Evaluation of trade-offs in the design of embedded systems via co-simulation

Claudio Passerone   Massimiliano Chiodo   Wilsin Gosti
Luciano Lavagno   Alberto Sangiovanni-Vincentelli

Abstract

Current design methodologies for embedded systems often force the designer to evaluate early in the design process architectural choices that will heavily impact the cost and performance of the final product. Examples of these choices are hardware/software partitioning, choice of the micro-controller, and choice of a run-time scheduling method. This paper describes how to help the designer in this task, by providing a flexible co-simulation environment in which these alternatives can be interactively evaluated.

Our approach is based on the Ptolemy co-simulation framework, but it uses a different modeling paradigm, well suited for control-dominated reactive systems (asynchronous extended Finite State Machines called CFSMs). We build on previous research on software synthesis and cost estimation to provide the designer with fast but reasonably accurate performance data. We demonstrate the effectiveness of the approach by showing the result of the trade-off analysis on a dashboard control system.

1 Introduction

One of the major problem facing an embedded system designer is the multitude of different design options, that often lead to dramatically different cost/performance results. In this paper we address the problem of trade-off evaluation via simulation, rather than via mathematical analysis. We believe that by providing efficient tools, supporting a variety of target implementation architectures and flexible mechanisms for evaluating design choices, we will help the designer in this difficult task more effectively than by providing relatively inflexible partitioning algorithms.

The problems that we must solve include:

- fast co-simulation of mixed hardware/software implementations, with accurate synchronization between the two components.

- evaluation of different processors and processor architectures, with different speed, memory size and I/O characteristics.
- co-simulation of heterogeneous systems, in which data processing is performed simultaneously to reactive processing, i.e. in which regular streams of data can be interrupted by urgent events. Even though our main focus is on reactive, control-dominated systems, we would like to allow the designer to freely mix the representation, validation and synthesis methods that best fit each particular sub-task.

The approach that we have chosen relies on the following basic ideas:

1. Use of an existing co-simulation environment for heterogeneous systems, which provides interfacing with a well-developed set of design aids for digital signal processing ([BHLM90, KL93, Buc93]).
2. Use of synthesis from a formal specification, as a mechanism of restricting embedded software syntactic structure, without restricting too much its expressive power[1]. This allows us to use powerful synthesis and analysis mechanisms, reducing the complexity and improving the quality of the results with respect to optimizing and analyzing programs written in an unrestricted general-purpose language. This also allows the designer to experiment with different implementation options for a given specification, e.g. as software or hardware, without requiring to always change the specification. In some cases (e.g. sorting), the choice of an algorithm is dictated by the envisaged implementation, but there are border cases for which the choice is not obvious a priori and an uncommitted specification can be very useful.

Software and hardware models are thus executed in the *same simulation environment*. Different implementation choices are reflected in different simulation times required to execute each task (one clock cycle for hardware, a number of cycles depending on the selected target processor for software) and in different execution constraints (concurrency for hardware, mutual exclusion for software). Efficient synchronized execution of each domain is a key element of any co-simulation methodology aimed at evaluating potential bottlenecks associated with a given hardware/software partition.

Past work in the area of performance prediction and trade-off evaluation has focused mostly on elaborate cost models to guide automated partitioning algorithms ([KAJW93], [VG92], [HEHB94], [HDMT94], [BRX93], [KL94], ...), or on co-simulation methods in which a rather detailed model of the processor may be required.

Specifically, co-simulation requires to satisfy two conflicting requirements:

1. accurate modeling of the interaction between software and hardware, which requires in the limit to use cycle-accurate execution of a stream of instructions on a hardware model (emulator) or on a software one (simulator).

[1] Coding standards for embedded systems often require a finite-state programming paradigm anyway, due to the absence of virtual memory mechanisms.

2. fast execution of application code in a flexible analysis and debugging environment, which would require in the limit to compile and run the embedded software on a host workstation, somehow simulating its interaction with the hardware and with the environment.

A first class of co-simulation methods, proposed for example by Gupta *et al.* in [GJM92], relies on a single custom simulator for hardware and software. This simulator uses a single event queue, and a high-level, bus-cycle model of the target CPU.

A second class, described by Wilson *et al.* in [Wil94], by Thomas *et al.* in [TAS93], and by Rowson in [Row94], loosely links a hardware simulator with a software process. Synchronization is achieved by using the standard interprocess communication mechanisms offered by the host Operating System. One of the problems with this approach is that the relative clocks of software and hardware simulation are not synchronized, thus requiring the use of handshaking protocols. This may impose an undue burden on the implementation, e.g. if hardware and software do not need such handshaking since the hardware part in reality runs much faster than in the simulation.

A third class, described in [tHM93], keeps track of time in software and hardware independently, using various mechanisms to synchronize them periodically. If the software is master, then it decides when to send a message, tagged with the current software clock cycle, to the hardware simulator. If the hardware time is already ahead, the simulator may need to back up, which is a capability that few hardware simulators currently have. If the hardware is master, then the hardware simulator calls communication procedures which in turn call user software code.

