Peer-Based Multithreaded Executable Co-Specification
Donald E. Thomas, JoAnn M. Paul, Simon N. Peffers, Sandra J. Weber
Carnegie Mellon University
ECE Department
Pittsburgh, PA 15213 USA
+1 412 268-3545
{thomas, jpaul, peffers, sweber}@ece.cmu.edu

1. Abstract
We are integrating language-based software and hardware behaviors in C/pthreads and Verilog for unrestricted peer execution of the domains, including bounded (finite) and unbounded notions of computer system modeling. Since we do not restrict the modeling currently available in each domain, our co-specification is inclusive of both reactive and data-intensive systems. By viewing all mixed system state as shared memory accessible by threads in each domain, we differentiate domains by system resource inferences. We introduce a unified multithreading model for execution and motivate the need to expand the specification capabilities currently available in each domain for mixed-systems using widely accepted languages as a basis. We discuss specific aspects of our cosimulator, provide examples and results, and indicate future directions of our work.

1.1 Keywords
Hardware/Software Co-Specification, Co-Simulation, Mixed-System Modeling, Multithreading.

2. Introduction
The specification of large, complex computer systems typically includes many separate software and hardware behaviors. Inferring system composition for mixed domain behavioral models and obtaining executable co-specification in widely accepted languages both require a true peer relationship between the domains. That is, neither paradigm can be thought of as being primary or in control. In bringing peer-based mixed domain modeling and simulation to a design team, we present: some differences in system resource composition when specifying a behavior using the two domains; how the domains interact for state updates in a multithreaded common state model; and some novel means of allowing the domains to interact. At this level of computer system modeling, full processor models are not required as in other work[1,4,9,13]; cycle accurate validation is not yet the issue. Rather we are modeling the complex interaction of existing hardware and software language-based models that represent system behavior in the advancement of state. Our system model includes multithreaded software models along with threaded hardware models (viewing Verilog always blocks and VHDL process statements as threads) in advancing mixed system state. For hardware and software design teams to be productive, they must be able to describe portions of the system’s functionality in a manner appropriate for each domain and then be able to simulate and co-execute a mixed specification. Software designers must not be forced to limit themselves to hardware models and vice versa. For these reasons, we allow modelers the full C language with threads along with Verilog for computer system description.

The primary difference in modeling using threads from one domain or the other has to do with the inference of resource modeling on system composition. Hardware descriptions model physical resources in the system with behavior. In contrast, software descriptions model behavior without specifying resources. It is important to allow the full use of software and hardware modeling languages to specify computer systems behaviorally.

To freely mix hardware and software models without any restriction to the language usage, a computer system model must: address the type of unbounded, Turing Machine models of memory currently utilized by software programmers; provide shared memory modeling between domains in a manner natural to hardware designers and software programmers; allow for dynamic threaded forms of concurrency; capture mixed hardware/software concurrency in a manner which is natural to the language domains of hardware and software modelers; and model data- and compute-intensive systems. These capabilities are needed to model large complex computer systems. Previous works such as those based on CFSMs [2,8] do not readily address these modeling issues.

Language-based models of computation utilize names for system state. Names of state in software correspond to addressable system memory and names of state in hardware correspond to labels on wires (that may be connected to the outputs of registers). Each domain has specific assumptions about the model of computation used to advance state. By allowing all state (wires & memory) to be considered shared by both domains in a common shared-memory model of mixed-domain system state, and focusing on the physical resource implications of both computation and state modeling abstractions in each domain, we provide the basis for hardware and software models to co-execute as concurrent threads that advance mixed system state. Threads both encapsulate behavior in each domain and allow sharing of state by name. This paper discusses the merging of hardware and software models of computation for the advancement of state by name. We do not yet have a formal model; our starting point for one is the abstractions already widely used in the vast majority of computer systems modeling today.

Unlike [6], we do not require component or interface object types to encapsulate both hardware and software models as event-driven entities for which shared memory models need to be built. Our threads encapsulate behavior in a manner natural to language-based models of concurrency already present in each domain, including event driven activation of hardware threads, and explicit activation of software threads. In addition, we introduce software resource threads, which have computation and state modeling characteristics which are a combination of hardware and software models.
We include the development of a co-simulator which, in contrast to [5], provides peer domains. Our co-simulation scheduler co-executes a unified multithreaded system model that includes software modeled with C/pthread[7] and hardware modeled with Verilog[12] on a peer basis. We present the differences in modeling in the hardware and software domains, define how these models interact during a simulation, and define methods of bringing time to the software domain using a shared state paradigm. We present co-simulation examples that demonstrate interactions between peer hardware and software models. Our dual language-based approach motivates the capabilities needed in modeling large computer systems which include both hardware and software paradigms for the advancement of system state.

