Experiences in Real-Time Hardware-Software Co-Simulation

W.M. Loucks,
B.J. Doray
and
D.G. Agnew
Bell-Northern Research Ltd.
Ottawa, Ont.

This paper describes the issues associated with the development of a system simulation which combines: the needs of Hardware and Software Co-Simulation; the simultaneous use of C and VHDL; the use of (graphical) real-time stimuli; and the use of (graphical) real-time display techniques. The major focus of the paper deals with merging separate versions of time: the time as seen by the hardware (measured in ns or ps); the time as seen by the software (measured in ms or seconds); and the time as seen by the user of the simulation (i.e. the person entering the graphical stimuli and interpreting the graphically displayed simulation results). The paper: defines co-simulation needs; examines the problems associated with the simulation of hardware-software and software-hardware interaction; describes the set of heuristics used to facilitate efficient interaction; and examines techniques to facilitate real-time interaction with the simulation. Resolution of these problems is fundamental to an interactive co-simulation system.

1.0 Introduction

This work has been undertaken to determine the feasibility of using a VHDL based simulation to develop both hardware and software simulation test environments. In the development of combined hardware and software systems, most of the software development is delayed until a hardware prototype exists. However, this delay discourages software issues from having an impact on the hardware development. One possible alternative would be to develop a high-level simulation of the entire system and then simultaneously refine both the Hardware and Software dimensions.

The co-design system described by Buchenreider et. al. [1] combines the use of C and VHDL to facilitate the design of combined Hardware and Software Systems. The goals of their work has been to develop a concurrent design environment. They have concentrated on issues such as: the use of a common specification and design strategy for both hardware and software sub-systems; design of the communication facilities between the hardware and software sub-systems; the use of commercial tools to facilitate their proposed specification structure; and a structured approach to combine these into one design framework. In the work described here, we concentrate on the issues associated with time management in a co-simulation system. We have not attempted to significantly alter the techniques currently used by the hardware and software design teams, but rather to provide them with a common simulation environment which can be used to test their sub-systems as they are refined.

The results presented in this paper have been developed by implementing a co-simulation of a hypothetical problem (See Section 1.1 and Section 2.0). Section 1.2 defines co-simulation and Section 1.3 specifies the terminology used in the paper. Section 3.0 examines the Co-Simulation Harness (CSH) which synchronizes the hardware and software simulation units. Section 3.0 also explains the techniques used to permit the software being developed to be run on the simulation workstation, while the hardware being developed is simulated using VHDL. Section 4.0 examines the problems involved in simulation with real-time input and output.
1.1 Hypothetical System

In order to examine the issues associated with Co-Simulation a small problem has been developed which incorporates all of the issues fundamental to co-simulation. The system being simulated is based on a simple telephone switch (shown in Figure 1a).

The telephones and the switch hardware are defined in VHDL to permit evolution of the design to an implementation. The switch provides only basic two-party connections among four telephone lines. It has been assumed that the Call Processing Computer (CPC), responsible for the execution of the Call Processing Software (CPS), is an available sub-system. The CPS however, is to be developed and tested using the simulation system. The CPS, written in C, is to be used for both simulation and development.

1.2 What is Co-Simulation?

Consider the development effort involved in a system, such as the one shown in Figure 1a. The major effort is to be expended in the development of the hardware associated with the switch and the design and implementation of the CPS. The goal of Co-Simulation is to provide the switch and CPS developers with a test bench which closely models the part of the system that they are not designing. The goals of the co-simulation study, are listed below.

- The use of a hardware description language which can be used to simulate the operation of the hardware and be used as input to synthesis tools (VHDL in this case).
- The use of a software description language which can be used to describe the algorithms to be implemented and be used as the implementation language (C in this case).
- An efficient simulation of the software component of the system. Interpreting C code, or operating using a simulated computer for complex algorithms would be too slow.
- Although not required for the co-simulation needs of this project, we have incorporated one extra goal for this implementation, it must be able to accept real-time stimuli.

