of each token exchange by the strong visualization facility of Ptolemy.

In this experiment, we manually partitioned the graph into software and hardware only to validate the correctness of the proposed cosimulation concept. The C module graph generates a complex number which is formed into a raised cosine waveform within a bit time by the VHDL module graph. We have run the QAM16 example on two different configurations of cosimulation as shown in figure 6. Figure 6(a) shows the first configuration with IVSIM VHDL simulator and (b) shows the configuration with VSS simulator. Both the backplane and client simulators run on the same machine in (a), while we can see the distributed cosimulation on the network of workstations in

V. Example of Cosimulation

This section shows an example and two experiments with different configurations. A 16-QAM modulation is chosen to demonstrate the proposed cosimulation environment. The QAM16 modulator produces a 16-point quadrature amplitude modulated signal. To make a modulated signal wave, a raised cosine pulse is used, where the excess bandwidth is 100% and the carrier frequency is twice the symbol rate. Figure 5 is the dumped screen of the design specification and the simulation results. A top view window, showed in upper left part of figure 5, has two wormholes. One has a C subgraph and the other has a VHDL subgraph that is displayed in QAM16_C and QAM16_V windows displayed in the bottom of figure 5.

(a) IVSIM runs on the same machine
(b) VSS runs on the different machine

Fig. 6. Two different configuration of cosimulation experiments

TABLE II

Interface code overhead in VSS and IVSIM simulators
(b) The VSS simulator is run on the other workstation. Though (b) performs cosimulation on two workstations, we get no speed-up because of the lock-step synchronization we are using. We are developing a more optimized synchronization scheme, which will achieve the cosimulation speed-up on the distributed cosimulation.

Table III shows the cosimulation execution on the configuration of figure 6(1). The pure cosimulation time is much smaller than total cosimulation time. Most of the cosimulation time is consumed in the network transfer time. Currently, we exchange many control packets between the backbone and the client VHDL simulator for conservative timed cosimulation. Since we do not have accurate timing estimates for functional blocks, we do not report timed cosimulation results in this paper. It will be our future main work to reduce the number of control packets to enhance the simulation speed.

VI. Conclusion

In this paper, we have described the backbone approach of cosimulation as well as the automatic interface generation. Using the backbone protocols, defined and implemented in this paper, we can achieve a cosimulation environment which integrates a new simulator seamlessly. Only a link module is needed to be created for integration and no modification of simulator source code is required. We assumed that the input specification is given with a coarse-grain dataflow graph which is partitioned into software and hardware subgraphs. By adding communication nodes to the partitioned subgraphs, we achieve automatic interface generation. The contents of the communication nodes are dependent on the simulator. The proposed approaches have been implemented based on Ptolemy, which provides our system with strong facilities such as graphical user interface, visualization, and C/VHDL code generation from the dataflow graph. To validate our cosimulation environment, we showed a QAM example which runs on two different configurations of cosimulation.

We are developing a more optimized synchronization scheme to accelerate the cosimulation speed. We will also pursue timed cosimulation with conservative and optimistic synchronization. To be more exact timed cosimulation, we will also integrate a processor simulator into our cosimulation environment.

Acknowledgements

This work was supported by Ministry of Education through Interuniversity Semiconductor Research Center (ISRC-96-E-2103) in Seoul National University.

References


<table>
<thead>
<tr>
<th>Total</th>
<th>VHDL</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>114.011</td>
<td>2.79</td>
<td>1.056</td>
</tr>
</tbody>
</table>

TABLE III
Execution time of QAM cosimulation