

# Hardware-Software Codesign of Embedded Systems

Designers generally implement embedded controllers for reactive real-time applications as mixed software-hardware systems. In our formal methodology for specifying, modeling, automatically synthesizing, and verifying such systems, design takes place within a unified framework that prejudices neither hardware nor software implementation. After interactive partitioning, this approach automatically synthesizes the entire design, including hardware-software interfaces. Maintaining a finite-state machine model throughout, it preserves the formal properties of the design. It also allows verification of both specification and implementation, as well as the use of specification refinement through formal verification.

**Massimiliano Chiodo**

**Paolo Giusto**

**Attila Jurecska**

**Magneti Marelli**


**Harry C. Hsieh**

**Alberto Sangiovanni-Vincentelli**

*University of California,  
Berkeley*

**Luciano Lavagno**

*Politecnico di Torino*

 Current approaches to the hardware-software codesign problem fall short, we believe, on one of two counts. The formal model they define is not abstract enough for use as an implementation-independent representation during the system design process. Otherwise, the model is not sufficiently detailed for efficient synthesis as a mix of hardware and software components.

Our formalism, an extension of classical finite-state machines called Codesign Finite State Machine (CFSM), is equally expressive for hardware or software implementations of control-dominated systems. It is also formally defined, hence we can directly use it to verify properties that all implementations will share.<sup>1</sup>

For our purposes, we take hardware-software codesign to mean the design of a special-purpose system composed of a few application-specific integrated circuits that cooperate with software procedures on general-purpose processors. A single, well-defined purpose at a definite point in time and a generally long lifetime also characterize such embedded systems. Development of these systems cannot proceed by trial and error after deployment, so designers must optimize them as completely as possible during early design phases.

Though restricted, this definition is still too broad to allow a useful formalization of generally applicable automated design methodologies.

Embedded controllers serve in everything from portable compact-disc players to the navigation control units of battle aircraft. Consequently, we limit our present focus to relatively small, real-time control systems composed of software on one (or few) microcontrollers and some semi-custom hardware components.

Ignored here as well are large systems requiring the coordination of many boards and hundreds of thousands of lines of code. Nor do we directly address computation-dominated tasks, such as robotics and vision, that require, for example, digital signal processors or powerful general-purpose computers. Typical applications of our proposed methodology include automotive electronics (as the accompanying box describes) and household appliance control (from elevators to microwave ovens).

Some current approaches to the codesign problem concern methods to implement software programs in hardware. For example, designers have translated various flavors of Hoare's Communicating Sequential Processes (CSP) into synchronous or asynchronous circuits. Languages for real-time software specification, such as Esterel,<sup>2</sup> StateCharts,<sup>3</sup> or some modification of C<sup>4</sup> have served directly or indirectly as hardware description languages. Conversely, investigators have proposed methods to implement hardware specifications in software.<sup>5</sup> Some research has focused on particular aspects of

## Automotive electronics

One application field deriving the greatest gains from the coming of age of codesign techniques is automotive electronics. Here, high production volumes go hand in hand with a demand for high quality and reliability, low costs and maintenance needs, and short time-to-market. The end product—usually a box with some connectors—must meet stringent physical constraints, such as weight and size.

The demand for embedded controllers in this application has steadily increased for the past few years, both in terms of the range of applications and the sophistication of the functions performed. In virtually all vehicles manufactured since the mid 1980s, an electronic device—the engine control unit (ECU)—controls fuel injection and ignition. Most medium- to high-range cars come equipped with microcontroller-based dashboards. Also electronically controlled and monitored are automatic transmissions, air-conditioning systems and, recently, shock absorbers. The boom of safety devices, such as air bags and anti-lock braking systems (ABS), has boosted the demand for in-vehicle electronics.

The increase in the number and complexity of electronic devices in vehicles is affecting the way designers conceive the entire automobile. For example, a high-range car equipped with a rich set of options can have over 100 wires tied to a dashboard (including 30 to 35 wires entering each front door), for a total length of about 3 miles and a weight of about 100 pounds (Mercedes S-series and Renault Safrane). Effectively replacing such a bundle of hard-wired devices is a local area network implemented, usually, with

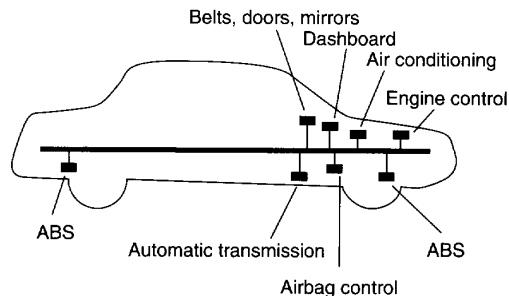


Figure A. An on-board local area network.

a serial bus (see Figure A). This bus replaces the numerous wires, allocating the functions to CPUs that no longer need be topologically close to the controlled part, such as a window or mirror. Compensating for the added cost of the electronics are savings in material (wires and connectors), manufacturing time, and reliability of the links.