A similar approach is used by Kalavade *et al.* in [KL92] and by Lee *et al.* in [LR93]. In both cases, the simulation and design environment Ptolemy (described in Section 3) is used to provide an interfacing mechanism between different domains. In [KL92] co-simulation is done:

- at the specification level by using a data flow model,
- at the implementation level by using a bus-cycle model of the target Digital Signal Processor and a hardware simulator, both built within Ptolemy.

In [LR93] the specification is simulated by using concurrent processes, communicating via queues. The same message exchanging mechanism is retained in the *implementation* (using a mix of micro-processor-based boards, DSPs, and ASICs), thus allowing to perform co-simulation of one part of the implementation with a simulation model of the rest. For example, the software running on the micro-processor can also be run on a host computer, while the DSP software runs on the DSP itself.

Our approach is different from those listed above, because:

1. it allows different components of the system to be scheduled independently, without requiring to write a custom environment,
2. software and hardware are simulated together in a unified environment, with the same debugging interface,

3. a bus-cycle model of the target processor is not required, yet a satisfactory level of accuracy is achieved.

The main emphasis of our work is on *speed*, both during simulation (a speed exceeding 1 million clock cycles per second can be easily achieved on a general-purpose workstation) and when the user changes some architectural parameters (changing the target processor or the hardware/software partition takes about 1 second).

The paper is organized as follows. Sections 2 and 3 (which are included just to provide the reviewers with some background, and will not be included in the final version) describe the *POLIS* co-synthesis system and the Ptolemy co-simulation and co-synthesis systems respectively. Section 4 describes our co-simulation methodology in detail. Section 5 shows with an example how co-simulation can be used to interactively evaluate the performance of various partitions of a system under various operating conditions. Section 6 concludes the paper and outlines opportunities for future research.

2 The *POLIS* co-design environment

Our co-simulation and trade-off evaluation method uses an existing co-design environment for reactive embedded systems, described in [CGH+94], for synthesizing software and hardware, and for analyzing their performance. The *POLIS* system is centered around a single Finite State Machine-like representation, which is well suited to our target class of control-dominated systems.

A Co-design Finite State Machine (CFSM), like a classical Finite State Machine, transforms a set of inputs into a set of outputs with only a finite amount of internal state. The difference between the two models is that the synchronous communication model of classical concurrent FSMs is replaced in the CFSM model by a finite, non-zero, a priori unbounded reaction time. Each element of a network of CFSMs describes a component of the system to be modeled. A CFSM consists of sets of inputs and outputs, and a transition function. Each transition is triggered by a set of input events and emits, after an unbounded non-zero time, a set of output events.

In other terms, each CFSM can be considered as composed of a "main" FSM, that represents the desired reactive behavior, surrounded by a set of one-place buffers modeling input detection and output reaction delays.

The design flow that is currently implemented in the *POLIS* system is depicted in Figure 1 and described more in detail below.

1. **High Level Language Translation** In the *POLIS* environment designers write their specifications in a high level language (e.g., ESTEREL [BCG91], StateCharts [DH89], subsets of Verilog or VHDL) that can be directly translated into CFSMs.

2. **Design Partitioning** Design partitioning is the process of choosing a software or hardware implementation for each CFSM in the system specification. A key

