Hardware and Software as Dual Languages for Computer System Modeling

JoAnn M. Paul, Donald E. Thomas, Sandra J. Weber, Simon N. Peffers
{jpaul, thomas, sweber, peffers}@ece.cmu.edu

Abstract

Complex computer systems can no longer be effectively designed without some consideration of the interaction of the hardware and software domains. Language-based behavioral specification for both simulation and synthesis in the hardware domain has made it possible to consider common models of computer system behavior with domain-specific inferences on the physical means of implementing the behavior. A dual computation modeling analogy is drawn for illustrating physical modeling inferences that overlap each domain and those which differ from domain to domain. The dual analogy illustrates the importance of preserving semantics of hardware and software modeling in separate languages. Peer-based co-execution of behavior specified in both modeling domains is possible because of reasoning about the interaction of the computation and state resources that are implied by modeling in each domain. We are uniting two existing hardware and software languages, Verilog and C using pthreads for an executable co-specification. We illustrate our approach with examples of our cosimulator.

1. Introduction

The complete description of computer systems requires a mix of hardware and software models. Although the hardware and software modeling domains have similar language-based means for behavioral modeling, there are very different implications on the physical means of implementing the behavior. Our view of a computer system modeling is as a collection of executing behaviors which may be described in either the hardware or software domains. Since the hardware and software domains are both used to describe the execution model of the computer system, the domains must interact. The most appropriate basis for interaction of the domains is with respect to the physical modeling inferred by the behavior.

This paper describes two related approaches for obtaining an executable co-specification for the description of arbitrarily complex computer systems using the existing languages C (using pthreads [1] for multithreading) and Verilog [2]: dual models and peer execution. Dual models draw on an analogy from systems modeling and allow the modeling domains to be understood on the basis of the physical resource implications of a common set of behavior. Peer-execution is a result of an analysis of the language overlap in modeling already present in each domain with respect to physical modeling.

Our approach is to first understand the basic essence of hardware and software models and their relationship to computer systems modeling. Instead of limiting one to fit the confines of the other, we take a peer-based approach. This raises the impact of our work from traditional codesign or embedded system design to that of computer system modeling and design.

By establishing peer execution of the domains, we bring the modeling capabilities of each domain to the same conceptual level of physical system modeling. Hardware and software models both perform digital computation on current state and inputs, producing a new current state and outputs. The primary differences in the way hardware and software languages advance state can be understood with respect to the way each domain models resources. We describe physical models that are inferred by modeling in each domain as state resources and computation resources. These serve to distinguish the physical models each domain represents in addition to providing a common physical basis by which the domains may be merged.

In contrast to peer execution, current co-simulation systems typically allow software function calls from a hardware simulator to model the software partition [3], where the software functionality must be defined in terms of the hardware and its simulator. This only provides co-verification near the completion of a relatively small scale design; it does not address the basic essence of modeling in each domain. Further, our approach extends previous work such as [4,5] in that it does not require processor models or instruction set emulators. We believe our view of peer execution of the domains is unique [6]. The range of modeling in the software domain in our peer-based execution model is not restricted and can accommodate data-intensive modeling, and multithreaded shared memory communications in contrast to [7].

The fundamental physical modeling difference between the domains is that hardware is an unbounded resource model of computation which is described using a fixed amount of state while software as an unbounded state model executing on a bounded resource. This observation motivates the types of physical systems that can be described by modeling in each domain and forms the basis a given behavior or set of behaviors in each domain as dual physical system representations.

This paper begins with an examination of the physical state and computation resources inferred by the models in each domain. The physical resource models provide the basis for establishing a peer-based executable co-specification modeled on interacting computational resources from both domains. Next, we describe the common models of physical resources currently available in language models in each domain, which forms the basis of language-based overlap. Then we consider some situations which are more suitable for modeling in one domain or the other and which illustrate the necessity to capture the full range of modeling currently available in computer systems in separate, co-executing languages. We conclude

with results from our peer-based executable co-simulator, which utilizes a shared memory unified multithreading model of computation.

2. State Resources

Any model for comprehensive system level design must differentiate system state as hardware state, software state, or shared state in a single execution model. The state in a software program is modeled by program memory, while the state in a hardware system is contained in registers and on wires interconnecting non-zero delay combinational elements. Since our goal is to allow for both hardware and software descriptions to execute within a simulation, this section will contrast these different models of memory with an eye toward understanding their commonality.