Figure 1 shows the evolution of the Co-Simulation Harness (CSH) used in this project. Figure 1a shows the true system which is composed of: the to-be-designed switch hardware, the CPC; and the to-be-written CPS. It would be possible to simulate the switch hardware and the CPC (with memory containing the binary image of the CPS) at the hardware level (as shown in Figure 1b), however this would be very slow. In fact, if part of the software development system (i.e. the compiler) were unavailable it would be impossible.

Figure 1c represents the ideal model of the co-simulation system. The CPC has been replaced with a Co-Simulation Harness (CSH) which provides the correct interfaces for the hardware designers (the switch) and the software designers (the production code). The CSH only simulates enough of the CPC so that the operation of the Switch and Software is very close the operation of the final system. Although much of the complexity of the CPC can be removed there remains one problem with this model — how to estimate the amount of time needed by the CSH to perform its operations.

To overcome this problem the software designers provide estimates for how much CPC time is required for the various activities. A CSH based on this approach is shown in Figure 1d. The port abstraction is a VHDL specification of the various signals which interconnect the Switch and the CPC. The kernel abstraction and hardware software time management function provide an interface for the software to access the (simulated) hardware and a technique to pace the execution of the software at a rate similar to that expected in the True system. The structure and limitations of the CSH are discussed in more detail in Section 3.0.

1.3 Times: True Time, Simulation Time and Execution Time.

There are three different time values associated with the simulation. In the definitions given below, consider a system, which is to be designed and simulated. In this system it is known that some operation \( O \) is to be performed. The three times associated with operation \( O \) are: \( T_{O} \), \( S_{O} \), and \( E_{O} \).
1. **True Time**: $T_O$ is the time that the operation will occur in the true system (that is the system and not the simulation of the system).

2. **Simulation Time** $S$: in order to simulate a true system it is necessary to estimate, or in some other manner determine the time required to perform various activities. The simulation time of an event ($S_O$), is an estimate of the time of that event in the true system ($T_O$). In most simulation problems the goal is to have $S_O = T_O$.

3. **Execution Time** $E$: each simulation requires a certain amount of computer time to execute. That time, sometimes called the wall-clock time, is referred to as execution time in this paper. $E_O$ is the time (as observed by a simulation operator) of the occurrence of $O$ while the simulation is executed on some platform. In most simulation problems the goal is to reduce the execution time, and thus there is seldom a need for an examination of the relationship between $E_O$ and $S_O$, however this relationship is key to dealing with real-time stimuli (see Section 4.0 for a more detailed discussion).

In the paper various atomic actions are labeled as $A_i$. $T_{A_i}$ is used to refer to the true time associated with atomic action $A_i$ (and similarly for $S_{A_i}$ and $E_{A_i}$). In the paper events are specified as $V_i$.

### 2.0 Implementation

The simulation problem, introduced in Section 1.1.1, and discussed in more detail in this section, has been implemented using the Vantage VHDL [3], the SL-GMS graphics package [2] and C.

The implementation, shown in Figure 2, involves 5 workstations and 10 unix processes. All of the simulation activity is performed on the co-simulation workstation. In addition there are 4 slave workstations. Each slave
workstation displays a graphical image of a telephone and accepts mouse input events to request changes in the telephone state (e.g., go off hook or dial a '3'). Each slave workstation uses a microphone and a speaker to provide a voice channel between the callee and the caller.

The UNIX processes needed for the system (also shown in Figure 2) are: the main co-simulation process; four voice processes—each responsible for accepting data from the microphone at a workstation and transferring the voice data to a voice process on a remote workstation, and similarly for the speaker; four graphics processes which accept input signals from the operator at a slave workstation, and display the status of the local phone on the screen of that workstation; and one console process which displays the current status of the simulation. The links shown in Figure 2 are non-blocking UNIX sockets.

The Co-Simulation process has been structured as a VHDL system, and thus time is managed by the scheduler associated with the VHDL simulator. It is beyond the scope of this paper to examine the implementation in detail. The only consideration which needs to be mentioned is that the interface to the CPS (implemented as a C architecture) was constrained to integer and enumerated data types.

