Fast Hardware-Software Co-simulation Using VHDL Models

Bassam Tabbara*
Marco Sgroi
Alberto Sangiovanni-Vincentelli

Enrica Filippi

Luciano Lavagno

EECS Department
U.C. Berkeley
Berkeley, CA 94720, USA

CSELT †
V. Reiss Romoli 274
I-10148 Torino, Italy

Cadence Berkeley Labs
2001 Addison St., 3rd Floor
Berkeley, CA 94704, USA

Abstract

We describe a technique for hardware-software co-simulation that is almost cycle-accurate, and does not require the use of interprocess communication nor a C language interface for the software components. Software is modeled by using behavioral VHDL constructs, annotated with timing information derived from basic block-level timing estimates. Hardware is also modeled in VHDL, and can be either pre-existing Intellectual Property or synthesized to RTL from a functional specification. Execution of the VHDL processes modeling software tasks is coordinated by a process emulating the target RTOS behavior. The effects of changing the hardware/software partition can be quickly estimated by changing a process parameter defining its target implementation and the processor on which it is running.

1 Introduction

Embedded systems include hardware and software components cooperating together to achieve a common goal, like implementing a cellular phone, controlling a motor or an engine, and so on. Their validation according to the current design practice requires *performance simulation* of both hardware and software, in order to assess the overall performance of the system and to check the correctness of the interfaces.

One common method used to perform such co-simulation involves running the software on a hardware model of the processor [15]. This solution has a major problem: RTL or behavioral processor models are difficult to develop, expensive and slow (up to tens of clock cycles per second for RTL and thousands of clock cycles per second for behavioral). Hence designers often use *bus-functional* processor models, that represent the bit-true behavior of the processor bus, but with a *statistical model* of the application. These models can be used to exercise and debug the hardware side of the hardware/software interface. The software code, on the other hand, is executed and debugged on *instruction set* (ISA) processor models. Such models represent explicit processor registers and interpret its binary code. Their speed can be up to tens of thousands of clock cycles per second [22], but they handle only approximate timing information, even at the clock cycle level, or no timing at all.

Recent commercial solutions, such as the Seamless environment described in [11], filter the data sent between the instruction set simulator (which must have cycle-exact simulation capabilities) and the hardware simulator. Even this approach, though, is not completely satisfactory, because it requires extensive manual intervention to “abstract” the interface, by hiding events such as instruction fetches or some memory accesses from the hardware simulation.

The co-simulation methodology described in this paper is aimed exactly at filling this “validation gap” between fast models without enough information (e.g., instructions without timing or bus cycles without instructions) and slow models with full detail. It assumes that *execution time estimates* are available for each basic block of software [12]. In particular, such estimates are easily available if one uses a software synthesis-based approach to co-design, as described in [18].

The basic idea is to create a *behavioral* model of the software and an RTL model of the synthesized hardware, with delay information derived from the timing estimation. The synthesized hardware and software can thus be simulated together with existing hardware components also modeled in VHDL. Moreover, changing processor or assigning a component to the hardware partition can be done by simply modifying a parameter that selects the timing estimates to be used for each basic block (or waits for the next clock edge in the case of hardware implementation).

The performance of the behavioral model can be much higher than that of an RTL processor model, because it reduces both the number of *simulation events*, and the number of *simulated bits*. Both reductions are effective when using an *event-driven* hardware simulator, and only the latter is ef-

*SRC Graduate Fellow
†Centro Studi e Laboratori Telecomunicazioni

fective when using a *cycle-based* hardware simulator.

Our approach is different from those described e.g. in [9, 13, 14], that rely on a single custom simulator for hardware and software, because we can use any commercial VHDL simulator. It is also different from the class of solutions described e.g. in [11, 19, 20, 6, 21] that execute the software and hardware partitions in separate processes, keeping track of time independently in the two domains, because it does not require elaborate mechanisms to synchronize them. In particular, we do not need a cycle-accurate nor a bus-cycle model of the target processor. Only performance estimate numbers for the software components are needed. The advantage in terms of simplicity and performance, of course, has a cost in terms of precision, as will be discussed below.

VHDL is a standard language and many well supported tools for efficient simulation are commercially available. This makes integration with external blocks perhaps designed with different methodologies quite straightforward. This feature of our method has a lot of added value in terms of:

- design re-use,
- incorporation of Intellectual Property (IP) Libraries, and
- integration of new co-design techniques into an established industrial design flow.

