Fast hardware/software co-simulation for virtual prototyping and trade-off analysis

Claudio Passerone, Politecnico di Torino – Cadence Europeans Labs, Italy
Luciano Lavagno, Politecnico di Torino – Cadence Europeans Labs, Italy
Massimiliano Chiodo, Alta Group of Cadence Design Systems, USA
Alberto Sangiovanni-Vincentelli, Dept. of EECS, University of California at Berkeley, USA

Abstract

Hardware/software co-simulation is generally performed with separate simulation models. This makes trade-off evaluation difficult, because the models must be re-compiled whenever some architectural choice is changed. We propose a technique to simulate hardware and software that is almost cycle-accurate, and uses the same model for both types of components. Only the timing information used for synchronization needs to be changed to modify the processor choice, the implementation choice, or the scheduling policy. We show how this technique can be used to decide the implementation of a real-life example, a car dashboard controller.

1 Introduction

Simulation of a mixed hardware/software system can be performed at all levels of abstraction. The most commonly used approach, including commercial tools, runs the software on a hardware model of the processor ([Row94]). 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 people often resort to using *bus-functional* processor models, that represent the bit-true behavior of the processor bus, but using 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* (or 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 ([ZM96]), 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 [KL96], 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 is based on using the very same code that will be run on the target processor, together with timing annotation obtained by performance estimation, as simulation model for software components. It runs the simulation at full speed on a workstation, by keeping the software synchronized with the hardware components (also simulated using the same behavioral model, but without timing) and with the environment model (testbed).

The paper is organized as follows. Section 2 describes our co-simulation methodology in detail. Section 3 shows with an example our implementation and how co-simulation can be used to interactively evaluate the performance of various partitions of a system under various RTOS conditions. Section 4 concludes the paper and outlines opportunities for future research.

2 Co-simulation via software synthesis

Our co-simulation methodology is heavily based on the use of software and hardware synthesis. This simplifies the customization of the generated code in order to adapt it to simulation and execution in the target system. We use an existing codesign environment for reactive embedded systems, called *POLIS*, described in [CGH+94] and available at URL [hp], for synthesizing software and hardware, and for analyzing their performance. It is centered around a single Finite State Machine representation, known as Codesign 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

DAC 97, Anaheim, California

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:
  1. The execution delay of a CFSM transition is unknown a priori. It is only assumed to be nonzero 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 in addition add constraints on this timing information that synthesis must satisfy. The overall design philosophy of *POLIS* is to provide the designer with *tools* to satisfy these constraints, rather than a push-button solution.
  2. Communication between CFSMs is not by means of shared variables (as in the classical composition of Finite State Machines), but by means of *events*. Events are a semi-synchronizing communication primitive that is both powerful enough to represent practical design specifications, and efficiently implementable within hardware, software, and between the two domains.

2.1 Software synthesis

Software synthesis techniques from formal specifications have reached a good degree of maturity for at least some specific application domains, like embedded control ([Ber96]). A design environment based on software synthesis allows the designer to specify the system in a high level formal language, describing the functionality of each block and how they are connected together. After translating this specification into an intermediate format, all sorts of optimization can be carried out to reduce the size and increase the speed of execution (see, e.g., [CGH+95]).

Software synthesis in *POLIS* is based on a Control-Data Flow Graph (CDFG) called S-graph [CGH+95]. The S-graph is considerably simpler than the CDFG used, for example, by a general purpose compiler because its purpose is only to specify the *transition function* of a *single* CFSM.

An S-graph computes a function from a set of finite-valued variables to a set of finite-valued variables.

1. The input variables correspond to input signals[1]. Each signal is a control signal, a data signal, or both, and can be associated with
   (a) a Boolean control variable, which is true when an event is present for the current transition, and
   (b) an enumerated or integer subrange variable.

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.

Traversing the S-graph from BEGIN to END computes the function represented by it. Output variables are initialized to an undefined value when beginning the traversal. Output values must have been assigned a defined value whenever a function depending on them is encountered during traversal of a *well-formed* S-graph.

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.

We use this 1-to-1 mapping 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. Any one of a number of estimation techniques can be applied, including the benchmark-based cost estimation method described in [SSV96].
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.). We are currently exploring ways of modeling interaction between nodes due to caches, multiple functional units, and pipelines, in order to model complex processors more accurately.

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

[1] States, i.e., signals that are fed back in the CFSM network, can be treated as a pair of input and output signals connected together for the purpose of this discussion.

of the set of parameters are necessary. These are part of a library distributed with *POLIS* for a growing number of micro-controllers.

