Communication and Co-Simulation Infrastructure for Heterogeneous System Integration

Guang Yang[1], Xi Chen[2], Felice Balarin[3], Harry Hsieh[2], Alberto Sangiovanni-Vincentelli[1]
[1] University of California, Berkeley, CA 94720, USA
[2] University of California, Riverside, CA 92521, USA
[3] Cadence Berkeley Laboratories, Berkeley, CA 94704, USA

Abstract

With the increasing complexity and heterogeneity of embedded electronic systems, a unified design methodology at higher levels of abstraction becomes a necessity. Meanwhile, it is also important to incorporate the current design practice emphasizing IP reuse at various abstraction levels. However, the abstraction gap prohibits easy communication and synchronization in IP integration and co-simulation. In this paper, we present a communication infrastructure for an integrated design framework that enables co-design and co-simulation of heterogeneous design components specified at different abstraction levels and in different languages. The core of the approach is to abstract different communication interfaces or protocols to a common high level communication semantics. Designers only need to specify the interfaces of the design components using extended regular expressions; communication adapters can then be automatically generated for the co-simulation or other co-design and co-verification purposes.

1. Introduction

As the complexity of electronic systems keeps increasing, people are seeking design methodologies with higher productivity. The system level design methodology, based on orthogonalization of design concerns, as well as pre-defined platforms, has therefore been proposed for the next major productivity gain [8]. Meanwhile, there is a wide consensus that intellectual property (IP) reuse is the key to most potential solutions. No design team or even company can still afford developing from scratch everything in an entire system under the short time-to-market. Because of this trend, there is a huge emerging demand of IPs with different functionalities and at various abstraction levels. IPs at different abstraction levels are usually specified in different languages, for instance, C/C++ and Simulink are used for generic algorithms; SystemC for behavior level and transaction level IPs; Verilog/VHDL for register transfer level and gate level IPs.

Although there are many IPs available on the market, designing a system by IP integration is still a challenging task. The main difficulty comes from two aspects:

- Unable to easily integrate IPs at different abstraction levels. The communication between IPs is not well-defined or has no obvious correspondence. Sometimes, even in the same abstraction level, the granularity of the communication still does not match.
- Unable to verify the system efficiently. In particular, there is not a systematic and integrated design environment to handle co-simulation of IPs across multiple abstraction levels and specification languages.

To solve these problems, we propose a generic IP integration infrastructure with event representation of services and communication adaptation based on regular expressions. The communication interfaces at different levels of abstraction are specified with regular expressions, and are abstracted (adapted) to a common semantics level. IP blocks or design components, regardless of their abstraction levels or specification languages, can then be integrated together and co-simulated as long as their interface specifications are available. In order to abstract different interface specifications to the common formal communication semantics, communication adapters are automatically generated for the integration, co-simulation, or other co-design and co-verification purposes.

This work is based on Metropolis [5] design framework and methodology, which aim to provide an integrated co-design environment that supports heterogeneous IPs from different IP providers, at different levels of abstraction, and specified in different languages. One of the most important features of Metropolis that greatly enhances the ability to explore design space is function-architecture mapping. The function of a system describes what needs to be implemented. The architecture specifies what can be implemented. They are specified and refined separately at a high abstraction level, and are eventually mapped together by synchronizing the functional components to the architectural components, which means if only one of the two synchronized

3-9810801-0-6/DATE06 © 2006 EDAA

events can be scheduled to occur, the process containing the event has to be blocked until the other event occurs also. In the rest of this paper, we will present how we support communication and mapping between heterogenous design components in a unified design framework. Another important feature in Metropolis is the ability to apply declarative constraints at the system level upon design components and their events. Designers can choose imperative programs, declarative constraints, or both due to convenience to specify behaviors. The mixture of them is regarded as the entire specification. Once a unified interfacing infrastructure is available, they can be easily applied to heterogenous design components.

The rest of the paper is organized as follows. Section 2 discusses related work. In Section 3, we describe the formal communication semantics. In Section 4, we propose a communication and co-simulation infrastructure for system level modeling with IP integration. Then, we demonstrate the effectiveness of our approach with a real world design examples in Section 5. Section 6 concludes the paper.