3. Codesign Systems Modeling

Hardware models describe resources that execute a computation. Adding behavior in the hardware domain adds the resources necessary to implement the behavior. Thus, if an extra hardware thread is added, the hardware resources of the system expand. Some hardware threads may produce FSMs which add concurrency and others may produce combinational logic. Software describes a computation without specifying the resources for it. Unlike a hardware thread, a software thread is not guaranteed concurrent resources on which to execute. While the specification of a behavior in a software thread may permit concurrent execution if the resources are made available, the resource is not a part of the software model of concurrency. Thus, true concurrency is not added to a system by adding a software thread. Indeed, the actual level of concurrency is not determined until the multithreaded software runs on a multiprocessor system, or the software is executed in simulation which permits the effects of additional processors on system behavior to be modeled.

Large portions of computer system behavior are modeled purely in the software domain. Software architects have borrowed hardware-like analogies in reasoning about the building of complex computer software and software components are becoming popular in the software community. But, there are important distinctions with the way structural terminology has been applied by software architects in the design of software components [10,11]. Software components are a software reuse technology and do not bring parallelism or resource models to a software description. Indeed, the resource modeling for the software domain is not captured without introducing simulation to provide pseudo-parallel execution in virtual time.

Figure 1 compares simple hardware and software pthreads. The two software threads on the left (SW1, SW2) specify behavior that corresponds to the behavior specified by the two hardware threads on the right (HW1, HW2). All of the threads shown are infinite loops. Hardware threads are statically declared and execute either once at time zero (initial blocks) or loop infinitely (always blocks). Software threads may loop infinitely or be dynamically created and destroyed by themselves or other threads.

```
while(1) { /* SW1 */
    pthread_mutex_lock(&mtx);
    C = A + F;
    B = D + E;
    pthread_mutex_unlock(&mtx);
} while(1) { /* SW2 */
    pthread_mutex_lock(&mtx);
    A = B + C;
    F = D + E;
    pthread_mutex_unlock(&mtx);
} C/pthread Software Threads
```

```
always @(A or D or E or F) // HW1
begin
    C = A + F;
    B = D + E;
end
```

```
always @(B or C or D or E) // HW2
begin
    A = B + C;
    F = D + E;
end
```

```

Figure 1 Comparison of Software and Hardware Threads
```

Unlike hardware, the software threads are protected by the same mutex such that the behavior inside is a critical section; only one thread can be executing it at a time independent of the number of processors executing the threads. Unlike software, the behavior of the hardware always blocks updates the simulation state of the variables written in each block atomically (B, C in HW1 and A, F in HW2). The update of C in HW1 cannot be seen by any other thread in the system without the update of B in HW1 also being available. In addition, both always blocks, HW1 and HW2 are simultaneously activated whenever there is a change in variable E in its sensitivity list; this behavior is ensured independent of the number of processors upon which the simulation executes. In fact, the HW1 and HW2 threads can be considered to be structurally interconnected with feedback. So long as the values of A produced by HW2 and B produced by HW1 change in a given simulation cycle, the execution models of the hardware threads are implicitly scheduled. This is true because a change in the value of B (A) in HW1 (2) implicitly causes thread HW2 (1) to be activated, since B (A) is on the sensitivity list of HW2 (1). The hardware scheduler is activated implicitly by the semantics of hardware specification. By contrast, software threads can only be scheduled by explicit calls to the thread scheduler. In the semantics of software modeling, access to shared resources (in this case shared memory) must be explicitly controlled.

With implicit invocation of computation via a hardware scheduler and atomic updates, hardware models of interconnected gates and components provide the hardware designer with an almost pure model of parallelism, even as large portions of behavior continue to be specified in similar syntax in both domains. More than just for timing verification, control over simulation time provides the appearance of parallel execution of behaviors in virtual time.

From a simulation perspective, the differences between domains are in the methods of updating system and simulation state and in the methods of synchronizing the execution of the threads. Our unified co-simulation thread considers hardware and software behavioral updates between potential scheduler events (PSEs) as in Figure 2. PSEs are mixed-domain events, such as wait or @or-ed_event, which may cause the executing thread to become blocked. Between PSEs, we find the “normal” behavioral code that describes the update of system state. In this view, software system state updates are immediate and have no side effects (i.e., there are no wires to implicitly execute other models). Hardware updates are atomic with respect to the update of system state, have guaranteed propagation effects on other threads within a simulation time, and changes are propagated on wires which implicitly cause other models to execute — without requiring an explicit call to a scheduler.