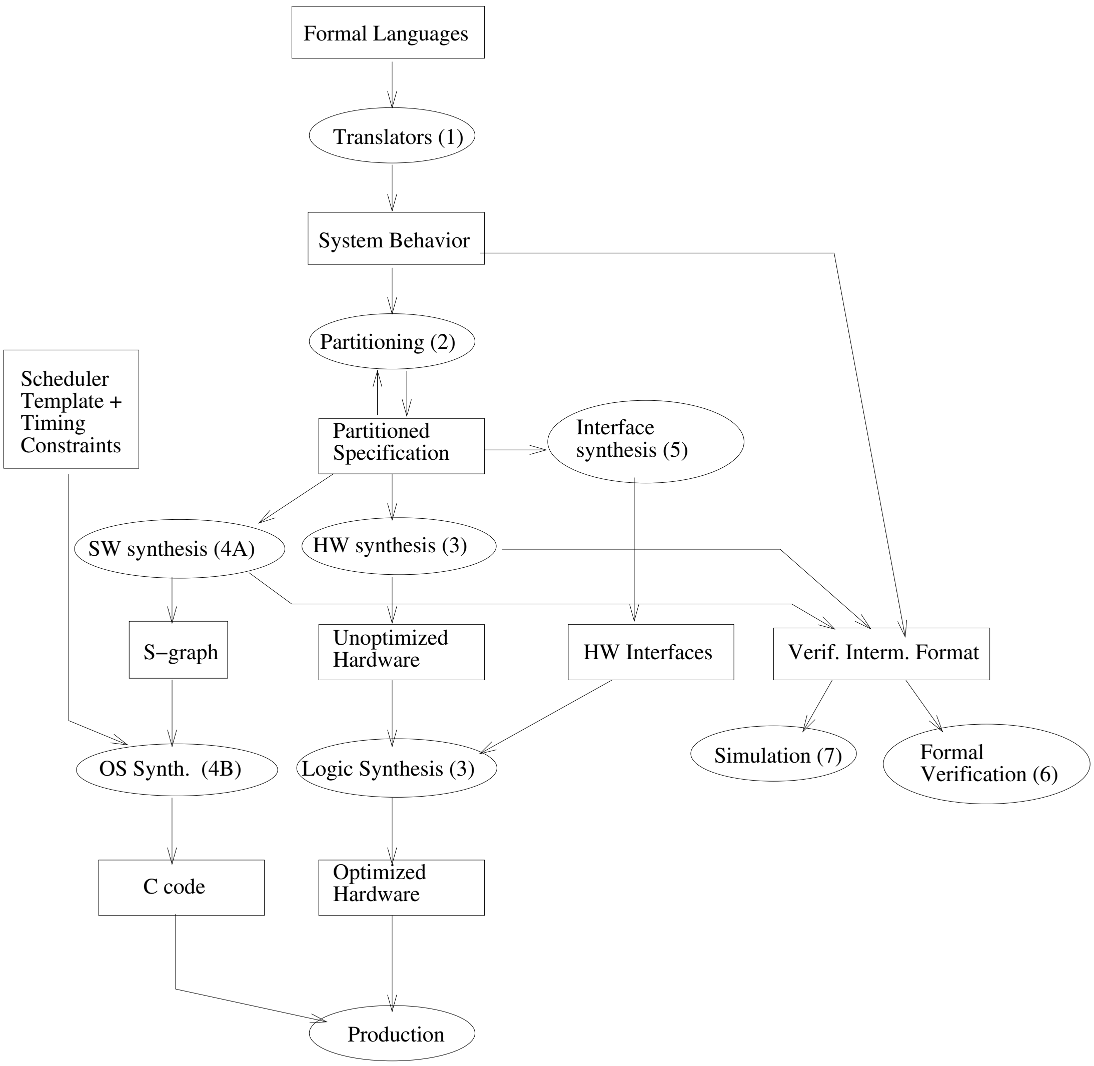

Figure 1: The *POLIS* system

point of this methodology is that the CFSM specification is implementation-independent, thus allowing the designer to interactively explore a number of implementation options.

3. **Hardware Synthesis** A CFSM sub-network chosen for hardware implementation is directly mapped into an abstract hardware description format, namely BLIF ([SSL+92]).

4. **Software Synthesis** A CFSM sub-network chosen for software implementation is mapped into a software structure that includes a procedure for each CFSM, together with a simple Real-time Operating System:

   (a) CFSMs. The reactive behavior is synthesized in a two-step process:

   - Implement and optimize the desired behavior in a high-level, processor-independent representation of the decision process similar to a control/data flow graph.
   - Translate the control/data flow graph into portable C code and use any available compiler to implement and optimize it in a specific, micro-controller-dependent instruction set.

   This methodology results in better optimization and tighter control of software size and timing than with a general-purpose compiler. A timing estimator quickly analyzes the program and reports code size and speed characteristics. The algorithm is similar to that used by [PS91], but requires no user input. It uses a formula, with parameters obtained from benchmark programs, to compute the delay of each node in the control/data flow graph for various micro-controller architectures (currently MIPS R3000 and Motorola 68HC11 and 68332). The precision of the estimator with respect to true cycle counting is currently on the order of $\pm 20\%$.

   The estimator is a key component of our *co-simulation* methodology, because it allows to obtain accurate estimates of program execution times on any characterized target processor, by:

   i. appending to each statement in the C code generated from the control/data flow graph instructions that accumulate clock cycles,

   ii. compiling and executing the software, as will be described more in detail in Section 4, on the host workstation.

   (b) Real-time Operating System. An application-specific OS, consisting of a scheduler and I/O drivers, is generated for each partitioned design. Currently *POLIS* allows the designer to choose from a set of classical scheduling algorithms (e.g. Rate-Monotonic and Deadline-Monotonic, [LL73]).

5. **Interfacing Implementation Domains** Interfaces between different implementation domains (hardware-software) are automatically synthesized within *POLIS*. These interfaces come in the form of cooperating circuits and software procedures (I/O drivers) embedded in the synthesized implementation.

6. **Formal Verification** The formal specification and synthesis methodology embedded within *POLIS* makes it possible to interface directly with existing formal verification algorithms that are based on FSMs. *POLIS* includes a translator from the CFSM to the FSM formalism which can be fed directly to verification systems ([BHJ$^+$96]).

7. **System Co-simulation** This step is used in a closed loop with system partitioning and depends on software synthesis for the generation of the executable models and the clock cycle estimates. It is described more in detail in Section 4.

3 The Ptolemy co-design environment

Ptolemy ([BHLM90, KL93, Buc93]) is a complete design environment for simulation and synthesis of mixed hardware/software data-dominated *embedded systems*. Here we will concentrate on its simulation aspects.

Ptolemy treats the system to be designed as a hierarchical collection of objects, described at different levels of abstraction and using different semantic models to communicate with each other:

- Each abstraction level, with its own semantic model, is called a "domain" (e.g., data flow, logic, . . .).

- Atomic objects (called "stars") are the primitives of the domain (e.g., data flow operators, logic gates, . . .). They can be used either in simulation mode (reacting to events by producing events) or in synthesis mode (producing either software code to be run on a target DSP, or an ASIC). Stars communicate by using point-to-point queues called "portholes".

- "Galaxies" are collections of instances of stars or other galaxies. Instantiated galaxies can possibly belong to domains different than the instantiating domain.