Software state is theoretically infinite as described by a Turing Machine (TM) model of computation. Pointers, dynamic memory allocation (and de-allocation), and nested function calls that save state on a stack are all drawn from a TM model of computation in which memory size is not considered to be finite in the specification of the computation. Additionally, multithreaded software descriptions allow the dynamic creation of threads in an unbounded fashion.

In Figure 1, the rectangular boxes represent state in either the hardware or software domain and the arrows illustrate updating of new state. The "unboundedness" or dynamic size to system state is indicated with dashes at the bottom of Figure 1. (Since all computer systems are physically finite, unbounded TM models of system state requires software computation specifications to be "well behaved" in not exceeding the true limits of the system.)

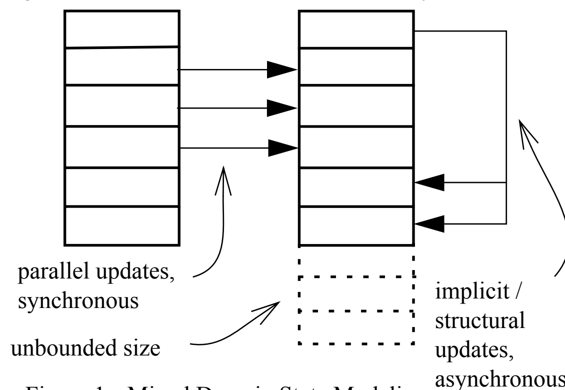

Figure 1 Mixed Domain State Modeling

Hardware state is completely specified in the hardware description using finite state machine (FSM) models. Since it is finite, it neither grows nor shrinks as a result of the execution. Wires which interconnect non-zero delay combinational blocks imply storage because of the physical delay property of a hardware description. As such, the boxes of Figure 1 also represent state on wires.

The arrows in the figure illustrates two methods of state update. Hardware implicitly updates state based on combinational logic and wire models as illustrated by the arrows on the right-hand side of the figure. Parallel updates, illustrated on the left-hand side, synchronize to a clock. These are a result of hardware's structural model of computational resources which contrast to software's explicit asynchronous updates. Software architects have tried to apply structural terminology in the design of software components [8,9]. However, software components are a software reuse technology and, in contrast to hardware models, do not bring parallelism or resource models to a software description.

In a hardware simulation, all state required to support the language semantic of the hardware description and model the state of the hardware system is in the memory of the computer platform executing the simulation. While there is a notion of global state update on a synchronous clock edge in a hardware description, there is no notion of global state addressability in hardware. A software model of computation infers access to a global unbounded addressable memory resource but without global state update. A mixed executable co-specification that utilizes a hardware simulation affords the possibility of considering all of the mixed hardware and software system state as addressable in a common, shared address space with globally synchronous updates.

The simulation aspect of physical state modeling is an important distinction in differentiating the modeling domains. Control over simulation time allows multiple memory locations to be updated in the same virtual time. By coordinating the interaction of dynamic, addressable memory access with simulated, synchronous access, it is possible to reason about all of the state in a peer-based executable co-specification as representing system state, with differentiation of hardware, software and shared state implied by the means of accessing the state resource.

3. Computation Resources

When a software program executes on a CPU, the program executes on a resource. The addition of lines of code to the program (i.e., added complexity) does not add computational resources to the system. Rather, we say that the software configures (through programming) a given resource to execute the specified function.

Multithreading is a shared-memory model of concurrency allowing a computation to be described in a concurrent fashion. The addition of software threads to an already-threaded program does not add computational resource to the system. While a software thread may allow for parallel execution if resources are available, the software multithreading model is one of allowing shared access to a limited resource, whether the resource is a CPU or shared system state.

In contrast, behavior in the hardware domain specifies the resources required to carry out the computation. Behavior in a hardware language is specified in threads — e.g., an always block in Verilog. Adding a thread of behavior to a hardware description adds the resources to the system necessary to carry out the computation. That is, a new component or module appears; the resources are determined by the specification. Hardware resources do not shrink or grow as a result of the execution of a computation. However, there is no conceptual limit to the amount of resources that can be added to a system

by adding hardware threads. The hardware model is an unbounded resource model at the time of specification.