2. Related Work

The MILAN project [4, 10] employs a model-based solution for hardware/software co-design and co-simulation. Different simulators can be integrated once different simulation models are interpreted into a common model supported in MILAN. Our approach mainly focuses on unifying communication semantics between models at different levels of abstraction. Much work has been done to solve communication gaps for hardware/software co-simulation, mostly focused on the register transfer level such as ISS and HDL. The existing solutions are usually targeting some particular design languages or simulators, and the connections across different platforms are usually done manually [3, 14, 9]. In this work, our focus is a generic co-design framework for heterogeneous design components regardless if they are software, hardware, or both.

In [6], communication adapters between the transaction level and register transfer level, which are called transactors, can be automatically generated from interface specification in regular expressions. This work is mainly targeting SystemC and Verilog co-simulation within a single environment such as NCSim or ModelSim. Using regular expressions to specify IP interfaces for the purpose of generating simulation monitors has been studied and presented in [12]. Protocol Compiler [13] is a design environment for designing and generating controllers in HDLs from a graphical specification of communication protocols. Our work is targeting a generic design framework that allows high level modeling with integration of heterogenous design components written in almost any languages.

CORBA [2] provides a middle communication layer that enables interoperability between objects from different operating systems, programming languages, and networks. CORBA is mainly software oriented and is powerful enough to take care of almost everything between software objects and underlying operating systems, so it might not be efficient to use directly in the embedded system co-design. Therefore, we use a more generic communication mechanism, inter-process communication (IPC), in our experiment.

3. Communication Semantics Formalism

In different abstraction levels and in different programming languages, the semantics of communication differ very much. e.g. in SystemC transaction level models, communication is done via port-interface calls; in Verilog RTL models, signals can be sent across modules to achieve communication. To unify them, we need to build a common semantics domain among all abstraction levels and applicable to all programming languages. We chose tagged signal model(TSM)[11]. In TSM, an event is a member of $T \times V$, where T is a set of tags and V is a set of values. In this paper, we take T as time and V as $action \times process$, where an *action* identifies a location in the description, e.g. a line of code; a *process* indicates the active entity that executes the *action*[5]. Depending on abstraction levels, T means differently, such as an event ordering at the behavior level, or the exact timing at the register transfer level. Regardless of the abstraction levels, a model always generates a sequence of events. In Metropolis, we abstract a sequence of events to a pair of representative beginning and end events, which is defined as a service. Following this idea, we can always transform a communication semantics into a Metropolis service. The communication between two design components can then be defined as one using services provided by the other.

A design component provides services through its input ports and utilizes services through its output ports. Each port is associated with a service. To clearly demonstrate the communication semantics in terms of services, see Figure 1 and Table 1. In Figure 1, *ob* and *oe* represent a service associated with the output port; *ib* and *ie* represent a service associated with the input port. For input ports, there are two kinds of services defined, *active* and *passive*. An *active* input service runs in its own thread. The calling of the active service requires the synchronization between the output service and the input service. This is one of the key concepts, function-architecture mapping, embodied in Metropolis design methodology[5]. A *passive* input service can only be initiated by an output service. Both the input and output services run in the thread of the output service. Note that for output ports, services are always active. The last row in Table 1 captures the timing relation, which is the key of the formal communication semantics. Intuitively, it says that active input service and active output service should execute simultaneously. While passive input service should be invoked by active output service and upon finishing, return to active output service.

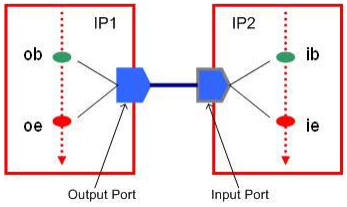

Figure 1. Service Example in Communication

$event =< time, action, process >$

Communication Semantics	**Active Input**	**Passive Input**
	$ib =< t_{ib}, a_{ib}, IP2 >$ $ie =< t_{ie}, a_{ie}, IP2 >$ $t_{ib} \le t_{ie}$	$ib =< t_{ib}, a_{ib}, p >$ $ie =< t_{ie}, a_{ie}, p >$ $t_{ib} \le t_{ie}$
Active Output		
$ob =< t_{ob}, a_{ob}, IP1 >$ $oe =< t_{oe}, a_{oe}, IP1 >$ $t_{ob} \le t_{oe}$	$t_{ob} = t_{ib} \le t_{oe} = t_{ie}$	$t_{ob} \le t_{ib} \le t_{ie} \le t_{oe}$ $p = IP1$

Table 1. Formal Communication Semantics

For design components at different levels of abstraction, according to the classification of their communication semantics, we can abstract their communication interfaces or protocols to a common higher level semantics with services denoted by a beginning event $e_b =< t_b, v_b >$ and an end event $e_e =< t_e, v_e >$, where $t_b \le t_e$.