The complexity of these functions and the precision dictated, for example, by the exhaust emission control laws, require the use of specialized fast microcontrollers. Such controllers, like the Motorola MC68332, feature 32-bit architectures (versus the 8 or 16 bits of older microcontrollers) along with specialized microprogrammable processors (time processing units) to handle hard real-time tasks more easily.

hardware-software cooperation, such as design of interfaces between hardware and software components<sup>6</sup> or formal specification of hardware-software system properties.<sup>7</sup>

Our basic model, by contrast, is a network of interacting CFSMs that communicate through a very low-level primitive: events. A CFSM, or the environment in which the system operates, emits events that one or more CFSMs or the environment can later detect. This scheme assumes a broadcast communication model, rather than point-to-point channels as in CSP.

Note that events are quite similar to the valued-token model used to model computation-intensive applications;<sup>8</sup> events can also serve to interface the two domains. CFSMs also bear some resemblance to behavioral finite-state machines,<sup>9</sup> because both are based on finite-state causal reaction. Our model, however, offers a more flexible communication mechanism, by using events with an arbitrary propagation time, rather than instantaneous broadcast.

Events directly implement a communication protocol that

does not require an acknowledgment. The receiver waits for the sender to emit the event, but the sender can proceed immediately after emission. An implicit one-place buffer between the sender and each receiver saves the event until it is detected or overwritten. This approach lends itself to an efficient hardware implementation with synchronous circuits, as well as a software implementation, either polling- or interrupt-based. If required, our approach can easily model an explicit full handshake mechanism in terms of event exchange.

The notion of communication our proposed model uses implies that the sender does not remove the event immediately after emitting it, but only when emitting another one. Each CFSM can detect an event at most once any time after the event's emission, until another event of the same type overwrites it. Thus, the event can be correctly received even with the rather unpredictable reaction times associated with a software implementation. Correct reception is ensured as long as either the sender data rate is lower than the receiver

### Applying our methodology

As an example of the application of our methodology, let's look at a simplified subsystem of a car dashboard. The functions considered here are the odometer and the speedometer. The system is structured as follows: a proximity sensor placed near the wheel shaft sends a pulse to the dashboard when an indentation goes by it. The dashboard then

- measures the instantaneous speed by counting wheel pulses in a given time interval;
- filters the speed value to improve resolution and reduce sensitivity to noise;
- drives a pulse width-modulated signal proportional to the value of the filtered speed; accumulates speed pulses and every *K* pulses refreshes the odometer display.

Figure B depicts a data-flow diagram of the system. Dashed arrows represent pure events. Solid lines represent data flows. Rectangles represent memories. Ellipses represent transformations. We can also see the diagram as an interconnection of CFSMs, where data flows are represented by valued events, and the memories are implicitly implemented by the input and output buffers of the CFSMs. We omit the details of the CFSM behavior for reasons of space.

The synthesis system we are describing produced various choices of implementation for each CFSM. Table A describes the cost of a hardware implementation of each CFSM (separately optimized) in terms of square microns in a 3- $\mu$ m technology. The delay of this implementation is almost negligible, and is supposed to be the same as the clock cycle of the microcontroller (a reasonable choice in an embedded system). The other tables describe the cost in terms of memory occupation (bytes) and execution time (average clock cycles over a set of random inputs) of a software implementation of the CFSMs, excluding the scheduler and I/O drivers, on a Motorola 68HC11 microcontroller. Table B refers to a straightforward implementation of each CFSM as a logic expression for each output and state variable.

**Table A. Cost of hardware implementation.**

CFSM	Area ( $\mu\text{m}^2$ )
AcqSpeed	163,792
OdoAcq	31,088
OdoDisplay	59,392
PWMDriver	31,088
StackFilter	128,064

**Table B. Cost of direct software implementation of hardware function.**

CFSM	Size (bytes)	Time (cycles)
AcqSpeed	117	130
OdoAcq	59	82
OdoDisplay	120	181
PWMDriver	177	250
StackFilter	104	152

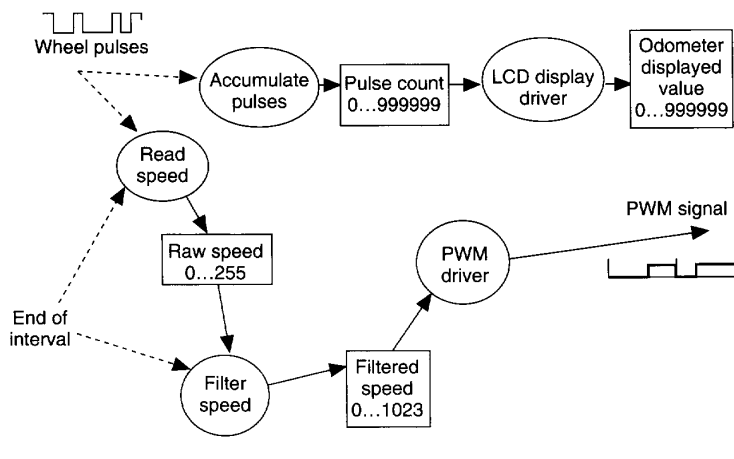


Figure B. Dashboard flowchart.

er processing ability, or the designer explicitly introduces some possibly complex form of synchronization between the two, using events as basic building blocks.

In our methodology, we use a discrete model of time, in

### Applying our methodology (continued)

Tables C and D refer to two different s-graph-based implementations of the same behavior. The former uses multivalued tests (implemented as switch statements, hence the large size of one case that required a 256-way jump). The latter uses binary tests on individual bits of each variable. No method has a clear advantage over the other in all cases, so both strategies should be part of the design toolbox.