Each domain includes a scheduler, which decides in which order stars are executed (both in simulation and in synthesis). Whenever a galaxy instantiates a galaxy belonging to another domain (which is a case typical of co-simulation), Ptolemy provides a mechanism called a "wormhole" for the two schedulers to communicate. The simplest form of communication is to pass time-stamped events across the interface between domains, with the appropriate data-type conversion. A more complex form involves de-scheduling in one domain as a result of events in another one.

In particular, we used the Discrete Event (DE) domain of Ptolemy to implement the event-driven communication mechanism among CFSMs. This domain is event-driven, rather than data-driven as most other domains in Ptolemy, and hence seems the most appropriate for our purposes. Each star of the DE domain has input and output portholes (which correspond to CFSM inputs and outputs) carrying *events*. When a star receives an input event, it fires and can emit output events. Events that are read from the input portholes are deleted (consumed). Events are kept in a global queue, sorted according to their time stamp, that is the instant in which they are emitted by the star. Scheduling is computed at run time, since there is no way to decide a priori which star should fire in a particular instant. The scheduler picks the set of (simultaneous) events with the smallest time stamp from the queue and distributes them to the destination stars, firing them in turn.

4 Simulation of CFSMs within the Ptolemy environment

4.1 Co-simulation methodology

The Ptolemy scheduler in the DE domain fires stars as if they were executed concurrently. Thus it does not directly provide a way to simulate CFSMs implemented in software and running on a limited amount of computational resources (in this paper we assume that a *single CPU* is available). Our goal was to modify the DE scheduler behavior without changing its code, to maintain compatibility with the original version. Thus we let the scheduler fire stars in its own preferred order, but every star may or may not actually execute the main part of its code, based on *global* information which characterizes the shared resources. All this is accomplished in a transparent way so that the Ptolemy scheduler sees a world of concurrent stars, while the software stars see the *POLIS* scheduling policy.

Having met this goal, it is now possible to simulate in Ptolemy a system designed using *POLIS*: this is accomplished by generating proper descriptions of every CFSM and loading them into Ptolemy together with the network of interconnections. It is then possible, through the nice graphical interface provided by Ptolemy, to evaluate trade-offs between hardware and software solutions, and to visualize the overall system behavior.

The detailed design flow for the co-simulation and trade-off analysis phases is as follows:

1. The control/data flow graph of every CFSM in the system specification is built, and the corresponding C code is generated (as described in [CGH+95]). The C code also includes run time estimations for each C code statement, based on information derived from benchmark analysis of the target processor.

2. The Ptolemy language source code for every CFSM is generated. This in turn includes the C code generated by the previous step.

3. All CFSMs are loaded into Ptolemy. Each of them is a single star, with input and output portholes corresponding to CFSM inputs and outputs.

4. The network of interconnections between stars (CFSMs) is created in the Ptolemy environment.

5. Each star is assigned (by interactively modifying one of its properties or one of the properties of a galaxy above it in the hierarchy):

- an implementation, either software of hardware, and
- a priority, used by the software scheduler.

Hardware stars run concurrently and terminate in a single clock cycle. Software stars are mutually exclusive (we assume a single CPU here), and use the run time estimation to determine how long it takes to emit outputs and complete firing of a single transition of the CFSM.

6. A single system-wide parameter (also modifiable interactively) describes which one of the pre-characterized processors must be used for cycle counting. This provides a mechanism to easily change the target processor for a given set of simulation stimuli, without the need to re-analyze or re-compile the specification. In the same way it is possible to specify the scheduling policy best suited for the given application. If the scheduler is priority-based, then it can use the priority level assigned to each star.

7. The simulation is started with appropriate stimuli generators and output monitors, to check the behavior of the system. Multiple simulations can be compared to evaluate timing constraint satisfaction, run time, processor occupation, and other interesting pieces of information.

   For example, in Section 5 we will describe how the CPU utilization can be monitored, to detect and analyze potential overload conditions.

4.2 Scheduling policy implementation

Simulating *POLIS* CFSMs in Ptolemy with the *correct semantics* required the solution of two main problems:

- ensure that each star behaves correctly,
- ensure that scheduling of the software components guarantees mutual exclusion and avoids deadlock or starvation.

The former problem was easy to solve. The C code generated by *POLIS* is included in the Ptolemy star description. The *POLIS* RT-OS calls for event detection and emission are translated into porthole-based communication procedures. These procedures also flush the event queue in case multiple events are received at the same time and at the same input (only one randomly chosen event is considered). Overwritten events are discarded, thus implementing the one-place buffer communication method among CFSMs (Section 2).

Scheduling the software components proved to be much more difficult. The Ptolemy DE Scheduler assumes that a star can be fired each time it receives an event at an input. This is not exactly our case since:

- a CFSM may need a *set* of input events to fire in a given state, which means that inputs at the current time may need to be "saved" until later, and

- the computational resource (the processor) is shared among multiple stars, which means that any star which may be fired, can do so only when the processor is free.

The solution that we have chosen *does not require to modify the DE scheduler*. From now on, we will call *software scheduler* our own scheduling policy, implemented by an automatically generated procedure on top of the Ptolemy DE scheduler. Each simulation cycle (identified by an integer number, and directly corresponding to a simulated system clock cycle) is divided into three phases:

1. **Request phase**, in which all stars receive events, and request from the software scheduler access to the processor if it is ready to fire, re-scheduling themselves at the grant phase.