The paper is organized as follows. In Section 2 we provide some background information. In Section 3 we describe the co-simulation technique in detail. In Section 4 we show results for an industrial-size application. Finally, in section 5 we discuss the status of this work and outline directions for future research.

2 Overview of the Co-design Environment

Our co-simulation methodology is heavily based on the use of software and hardware synthesis. This simplifies the estimation of performance without requiring any user input, as well as the customization of the generated code in order to adapt it to simulation and execution in the target system. We use the POLIS co-design environment for reactive embedded systems ([8]) for synthesizing software and hardware, and for analyzing their performance since that co-design environment is open, available to the public, and provides software synthesis and estimation capabilities. In this Section, we briefly describe the hardware and software synthesis strategies in POLIS as they relate to the modeling for co-simulation.

POLIS is centered around a single Finite State Machine representation, known as Co-design Finite State Machine (CFSM). Each element of a network of CFSMs describes a component of the system to be modeled, and defines the partitioning and scheduling granularity. The CFSM model is based on:

- Extended Finite State Machines, operating on a set of finite-valued enumerated or integer subrange variables by using arithmetic, relational, and Boolean operators, as well as user-defined functions. Each transition of a CFSM is an atomic operation. All the analysis and synthesis steps ensure that:
  1. a snapshot of the system state is taken just before the transition is executed,
  2. the transition is executed, thus updating the internal state and outputs of the CFSM,
  3. the result of the transition is propagated to the other CFSMs and to the environment.
- The interaction between CFSMs is *asynchronous* in order to support "neutral" specification of hardware and software components by means of a single CFSM network. This means that:
  - The execution delay of a CFSM transition is unknown a priori. It is only assumed to be non-zero in order to avoid the composition problems of Mealy machines, due to undelayed feedback loops. The synthesis procedure refines this initial specification by adding more precise timing information as more design choices are made (e.g., partitioning, processor selection, and compilation). The designer or the analysis steps may also add constraints on this timing information that synthesis must satisfy.
  - Communication between CFSMs is not by means of shared variables (as in the classical composition of Finite State Machines), but by means of events.

Software synthesis and performance estimation in POLIS is based on a simplified Control-Data Flow Graph (CDFG) called S-GRAPH.

An S-GRAPH is a Directed Acyclic Graph (DAG) consisting of the following types of nodes:

- **BEGIN**, **END** are the DAG source and sink nodes, and have one and zero children respectively,
- **TEST** nodes are labeled with a finite-valued function, defined over the set of input and output variables of the S-GRAPH. They have as many children as the possible values of the associated function.
- **ASSIGN** nodes are labeled with an output variable and a function, whose value is assigned to the variable. They each have one child.

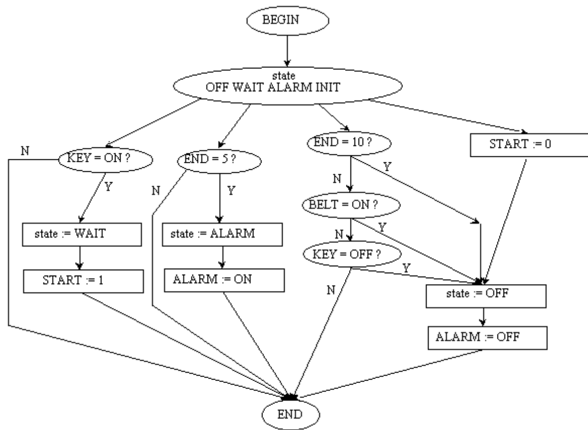

Figure 1: The S-GRAPH of a Seat Belt Alarm Controller

Figure 1 shows the S-GRAPH of a simple CFSM implementing a seat belt controller that turns on the alarm if the driver does not fasten the seat belt 5 seconds after turning on the ignition key, and turns off the alarm after 10 seconds, or when the seat belt is fastened.

It should be clear that an S-GRAPH has a straightforward, efficient implementation as sequential code on a processor. Moreover, the mapping to object code, whether directly or via an intermediate high-level language such as C, is almost 1-to-1. This 1-to-1 mapping is used in POLIS to provide *accurate estimates* of the code size and execution time of each S-GRAPH node. This estimation method works satisfactorily if:

1. The cost of each node is accurately analyzed. This is a relatively well-understood problem, since each S-GRAPH node corresponds *roughly* to a basic block of code, that is a single-input, single-output sequence of C code statements.