The graphics processes use a commercial C graphics package (SL-GMS [2]) to facilitate the development of a Graphical User Interface (GUI) for real-time user input and feedback. We have been able to develop the graphics application by re-use of a previously developed graphical telephone interface.

3.0 Hardware vs. Software Time Resolution — CSH Operation

3.1 CSH Issues

The CSH must provide an efficient way to execute the CPS, while at the same time not sacrificing the fidelity of the simulation. Thus it was decided to permit the CPS code to execute on the simulation workstation and to require the programmer to provide estimates for the expected execution time for the CPS operations.

The fidelity of a simulation indicates how close the simulation operation is to the operation of the real system. Fidelity can be evaluated using two criteria: the sequence of events performed and the time of these events. The goal has been to perform correct operations at approximately the correct time. This is in keeping
with the general philosophy of event driven simulation; although the event may occur at a slightly different time in the true system, the event does occur.

The timing estimates for events within the hardware and software domains are fairly precise. (If a hardware event causes another hardware event, the timing is reasonably correct and similarly when a software event causes another software event.) The major source of error is when the two domains interact, that is an event from the Switch causes an interrupt in the software (s⇒s), or an action of the software requires a change in the switch hardware (s⇒h). In the case of s⇒h requests, since the switch is modelled with a finer time resolution, if the software request is issued at the correct simulation time, the correct hardware action should take place.

However, in the case of a h⇒s request, given that the software may be in the middle of a complex algorithm (in the true system), it is necessary to arrange for the software to permit hardware requests in a timely manner. However, if the software executes in a native mode on the simulation workstation, there is no way to determine how long an action should take when executed on the CPC, or what the impact would be if an operation were to be interrupted when partially performed.

3.1.1 Software to Hardware (s⇒h) Interaction

To facilitate interaction the CPS and the CSH must arrange for each s⇒h change request to be transferred such that $S_s \rightarrow h = T_s \rightarrow h$. In order to accomplish this the programmer partitions the software problem into a number of atomic operations ($A_1, A_2, \ldots A_n$). Although in the true system, action $A_i$ takes place gradually, in the simulation, it starts (correctly) and finishes at time $S_{A_i}^{start}$, as shown in Figure 3a. At the end of the simulation of $A_i$, a wait is issued which re-synchronizes the software simulation with the estimate of $T$ using the delays provided by the programmer ($D_{A_i}$).

The operation in this case relies only on an accurate estimate for $D_{A_i}$, and a reasonable set of atomic actions. Since it is important for the s⇒h interactions to occur at the correct time, it was decided that the programmer should be required to partition the problem into atomic actions comprised of at most one s⇒h interaction. Although this interaction could be assumed to occur at either the beginning or the end of the action, it was found to be simpler to require the partition to have atomic actions with s⇒h interactions at the beginning of the action.

Consider the operation in response to the (asynchronous) request R shown in Figure 3b. The request from the hardware arrives at time $T_R^{start}$ and service starts immediately (see Section 3.1.2 for more discussion of h⇒s interaction). At some point in the service, a decision is made to send a message to the hardware (i.e. a change to a register value), $T_R^{emit}$ in Figure 3b. A little later the operations involved in request R are completed ($T_R^{stop}$). One atomic action $A_{R_1}$ performs all of the operations between $T_R^{start}$ and $T_R^{emit}$ (i.e. a small amount of time before $T_R^{emit}$) and $A_{R_2}$ performs all of the actions between $T_R^{emit}$ and $T_R^{stop}$. The message to the hardware is sent at the start of event $A_{R_2}$. Thus the programmer estimates the duration of these actions and the CSH is responsible for causing each action to start at the correct (simulation) time.
3.1.2 Hardware Software Interaction (h>s)

The simple h>s and s<h system described in Section 3.1.1 could be made to work except for the presence of interrupts in the system. Since there are a number possible sources of requests to the CPS (one for each phone), it is possible that an interrupt will occur (in the true system) while some atomic action is active. In order to compensate for this it is desirable to break the code into atomic actions so that it is not necessary to consider partially processed atomic actions. There is a clear trade-off between requiring the programmer to permit interrupts too frequently (for high fidelity simulation) and too infrequently (for high efficiency simulation of the software).