The parameters can be derived either automatically, or by hand (e.g., by inspecting the assembly code after synthesis and compilation for the target processor). In the former case, the processor library maintainer needs to compile a set of benchmarks and analyze their size and timing by using a profiler for the target system.

Clearly a more accurate analysis technique, for example based on a cycle-accurate model of the processor ([Row94, KL96]), 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.

Note that there are some aspects of the final system which are ignored by using this scheme:

- 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 claim that such levels of detail may not be necessary in the initial phases of system architecture definition, and that their lack is well compensated by the increased flexibility and speed of the approach. We are also planning to look at lifting those limitations as part of our future development work.

2.2 The simulation

We use as our simulation engine the tool Ptolemy ([BHLM90]), which is a complete design environment for simulation and synthesis of mixed hardware/software data-dominated *embedded systems*. In particular, we used the Discrete Event (DE) domain of Ptolemy to implement the event-driven communication mechanism among CFSMs. The formal specification of the system to be modeled is first translated by *POLIS* into a network of CFSMs, and then synthesized as timing-annotated C code as described above. The C code is used in Ptolemy as a model for both hardware and software components. The scheduler in the DE domain fires components as if they were executed concurrently, so it does not provide a way to directly simulate CFSMs implemented in software, and running on a limited amount of computational resources. We therefore changed its behavior without modifying its code, by adding a procedure on top of it (see [PLS+97]), and letting Ptolemy see a world of concurrent components, while the software ones see the *POLIS* scheduling policy.

Different scheduling policies, corresponding to those implemented by the *POLIS* RTOS, can be provided, which enable the designer to experiment several solutions and find the one which gives the best results. However, all the policies should have the following common features:

- Software components which run on the same microprocessor should be executed in a mutual exclusion fashion, since they share the same computational resource. Therefore we need a way to hold the execution of a process until the CPU is free.
- Hardware components should be executed as concurrent processes, which terminate in one clock cycle.

The various schedulers will differ on how the software task which should be executed is chosen among all the enabled ones. Some of them can also implement priorities, and provide a way to suspend a running task and resume it at a later time, thus implementing a preemptive scheme.

All the software models of the blocks representing the system are compiled on the host workstation, and the scheduler is used to invoke each process when necessary. Based on the implementation choice and the running time estimations, the simulator can determine how many clock cycles are required to execute a component and the availability of the computational resources, so the scheduler can effectively implement concurrent and mutually exclusive behaviours.

Since our representation is event driven, we chose to use a Discrete Event simulator. Aiming our co-simulation tool at reactive real time control-dominated systems, a static schedule cannot be computed. Processes are queued and scheduled according to the communication event ordering in time and the selected policy. Tasks are executed atomically, and only their outputs are delayed to the proper time-stamp, thus sensibly reducing the number of scheduling operations to be performed. In this way we do not have to reschedule the same task for every single statement in the C code, thus saving simulation time.

The only problem with this mechanism is that if an interrupt occurs, the computed delays are no longer valid, and must be *updated*, while they are still in the event queue, based on the time taken by the interrupt service routine. We rely on the fact that in our model *and in the final implementation* the input values of each task cannot change once it has been started. The runtime RTOS ensures that a consistent snapshot of the system status is provided to each CFSM, thus implementing the synchronous layer of the GALS model. With this technique we can handle multiple priority levels, nested and maskable interrupts at the same time, with very little overhead.

We can therefore have a very fast simulation, sometimes even faster than the target system, for purely reactive systems with low frequency inputs, like many embedded control applications. In this case the co-simulation acts more as a virtual prototyping tool, with all the flexibility of an entirely software simulation.

Changing implementation or target microprocessor is just a matter of using the corresponding estimations while counting the cycles, and does not require one to change the model or to recompile the entire system. The timing estimations for all the target processors are embedded in the C code, and can be selected at run–time by simply changing a simulation parameter. This also allows us to simulate the behaviour of tasks running on a multiprocessor system, where the processors can be either of the same kind, or differ among each other. However, this feature is not yet implemented in the current version of our tool. Therefore the software model simply captures the behaviour for hardware components, and one clock cycle running time is assumed, and describe both behaviour and timing information for software components.

2.3 Performance

We wanted to obtain both a high speed during the simulation, and a high speed during the interaction with the user. The former enables us to simulate a larger number of input patterns, providing a more extensive test of the algorithm for different operating conditions. The latter lets the designer easily select various architectures and compare their performance and costs. Both these goals were achieved in our implementation:

Execution speed : the simulation is fast for the following reasons:

- hardware components use a cycle-based C model derived directly from their CFSM representation, which is the fastest known method of simulating them ([MMS+95, AM95]),
- software components are compiled on the host workstation and thus can run faster than in the target micro-controller, which is often less powerful.