Software state is only advanced explicitly in single addressable memory locations as each assignment occurs in a software computation. By contrast, hardware state advances in synchrony with a global synchronizing signal, the clock edge. Specifying resources to carry out a computation allows for atomic, parallel updates in the hardware domain. Since multiple registers may be loaded on any clock edge, hardware updates are atomic within the assumptions of global synchrony. Multithreaded software does not have a notion of edge synchrony, although the appearance of atomicity needed for critical sections can be generated through mutual exclusion primitives — an asynchronous approach to synchronization. The notion that software advances concurrent state atomically using asynchronous means, and that hardware advances concurrent state atomically through a global synchronization edge is an effect of the way each domain models computation and state resources.

In addition to explicit updates of state through register loading, hardware state changes have implicit effects on other state. Such implicit changes in hardware state are asynchronous state advancements on wires and combinational logic that are not directly synchronized to a clock signal (although they react to changes in state that are synchronized). Asynchronous hardware updates may also be atomic, which is consistent with hardware as a structural model of computation that allows for specification of unbounded computation resources.

Software threads may be mapped to available resources in many-to-one, many-to-many, or one-to-one fashion. In all cases, software threads may be eligible to execute, but be resource starved. By contrast, hardware threads are always one-to-one mapped to the resources their behavior describes. Since the behavior of hardware models includes both computation resource and state resource, hardware threads are always eligible to execute and always active. The contrast in the activation of hardware and software threads is an effect of the model of a thread in software as bound by the availability of resources it may ultimately execute on, while hardware threads co-specify computational behavior with the resources required to carry out the behavior.

The unbounded specification of concurrent hardware resources requires a simulator to model execution. Through simulation, virtual time is controlled and the resources appear to execute concurrently regardless of the underlying hardware to execute the simulation. Software, with its fixed/bounded resource model, is not simulated. The interaction of simulated and non-simulated computation resources requires a scheduler that can interleave simulated execution with execution that is conceptually executing on the resources provided by a host computer. To be truly simulated, resources for the software to execute on must be specified (i.e., the number of processors for the threads and the mapping approaches). Whether the resources for software to execute on are conceptual or specified, the interleaving of behavior is based upon the interaction of physical FSM resources implied by the mixed domain computation models as described in the next section.

4. Common Models

The physical models that hardware and software represent may best be understood by the conceptual models from each domain and their overlap. Overlap between computation models of a multithreaded TM software description and a structural hardware description is shown in Figure 2. The first model in the overlap is the FSM itself. FSM models of system state are available in each domain by requiring bounded state size in software. The second overlapping model, semantic FSMs, is used to model behavior which does not imply a finite state machine. Rather the FSM is an artifact of modeling in a language domain. The third overlapping model is FSM-implied connectivity, utilized by each domain to allow threads to communicate. These will be described in terms of the means of specifying state in each of the domains.

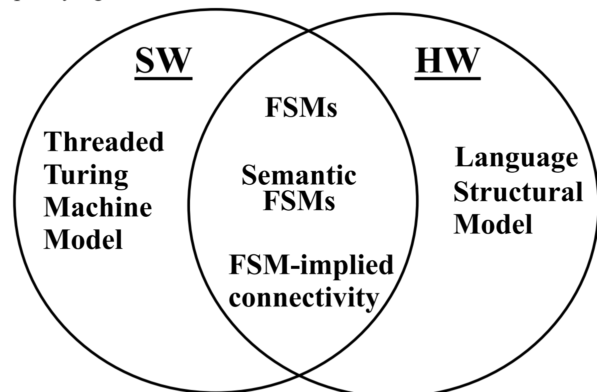

Figure 2 Computation Models for State Advancement

In the hardware domain, FSMs are specified behaviorally. Latches are inferred in non-combinational hardware descriptions and imply state that is updated asynchronously. Synchronous FSM behavior is implied by hardware state that is updated by sensitivity to a clock edge. When state is implied in a hardware description, a resource is added to the system.

Software programs physically represent one FSM in a system, while multiple FSMs are implied by a multithreaded software program. However, the actual number of FSMs in the physical system is determined by the mapping of threads to resources.

State for an FSM falls into three common categories as illustrated in Figure 3. Static state is persistent through execution of a system but is limited in visibility. Local state is limited in visibility, is not persistent, and is unbounded in the software domain. Global state is persistent and not limited in visibility. These are used to produce the three FSMs of Figure 2 that are common between the domains.

Each domain has different means of specifying system state that is advanced by FSMs. Hardware descriptions imply the addition of the static state through analysis of the behavioral model, as illustrated on the first line of Figure 3. State is explicitly declared in software.