To evaluate the trade-off between hardware and software implementations of this subsystem, we can estimate the RAM/ROM occupation of the implementation in Table D at around 100,000- $\mu\text{m}$  (in a 0.5- $\mu\text{m}$  technology). Hence it is about eight times smaller (neglecting the microcontroller area, since the microcontroller is supposed to be required by other functions anyway), but also approximately 100 times slower than the hardware implementation.

**Table C. Cost of multiway s-graph implementation.**

CFSM	Size (bytes)	Time (cycles)
AcqSpeed	62	32
OdoAcq	33	27
OdoDisplay	55	82
PWMDriver	581	77

**Table D. Cost of bit test s-graph implementation.**

CFSM	Size (bytes)	Time (cycles)
AcqSpeed	55	45
OdoAcq	36	34
OdoDisplay	97	107
PWMDriver	94	83
StackFilter	63	40

which each computing element takes a nonzero unbounded (at least before an implementation is chosen) time to perform its task. This model is quite realistic for synchronous systems and lends itself to efficient formal verification techniques.<sup>1</sup>

The synthesis methodology uses two auxiliary models, derived from CFSM specifications, to describe a hardware and a software component. The first model is a standard logic netlist used by logic synthesis systems. The second model, which is new, is an abstraction of the basic instructions of a very simple computer model, called an s-graph (for software graph).

Since the s-graph is much simpler than a full-blown programming or assembly language, it lets us perform optimizations that real compilers and instruction schedulers could not easily do. They must solve a much more difficult and general optimization problem. Our approach then can map the s-graph into a high-level or assembly language implementation for a specific microcontroller, compile it, and load it.

Completing the methodology is a validation paradigm allowing us to verify that a synthesized design satisfies its specification. We use formal verification to debug both the specification with respect to high-level properties—safety constraints—and the implementation with respect to lower-level properties—timing constraints. It can also help the designer fix errors and try alternate solutions, by providing error traces that describe the reason for failing to satisfy a desired property. Designers can use simulation to complement verification, thus quickly ruling out special cases that are considered potentially troublesome by formal verification, but that are actually impossible in the specified operating conditions. (For an example of this approach in action, see the accompanying Applying our methodology box.)

### Codesign finite-state machines

A CFSM, like a standard FSM, 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 standard definition of concurrent FSMs implies that all the FSMs change state exactly at the same time. On the other hand, a software implementation of a set of FSMs generally interleaves them in time. Hence replacing the synchrony hypothesis (which is often quite satisfactory for synchronous hardware) in our model is a finite, nonzero, a priori unbounded reaction time.

In this article, we describe the set of assumptions that we chose to add to this basic intuition to obtain a powerful general model for control-dominated reactive systems. For a more extensive treatment, see Chiodo et al.<sup>10</sup>

Suppose we want to specify a simple safety function of an automobile: a seat belt alarm. A typical specification a designer receives would be: "Five seconds after the key is turned on, if the belt has not been fastened, an alarm will beep for ten seconds or until the key is turned off." We can represent the

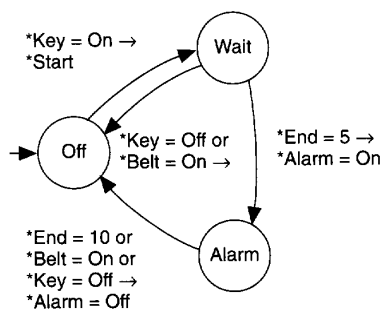


Figure 1. Codesign finite-state machine specification of a simple system.

specification in a reactive finite state form as shown in Figure 1. Input events, such as the fact that the key has been turned on or that a 5-second timer has expired, trigger reactions, such as the starting of the timer or the beeping of the alarm.

The basic observable entities defining the behavior of the system that we want to model are events. Sequences of time-stamped events are timed traces; defining the behavior of the system are the set of timed traces that can be observed when it interacts with the environment. Modeling the system itself and possibly its environment will be a set of CFSMs that produce those traces.

Identifying an event are a name (such as, \*Key), a value from a finite set (such as On, Off), and a time of occurrence. We also refer to the event-name, set-of-values pair as an event type. Some events, such as the user hitting the reset button, may not have an interesting value.

Each element of a network of CFSMs describes a component of the system to be modeled. (For the sake of clarity, we give the definition here in terms of a flat view, even though both the methodology and its implementation support hierarchical decomposition.) A CFSM consists of sets of input and output event types (the latter with an optional initial value), and a transition relation (a set of cause-reaction pairs, in which each cause is a set of event names and values and each reaction is also a set of events and values). Triggered by the input events, each transition emits, after an unbounded nonzero time, the output events. (Response bounds must be assumed when we want to verify time-dependent constraints of the system, as we will discuss later.) The state of the CFSM consists of the set of those event types that are at the same time input and output for it. The non-zero reaction time provides the storage capability required to implement the concept of state.

For example, in Figure 1, the fact that the state event has value Off and that input event \*Key occurs with value On causes the state event value to become Wait and the value-

less event \*Start to be emitted. We model causality by a relation rather than a function, because the specification may be nondeterministic. We can use this to abstract implementation details in the early phases of a design or during validation, or to model the unpredictable behavior of the environment.