In the CSH (described in Section 3.2), the programmer selects atomic actions, such that if an interrupt arrives in the middle of an action, then the fact that action has completed before the interrupt has been serviced will not alter the operation or the results of the simulation.

3.2 CSH Implementation

The CSH structure resembles that shown in Figure 1d, and is shown in more detail in Figure 4. The Rx, Tx, and Scheduler unit are simulated using VHDL to implement a version of the port abstraction and kernel functions shown in Figure 1d. The User architecture is implemented in C and implements the CPS. The CPS has been structured as a sequence of atomic actions, composed of messages to the switch and changes to the local state of each phone. The User maintains a list of operations to perform. The scheduler can either: activate the User in a normal mode, in which case the CPS selects the next atomic action and proceeds; or activate the User in an interrupt mode, in which case it deals with the immediate needs of the interrupt (i.e. modifies a future action on the list, or causes the current action to be terminated prematurely). In a normal activation, the User completes \( A_i \) and requests that a message be sent to the Tx (by the Scheduler). The User also specifies, to the Scheduler, the delay \( (D_{A_i}) \) which is to elapse prior to the next scheduler activation.

The implementation of this part of the simulation involves the partitioning of the problem (the CPS) into atomic components using the two rules described in Section 3.1.1 and Section 3.1.2.
1. Each atomic action, $A_i$, can cause, at most, one $s \rightarrow h$ request and it must occur at the beginning of the time step associated with $A_i$ ($s_{AR_2}^{start}$ for $A_{R_2}$ in Figure 3b).

2. Each atomic action is completed at $s_{A_i}^{start}$ in the simulation, even though it would complete at $T_{A_i}^{Stop}$ in the true system (see Figure 3a). This means that an interrupt, even though it may occur at $s_{A_i}^{start +}$ does not alter the operation of $A_i$.

![Figure 4. Co-Simulation Harness](image)

There are three possible scenarios for the execution of an atomic action, as shown in Figure 5. Although the true system time $T$ and the simulation system time $S$ are only approximately equal, in the discussion below it has been assumed that they are equal and thus the discussion is all in terms of $T$. For this discussion assume that at the start of the execution of $A_1$ the next atomic action to be executed is $A_2$.

1. Figure 5a. There are no interrupts following the start of the atomic action. In this case the operation proceeds (and completes following a delay of $D_{A_1}$ at time $T_{A_2}^{start}$) for the True System (TS). In the simulation system, using the CSH, the operation is performed while the simulation time is $T_{A_1}^{start}$. The initiation of $A_2$ is delayed by $D_{A_1}$. Any $s \rightarrow h$ interaction required by $A_1$ takes place at time $T_{A_1}^{start}$, and any state change in the program state becomes effective at the same time. The CSH prevents any further action by the CPS until the estimated delay of $A_1$ has elapsed ($D_{A_1}$).

2. Figure 5b. There is an interrupt, however it does not affect the current operation, other than to delay $A_1$'s completion by the time needed to service the interrupt. The interrupt service routine (ISR) could alter the selection of the atomic action to be initiated following the completion of $A_1$. Thus, as in case 1 $A_1$'s message and actions occur at $T_{A_1}^{start}$, however the next atomic action starts after the delay of $A_1$ ($D_{A_1}$) plus the CPC time required by the ISR ($D_{ISR}$). Multiple interrupts are handled in a similar manner.

3. Figure 5c. There is an interacting interrupt, that is one which alters the sequence of execution, such that $A_1$ would not complete in the TS (in this case it is replaced by $A_3$). However, the activity associated with the original action is complete in the simulation and cannot be unwound. The new action ($A_3$) is started at the correct time (following the completion of the ISR).
4.0 Real-Time Synchronization

This project has also examined the issues associated with accepting (and displaying) data in real-time. There are two basic issues associated with real-time interaction: user perception; and simulation independence.