2. **Grant phase**, in which only one star is granted access to the processor and executes its user code, while all other enabled stars re-schedule themselves at the update phase. The identifier of the star which currently runs on the processor is kept in a shared variable called *star_ack*. This star also computes the time at which the processor will become available again, in a shared variable called *next*, by accumulating estimated clock cycles during the execution of user code and RT-OS calls.

3. **Update phase**, in which all enabled stars re-schedule themselves at the next time the processor is available.

The request phase is characterized by an integer simulation time, while the grant and update phases are characterized by fractional simulation times. All events are received and emitted at integer times.

Scheduling is thus performed in the following way. Whenever a software star receives an event:

- If the current time is greater than or equal to *next*, then the processor is available:
  - If the current time identifies a request phase, then the star sends the request to the software scheduler, and re-schedules itself at the next grant phase.
  - If the current time identifies a grant phase, then the star must check the variable *star_ack* to know if it has been chosen by the software scheduler. In this case, it executes the user-specified part of its own code, accumulating estimated clock cycles depending on the sequence of instructions that is executed, and sets the variable *next* according to the clock cycle estimate. Otherwise it must re-schedule itself again at the next update phase (to make sure that the selected star has had time to execute its code and update the variable *next*).
  - If the current time identifies a grant phase, then the star reads the variable *next* and re-schedule itself to try to get the processor at that time.

- If the processor is not available, but the priority of the star is greater than that of the currently executing star, and *the chosen scheduling policy allows interrupts* (i.e. it is pre-emptive), then an interrupt occurs. The variable *next* is incremented by the estimated execution time of the interrupting star, and the interrupted one is prevented from emitting output events until the end of the interrupt. Interrupts may be arbitrarily nested, and can only cause delays in the interrupted stars, without changing their behaviour (input variables to stars are buffered in our software implementation scheme, to improve the predictability of the system behavior).

- If the processor is not available and the priority of the star is less than or equal to that of the currently executing star, or if the software scheduling policy is non-pre-emptive, then the star must re-schedule itself at time *next*. Hence, stars with the same priority level do not interrupt each other.

The communication queues among stars are forced to hold at most one event, to match the CFSM communication model using one-place buffers. This means that events may be overwritten, if they are emitted twice without being detected. This is *legal* in the CFSM model of computation, but can optionally be *logged to a file*, because often losing events means violating timing constraints and is interesting for the designer.

Hardware stars run concurrently, and take only one unit of time to execute. They simply check, upon receiving an event, if they are still processing a previous event. In this case they re-fire at the next time unit.

The implementation (hardware or software) of each star can be dynamically and independently changed during a Ptolemy session, by just updating a parameter of the star or of a galaxy enclosing it in the hierarchy. Thus it is possible to experiment with different solutions in a very straightforward manner and in a short period of time. In fact, it is not necessary to rebuild the system, but it is sufficient to start a new simulation.

5 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 timing pulse generator, providing the time basis for the whole system.

It provides as outputs:

1. four Pulse Width-Modulated signals, directly driving a pair of gauges (vehicle and engine speed),

2. a 16-bit number controlling the odometer display,

3. an 8-bit number controlling the fuel level display.

Its operation is divided into various components:

- an event-triggered front-end, which converts pulses into revolution counts,

- a time-triggered back-end which converts revolution counts into car and engine speed, normalizes them, and presents them in the appropriate form required by the displays,

- a background task which updates the fuel gauge.

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 5000 pulses per second, while for the wheel this means up to 16 pulses per second. Outputs had to 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 (379 clock cycles at most for the most time critical task, the engine pulse recorder) showed that we could hope to satisfy the requirements with a 1 MHz processor.

The system was modeled using 13 CFSMs, each specified in ESTEREL. Their interconnection was described graphically with the Ptolemy user interface, as illustrated in Figure 2. One CFSM is devoted to input conversion from level to pulse, three CFSMs derive timing events from the base time reference, four CFSMs filter and normalize the data, one CFSM converts potentiometer readings to fuel level (taking into account the shape of the tank), and the remaining four CFSMs perform the PWM conversion. Scheduling was performed by hand, due to the simplicity of the system, resulting in an assignment of two priority levels to CFSMs. The highest 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.

We first tried to implement everything in software, then moved part of the components to a hardware implementation in order to satisfy the timing constraints. The PWM converters were an obvious choice for hardware implementation, because their low jitter constraints made a software implementation on a 68HC11 infeasible. Figure 3 shows the priority level of the star currently executed on the processor as a function of time[2]. The processor utilization is still fairly high, especially considering that the simulated car speed was about 50 Km/h. Hence we moved also the timing event generators to hardware (a typical choice in embedded systems, in which timing functions are often performed by special-purpose timers which are part of standard micro-controllers).

[2]A value of -1 means that the processor is idle, 1 is the highest level, each vertical bar represents a Conte switch among CFSMs with the same priority level.