Defining the behavior of the system is the evolution of the CFSM network in time, governed by a set of rules that define valid timed traces of a network of CFSMs. Such rules ensure that the transitions of each CFSM are atomic; all output events of a given transition must be emitted (not necessarily at the same time) before the next transition can occur. Note that a CFSM may ignore events, if it is not ready to accept them when they occur. Our methodology requires only that reactions to successive events with the same name be ordered, thus ensuring that one-place buffers can serve to implement events.

The behavior defined by a CFSM network does not assume fairness per se, because this would impose too tight a constraint on the software implementation. (Loosely speaking, fairness means that each CFSM that is enabled to react will do so within a finite amount of time.) However, a suitable software scheduler may impose fairness in practice.

For showing implementation correctness and performing design validation, we can also interpret the behavior of a CFSM network as a (less compact) network of nondeterministic FSMs. Each CFSM corresponds to a main FSM, which represents the desired reactive behavior, surrounded by a set of FSMs (one for each input and output event type) implementing one-place buffers modeling input (detection) and output (reaction) delays.

### Synthesis of a CFSM network

Figure 2 represents our complete framework for hardware-software codesign. Before describing the system validation approach, we need to briefly discuss the design flow through the synthesis portion of the framework.

**Specification language translation.** Due to its relatively low-level view of the world, the CFSM model is not meant to be used directly by designers. They will conceivably write their specifications in a higher level language—Esterel, StateCharts, or a subset of VHDL<sup>1</sup>—that directly translates into CFSMs.

These languages or language subsets all share a common zero-delay hypothesis, also called the perfect synchrony hypothesis. We assume that the system reacts infinitely fast to environmental stimuli. This allows us to eliminate all internal communication between interacting system components, and produces very fast software implementations, at the expense of relatively large code size.<sup>2</sup>

Our CFSM model comes into play after this collapsing of functions into a single reactive block has produced the desired level of granularity. We take into account the non-zero response time typical of a software implementation of the reaction, and study how an interconnection of such reactive blocks, each represented by a CFSM, behaves. In this

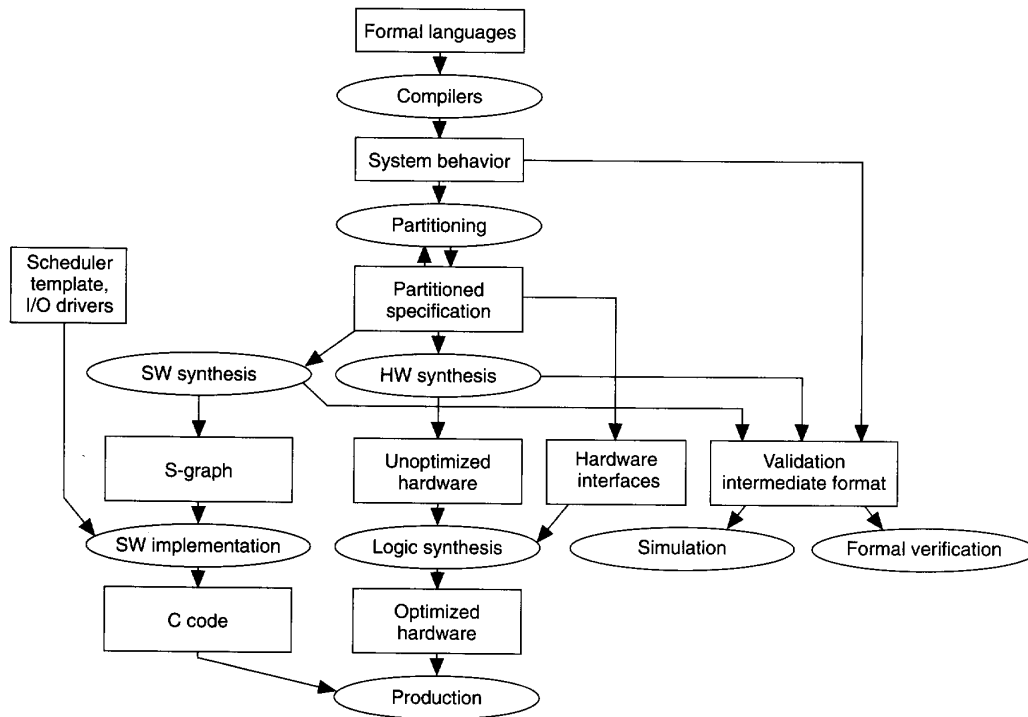


Figure 2. Codesign framework.

sense our proposed methodology is orthogonal to compilation issues for synchronous languages.

**Design partitioning.** The second step of the design process is design partitioning, that is, choosing the software or hardware implementation for each component of the system specification. The CFSM specification is totally implementation-independent in our approach, a key point that allows designers to experiment with a number of implementation options.

This article does not address directly the issue of automated partitioning; rather it describes a framework where algorithms to solve it can be transparently embedded.

**Hardware synthesis.** The third step of the proposed design process is to implement each CFSM in the chosen style. The synthesis algorithms we propose are based on restrictions common to the design of most industrial embedded control systems:

- Each hardware partition is implemented as a fully synchronous circuit.
- Each software partition is implemented as a C stand-

alone program (with an operating system skeleton that includes the scheduler and I/O drivers) embedded in a microcontroller.

- All partitions have the same clock.