2. The interaction between nodes is limited (also known as the "additivity hypothesis" in the literature). This is approximately true *in the case of* an S-GRAPH, since there is little regularity that even an optimizing compiler can exploit (no looping, etc.).

If the level of accuracy is not sufficient, one could use the techniques described in [12] in order to refine the estimation. Note that in this case the S-GRAPH structural restrictions mean that no user input to the estimation tool would be required.

In the POLIS system, code cost (size in bytes and time in clock cycles) is computed by analyzing the structure of each S-GRAPH node, for example:

- the number of children of a **TEST** node (a different timing cost is associated with each child),
- the type of tested expression. For example, a test for event presence must include the RTOS overhead for event handling, and reading an external value must include the execution time of the driver routine.

A set of cost parameters is associated with every such aspect, and is used to estimate the total cost of each node. These costs are then used by the co-simulation environment to accumulate clock cycles, and hence to synchronize the execution of software CFSMs with each other and with the rest of the system (hardware CFSMs and the environment). In this way, neither estimation nor co-simulation require the designer to have access to any sort of model (RTL, instruction set, user's manual) for a processor whose performance is to be evaluated for a given application. Only the values of the set of parameters are necessary. These are part of a library distributed with POLIS for a growing number of microcontrollers.

Clearly a more accurate analysis technique, for example based on a cycle-accurate model of the processor [15, 11], is needed to validate the *final implementation*. But the architecture exploration phase can be carried out much faster, as long as the precision of estimation (currently within 20%) is acceptable for the task at hand.

CFSMs implemented in hardware are currently synthesized assuming that each transition requires exactly one clock cycle, by using classical RTL and logic synthesis techniques.

3 High-level Co-simulation Using VHDL

Our approach to co-simulation is based on the decomposition of the system into three classes of components:

1. software CFSMs, synthesized by POLIS and executed on a single processor under the control of a Real-Time Operating System (RTOS). The RTOS, also synthesized by POLIS, handles communication within the processor and with the rest of the system (the limitation to only one processor is not inherent in our method, but is only due to the current status of the RTOS synthesis),

2. hardware CFSMs, also synthesized by POLIS and communicating via a standardized protocol with the rest of the system,

3. existing pieces of hardware IP, modeled in VHDL (behavioral or RTL).

We synthesize a VHDL model for each CFSM, for the RTOS scheduler, and for the interfaces. We assume that existing IP has been adapted to use the POLIS communication protocol, as described below. This adaptation task proved to be very simple in the case study discussed in Section 4, and techniques such as those described in [17] can be used to automate it. In this section we describe each element in detail.

3.1 Modeling The Software Tasks

The behavioral VHDL simulation model of a piece of software implementing a CFSM is generated automatically using the same mechanism that is used for software synthesis. In this way, we keep a 1-to-1 correspondence between C basic blocks (S-GRAPH nodes) in the implementation and groups of VHDL statements in the simulation model. Each such group is annotated with the performance numbers for the target processor, so as to keep track of the estimated timing while executing the VHDL simulation.

Behavioral VHDL lacks the infamous `goto` statement, that is the basis of the 1-to-1 S-GRAPH implementation in C. Hence, representing an S-GRAPH in VHDL using `if` and `case` statements could *potentially* lead to an exponential explosion in code size. Thus a mechanism is needed to retain linear complexity also for the VHDL implementation.

The basic idea of our modeling approach hence is as follows: the S-GRAPH is interpreted as an FSM, with one state for each S-GRAPH node. The execution of the S-GRAPH from BEGIN to END then becomes an ordered traversal of a sequence of states of this FSM. In some sense, this FSM is a *sequential implementation* of the transition function of the CFSM, just like the synthesized C code. The structure of the VHDL code for the belt controller software task described in Section 2 is shown in Figure 2.

Parameter `SW_CLOCK` determines the time unit for a CPU clock cycle, and is used to keep the simulation synchronized with the hardware partition and with the external world. In this case the delays were estimated assuming the use of a Motorola 68HC11 micro-controller.