Local state in hardware and software supports descriptions of computational behavior that are FSM-like in appearance, but not in effect. An example of these *semantic FSM* models are the combinational Verilog always blocks used for synthesizing

	HW	SW
static	implied behaviorally	declared static
local	bounded	stack/unbounded
global	hierarchical name	declared public

Figure 3 Mixed Models of State

combinational behavior. The local variables used in a software language function to model these are typically allocated to the stack, allowing for unbounded levels of computation nesting to take place. By contrast, local state in hardware functions and tasks have bounded system state. There are no hardware stacks for dynamic state; hardware does not follow the TM model.

The hardware description appears as an FSM because the always block is considered to contain sequential statements. However, in effect it is not; it contains no persistent state and thus synthesizes to combinational behavior. Functions in software are modeled using an FSM style of computation, including the declaration of local variables upon which the computation may be carried out. If a software function is stateless (i.e., there is no state stored in statics or globals), then the software function is also a semantic FSM. Thus both languages contain the notion of a semantic FSM.

Both domains have some notion of global state access. While hardware supports global access via hierarchical naming conventions, software allows for global access by any scope simply by variable name. Software state is modeled by name, where each name resolves to a unique location in memory. Because of the unbounded memory model in software, the physical memory location in which a named variable resides may actually be shared with other named state. This is most commonly seen when local state is stored on a stack or virtual memory is included in the system. But the essence of the naming convention in software modeling is one of addressability. Global variables in software are shared by all scope, and so are commonly used as a means of communicating between threads.

Hardware state is not globally addressable in the same way software state is. The hierarchical naming convention in hardware allows a simulation to access variables which may be outside the scope of a given module. Hardware globals, therefore, are a simulation convenience. Because hardware models unbounded physical copies of behavior and state required for the computation, hardware state is encapsulated with behavior.

The last common computation model of Figure 2 is one where FSM behavior is implied for the purpose of inter-thread communications. This "FSM-implied connectivity" is common to both domains. In the hardware domain, when a thread is waiting for a certain condition to be met (i.e., a level) the semantics of the wait implies an FSM state to carry out the wait. The notion of a thread waiting for a condition to become true is the only common inter-thread communication model between the domains. In software, a single scheduler FSM is implied to carry out the specified behavior. In hardware, a unique FSM state is implied for each wait condition.

Thus, both models of computation contain FSMs which allow for asynchronous advancement of system state in their internal models. Inter-thread advancement of system state in the software domain requires explicit activation of a scheduling mechanism which allows for other threads to execute.

Because software is not modeled as a resource, global inter-thread synchrony is not possible in the software domain. By contrast, inter-thread advancement of system state in the hardware domain is activated implicitly by updates to system state variables modeled as wires. Because of hardware as a computational resource model, true parallelism and global synchrony is possible in the hardware domain. Thus software activates threads explicitly and asynchronously, while hardware implicitly activates threads globally and synchronously.

The common FSM models of computation allow for the interaction of behaviors on the basis of the domain in which they are modeled. Thus a computer system is modeled as a collection of interacting behaviors, with overall performance and system level behavior defined by the domains in which the behaviors are described. This is the basis of our integrated dual model of computation.

5. Mixed System Modeling

Comprehensive computer system modeling requires the modeling situations in Table 1 to be available and to co-execute. The two columns are labeled as hardware and software models and contain modeling situations not readily available in the other domain because of the physical resource modeling implied by each domain. The complementary nature of the modeling situations underlies our choice to include both C and Verilog in our modeling capability.

Table 1: Complementary H/S Modeling

HW, Unbounded Structural Resource Model	SW, Unbounded Memory Model
co-specified behavior/resources	dynamic, addressable, de-referencable memory
synchronous global state update	global state addressability
synchronous state advancement	time independence
implicit state updates via modular interconnect	explicit state updates via compiled library function calls
time-based simulation	interactive, operational input
fan-out/fan-in	unbounded buffers, mutexes, semaphores
threads always eligible for execution	dynamic threads
feedback	recursion

Consider the calculation of the factorial of a value, n, as part of a larger computer system behavior. The behavior might be considered for implementation in the software domain using recursion or in the hardware domain using structural feedback.

A recursive implementation of a stateless (semantic FSM) factorial function in the software domain is shown below. The software recursive factorial requires a dynamic stack for local calling state. The stack is an inferred physical model required to carry out the behavior of the recursive software model.

```
int factorial (int n){
    if (n == 0) return 1;
    else return (n * factorial(n - 1));}
```

Not having unbounded dynamic memory, a hardware domain model uses structural feedback as shown in Figure 4 to implement the factorial. The hardware behavior is written in a manner that implies the feedback state.