Interactivity : the user can change several parameters, in order to explore the design space *without recompiling* the model, but just running a new simulation. These parameters include, for example:

- the implementation (and CPU assignment in case of software) of each single block,
- the type of each CPU and its clock frequency,
- the scheduling policy.

The accuracy of this method of performing co-simulation strongly depends on the precision of the running time estimations of the generated code. Obviously precision beyond the cycle level is not within the scope of this technique, which should be used as a high level tool to evaluate trade-offs among different architectures of the same system.

The co-simulator can produce useful information, in order to help the designer in comparing different implementations of the same system. Some of the possible applications are:

- Functional debugging to check the correctness of the algorithm used. In this case, modules are executed concurrently without any timing information.
- Missed deadlines, which can be flagged to the user whenever an event is overwritten in the input queue of a block. This can happen when communication is done by using finite-length queues. This information permits to identify the critical components in the system.
- Processor utilization and task scheduling charts, including interrupt exceptions, that clearly identify which task is running on which processor. They are useful to determine the speed of the processor and the scheduling mechanism which best fits the timing constraints.
- Cost of the implementation, both for hardware (area), and for software (memory size for code and data, estimated execution time).

All this information can greatly help driving the correct selection of an architecture at an early stage of the design, reducing the number of iterations of the design process, and decreasing the time to market. Moreover, high level languages add flexibility, and the design can be easily re-targeted if new hardware becomes available.

3 An application example

We consider an application from the automotive domain: a dashboard controller. We will evaluate different implementation choices, under various possible operating conditions.

The system receives inputs from:

1. a magnetic sensor located near the wheel, producing a pulse every 4^{th} of a revolution,
2. a magnetic sensor located on the engine, producing a pulse every revolution,
3. a potentiometer measuring the fuel level,
4. a thermistor measuring the water temperature,
5. a belt sensor, producing a pulse when the seat belt is fastened,

It provides as outputs:

1. four Pulse Width-Modulated signals, directly driving a pair of gauges (vehicle and engine speed),
2. two 16-bit numbers controlling the odometer display (total and partial kilometers),
3. three flag bits controlling respectively the alarms for the fuel level, the water temperature and the seat belt.

The timing constraints, which in this case are relatively soft, derive mainly from the need not to miss any incoming pulse. For the engine this means up to 200 pulses per second, while for the wheel this means up to 250 pulses per second. Outputs must be produced at a rate of at least 100 Hz and with a maximum jitter of 100 microseconds to drive the gauge coils. In this case, we first assumed to use a Motorola 68HC11, because the timing estimate available from synthesis (e.g., 379 clock cycles at most for one of the time-critical tasks, the engine pulse recorder) showed that we could *hope* to satisfy the requirements with a 1 MHz processor.

The system was modeled using several CFSMs, each specified in ESTEREL (see Figure 1). Their interconnection was described graphically with the Ptolemy user interface, as illustrated in Figure 2. Some of the components in the design are software models for hardware resources commonly found in micro-controller, such as the timer unit. Three blocks are devoted to input conversion from level to pulse, three blocks derive timing events from the base time reference, two blocks filter and normalize the data, one block converts potentiometer readings to fuel level (taking into account the shape of the tank), one reads the thermistor and another one checks the seat belt. The remaining two blocks perform the PWM conversion. The two critical tasks are counting the pulses coming from the wheel and the engine, therefore the highest priority level was assigned to the CFSMs that had to handle input events, and corresponds, in practical terms, to interrupt-driven I/O without interrupt nesting.

Table 1 reports the number of missed deadlines every 1,000,000 clock cycles, under several operating conditions. They can be used as a guidance when deciding the implementation of each block and the processor choice. We simulated in all cases the same amount of *real* time (1 second), independent of the processor speed. We chose to use a Motorola 68HC11 (with clock speed ranging between 1 and 4 MHz) for the mixed hardware/software implementation, and a Motorola 68332 (with clock speed ranging between 20 and 40 MHz) for the all-software implementation. Partition 1 implements in hardware the timing generators, pulse counters