This solution is slower than using a "tree-like" explosion of the S-GRAPH structure with VHDL `if-then-else` and case statements, because an event must be posted to the global VHDL timing queue once for each traversed S-GRAPH node, rather than once for each CFSM transition. However, we chose this solution for two main reasons: First, it keeps the synthesized code small (linear in the size of the S-GRAPH, and hence of the code that will be loaded on the target processor), and second, it makes modeling the software scheduler (Real Time Operating System) much easier.

3.2 Modeling The Scheduler

A scheduler VHDL process is necessary to coordinate those implementing software CFSMs. The scheduler must ensure that such CFSMs are active one at a time, by receiving a request from every one of them, and deciding which one is going to use the processor next. All other software CFSM processes will be delayed by the appropriate time to allow the current CFSM to complete one transition. If the scheduler is preemptive, such request/acknowledge should take place once for each instruction execution on the simulated CPU, since the currently executing CFSM may be interrupted in the middle of a transition to allow a higher-priority CFSM to respond to an urgent request. This can be quite inefficient, since it requires performing signal-based handshaking once for each simulated CPU instruction. The mechanism for implementing an S-GRAPH in VHDL allows us to choose an intermediate solution: to allow pre-emption at the S-GRAPH node level. This solution has a relatively low overhead (one handshake for several target CPU instructions, implementing one S-GRAPH node) and a generally satisfactory level of granularity in responding to preemption requests (e.g. from interrupt sources). Figure 3 shows an illustrative example of a *Round Robin* scheduler and an I/O polling task for the belt controller (priority-based scheduling has also been implemented).

```
--Round Robin Scheduler
scheduler_round_robin:
process
begin
--first task
if (activate_belt_control = '1') then
   move_belt_control <= '1';
   wait until ready_belt_control'event and
      (ready_belt_control = '0');
   move_belt_control <= '0';
   wait until ready_belt_control'event and
      (ready_belt_control = '1');
end if;
--second task
if (activate_... = '1') then move_... <= '1';
...
end process scheduler_round_robin;

--Event polling processes
activate_belt_control <= (mid_belt_control or
            e_END_1_to_belt_control...);
activate_... <= ...
```

Figure 3: The Behavioral VHDL Code for a Round Robin Scheduler

3.3 Modeling Hardware and Interfaces

The technique used for modeling hardware components of the embedded system depends on whether they have been synthesized from CFSMs or have been designed by using different techniques (e.g. directly in synthesizable VHDL). In the former case, we can just use the same S-GRAPH-like VHDL model, by just setting the task execution delay to the number of clock cycles that will be used for the final hardware implementation. In the latter case, the designer must make sure that those modules use the CFSM communication protocol (1 bit active for 1 cycle for an event, and n bits sampled only when the corresponding event is at 1 for the

```
Architecture CFSM of belt is
    --internal signal (sender side)
    signal e_timer_e_end_5_o_tmp : bit := '0';
    --internal signal (receiver side)
    signal e_timer_e_end_5_to_belt_control : bit := '0';
    --input from environment
    signal e_key_on_to_belt_control : bit := '0';
    --task activation and scheduling signals
    signal move_belt_control : BIT := '0';
    signal ready_belt_control : BIT := '1';
    signal cleanup_belt_control : BIT := '0';
Begin
belt_control: process
  type S_Type is (STB,ST1,...,STend); --s-graph nodes
  variable Lbl, Next_Lbl: S_Type := STB;
  ...
  variable e_key_on_tmp: bit; --buffered event
  variable e_timer_e_end_5_tmp: bit; --buffered event
begin
wait on move_belt_control;
if ( move_belt_control = '1' ) then
    ready_belt_control <= '0'; --trigger to move
    Lbl := Next_Lbl;
else
case Lbl is
when STB =>
    mid_belt_control <= '1'; --starting transition
    -- sample input events
    e_key_on_tmp := e_key_on_to_belt_control;
    e_timer_e_end_5_tmp := e_timer_e_end_5_to_belt_control;
    ...
    cleanup_belt_control <= '1';
    --base delay
    ready_belt_control <= '1' after 56 * SW_CLOCK;
...
when ST2 =>
    if (e_key_on_tmp = '1') then -- check event key_on
        Next_Lbl := ST17;
        ready_belt_control <= '1' after 40 * SW_CLOCK;
    else
        Next_Lbl := ST3;
        ready_belt_control <= '1' after 26 * SW_CLOCK;
    end if;
...
when ST9 =>
    -- check event end_5
    if (e_timer_e_end_5_tmp = '1') then
    ...
```

Figure 2: The Behavioral VHDL Code for the Software Seat Belt Controller

value). The simple buffers, automatically inserted by the VHDL code generator to implement the hardware side of the interfaces (including those with the test-bench) are shown in Figure 4.