1. Register Transfer Level: Usually, register transfer level and lower level design components are described in HDLs such as Verilog or VHDL. Their communication is done via signals, which correspond to physical wires and voltage transitions. There are two cases when abstracting signals into a service.

(1). One event or one signal occurring on a wire corresponds to a service. For example, a system reset service could be triggered by an asynchronous reset signal. In this case, we split the single reset event $e' =< t, v >$ into two separate events $e'_b =< t, v >$ and $e'_e =< t+\Delta, v >$, where Δ is a positive infinitesimal, i.e. infinitely small but positive real number. This is the same trick played by Verilog/VHDL simulators. In this case, it actually implies that e'_e occurs infinitesimally close but after e'_b, i.e. $t_b = t_e = t$.

(2). A set of signals work altogether to perform a communication task. This usually occurs when there is a communication protocol, such as memory access or bus transaction. During an entire communication session, the set of signals will generate and receive a sequence of events $< e_1, e_2, ..., e_n >$, whose time tags satisfy $t_1 \le t_2 \le ... \le t_n$. Up to the protocol or designer, the sequence of events can be abstracted to a service with a beginning event e_i and an end events e_j, where $1 \le i < j \le n$, and $t_i = t_b$, $t_j = t_e$.

2. Transaction Level: In a transaction level model, services are usually modeled as function calls. A function naturally generates a sequence of events $< e_1, e_2, ..., e_n >$, and is usually represented by the first and the last events in the sequence. i.e. $e_i = e_1$ and $e_j = e_n$. Therefore, the communication semantics is $t_1 = t_b$ and $t_n = t_e$.

3. Behavior Level: Behavior level often just dictate the algorithm to solve a problem without detail correspondence to low level data or timing. It is very likely that a service is just a portion of the system behavior. Designers again need to extract the beginning and end events representing the portion out of the sequence of events generated by the system. After the extraction, it is exactly like the case in transaction level.

4. Arbitrary Mixture of Abstraction Levels: The definition of services are powerful enough to capture arbitrary combinations of abstraction levels. For example, a few transactions at a lower granularity might correspond to a single transaction at a higher granularity, or a set of services can be combined to form another higher level service. The formalism is the same as in previous two cases, where the beginning and end events are extracted and used to present the new service.

4. Communication and Co-Simulation Infrastructure

4.1. Co-Simulation Flow

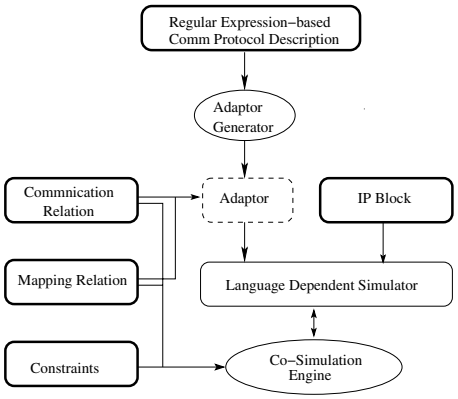

Figure 2. Co-Simulation Work Flow

The proposed co-simulation work flow is shown in Figure 2. The bold boxes are provided by designers. The regular bordered shapes are tools. The dotted boxes are generated by tools. The co-simulation infrastructure consists of an automatic generator for communication adapters and the co-simulation engine in charge of event synchronization between design components. Designers specify the communication protocols using regular expressions. IP blocks and the automatically generated adapters form the new IPs then can communicate at the service level. The service level communication relations (connections) between IP blocks are also provided by designers. In addition to the regular communication, mapping relations and declarative constraints are taken into account all together at simulation stage by the co-simulation engine.

4.2. Specification

In the specification of the communication interfaces at different levels of abstraction, there are four issues that we need to solve: event order, data mapping, service mapping and event generation.

4.2.1. Event Definition There exists a standard definition of a Metropolis event, which is a three tuple $e :< time, action, process >$. In order to capture explicit data mapping, we add additional events, i.e. $e_m : D_c \leftrightarrow D_s$, where D_c is the datum/variable in the communication protocol being abstracted and D_s is the variable in the scope of beginning or end event of a service (The scope of an event includes all variables visible at the action of the event.) This definition captures the data correspondence between the communication protocol and the service.