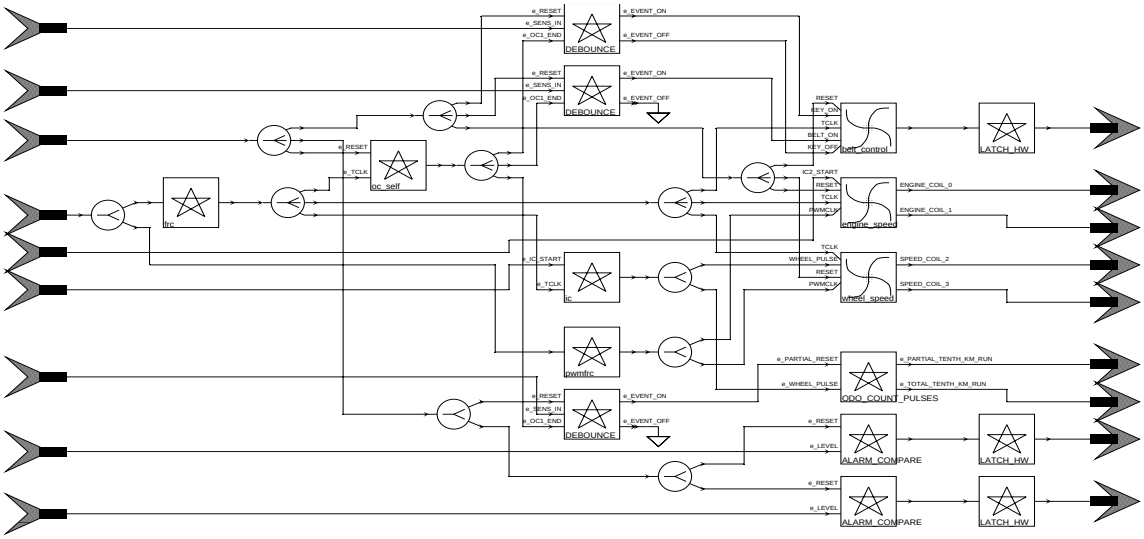

Figure 2: The dashboard controller netlist

```
% count wheel pulses in a time frame (1/10 sec)
module SPEED_COUNT_PULSES:
input RESET, WHEEL_PULSE(integer), OC3_END;
output OC3_START(integer), WHEEL_PULSES(integer);
constant CONST_TIME_UP_WHEEL_PULSE: integer;
var counter: integer
in loop
   do loop
      counter := 0;
      % start the 1/10 sec watchdog timer
      emit OC3_START(CONST_TIME_UP_WHEEL_PULSE);
      do loop
         await WHEEL_PULSE;
         counter := counter + 1;
       end;
       % stop counting when watchdog occurs
      watching OC3_END;
      emit WHEEL_PULSES (counter);
    end;
   watching RESET;
 end;
end.
```

Figure 1: ESTEREL code for a module of dashboard

CPU	Clock	Part.	Wheel. sp.	Eng. sp.	Missed
68HC11	4	1	50	3000	0
68HC11	4	1	180	6000	0
68HC11	4	1	260	8000	0
68HC11	1	1	50	3000	0
68HC11	1	1	180	6000	900
68HC11	4	2	50	3000	5000
68332	20	3	50	3000	30
68332	40	3	50	3000	0
68332	40	3	180	6000	6
68332	40	3	260	8000	20

Table 1: Missed deadlines for various types of system architectures

and PWM generators. Partition 2 implements in hardware the timing generators and pulse counters. Partition 3 implements everything in software. Clock speed is given in MHz, engine speed is given in RPM, wheel speed in Km/H.

The only acceptable solutions, from a performance standpoint, are a 4 MHz 68HC11 with hardware support (versions of the 68HC11 with on-board counters and PWM generators are currently available) and a 40 MHz 68332 without hardware support (in that case, the missed deadlines were due to a slow update in the PWM duty cycle, and hence were considered quite acceptable in practice). Both the 68HC11 and the 68332 were utilized at about 50%, which meant that simple static priority-based scheduling techniques could be used to guarantee deadline satisfaction. The solution with the 68HC11 will most likely be chosen in the final system, due to cost reasons, but the analysis shows that a *prototype board* using a 68332 can be used for functional debugging in the field, with the added convenience of a software-only debugging style.

Figure 3 shows the priority level as a function of time in this second case[2]. An interrupt can be recognized when a low priority task is suspended to execute a higher priority one. The Ptolemy parameter specification mechanism is used to allow the designer to change all the architectural parameters without re-compilation. Currently supported parameters are:

- CPU type, clock speed, scheduler type for the whole system,
- implementation (hardware or software) and priority (used only for software stars) for each star or hierarchical star group (galaxy).

The performance of the simulator is very high, especially if there is no component which is active at every clock cycle, because in that case we can exploit the inactivity of the system; in this case, we could achieve a speed of 300,000 clock cycles per second on a SparcStation 10 (90% of the time is spent executing the Ptolemy scheduler code that keeps the time-sorted event queue).

[2] A value of -1 means that the processor is idle, 1 is the highest level, and each vertical bar represents a context switch.