The properties of atomicity and implicit invocation of other hardware threads are necessary to model hardware behaviorally (using a threading paradigm) and to model the effects of true, concurrent structural parallelism. By contrast, pure SW threads update state immediately, have no implicit side effects when a value is updated, and can only explicitly cause other threads to execute via function calls to a thread scheduler. These are language considerations.
In a co-simulation based upon a unified multithreading model, when memory shared between hardware and software is written to by a software or hardware thread there are three actions that may occur as illustrated in Figure 3. If a hardware model is writing to memory shared by other hardware models, then the normal actions of a hardware simulation scheduler would be invoked. If software is only writing to a memory shared with another software thread, there is no need to invoke a scheduler. However, if hardware is waiting for a level or an or-ed_event from memory shared between domains (e.g., as you might find in memory mapped hardware), a co-simulation scheduler that propagates the event between domains is needed. That is, when software writes to the memory location, hardware wakes up and executes.

The merging of multithreaded mixed-domain modeling is supported by the addition of two shared memory updates to the software domain.

- Non-Atomic Shared Memory Concurrent Updates (|=)

When a software thread updates a memory that may be connected to a wire in the hardware domain, other threads (both hardware and software) may be connected to the wire, such that immediate, implicit action is required by the co-simulation scheduler. The invoking of the hardware scheduler is implicit and does not require an explicit call similar to the way a hardware scheduler is invoked at the end of a block of behavior in the hardware domain. However, the scheduler is invoked immediately, not at the next PSE, unlike the implicit invoking of the hardware scheduler at the next PSE. The |= assignment guarantees that hardware models will see the change without the necessity of a subsequent PSE. This is useful in modeling situations where hardware is mapped to shared memory.

- Atomic Shared Memory Concurrent Updates (:=)

We can also allow software threads to perform updates in the same way that hardware does. With the := assignment operator used in our examples, the change in a value as seen by other software threads does not always depend on the hardware scheduler being invoked. The := operator is analogous to the blocking assignment (=) in Verilog.

Interestingly, if we allow the software domain to update state atomically, using :=, it begins to appear as a model of a hardware resource. Behavioral hardware descriptions trade serialized descriptions of behavior (which require local state) for larger groupings of behavior and structure. A single line of a behavioral description is typically meaningless to hardware system state; all lines between PSEs must be considered atomically. The large grouping of behavior permits sequential descriptions to appear truly parallel and to represent the resources in conjunction with the specified behavior. Providing atomic update in the software domain starts to bring the notion of resource modeling to software, since true atomic updates can not be done without the resources (bus width) required to complete the transfer atomically. The resource modeling is still different from pure hardware modeling, however, since the memory (state) modeling in the software domain remains unbounded, possibly even specified globally and shared by other software threads.

To further the use of the software domain to model and simulate resources, we are adding the @or-ed_event operator along with a software wire as implied by the middle arrow of Figure 3. @or-ed_event and := provide a modeling situation which has execution properties similar to hardware resource models, and state properties similar to software memory models. The addition of resource modeling to the software domain using @or-ed_event and := further serves to illustrate the differences between domains and indicates the system-level architectural modeling possibilities of extended forms of resource modeling to the software domain.

4. Multithreaded Co-Simulator

We have developed a co-simulator based on our unified model of multithreading as illustrated in Figure 4.
5. Demonstration And Examples

We demonstrate mixed-domain multithreaded modeling in our co-simulator with two examples.