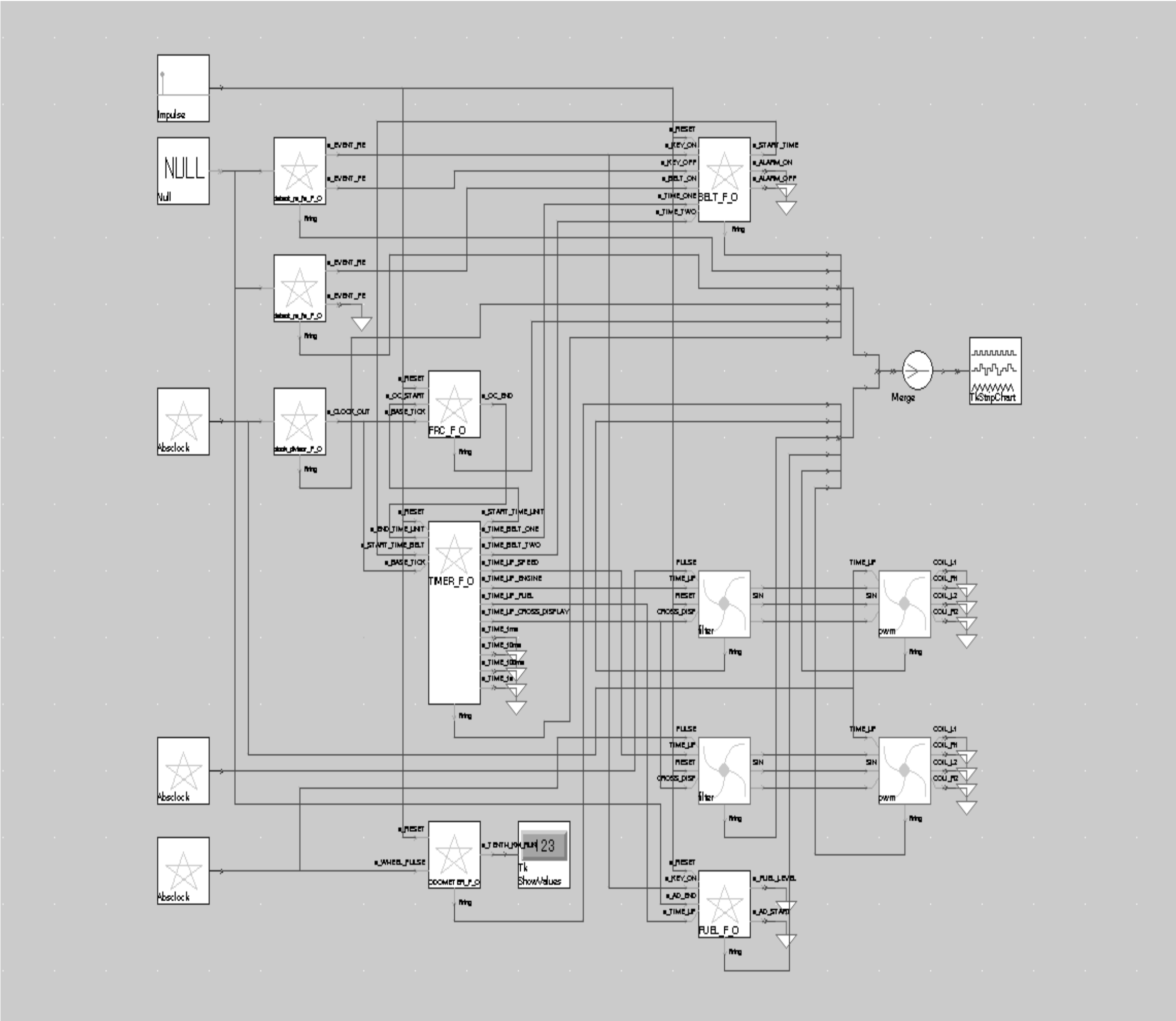

Figure 2: The dashboard controller netlist

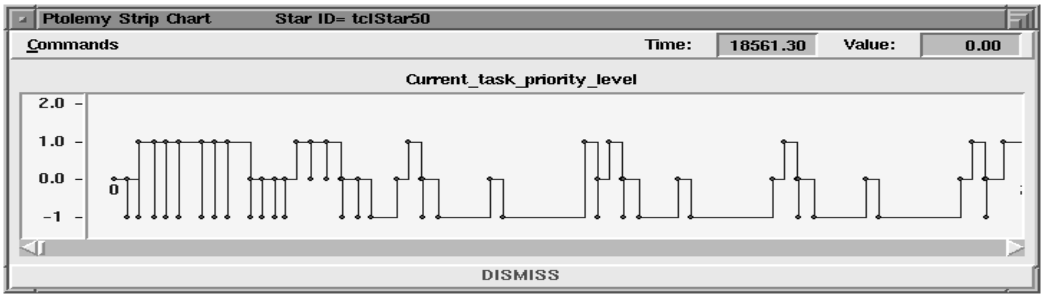

Figure 3: Processor utilization analysis with few hardware components

Figure 4 shows the priority level in this second case. Figure 5 shows the user interface, that uses the standard constant specification mechanism provided by Ptolemy, and allows 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. The dashboard example is ideal in this respect, because the highest frequency input events occur about once every 100 clock cycles (assuming a 1 MHz clock). The results for the dashboard simulation, using various types of architectural choices, are reported in Table 1. In this case, we used estimated execution times for a Motorola 68HC11 micro-controller (which was eventually chosen for the final implementation, reported in [CCG$^+$96]), with a 1 MHz clock speed, and a MIPS R3000, with a 1 MHz and a 10 MHz clock speed. The partitions shown in the column labeled "Part." are respectively:

- SW: all modules are in software,
- HW/SW: the PWM drivers and timing generators are in hardware, and the rest is in software,
- HW: all modules are in hardware.