4 A Practical Case Study: An ATM Server

The co-simulation technique described in this paper has been used to validate the design of an industrial case study from the communication networks domain: an ATM server suitable for implementing Virtual Private Networks (VPN) in ATM nodes, which is a re-engineering of the system described in [5].

Our target system is essentially a statistical multiplexing unit capable of performing traffic management functions like controlling the bandwidth of the outgoing flows, and preserving the flow integrity at the message level. Message discarding techniques and a per-flow queuing service discipline are implemented.

The input of the system is a stream of ATM cells belonging to the set of active Virtual Channel Connections (VCC). Cells are buffered inside the server. The buffer is divided into FIFO queues, one for each output Virtual Path Connection (VPC). The incoming cells are forwarded to the proper FIFO according to the entries in the internal routing table as shown in Figure 5. The timing constraints of the system are tight.

```
-- communication interfaces
  --between 2 SW processes
  process
  begin
   wait until e_timer_e_end_5_o_tmp = '1';
   e_timer_e_end_5_to_belt_control <= '1';
   wait until cleanup_belt_control = '1';
   e_timer_e_end_5_to_belt_control <= '0';
  end process;

  --between external world and SW process
  process
  begin
   wait until e_key_on = '1';
   e_key_on_to_belt_control <= '1';
   wait until cleanup_belt_control = '1';
   e_key_on_to_belt_control <= '0';
  end process;
```

Figure 4: The Behavioral VHDL Code for the Communication Interfaces

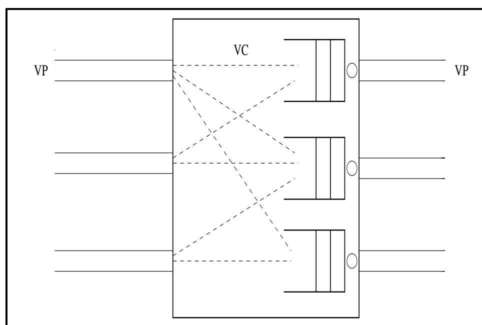

Figure 5: Operational View of the Buffers Inside the ATM Server

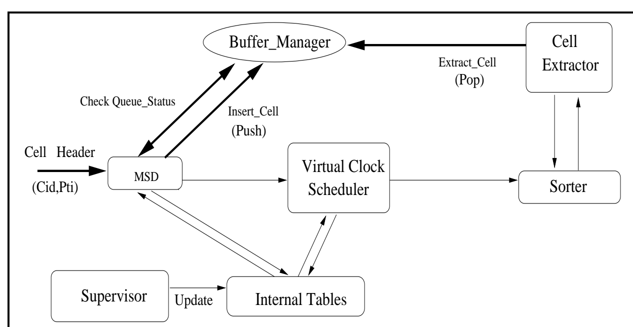

Figure 6: A High Level Description of the ATM Server Control Unit

The processing of every incoming cell has to be done before the next cell arrives, i.e. within $2.72\mu s$, for a link rate of 155 Mbit/s.

The system is composed of two parts: a fast hardware data path, and a control unit.

The fast data path includes two standard interfaces (UTOPIA see [1]), an ATM cell address lookup unit, a buffer logic queue manager, and a large buffer memory. It is implemented with a set of VHDL synthesizable IP models [7] and some commercial memories.

The control unit has been designed using POLIS, and implements the server core custom functionalities:

- Buffer management: the Message Selective Discarding (MSD) technique avoids node congestion by preserving the integrity of messages.
- Egress policing: the bandwidth of the outgoing flows is controlled by a Virtual Clock scheduling technique, that provides fair bandwidth allocation among the queues.

In Figure 6 a high-level description of the control unit functional blocks is given.

VHDL co-simulation has been used to validate the whole system (including both the data path and the control unit). The ATM server design is composed of about 14000 VHDL code lines, of which about 7000 lines are from RT-level IP modules, about 6700 lines have been synthesized from CFSMs, and the rest come from hand-written code. In other words, one half of the design comes from reusable IP RT-level code.