It is necessary for the user (the operator who enters and interprets the real-time data) to have a simulation which performs in a manner very similar to the true system. It would be very difficult to interact with a simulation in which the simulation time ($S$) progressed erratically with respect to the current (wall-clock) execution time ($E$). Section 4.1 examines the techniques used to linearize the relationship between $S$ and $E$.

Even if the simulation has been implemented such that $E = S$ it is also necessary for the simulation to be independent of the real-time input-output system. Section 4.2 examines the interface between the GUI and the simulation. This interface has been designed to permit the user to only enter real events. Thus it is not necessary for the user to enter instructions such as proceed for $x$ seconds.
4.1 Linearizing the Relationship Between $S$ and $E$

In order to introduce real-time stimuli, it is necessary for there to be a predictable relationship between $S$ and $E$. It would be feasible to use any linear relationship ($E = kS$, where $k$ is any constant) however, we have assumed that a value of $k = 1$ is the goal.

In any simulation the relationship between $E$ and $S$ is determined by the execution power of the simulation platform and as a result some values of $k$ are infeasible. It is possible to increase the value of $E$, for a given value of $S$ (by slowing the execution) and thus to increase $k$. However, for a given platform, $k$ cannot be decreased below its intrinsic value. It should also be noted that in general, a simulation will not exhibit a linear relationship between $S$ and $E$, however if it is possible to artificially slow the simulation, then it is possible to provide the appearance of a linear relationship.

This section describes the synchronization of $S$ and $E$ for the system described in this paper. When simulation interacts with a user, $E$ and $S$ can progress in two different ways:

1. $S$ can progress faster than $E$ ($k < 1$);
2. $S$ can progress slower than $E$ ($k > 1$).

4.1.1 Simulation is Faster ($k < 1$)

In this situation, the simulation running on the simulation platform can outperform the true system, and it is possible to force the platform to perform operations at the same time $E$ as the true system $T$.

Consider an operator who wishes to issue three commands to the simulation of a system at true times $T_{V_1}, T_{V_2}$ and $T_{V_3}$, corresponding to events $V_1, V_2$ and $V_3$. Figure 6 indicates two synchronization alternatives. Following each event, there is a period of intense computation. During this period the simulation time progresses very slowly, as indicated by the lines of slope $S_2$. Following completion of the activity associated with the event, the simulation returns to a normal progression rate of execution time, indicated by the lines of slope $S_1$. Synchronization scheme Synch1 is based on no special attempt to synchronize $E$ and $S$. In this case, the difference between $E_{V}$ and $S_{V}$ for event $V$ increases as the simulation progresses. In the implemented simulation (Synch2), $S$ and $E$ are re-synchronized at regular intervals. For example, assume that $E_{V} = S_{V}$ for some event $V$, as $S$ progresses at slope $S_2$, it lags $E$. When the intense execution is completed $S$ approaches and then surpasses $E$ (as shown in the expanded view in Figure 6) at a rate of $S_1$. Once $S > E$ (point A), the co-simulation process stops until $S = E$ (point B).

To handle this case we have used the foreign language interface provided with the Vantage toolset to access system utilities to suspend the co-simulation process for a specified period of time. The (VHDL) process (implemented in C) which wakes up at regular simulation time intervals (i.e. every $\Delta S$ simulation time units) to control the progression of simulation time is referred to as the synchronization process (SP).

There are two slightly different approaches to synchronize execution time and simulation time:

- Each time the synchronization process is activated it calculates the change in $E$ (referred to as $\Delta E$). If $\Delta E < \Delta S$ then the simulation execution is suspended for $\Delta S - \Delta E$ real time units. At this point it is assumed that $E$ and $S$ have been re-synchronized and the SP is suspended for $\Delta S$ simulation time units ($\Delta S$ is adjusted as a function of the characteristics of the simulation). This approach works when $\Delta E \leq \Delta S$; however once $E > S$ (i.e. as the result of a heavy workload) $E$ will remain larger than $S$.
• The second scheme is based on an attempt to keep $E = S$ at all points in the simulation. If $E < S$ then the simulation execution is suspended for $S - E$ real time units. Once this time has elapsed, the SP is suspended for $\Delta S$ simulation time units as discussed previously. This is the technique we have used since it allows the simulation time to catch up with the execution time after a large workload. We used $\Delta S = 0.25$ seconds.