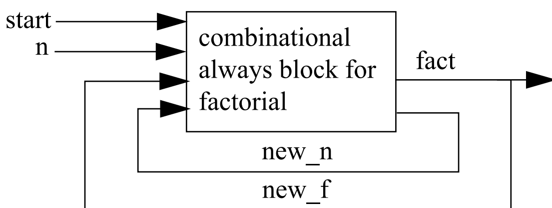

Figure 4 Factorial in Hardware Using Structural Feedback

In the recursive model from the software domain, the state required to carry out the overall factorial calculation is saved on the stack, while in the hardware domain the state required to carry out the calculation is implied into registers. In both cases, the factorial behavior being modeled is conceptually stateless inside the function and the always block. The state required to carry out the computation is inferred by a larger context in each case. The dual models of the hardware and software implementations infer very different physical models and so have very different implications on the behavior of the overall system in which they appear.

The list in the table and the factorial example illustrates the necessity of bringing together all of the modeling situations described in a more seamless system view using a dual modeling approach. Comprehensive system modeling requires the ability to consider the implementation of a given behavior within a larger system context — a context which is inclusive of all of the physical means of implementing the behavior. Comprehensive system modeling also requires the ability to model all of the means of implementing a given behavior that are currently possible, with the ability to seamless integrate all of the physical models into a single execution model.

6. Examples

We are developing a peer-based executable hardware/software co-specification based upon a multithreaded shared memory model. The cosimulator allows the domains to co-execute as peers in the advancement of mixed system state. The cosimulator allows for the TM and structural models of Figure 2 to be preserved in separate modeling domains, while it couples the domains by resolving common notions of system state as shown in Figure 3. This will eventually allow all the modeling situations of Table 1.

There is no restriction on the modeling possible in each domain. The software component of a co-simulation is comprised of a user defined multithreaded C program written using function calls to the Solaris Threads Library. All features of the threads library are available to the user. We are developing our own hardware scheduler to work with the Solaris thread scheduler and our cosimulation scheduler and will eventually develop a mixed domain cosimulation scheduler for both domains. These are implemented in our own hardware scheduler. Since we are focusing on the demonstration of the integration of concurrent hardware and software models, all models are currently specified in stylized C-syntax. Eventually, we will develop Verilog and C parsers to generate the inter-domain communication primitives.

The dining philosophers problem [10] illustrates the peer execution of our co-simulator. Mixed-domain modeling has philosophers in both hardware and software and chopsticks in hardware. Figure 5 shows the allocation of resources between the hardware and software domains. Mutual exclusion is through multi-phased clocks in hardware and mutexes in software. All of the software philosophers share one clock in the hardware simulation, VCLK in Figure 5, to avoid contention with hardware philosophers. 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.

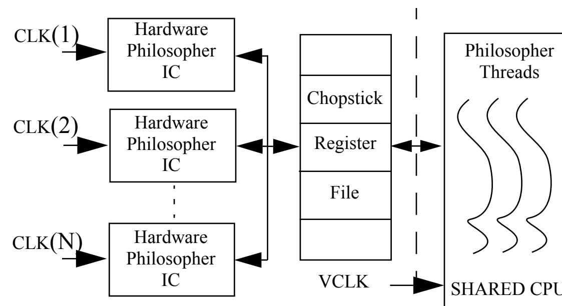

Figure 5 Hardware/Software Resource Partitions

Philosophers in both domains have identical behavior and, for the purposes of illustration, require the same amount of simulated time to execute. Figure 6 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. 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 multi-

threaded model to a hardware model effectively increases the resources and thus the true parallelism modeled by the system.

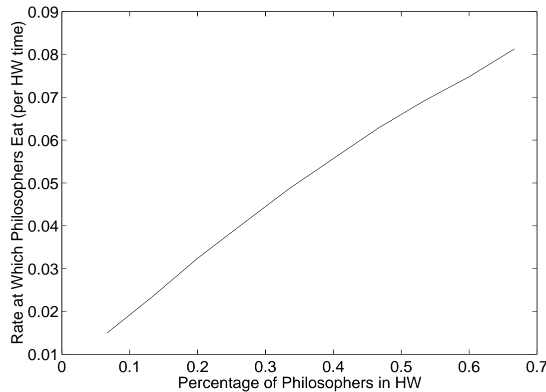

Figure 6 Performance Benefits of Resource Allocation

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. Thus, the mixed domain testbenches are peers along with the mixed domain system models. 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 that controls eating and thinking times. The testbenches read a new pair each time one of their philosophers eat.