Generally speaking, the above definitions work for all imperative languages, such as C, C++, SystemC, and HDLs. However, if we look at the native communication mechanisms in these languages, we can tune the definition to make it better fit in the languages. For example, C and C++ communicates via function calls, which are consistent with the service definition. SystemC communicates through transaction ports, which are similar to function calls. HDLs communicate via hardware signals. Events flowing on signals have not only a time stamp, but also the new value of the signal. We can extend the event definition for this case and let it carry the value as well, i.e. $e(v)$, where $v \in \{0, 1, DC\}$. We use DC and $*$ interchangeably to denote a don't-care value. Similarly, for a multiple-bit signal, any event occurred in individual bit is considered an event for the entire signal.

4.2.2. Event Order When abstracting an event or a set of events to a service, especially when there exists a complex communication protocol, the event order needs to be specified. This is the template to identify the execution and invocation of the service. Therefore, it can be correctly relayed to the corresponding design component. To specify the event order, we chose to use regular expressions. First, it is the most well-known technique for specifying sequences, which most designers feel comfortable with. Second, it can be extended easily to have strong enough expressive power to specify most communication interfaces at various abstraction levels[1][7][12].

The alphabet of the regular expression is $\Sigma = \{\mathbf{B}, \mathbf{E}, e(v), e_m\}$, where **B** and **E** are two special events indicating beginning and end events of a service respectively, $e(v)$ are events on all signals, and e_m are the data mapping between a communication protocol and a service. Designers can specify the event ordering in the communication protocol using $e(v)$. For example, in the simplest hand-shaking protocol, we have *req*, *ack* and *go* signals. Suppose we use value one to denote the low-to-high event of the signals. Then we can use the expression $\{req(1), ack(1), go(1)\}$ to specify the hand-shaking protocol.

4.2.3. Service Mapping Having the event ordering, we need to decide how the event sequence should correspond to a service represented by a beginning and an end events. Designers could insert **B** into the regular expression where they think the service should begin once the prefix event sequence has been detected. Similarly, **E** is inserted as the end of the service. For the hand-shaking example, it could be $\{req(1), ack(1), \mathbf{B}, go(1), \mathbf{E}\}$.

4.2.4. Data Mapping Another very important ingredient of a service is data. Let's modify the hand-shaking protocol and let it send a datum *out* and receive a processed datum *in* after the *ready* event occurs. On the service side, there are corresponding variables *input* and *output*. If we look at the protocol as a function call, *out* is the function argument and *in* is the return value. The communication protocol now becomes $\{req(1), ack(1), \mathbf{B}, out \leftrightarrow input, ready(1), in \leftrightarrow output, \mathbf{E}\}$. Note that naturally the data transfer for a service should happen right after event **B** and right before event **E**, but this is not a hard requirement. Designers can insert data mapping anywhere needed in the regular expression.

4.2.5. Event Generation The semantics of services we are proposing is at a virtual level, which could sweep from below register transfer level to above behavior level. In reality, it is often the case that when we move from a lower abstraction level to a higher abstraction level, we lose information, e.g. timing. Conversely, when going from a higher abstraction level to a lower abstraction level, we need to synthesize with additional information. In bridging the communication across IPs, the adapter receives all the input events and detect event sequence due to regular expressions. At the same time, adapters will generate output events (including data mapping) at the 'right' time. If the design is not sensitive to the exact timing but to the event ordering, we can depend on the regular expression to generate output events at the correct points. Otherwise, if the timing is crucial, we can augment the regular expression to include that information as well.

4.3. Discussion on Expressiveness

In this paper, it is not our primary goal to extend regular expression such that it is expressive enough to describe any communication protocols. In fact, there are some existing work to do that. For instance, in [12], two extensions are found helpful to express protocols with state storage and pipelining. PSL also extends the regular expression in a similar way plus the more versatile repetition expressions. Any of them could be chosen to specify communication protocols based on their expressiveness and convenience. Our infras-

tructure is not biasing one over another and could accommodate all.

4.4. Implementation

The most complicated step in the implementation of the co-simulation infrastructure is the generation of adapters. The input of the generator includes the four pieces of information described in Section 4.2. From a regular expression, we can generate an equivalent finite automaton. The algorithm to do that can be found in any introductory computing theory books. In the generated finite automaton, a state represents a particular prefix matching; edges are labeled with events. If the events are inputs to the adapter, on observing such events, the automaton transfers from the current state to the next state; if the events are data mappings or need to be generated, the adapter will do so and transfer to the new state. Among all the states, an initial state is always the place from which event matching starts; accepting states indicate the successful matching of regular expressions. In later co-simulation, the finite automata perform the adaptation of communication protocols by detecting event sequences, passing mapped data and generating events as needed.