In the case of hardware synthesis, we map a CFSM into an abstract hardware description format. We implement<sup>12</sup> each transition function with a combinational circuit, latching the circuit outputs to ensure the nonzero reaction delay.

At a given point in time, the significant information elements regarding an event name are its presence or absence, and its optional value. A straightforward hardware mapping of this model uses a wire that has value 1 in all the clock cycles when an event with that name is present and 0 when it is absent. (That is, if the signal stays at 1 for  $n$  cycles, there are  $n$  successive events with the same name.) The value, if present, is encoded on a bundle of auxiliary wires. Unlike other communication paradigms used originally for software specifications—channels or rendezvous—this communication scheme imposes almost no overhead due to event detection in hardware.

**Software synthesis.** We map a CFSM subnetwork into a software structure that includes a number of procedures and a simple operating system.

*CFSM implementation.* A two-step process implements the reactive behavior:

- Implementing and optimizing the desired behavior in a high-level, technology-independent representation of the decision process (called s-graph).
- Translating the s-graph into portable C code and using any available compiler to implement and optimize it in a specific, microcontroller-dependent instruction set.

The methodology-specific, processor-independent first step allows a much broader exploration of the design space than a general-purpose compiler can generally achieve. We can take advantage of the fact that CFSMs compute a finite state reaction to a set of events, which is much simpler to optimize than a generic high-level program. The second step, conversely, allows us to capitalize on predeveloped, microcontroller-specific optimizations such as register allocation or instruction selection and scheduling.

Another major advantage of our proposed approach is a much tighter control of software cost than generally possible with a general-purpose compiler. For this discussion, the cost of a software program is a weighted mixture of code size and execution speed, as real-time reactive systems usually have precise memory occupation as well as timing constraints. With these requirements, the s-graph needs to be detailed enough to make prediction of code size and execution times easy and accurate. On the other hand, it must be high-level enough for easy translation into various dialects of programming languages for the target microcontrollers (whose interpretation even of a standardized language such as C may vary widely).

Hence the s-graph is a reduced form of the control-flow graphs used in compiler technology (see, for example, Aho, Sethi, and Ullman<sup>13</sup>). As such, it is amenable to the standard set of optimizations done by compilers, plus some specific ones.

An s-graph is a directed acyclic graph (DAG), containing Begin, End, Call, Test, and Assign vertices. It is associated with a set of finite-valued variables, corresponding to the input and output events of the CFSM it implements.

An s-graph has one source vertex Begin, and one sink vertex End. Each Test vertex  $v$  has two children,  $true(v)$  and  $false(v)$ . (We use the limit of two children for the sake of explanation only. The implementation caters to multiway branching.) Each Begin or Assign vertex  $v$  has one child  $next(v)$ . Each Call vertex  $v$  has two children,  $sub(v)$  and  $next(v)$ . Any nonroot vertex can have one or more parents. A label associates each Test vertex with a Boolean-valued function that determines which child is traversed. It also associates each Assign vertex with a CFSM output variable and

with a function defined on the s-graph variables.

The semantics of the s-graph is simply defined by a traversal from Begin to End. The s-graph execution process visits the next vertex at every step, except for the Call and Test vertices. Every time it reaches an Assign vertex, it evaluates the Assign vertex's associated function and assigns its value to the labeling variable. Every time it reaches a Test vertex, it evaluates the function and visits the true or false child. Every time it reaches a Call vertex, it visits the sub vertex of the Call vertex, traverses until the End vertex, then visits the next vertex of the last visited parent Call (no recursion is allowed).

Figure 3 shows an example of a simple s-graph, computing the transition function of Figure 1 (variable  $S$  denotes the CFSM state).

We deal with speed and size requirements at the s-graph level. The components of the cost function we consider are:

- the total number of vertices, which is related to the program code (ROM) size,
- the maximum distance between Begin and End vertices, which is related to the execution time, and
- the number of variables, which is proportional to the data size (RAM, including CPU registers).

Speed optimization makes the s-graph as flat as possible without exceeding a bound on the number of vertices.

Driving the software synthesis procedure is a cost estimation done on the s-graph. We use appropriately chosen benchmarks to obtain a cost (code/data size and time) estimation for the various software constructs corresponding to s-graph vertices in various combinations.

We can perform local optimization by collapsing groups of s-graph vertices with one entry point and two exit points into a single Test vertex, whose label we can obtain by function composition. We can estimate the potential gain of this collapsing as the reduction in code size and execution time due to the use of Boolean operations (used to compute the function) rather than tests and jumps.

Finally, we can perform more global optimizations across multiple CFSMs, because composing CFSMs together can reduce the execution time and RAM occupation, thus eliminating the variables used for communication.<sup>2</sup>

*Real-time operating system.* The customized operating system for each microcontroller consists of a scheduler and drivers for the I/O channels. Hence it is extremely small and imposes little overhead, compared with standard operating systems for real-time applications.

To correctly implement the CFSM behavior, the operating system must satisfy the following constraints:

- Each transition of a task must be performed atomically; that is, the values of the input event buffers for that task must not change once it has been started.