Table 2: Dining Philosophers Testbench

HW TestBench (Eat Time, Think Time) in Clock Cycles	SW TestBench (Eat Time, Think Time) in HW Time	Real Time (sec)	Number of Times SW Eats	Number of Times HW Eats
(1,2)	(20,40)	0.74	74.3	78.0
(2,4)	(20,40)	0.72	69.3	69.5
(4,8)	(20,40)	0.76	63.7	39.5
(1,2)	(40,80)	0.67	57.0	82.0
(1,2) (2,4)	(20,40)(40,80) (80,160)(160,320)	0.68	38.0	65.5

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 truly concurrent threads are a completely foreign concept in software multithreading. Figure 7 shows the results of dynamically creating and destroying software philosophers. One philosopher is destroyed every 25000 HW time units for two iterations, then one is created every 25000 HW time units for two iterations, etc. The graph shows the rate of food consumption by the philosophers over time. As philosophers are destroyed, food consumption decreases and as they are created it increases.

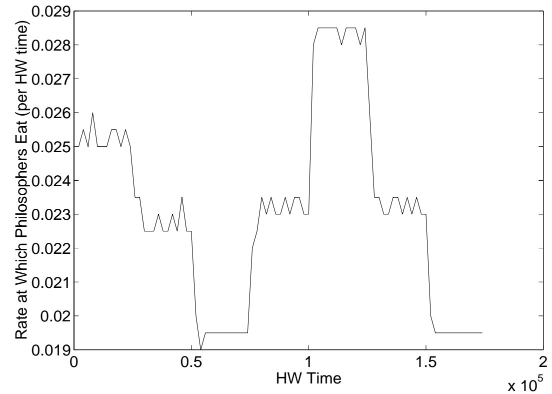

Figure 7 Rate at Which Philosophers Eat per Time

7. Summary

Peer-based executable co-specification of mixed hardware/software modeling of computer systems has been described in terms of the physical models inferred by each domain. A language based approach for specifying common behavioral models permits an analogy of the hardware and software domains to physical dual models. The range of computer system modeling currently available illustrates the necessity to preserve separate language-based domains but also motivates the need for the domains to co-execute. Comprehensive computer system modeling is viewed as a collection of interacting behaviors with domain specific inferences that impact the system composition. Examples of our cosimulator have been described along with results that illustrate how the hardware and software domains co-execute on a peer basis.

Acknowledgment

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

References

[1] B Lewis and D.J. Berg, *Multithreaded Programming with Pthreads,* Mountain View: Sun Microsystems Press, 1998.

[2] D. E. Thomas and P. R. Moorby, *The Verilog Hardware Description Language, 4th Edition*, Boston: Kluwer Academic Press, '98.

[3] J.A. Rowson, "Hardware/Software Co-Simulation," *Proc. of 31st DAC*, 1994, 439-440.

[4] S. L. Coumeri and D. E. Thomas, "A Simulation Environment for Hardware-Software Codesign," in *Proc. ICCD '95*, 1995, 58-63.

[5] J. Wilson, "Hardware/Software Selected Cycle Solution," *Proc. Int. Workshop on Hardware-Software Codesign*, 1994, 190-194.

[6] S. Edwards, L. Lavagno, E. Lee, A. Sangiovanni-Vincentelli, "Design of Embedded Systems: Formal Models, Validation, and Synthesis," *Proc. of the IEEE* 85:3 (March, 1997) 366-390.

[7] C. Passerone, L. Lavagno, M. Chiodo, and A. Sangiovanni-Vincentelli, "Fast hardware/software co-simulation for virtual prototyping and trade-off analysis," *Proc. of 34th DAC,* 1997, 389-394.

[8] C. Szyperski, *Component Software*, Essex: Addison-Wesley, '98.

[9] M. Shaw, R. DeLine, D. Klein, T. Ross, and D. Young, "Abstractions for Software Architecture and Tools to Support Them," *IEEE Trans. on Software Engr.*, 21:4 (April, 1995), 314-335.

[10] M. Ben-Ari, *Principles of Concurrent Programming*, Cornwall: Hartnolls Limited, 1982.