5.1 HDLC Example

HDLC is a simple serial transfer protocol. The example in Figure 5 demonstrates how the hardware component of the co-simulation can be stimulated through assignments to wires and the use of hardware-like PSEs in the software component. Software threads also can wake-up in response to events generated by the hardware.

```plaintext
/* Software thread */
2 void *feed_encoder(void *arg) {
3     char *buffer;
4     /* addr, num_char, and done are global wires */
5     buffer = (char *)malloc(MAXLINE*sizeof(char));
6     while (fgets(buffer,MAXLINE,input_file) != NULL) {
7         addr := buffer; num_char := strlen(buffer);
8     }
9     hwWait(done == 1); // suspended until HW sets done
10    addr := buffer; num_char := strlen(buffer);
11     // increment of 1 to a memory pointer translates to the
12     begin
13     always @(addr)
14     mem charbuf [ ];
15     output done;
16     . . .
17     input [31:0] addr, num_char;
18     thrSleep(5); // suspended for 5 HW simulation time units
19     hwWait(done == 1); // suspended until HW sets done
20     }
21 }
22 /* Hardware module */
23 module hdlc_encode (addr, num_char, ..., done);
24     addr must be large enough to store SW virtual address
25     input [31:0] addr, num_char;
26     . . .
27     output done;
28     mem charbuf [ ];
29     always (@(addr))
30     begin
31         data[31:0] = charbuf[addr];
32         // increment of 1 to a memory pointer translates to the
33         // smallest addressable unit (a byte) in the host memory
34         addr := addr + 4;
35         . . .
36         done := 1;
37     endmodule
```

Figure 5  Pseudo-Code for Hardware and Software Threads

The software thread reads each string (line) from a data file and stores it in a local buffer in the thread’s address space. Once the string has been written to the buffer, the software thread writes the address of the buffer and the number of characters in the buffer to wires in the port list of the hardware encoder with the := operator (lines 7, Figure 5). We use the := operator because addr and num_char are updated atomically, with respect to the hardware simulation. One value is of no use to the hardware without the other. Using the := operator for these assignments would result in the hardware scheduler being unnecessarily invoked twice.

The hardware encoder is on the fanout of the address, so at the next PSE, thrSleep, the hardware simulation is invoked and the encoder is activated. Note, however, that the buffer holding the string is not part of the I/O of the encoder module. An interesting aspect of this example is the DMA style interface between the software source thread and the hardware encoder. The mem type is used by the hardware to access a location in the software memory (line 19, Figure 5). Memory is byte addressable; therefore, indexes into charbuf are interpreted as byte addresses. Currently, the value accessed is the size of an integer in C. Knowledge of the simulation platform is required to use the mem type properly.

5.2 Dining Philosophers Example

The dining philosophers problem [3] further illustrates some of the inter-domain PSEs and scheduling methods in our co-simulator. Several versions of the dining philosophers were modeled including purely hardware, purely software, and several mixed-domain versions. The multithreaded pure software implementation is relatively straightforward. Each philosopher is in his own thread and the chopsticks are represented by shared variables. Each chopstick variable is protected by a mutex to guarantee atomic updates. In the pure HW implementation, registers represent the chopsticks and hardware state machines represent the philosophers. The problem is more interesting when the implementation is mixed between hardware and software. We will present one mixed-domain implementation here.

This version of dining philosophers has philosophers in both hardware and software and chopsticks in hardware. Figure 6 shows this allocation of resources between the hardware and software domains. Mutual exclusion to the shared chopstick register file is through multi-phased clocks in hardware and mutexes in software. Hardware and software testbenches allow us to change the amount of time the philosophers spend eating and thinking. Each time a philosopher eats, the appropriate testbench reads in a new pair of values from a file and gives them to the philosopher. The software testbench is a pure software thread controlled entirely by the Solaris thread scheduler. It synchronizes with the software philosophers via mutexes.

![Figure 6 Hardware/Software Resource Partitions](image)

One of the benefits of modeling using a co-simulator is the ability to move functionality between hardware and software. In this example, philosophers in both domains have identical behavior and, for the purposes of illustration, require the same amount of simulated time to execute. Figure 7 shows the performance benefits of moving philosophers from software to hardware while keeping the total number of philosophers constant. Thus, the only change we make is moving threads from the software domain to the hardware domain — no other behavioral or timing changes are made. To convert a HW philosopher to a SW philosopher with the same behavior, the required state is moved from registers to local variables or data structures and software threads are created instead of hardware instantiations. Software threads use a mixed h/s model of mutex activation with updates to the shared domain chopsticks made through the atomic shared memory update operator (:=).

The vertical axis shows the rate at which philosophers can acquire two chopsticks. As more philosophers are moved to hardware, the philosophers eat more times overall since hardware philosophers have devoted execution resources, unlike software threads, which have to share the CPU. Thus, moving a behavior from a software multithreaded model to a hardware model effectively increases the resources and thus the true parallelism modeled by the system. This type of information can help the designer achieve a desired cost/performance balance.