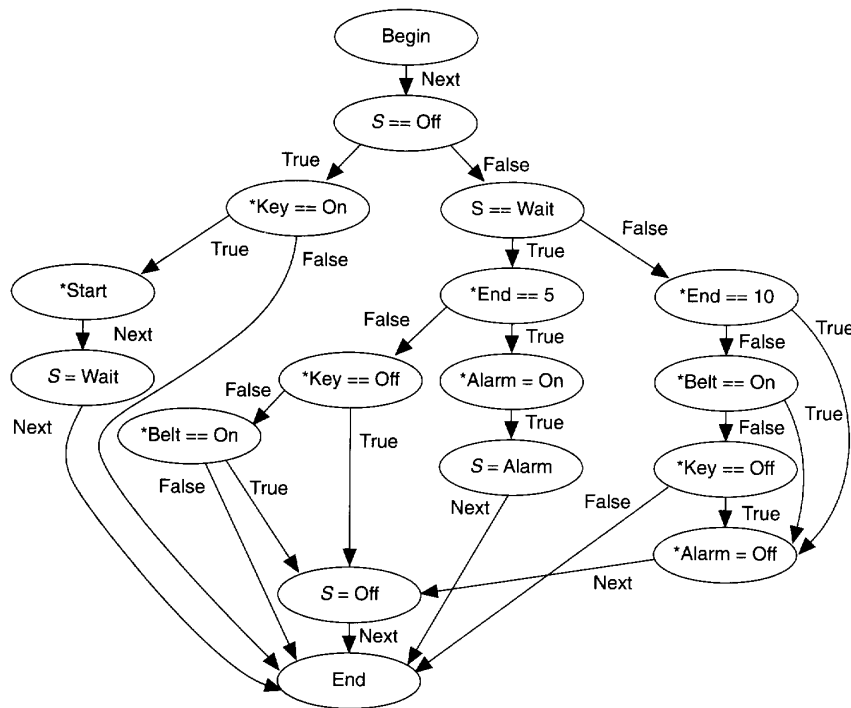


Figure 3. An s-graph implementing the seat belt system.

- The operating system must reset consumed events before invoking a task again.
- The operating system must transfer all output events from a task by setting the signal or variable denoting the event presence only after it has updated the corresponding value.

This scheme lends itself both to polling and interrupt implementations of event detection, as it can suspend and resume an active task at any time. We are still investigating means of specifying real-time constraints, and scheduling algorithms used to satisfy them.

Handling the communication mechanism between a task and the external world—other tasks or external devices—are two layers of software services:

- a general event-driver layer that implements the CFSM event emission/detection primitives, and
- a microcontroller-specific peripheral-driver layer that interfaces the software tasks with the physical I/O channels such as parallel I/O ports, serial ports, and analog-to-digital and digital-to-analog converters.

The latter, which is generally implemented via memory-mapped peripherals, also constitutes the software side of the interfacing mechanism we describe later.

**Modeling data flow.** Although general in terms of expressiveness, CFSMs are not specifically designed for computation-intensive tasks, but only for control-dominated ones. The idea is that a CFSM specifies the reactive part of the behavior, whereas standard functions can specify the details of the algorithms associated with the actions invoked. This model corresponds to the classic data/control dichotomy both compilers and high-level synthesis use. Libraries thus are made available that provide standard components—adders, counters—already mapped in hardware and software. Designers can incorporate netlists in the hardware implementation and optimize them with it. Similarly, software macros can occur as Test and Assign vertex labels in the s-graph; we must provide cost estimates for them to the s-graph optimizer.

**Interfacing implementation domains.** Event emission and detection are implemented differently in each domain, so we need an interfacing mechanism.

Conceptually, we can think of an interface mechanism as



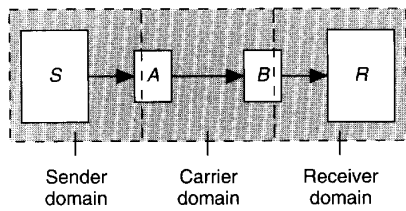


Figure 4. Interface between heterogeneous domains.

a three-layer block (see Figure 4 and Chou, Ortega, and Borriello<sup>6</sup>). A carrier here can be, for example, a printed circuit board track or a bus. Each block acts as a translator between representations in different domains. For example, Figure 5a describes the A and B blocks of a hardware-to-hardware interface (a simple wire). Figure 5b describes the B block of a software-to-hardware interface. Figure 5c describes the A block of a hardware-to-software interface. The software counterparts are embedded in the real-time operating system (for example, as an interrupt handler or an output port driver).

**System validation**

Partially driving the CFSM model was our desire to use formal verification techniques, which essentially involve proving mathematically that a certain formally specified property is true of a design.

Among the several methods proposed for formal verification, which is best suited for the task at hand, was an approach based on modeling the system as a network of FSMs. We can formally define the time behavior of a CFSM in terms of an equivalent FSM network. Any implementation of the CFSM network exhibits a behavior (allowed sequences of events) contained in the behavior of its specification.

Verification, for example, involves checking whether an undesirable behavior, expressed as a timed sequence of events  $\sigma$ , is consistent with the specification. If  $\sigma$  cannot occur in the specification, neither can it occur in a correctly derived implementation; the latter defines a subset of behaviors of the former. Examples of properties of our seat-belt system are "The alarm will not be on forever" (untimed) and "The alarm will not be on for more than 6 seconds" (timed). After modeling both the system and the properties, we can perform implementation-independent verification with existing formal verification tools, such as Kurshan<sup>1</sup> describes.