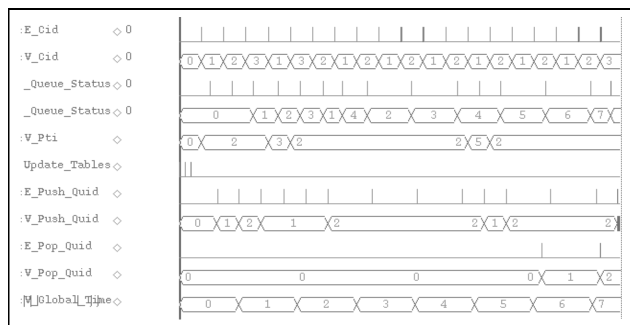

Figure 7: Screen Capture of a Simulation Run

Implementation	Clock Cycles per CPU Sec.
All SW (no scheduler)	50,000
All SW	214,000
All HW	7,000
Mixed HW/SW	15,000

Table 1: VHDL Co-simulation Results for ATM Switch

The control unit has been modeled as a network of 25 CFSMs in POLIS, resulting in the 6700 lines of behavioral VHDL mentioned above or about 2450 lines of C code (used for the final implementation on a microprocessor). In the following we report results relative only to the part of the design that has been *fully* synthesized using POLIS.

A screen capture of a simulation run is shown in Figure 7.

The VHDL description took less than a minute to compile. The co-simulation results for different system implementation alternatives are displayed in Table 1(data collected from a commercial VHDL simulator on a Sun Ultra 2 workstation with 256MB of memory and 2 CPUs). The first row in the results table is for a concurrent software implementation (no scheduler). The second row is for an entirely software implementation with a Round Robin scheduler. The third row is for an entirely hardware implementation. The last row is for an implementation where the *MSD*, *Virtual Clock Scheduler*, *Cell Extractor*, and *Supervisor* tasks of Figure 6 were implemented in software and the *Sorter*, *Buffer_Manager*, and *Internal Tables* tasks in hardware.

5 Conclusions and Future Work

In this paper we have presented a mechanism for co-simulating synthesized hardware and software together with existing hardware Intellectual Property in a single VHDL-based environment. This technique uses software timing estimation to efficiently synchronize the VHDL processes

modeling software tasks with those modeling hardware components and the test-bench. It permits *easy exploration of the design space* in terms of choice of the:

- partition,
- processor, and
- target clock(s) speed.

Since the VHDL entity (that is, the interface to the external world) does not depend on the chosen partition, no changes are needed in the VHDL test-bench code when a different partition is tried. Also, as a side effect of the fact that our approach does not require costly microprocessor models, and that hardware and software clock cycles are generic parameters, a quick and cheap evaluation of many different processors is possible.

The VHDL code generated by our package within POLIS can be integrated with other VHDL blocks into a global system test-bench. Usually however, external pre-existing modules (in our test case, IP blocks and memories) do not follow the POLIS event-based communication model, and some hand-written protocol adapters are required. In most cases the needed interfaces are very simple and efficient.

Integrating modules which are intrinsically based on rendez-vous protocols (like memories) results in an expected performance loss at the implementation level; some extensions to the POLIS hardware to hardware, and hardware to software interface models would help to improve overall performance in this case. However, the current POLIS implementation of software CFSMs is tuned to reactive systems in which events have little temporal correlation. Hence the implementation of tight handshaking sequences, such as for example the address and data generation for a memory, would require a different software and RTOS synthesis strategy. When IP blocks with complex functionalities are used, some events are usually more important than others, and behave as "trigger" events for the whole system; our strategy is to use some interrupts in this case and that usually alleviates the loss in performance.

The need to achieve fast simulation speed within a single co-simulation environment has forced us to ignore some aspects of the final embedded system implementation. These include:

- the overhead due to the scheduling mechanism, which may depend on the number of tasks (e.g., for priority-based schemes the choice of a task is generally logarithmic in the number of tasks in the worst case),
- the cost of inter-processor or hardware/software communication

We plan to look at lifting both limitations as part of our future development work. Lifting the first limitation is straightforward, as it is easy to also include the RTOS performance figures with the chosen approach. Lifting the second limitation requires modeling bus resource contention (see [16, 10] for a good discussion of this issue).