Because IP blocks can be specified in any design languages and at any abstraction level, each communication adapter associated with an IP block needs to include two parts, a language dependent part that can directly communicate with the IP block and a language independent part that has a common service level interface and can be synchronized by the co-simulation engine. Since the generated finite automaton needs to handle signals from the original IP block, we put it in the language dependent part of the adapter. In our experiment, we have implemented our co-simulation engine using standard C++, so is the language independent part of an adapter. Between co-simulation engine and adapters, the communication is implemented with UNIX Inter-Process Communication (IPC) library. For example, in the case of Verilog, the automaton is generated in Verilog; Programming Language Interface (PLI) is used to access the IPC at the operating system level to communicate with co-simulation engine. VHDL, Matlab, and Metropolis all have standard interfaces that can talk with the operating system. We believe that there is no technical difficulty in using any other modern languages or platforms for implementation.

5. Case Study

We use a JPEG encoder design to illustrate the effectiveness of our approach for the communication and co-simulation infrastructure. In Figure 3, the upper portion shows the high level functional model of a JPEG encoder consisting of discrete cosine transform (DCT), Quantization and Huffman encoder. The lower portion is an abstract dual-processor architecture. Each micro-processor has

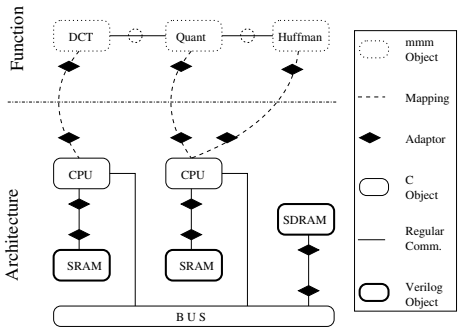

Figure 3. *JPEG Encoder Block Diagram*

a local SRAM module. They also connect to a bus to communicate with each other through another SDRAM memory module. Figure 4 shows the abstraction level and design language for each block.

In this case study, we apply a particular mapping between the function and architecture. Because the architecture has two processors, we partition the function into two stages and let each stage run on a separate processor. This way we achieve a two-stage pipelined JPEG encoder. We group the Quantization and Huffman encoder together and let them share one processor. The DCT block uses the other. Note that mapping is a very flexible and efficient design exploration step in our system level design methodology. There are mechanisms in Metropolis to map two separate services onto a single service. Designers can even specify the scheduling of the two services using either imperative schedulers or declarative constraints. Then, the co-simulation engine is able to take the scheduling into account. Since these are not the focus of this paper, we will not discuss them in details.

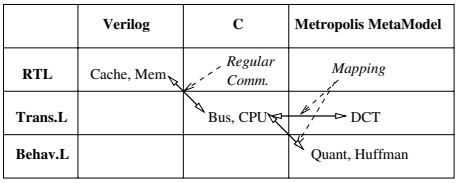

Figure 4. *Design Component Classification*

In this example, there exist multiple abstraction levels, multiple design languages, both regular communications and mappings. In order to make design components talk across abstraction levels, adaptors are generated based on the communication protocol description and inserted between abstraction levels. Figure 5 shows the SRAM timing diagram for read operation. The following regular expression captures this protocol, which can be used to adapt to a read service with *addr* as the address argument and *data* as the memory data.
$\{\mathbf{B}, data\_read(1), m\_addr \leftrightarrow addr, clk(1), clk(1)^+, data\_ready(1), clk(1), m\_data \leftrightarrow data, clk(0), data\_read(0), m\_addr \leftrightarrow Z, clk(1), \mathbf{E}\}$

The generated automaton is shown in Figure 6. The events

on the edges without boxes will be observed by the adapter. As soon as an event is observed, the automaton transfers to the next state. If there are events in boxes on outgoing edges from a state, the automaton takes that transition immediately and generate the events accordingly. This could result in either generating new regular events or passing mapped data. The automaton is realized by a standard Verilog FSM. The code size is linear in the number of states in the automaton. Memory write service, and SDRAM read/write services are similar to the SRAM read service. For the other kind of communication, mapping between function blocks and CPUs, the adaptation is much easier than memory read/write services. This is because the CPUs are written in transaction levels. Their operations are abstracted with read, write and execute services. On the function blocks side, we also extract the same set of operations by chopping the behaviors into corresponding pieces. This way the mapping relation becomes one-to-one.