The column labeled "Graph." shows when the graphical priority display (useful for debugging the software scheduler) is used. All CPU and User times are in seconds,

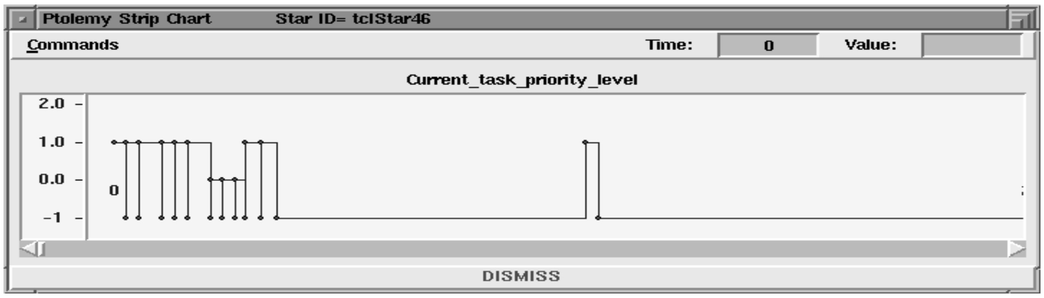

Figure 4: Processor utilization analysis with more hardware components

and were obtained on a SPARCSTATION 10 with 16 Mbytes of RAM, simulating 20 million clock cycles (except for the 10 MHz MIPS, for which 200 million cycles were simulated). The user time required to restart a simulation when the partition or the target processor are changed is about 1-2 seconds.

The execution time for the all-software partition on the 68HC11 is very high because the processor does not meet the timing constraints, and hence the simulation is interrupted very often to log the information about missed deadlines on a file.

The simulation performance (without graphics output, because the display time drastically slows down the system) is around 1 million clock cycles per second. This speed, which can be achieved thanks to the extremely low overhead imposed by our cycle counting technique, is sufficient in many cases to run simulations almost at the same speed as the real target system (*virtual prototyping*).

6 Conclusions and future work

In this paper we have shown that fast co-simulation can be done at 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 a powerful co-simulation environment built in the Ptolemy system.

We noticed, by profiling the simulator code, that over 90% of the time (when the graphic output is not used) is spent executing the DE scheduler code. This means that a faster simulator could be obtained by re-writing the Ptolemy DE Scheduler to take into account the required behavior of CFSM stars, and eliminate the overhead introduced by the re-firing method. On the other hand, this option may not be desirable for reasons of compatibility, both with future versions of Ptolemy, and with other simulation domains within Ptolemy.