Between the abstracted behavior of a specification (an FSM network in our case) and the actual behavior of a given implementation there are a number of intermediate models that we can use as input to a formal verification algorithm. We can obtain one such intermediate model by composing the specification (as proposed in Alur, Courcoubetis, and

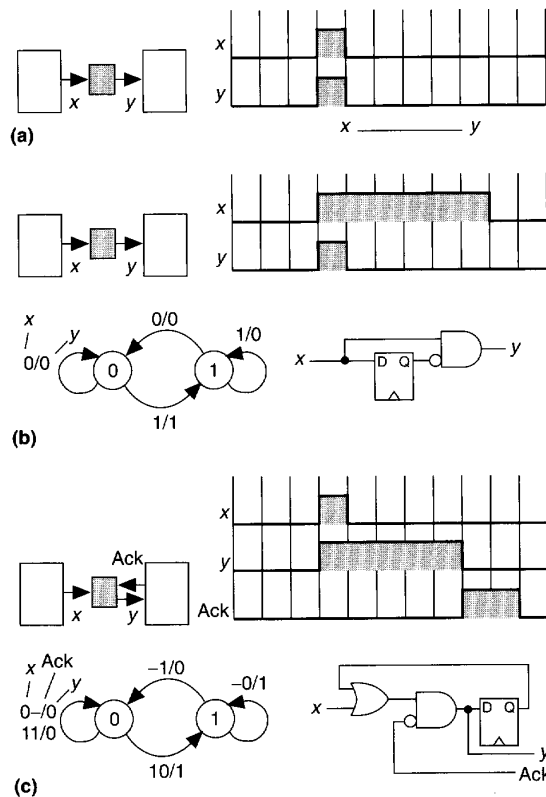


Figure 5. Interface types: the A and B blocks of a hardware-to-hardware interface (a); B block of a software-to-hardware interface (b); and A block of a hardware-to-software interface (c).

Dill<sup>14</sup>) with an implementation-specific component called a timing descriptor. The timing descriptor captures information about the allowed delays in a given implementation. Obtained trivially as 1-clock delays are portions of the timing descriptor related to hardware. For software, computing the maximum/minimum/average runtime, or giving a probability distribution, is desirable.

Specification refinement is a novel way of using formal verification. Initially, the designer specifies a set of properties that the system should satisfy, along with the description of the system in the form of a network of CFSMs with unbounded delays. Then, the designer checks the model with respect to some properties; the results define the information required to build the timing descriptor.

One major obstacle in formal verification is that deci-

phering an error trace and determining the exact cause of the failure is generally difficult. Active research is underway to efficiently return a set of minimal error traces that can help the designer pinpoint the cause of the failure.

Another obstacle is the complexity of models (also known as the state explosion problem). The size of a model can easily grow, especially when precise timing information must be considered. As an example, we modeled the seat-belt system described earlier as an implementation-independent network of two CFSMs, plus a CFSM that models the unpredictable behavior of the user. There, hardware, software, and human speeds differ by several orders of magnitude. Even with an unrealistically coarse base clock of 0.1 seconds, we have a total of 60,000,000 states. The longest run requires about 6.5 hours of CPU time on a DEC5000/125 with 64 Mbytes of memory. Dense-time algorithms<sup>14</sup> may be one way to cope with complexity, but more research needs to be done to make these algorithms applicable to problems of real-life magnitude.

**OUR METHODOLOGY FOR DESIGNING** hardware-software systems exhibits the following characteristics:

- It is well-suited to small control-dominated embedded systems.
- It is based on events as a basic communication primitive. Events are low-level enough to be efficiently implemented both in hardware and software, without imposing unnecessary overheads, and yet general enough to allow the construction of more powerful communication schemes.
- Since both hardware and software can be transparently derived from the same CFSM specification, we need not commit ourselves to a particular mix of software-hardware implementation a priori.
- We use an FSM-based model throughout the design process, thus preserving formal properties. The FSM model derived from a CFSM is compatible with the input format of many formal verification algorithms.
- Verification can proceed both at the specification and implementation levels. In addition, the results of formal verification can serve as a guide in specification refinement.

Already in place are many of the tools necessary for a complete synthesis system. In particular, an early version of the ESTEREL translator and of the synthesis environment described in Figure 2, including hardware and software synthesis, are functional.

In the future, we plan to explore the possibility of adopting formal verification methods that do not require an expen-

sive translation of a CFSM into equivalent FSMs. We would like to exploit different time scales pertaining to different implementation and environment domains.

We also want to investigate the possibility of automated constraint-driven partitioning algorithms for mixed hardware-software systems. □