4.1.2 Simulation is Slower ($k > 1$)

In this situation, the platform cannot operate as fast as the system. To overcome this we provide feedback to the user to request slower data entry until simulation has had a chance to catch up. In our case, the phone in the GUI changes colour until the previous data entry has been processed and the switch has received the previous state change. If the GUI was not synchronized with the simulation you could, if the simulation was very slow, send two events at the same ($S$) time which is impossible in the true system. The protocol used to synchronize the simulation and the handset is described in the next section.

4.2 Accepting Real-Time Input

To integrate the GUI (which deals with execution time) with the switch model (which deals with simulation time), a synchronization scheme must be used between the switch and the handset. The simulation cannot just block waiting for an event to come from a handset since this would stop simulation progress until there is an event (this problem is compounded by the fact that there are four such handsets driving the switch). Also the GUI must be independent of the simulation so that it can accept real-time input from the user. Since both processes must be independent, it was necessary to use a non-blocking communication protocol.

The switch sends a polling message at regular intervals to the handset; it also attempts to read a message from the handset at the same regular intervals. Each time the handset receives a polling message, it replies with a message containing its status. When the switch attempts to read from the handset it processes any messages received.
5.0 Concluding Remarks

The project discussed in this paper has examined some of the timing issues associated with the simulation of a system which requires: simulation of a hardware component suitable for VHDL specification and a software component which is to execute prototype code in a native mode on the simulation platform; real-time user interaction with the simulation; and the use of both VHDL and C as specification languages.

The major problem to overcome is the difference in simulation strategies for the hardware and software. In the hardware simulation there is an assumption that the time needed for each unit can be estimated very accurately (the delay of a gate or component can be found or estimated using established techniques). In the simulation of the software there is a desire to execute a program, very similar (or identical) to the program to be used in the true system. However, that program is to be executed in a native mode, on the simulation computer, and not on the computer (or even a simulation of the computer) to be used in the true system. To overcome this problem we have forced the programmer to specify their program in atomic actions, with the following constraints:

1. There is at most one change to a hardware value.
2. Once started it cannot be prevented from completing by an interrupt from the hardware.
3. The time needed for each atomic action must be specified.

Although it would be possible to do this in an automatic manner, there has been no attempt to automate the process in this project.

This system has been implemented by means of a Co-Simulation Harness (CSH) which is responsible for providing an interface for both the hardware and software, which is similar to the one found in the true system. The CSH also must provide for efficient execution of the software.

In the problem discussed, we have also provided a scheme to demonstrate the use of real-time input and output from the simulation. Thus, there is a need for the simulation to progress at a rate comparable to real-time. To accomplish this the system compares the elapsed time with the current simulation time and, if the current simulation time is greater than the elapsed time, the simulation is suspended until the two times are the same.

Although this scheme requires the programmer to prepare their solution in a particular manner, it could be argued that preventing interrupts from occurring at certain times, or interrupting certain code could lead to better specified programs.

6.0 Acknowledgments and Addresses

The authors are indebted to Moustapha Bouralha and Trevor May for their tireless effort to develop the voice processes for use in the demonstration.

Wayne Loucks can be reached at the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo Ontario Canada, N2L 3G1, (e-mail Wayne.Loucks@Uwaterloo.ca). Bernard Doray and David Agnew can be reached at BNR, Mail Stop CRK-49, P.O. Box 3511, Station C, Ottawa, Ontario, Canada K1Y 4H7 (e-mail doray@bnr.ca and crm24@bnr.ca respectively)

7.0 References