In the future we also plan to explore the trade-off between simulation speed and VHDL code size due to using the "FSM-like" mechanism for unstructured, `goto`-based programming. This trade-off also involves the precision at which the preemptive scheduling mechanism must be modeled. Most likely, the final choice will be a mix of "state-based" and "if-based" implementation, to get the best performance with a bounded increase in code size.

References

[1] The ATM Forum "Utopia, An ATM-PHY Interface Specification, Level 1, v2.01" March 1994.

[2] G. Berry, 1996. See http://cma.cma.fr/Esterel

[3] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. "Hardware/software Codesign of Embedded Systems" *IEEE Micro*, Vol. 14, Number 4, pp. 26-36, 1994.

[4] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. "Synthesis of Software Programs from CFSM Specifications" *Proceedings of the Design Automation Conference*, June 1995.

[5] P. Coppo, M. D'Ambrosio, V. Vercellone "The A-VPN Server, a Solution for ATM Virtual Private Networks" *ICCS*, November 1994.

[6] J. Ernst, C. Prasad, G.S. Thurston "Cosimulation with Heterogeneous Simulation Algorithms Using Distributed Objects" *Summer Computer Simulation Conference*, July 1997.

[7] E. Filippi, L. Licciardi, A. Montanaro, M. Paolini, M. Turolla, M. Taliercio "The Virtual Chip Set: A Parametric IP Library for System on a Chip Design" CICC, Santa Clara, May 1998.

[8] The POLIS group, POLIS Home Page, http://www-cad.eecs.berkeley.edu/~polis

[9] R. K. Gupta, C. N. C. Jr., and G. D. Micheli "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components" *In Proceedings of the Design Automation Conference*, June 1992.

[10] K. Hines, G. Borriello "Selective Focus as a Means of Improving Geographically Distributed Embedded System Co-simulation. *IEEE International Workshop on Rapid System Prototyping* p. 58-62, 1997.

[11] R. Klein and S. Leef "New Technology Links Hardware and Software Simulators" *In Electronic Engineering Times*, June 1996.

[12] Y. Li and S. Malik "Performance Analysis of Embedded Software Using Implicit Path Enumeration" *In Proceedings of the Design Automation Conference*, June 1995.

[13] K.A. Olukotun, R. Helaihel, J. Levitt, R. Ramirez "A Software-Hardware Cosynthesis Approach to Digital System Simulation" *IEEE Micro*, vol. 14(4):48-58, Aug. 1994.

[14] C. Passerone, L. Lavagno, C. Sansoè, M. Chiodo, A. Sangiovanni-Vincentelli "Trade-off Evaluation in Embedded System Design Via Co-simulation *ASPDAC*, Jan. 1997.

[15] J. Rowson "Hardware/Software Co-simulation" *In Proceedings of the Design Automation Conference*, pp. 439-440, June 1994.

[16] J. Rowson "Interface-Based Design" *In Proceedings of the Design Automation Conference*, June 1997.

[17] A. Seawright, U. Holtmann, W. Meyer, B. Pangrle, et. al. "A System for Compiling and Debugging Structured Data Processing Controllers" *EURO-DAC* p. 86-91, Sept. 1996.

[18] K. Suzuki and A. Sangiovanni-Vincentelli "Efficient Software Performance Estimation Methods for Hardware/Software Codesign" *In Proceedings of the Design Automation Conference*, pp. 605-610, June 1996.

[19] K. Ten Hagen and H. Meyr "Timed and Untimed Hardware/Software Co-simulation: Application and Efficient Implementation" *In Proceedings of the International Workshop on Hardware-Software Codesign*, October 1993.

[20] D.E. Thomas, J.K. Adams, H. Schmit "A Model and Methodology for Hardware-Software Codesign" *IEEE Design and Test of Computers*, vol. 10(3):6-15, Sept. 1993.

[21] C.A. Valderrama, A. Changuel, A.A. Jerraya "Virtual Prototyping for Modular and Flexible Hardware-Software Systems" *Design Automation for Embedded Systems*, vol. 2(3-4):267-82, May 1997.

[22] V. Zivojnovic and H. Meyr "Compiled HW/SW Co-simulation" *In Proceedings of the Design Automation Conference*, June 1996.