Table 1 shows the number of times HW and SW philosophers eat using two different timing models. The first row in the table shows results assuming that each line of software takes zero hardware time. The remaining rows show the results of varying the number of hardware simulation time slots equated to a line of code in a software thread. The results are more representative of a real system: hardware runs relatively faster than software. The ability to relate the execution speed of a line of specified behavior in each domain allows for system level consideration of a line of specified behavior, without full processor models.
Figure 7 Performance Benefits of Resource Allocation

Table 1: Varying Simulation Time Per Line of Code

<table>
<thead>
<tr>
<th>Time per Line of Code</th>
<th>Real Time (sec)</th>
<th>Number of Times SW Eats</th>
<th>Number of Times HW Eats</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (Untimed)</td>
<td>3.70</td>
<td>777</td>
<td>381</td>
</tr>
<tr>
<td>20 (HW clk speed)</td>
<td>3.73</td>
<td>52</td>
<td>437</td>
</tr>
<tr>
<td>40</td>
<td>3.43</td>
<td>28</td>
<td>426</td>
</tr>
<tr>
<td>80</td>
<td>3.36</td>
<td>15</td>
<td>422</td>
</tr>
<tr>
<td>160</td>
<td>3.24</td>
<td>7</td>
<td>431</td>
</tr>
<tr>
<td>320</td>
<td>3.14</td>
<td>3</td>
<td>419</td>
</tr>
</tbody>
</table>

A mixed simulation also presents the possibility of using two testbenches, one each for hardware and software. In our implementation of dining philosophers, the HW testbench controls the clocks. Both testbenches control the amount of time the philosophers in their domain spend eating and thinking. Having two testbenches allows the software and hardware domains to drive the simulation independently rather than making one testbench the master. Likewise, the software and hardware philosophers are peers in the co-simulation. Table 2 shows the results of changing the testbenches for the number of times the philosophers in each domain eat. The first two columns list the contents of the file controlling eating and thinking times. Each testbench reads a new pair each time their philosopher eats.

Table 2: Dining Philosophers Testbench

<table>
<thead>
<tr>
<th>HW TestBench (Eat Time, Think Time) in Clock Cycles</th>
<th>SW TestBench (Eat Time, Think Time) in HW Time</th>
<th>Real Time (sec)</th>
<th>Number of Times SW Eats</th>
<th>Number of Times HW Eats</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2) (20,40)</td>
<td>(20,40)</td>
<td>0.74</td>
<td>74.3</td>
<td>78.0</td>
</tr>
<tr>
<td>(4,8) (20,40)</td>
<td>(20,40)</td>
<td>0.76</td>
<td>63.7</td>
<td>39.5</td>
</tr>
<tr>
<td>(4,2) (40,80)</td>
<td>(40,80)</td>
<td>0.67</td>
<td>57.7</td>
<td>82.0</td>
</tr>
<tr>
<td>(1,2) (2,4) (20,40) (40,80) (80,160) (160,320)</td>
<td>(20,40)</td>
<td>0.68</td>
<td>38.0</td>
<td>65.5</td>
</tr>
</tbody>
</table>

Another feature of our multithreaded approach is the ability to dynamically create and destroy software threads throughout a co-simulation — a completely foreign concept in traditional hardware simulations just as resource threading is a completely foreign concept in software. Each of the threads in this example are peers in the co-simulation. In the previous discussion, the software philosophers were created at the beginning of the co-simulation and remained until the hardware simulation finished. However, the example is easily modified such that software philosophers are created dynamically throughout the co-simulation and exit before the co-simulation ends, as specified in the software testbench. Figure 8 shows the results of dynamically creating and destroying software philosophers. One philosopher is destroyed every 25000 time units for two iterations, then one is created every 25000 time units for two iterations, etc. The graph shows the rate of food consumption over time. As philosophers are destroyed food consumption decreases, as they are created it increases.

6. Summary

Our unified multithreading approach has formed the basis of mixed domain co-execution where hardware and software domains co-simulate as peers in the advancement of mixed-domain system state modeled in the same shared memory space. Our shared memory co-simulation includes notions of time that support mixed resource modeling, expanded notions of behavioral updates between domains, expanded resource modeling in the software domains, and host memory access in the hardware simulation. Peer-based language modeling permits the inference of system resource composition from the domains in which behaviors are described. It also establishes the possibility of forming arbitrary relationships between domains at arbitrary points in the system design process and levels in system design views. We are further developing a codesign system methodology based upon this work.

7. Acknowledgments

This work was supported in part by NSF Award EIA-9812939.

8. References