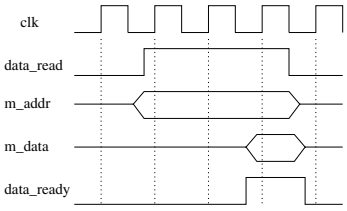

Figure 5. *SRAM Read Timing Diagram*

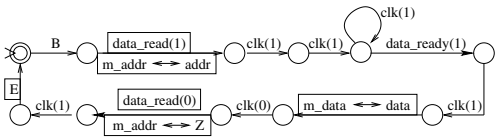

Figure 6. *SRAM Read Adaptor Automaton*

Upon finishing the communication adaptation, we run co-simulation on the function-architecture model. The simulation outputs the correct JPEG image converted from a raw image. The CPU running DCT takes 208216 cycles; however, the other CPU running both Quantization and Huffman takes only 34736 cycles. The numbers show the imbalance of the two pipeline stages, which suggests a better function partition, e.g. move more work over from DCT to Quantization/Huffman. Based on this information, we go down into details of DCT, which consists of smaller blocks of Preprocess, DCT1, Transpose1, DCT2 and Transpose2. We then re-map DCT2 and Transpose2 to the other CPU. The new simulation result justifies the new mapping, where the CPU running Pre-process, DCT1 and Transpose1 takes 102699 cycles; the other CPU takes 133892 cycles. This kind of exploration is exactly what we want to achieve by co-simulation across abstraction levels and languages. One measurement we planned to do is to analyze the simulation overhead on adapters. Due to the simple automaton-based mechanism, we believe the overhead should be small.

6. Conclusions

In this paper, we has presented a unified communication and co-simulation infrastructure for integration of heterogenous design components and IPs. Our approach is based on a standard high level communication semantics formalism that enables easy communication and synchronization between different abstraction levels. Co-simulation adapters can be automatically generated from the specification of design component interfaces. Verification is a future research direction in the topic of co-design infrastructure for heterogeneous IP blocks. We are also interested in more complex communication protocols with pipelines or conditionals.

References

[1] http://www.eda.org/vfv, 2003.

[2] CORBA homepage. http://www.corba.org, 2005.

[3] A. Amory and et al. A heterogeneous and distributed co-simulation environment. In *Proceedings of the 15 th Symposium on Integrated Circuits and Systems Design*, 2002.

[4] A. Bakshi, V. Prasanna, and A. Ledeczi. MILAN: A model based integrated simulation framework for design of embedded systems. In *Proceedings of Workshop on Languages, Compilers, and Tools for Embedded Systems*, June 2001.

[5] F. Balarin and et al. Metropolis: an Integrated Electronic System Design Environment. *IEEE Computer*, 36(4):45– 52, Apr. 2003.

[6] F. Balarin and R. Passerone. Functional Verification Methodology Based on Formal Interface Specification and Transactor Generation. *Design Automation and Test in Europe*, 2006.

[7] C. Eisner and D. Fisman. Sugar 2.0 proposal presented to the accellera formal verification technical committee. Mar. 2002.

[8] K. Keutzer and et al. System level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design*, 19(12):1523–1543, Dec. 2000.

[9] C. Kreiner, C. Steger, and R. Weiss. A hardware/software cosimulation environment for DSP applications. In *Proceedings of 25th Euromicro Conference*, 1999.

[10] A. Ledeczi and et al. Overview of the model-based integrated simulation framework. Technical Report ISIS-01-201, Vanderbilt University, Jan. 2001.

[11] A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–29, Dec 1998.

[12] M. T. Oliveira and A. J. Hu. High-level specification and automatic generation of IP interface monitors. In *Proceedings of the 39th Design Automation Conference*, 2002.

[13] A. Seawright and et al. A system for compiling and debugging structured data processing controllers. In *Proceedings of the European Design Automation Conference*, 1996.

[14] C. Valderrama and et al. Automatic generation of interfaces for distributed C-VHDL cosimulation of embedded systems: an industrial experience. In *Proceedings of Seventh IEEE International Workshop on Rapid System Prototyping*, 1996.