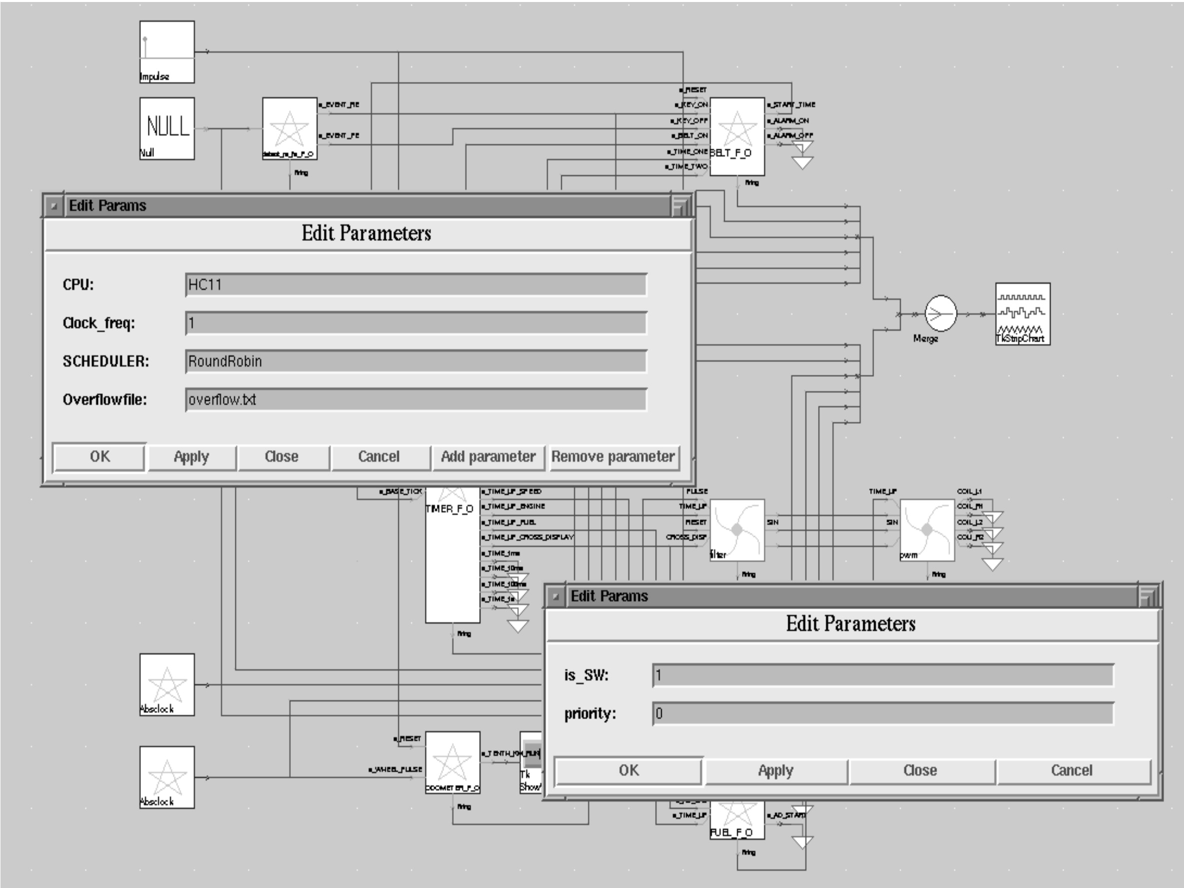

Figure 5: The user interface

Target Proc.	MHz	Part.	Graph.	Time CPU	Time User
HC11	1	SW	No	> 1000	> 1000
HC11	1	HW/SW	No	30	38
HC11	1	HW	No	25	27
MIPS	1	SW	No	161	162
MIPS	1	HW/SW	No	23	25
MIPS	1	HW	No	25	26
MIPS	10	HW/SW	No	23	23
HC11	1	HW/SW	Yes	202	205
MIPS	1	HW/SW	Yes	258	270

Table 1: Simulation speed for various types of system partitions

In the future, we would like to allow the designer to create more than one software partition, thus simulating multiprocessor environments, and to specify hand-estimated execution times for software modules that were not synthesized using POLIS (e.g., data-intensive modules designed using Ptolemy).

References

[BCG91] G. Berry, P. Couronné, and G. Gonthier. The synchronous approach to reactive and real-time systems. *IEEE Proceedings*, 79, September 1991.

[BHJ+96] F. Balarin, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal verification of embedded systems based on CFSM networks. In *Proceedings of the Design Automation Conference*, 1996.

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

[BRX93] E. Barros, W. Rosenstiel, and X. Xiong. Hardware/software partitioning with UNITY. In *Proceedings of the International Workshop on Hardware-Software Codesign*, October 1993.

[Buc93] J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, U.C. Berkeley, 1993. UCB/ERL Memo M93/69.

[CCG+96] S. Cardelli, M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. Rapid-prototyping of embedded systems via reprogrammable devices. In *7th IEEE International Workshop on Rapid System Prototyping*, 1996.

[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.

[DH89] D. Drusinski and D. Har'el. Using statecharts for hardware description and synthesis. *IEEE Transactions on Computer-Aided Design*, 8(7), July 1989.

[GJM92] R. K. Gupta, C. N. Coelho Jr., and G. De Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proceedings of the Design Automation Conference*, June 1992.

[HDMT94] X. Hu, J.G. D'Ambrosio, B. T. Murray, and D-L Tang. Codesign of architectures for powertrain modules. *IEEE Micro*, 14(4):48–58, August 1994.

[HEHB94] J. Henkel, R. Ernst, U. Holtmann, and T. Benner. Adaptation of partitioning and high-level synthesis in hardware/software co-synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, November 1994.

[KAJW93] S. Kumar, J. H. Aylor, B. Johnson, and W. Wulf. Exploring hardware/software abstractions and alternatives for codesign. In *Proceedings of the International Workshop on Hardware-Software Codesign*, October 1993.

[KL92] A. Kalavade and E. A. Lee. Hardware/software co-design using Ptolemy – a case study. In *Proceedings of the International Workshop on Hardware-Software Codesign*, September 1992.

[KL93] A. Kalavade and E.A. Lee. A hardware-software codesign methodology for DSP applications. *IEEE Design and Test of Computers*, 10(3):16–28, September 1993.

[KL94] A. Kalavade and E.A. Lee. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *Proceedings of the International Workshop on Hardware-Software Codesign*, 1994.

[LL73] C.L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46 – 61, January 1973.

[LR93] S. Lee and J.M. Rabaey. A hardware-software co-simulation environment. In *Proceedings of the International Workshop on Hardware-Software Codesign*, October 1993.

[PS91] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24(5):48–57, 1991.

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

[SSL+92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, U.C. Berkeley, May 1992.

[TAS93] D.E. Thomas, J.K. Adams, and H. Schmit. A model and methodology for hardware-software codesign. *IEEE Design and Test of Computers*, 10(3):6–15, September 1993.

[tHM93] K. ten Hagen and H. Meyr. Timed and untimed hardware/software cosimulation: application and efficient implementation. In *Proceedings of the International Workshop on Hardware-Software Codesign*, October 1993.

[VG92] F. Vahid and D. G. Gajski. Specifcation partitioning for system design. In *Proceedings of the Design Automation Conference*, June 1992.

[Wil94] J. Wilson. Hardware/software selected cycle solution. In *Proceedings of the International Workshop on Hardware-Software Codesign*, 1994.