## References

1. R.P. Kurshan, "Analysis of Discrete Event Coordination," *Lecture Notes in Computer Science*, J.W. de Bakker, W.P. de Roever, and G. Rozenberg, eds., Springer-Verlag, Berlin, Heidelberg, Germany, 1990, pp. 414-453.
2. A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proc. IEEE*, Vol. 79, No. 9, IEEE, Piscataway, N.J., 1991, pp. 1270-1282.
3. D. Druzinski and D. Har'el, "Using Statecharts for Hardware Description and Synthesis," *IEEE Trans. Computer-Aided Design*, Vol. 8, No. 7, 1989, pp. 798-807.
4. N. Woo, A. Dunlop, and W. Wolf, "Codesign from Cospecification," *Computer*, Vol. 27, No. 1, Jan. 1994, pp. 42-47.
5. R.K. Gupta, C.N. Coelho, Jr., and G. DeMicheli, "Program Implementation Schemes for Hardware-Software Systems," *Computer*, Vol. 27, No. 1, Jan. 1994, pp. 48-55.
6. P. Chou, R. Ortega, and G. Borriello, "Synthesis of Hardware/Software Interface in Microcontroller-Based Systems," *Proc. Int'l Conf. Computer-Aided Design*, IEEE Computer Society Press, Los Alamitos, Calif., 1992, pp. 488-495.
7. M.C. McFarland, T.J. Kowalski, and M.J. Peman, "Language and Formal Semantics of the Specification System CPA," *Proc. Int'l Workshop Hardware-Software Codesign*, CS Press, 1991, pp. 342-345.
8. N. Halbwachs et al., "The Synchronous Data Flow Programming Language LUSTRE," *Proc. IEEE*, Vol. 79, No. 9, IEEE, Sept. 1991, pp. 1305-1320.
9. W. Wolf et al., "The Princeton University Behavioral Synthesis System," *Proc. 29th ACM/IEEE Design Automation Conf.*, Assn. Computing Machinery, June 1992, pp. 182-187.
10. M. Chiodo et al., "A Formal Specification Model for Hardware/Software Codesign," Tech. Report UCB/ERL M93/48, University of California at Berkeley, Berkeley, Calif., June 1993.
11. W. Baker, "Application of the Synchronous/Reactive Model to the VHDL Language," Tech. Report, UCB/ERL-93-10, U.C. Berkeley, 1993.
12. E.M. Sentovich et al., "Sequential Circuit Design Using Synthesis and Optimization," *Proc. Int'l Conf. Computer Design*, CS Press, Oct. 1992, pp. 328-333.
13. A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass., 1988.
14. R. Alur, C. Courcoubetis, and D. Dill, "Model-Checking in Dense Real Time," *Information and Computation*, Vol. 104, No. 1, May 1993, pp. 2-34.



**Massimiliano Chiodo** currently serves as a software area leader at Magneti Marelli Electronics Division, Pavia, Italy. Recently, he was a visiting industrial fellow for Magneti Marelli at the University of California, Berkeley, where he worked on formal verification and hardware-software codesign.

Chiodo received his doctorate in physics from the Università degli Studi di Milano, Italy. He is a member of the IEEE.



**Paolo Giusto** has been working for Magneti Marelli, Turin, as software engineer. He is currently a Visiting Industrial Fellow for Magneti Marelli at the University of California, Berkeley, where he is working with the hardware-software codesign group

Giusto received the Doctor of Information Sciences from Università di Torino, Italy, and received his MSc degree in hardware engineering from CEFRIEL in Milan, where he developed a tool for synthesis of PLAs. He is a member of the IEEE.



**Attila Jurecska** works for Magneti Marelli, Turin, Italy, as a software engineer, where he is currently participating in the Formal Languages project. He is also working on the development of a codesign environment, together with the hardware-software codesign group of the University of California, Berkeley.

Jurecska received his first MSc degree in electrical engineering from the Technical University of Budapest. He received his second M.Sc. degree in hardware engineering from CEFRIEL in Milan.



**Harry C. Hsieh** is currently a PhD student in the Electrical Engineering and Computer Science Department at the University of California, Berkeley. He has been a member of the technical staff at Hewlett-Packard's Mainline Systems Laboratory, and has worked at IBM's Federal System Division and T. J. Watson Research Center. His primary research interests include system-level design

methodologies, logic synthesis, and hardware-software codesign.

Hsieh received a BS degree from University of Wisconsin, Madison, and an MS degree from Stanford University. He is a member of the IEEE.



**Alberto Sangiovanni-Vincentelli** is a professor of electrical engineering and computer sciences at the University of California, Berkeley, where he recently received the Distinguished Teaching Award. He has been a visiting professor at the Universities of Torino, Bologna, Pavia, Pisa, and Rome, and has served as advisor, founder, or director with a number of companies. He has published over 250 papers and three books in the area of CAD for VLSI.

Sangiovanni-Vincentelli holds a D.Eng. degree from the Politecnico di Milano, Italy. He is on the International Advisory Board of the Institute for Micro-Electronics of Singapore, is a Fellow of the IEEE, and belongs to the Berkeley Roundtable for International Economy (BRIE). He has also been the technical program chair and the general chair of the International Conference on CAD. He is a member of the IEEE and the IEEE Computer Society.

He is a member of the IEEE and the IEEE Computer Society.



**Luciano Lavagno** is an assistant professor at the Politecnico di Torino, Italy, where his current research interests include synthesis of asynchronous and low-power circuits, concurrent design of mixed hardware and software systems, and formal verification of digital systems.

Lavagno received the D.Eng. from the Politecnico di Torino and his PhD in electrical engineering from the University of California, Berkeley. In 1991, he received the Best Paper award at the 28th Design Automation Conference. He is a member of the IEEE and the IEEE Computer Society.

Direct questions concerning this article to Luciano Lavagno, Dip. di Elettronica, Politecnico di Torino, C. Duca degli Abruzzi 24, 10129, Torino, Italy; lavagno@polito.it.

### Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 159

Medium 160

High 161