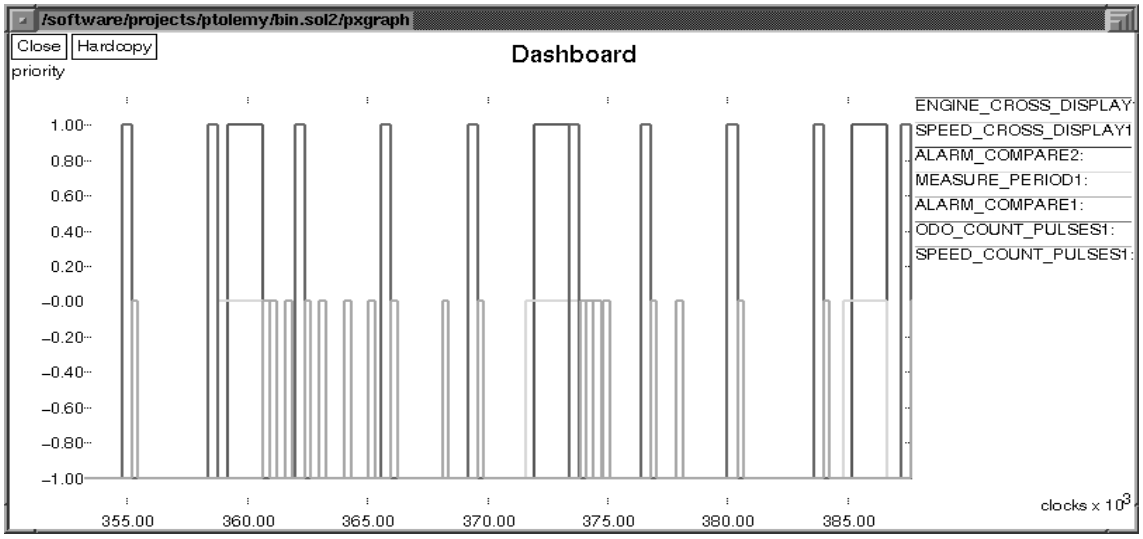

Figure 3: Task scheduling chart with priority levels

4 Conclusions and future work

In this paper we have shown that fast co-simulation can be done at the early stages of a design, for partition evaluation and functional verification purposes. The methodology relies on the use of constrained software synthesis, that permits easy run time estimation for a target processor, and of an event-based co-simulation environment. We intend it to be a mechanism to evaluate trade-offs during the high level phase of a system design, by providing fast design space exploration and performance measures.

Currently simulation of multiprocessor systems is not yet implemented, but we are planning to add it soon. In the future we would like also to improve our estimation technology, and to be able to handle communication costs, caches and pipelined architectures, which limit the current estimation accuracy.

References

[AM95] P. Ashar and S. Malik. Fast functional simulation using branching programs. In *Proceedings of the International Conference on Computer-Aided Design*, November 1995.

[Ber96] Gérard Berry, 1996. See http://cma.cma.fr/Esterel.

[BHLM90] J. Buck, S. Ha, E.A. Lee, and D.G. Masserschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, special issue on Simulation Software Development, January 1990.

[CGH+94] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. Hardware/software codesign of embedded systems. *IEEE Micro*, 14(4):26–36, August 1994.

[CGH+95] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. Synthesis of software programs from CFSM specifications. In *Proceedings of the Design Automation Conference*, June 1995.

[hp] The POLIS home page. `http://www-cad.eecs.berkeley.edu/Respep/Research/hsc/abstract.html`.

[KL96] R. Klein and S. Leef. New technology links hardware and software simulators. *Electronic Engineering Times*, June 1996.

[MMS+95] P. McGeer, K. McMillan, A. Saldanha, A. Sangiovanni-Vincentelli, and P. Scaglia. Fast discrete function evaluation using decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, November 1995.

[PLS+97] C. Passerone, L. Lavagno, C. Sansoè, M. Chiodo, and A. Sangiovanni-Vincentelli. Trade-off evaluation in embedded system design via co-simulation. In *Proceedings of ASP-DAC*, pages 291–297, 1997.

[Row94] J. Rowson. Hardware/software co-simulation. In *Proceedings of the Design Automation Conference*, pages 439–440, 1994.

[SSV96] K. Suzuki and A. Sangiovanni-Vincentelli. Efficient software performance estimation methods for hardware/software codesign. In *Proceedings of the Design Automation Conference*, pages 605–610, 1996.

[ZM96] V. Zivojnovic and H Meyr. Compiled HW/SW co-simulation. In *Proceedings of the Design Automation Conference*, 1996.