Synthesis of Software Programs for Embedded Control Applications

Felice Balarin, *Member, IEEE*, Massimiliano Chiodo, *Member, IEEE*, Paolo Giusto, Harry Hsieh, *Student Member, IEEE*, Attila Jurecska, Luciano Lavagno, *Member, IEEE*, Alberto Sangiovanni-Vincentelli, *Fellow, IEEE*, Ellen M. Sentovich, and Kei Suzuki

***Abstract*— Software components for embedded reactive real-time applications must satisfy tight code size and run-time constraints. Cooperating finite state machines provide a convenient intermediate format for embedded system co-synthesis, between high-level specification languages and software or hardware implementations. We propose a software generation methodology that takes advantage of a restricted class of specifications and allows for tight control over the implementation cost. The methodology exploits several techniques from the domain of Boolean function optimization. We also describe how the simplified control/data-flow graph used as an intermediate representation can be used to accurately estimate the size and timing cost of the final executable code.**

***Index Terms*— Boolean functions, estimation, finite state machines, high-level synthesis, optimizing compilers, real-time systems scheduling, software performance.**

I. Introduction

A. The Context: Embedded Systems

Embedded systems are electronic components of a physical system such as a vehicle, a chemical plant, a nuclear plant, or a communications system, that typically:

- monitor variables of the physical system such as temperature, pressure, traffic, chemical composition;
- process this information making use of one or more mathematical models of the physical system;
- output signals that influence the behavior of the physical system to control its function and optimize its performance.

Embedded systems cover a broad range of applications, from microwave ovens and watches to telecommunication network management and control functions. These embedded systems are integrated onto the physical system itself and hidden from the user. The implementation of such systems can vary from a full hardware configuration, where all the tasks to be performed by the embedded system are translated into a suitable set of customized integrated circuits, to a full software implementation, where all the tasks are implemented as software routines run on a standard component, such as a microprocessor or a digital signal processor (DSP). While in the past hardware configurations dominated the field, today most of the applications are implemented in a mixed configuration, where software has the lion's share. This shift has been basically due to the flexibility offered by software implementations and to the increasing importance of time-to-market considerations in engineering design.

B. The Problem: Software Synthesis

The bottleneck for the implementation of embedded systems has long been considered the development of software, its debugging, and its integration with the hardware components. Recently, it has been pointed out that the capability of analyzing a system before a particular technology is chosen as a target implementation is of paramount importance to have "right-for-the-first-time" designs.

This scenario fueled the quest for a design methodology that favors system-level descriptions of functionality and constraints, technology-independent verification, and automaticoptimized mapping from the system-level descriptions and constraints to software and hardware implementations. While hardware synthesis has been the object of considerable attention over the recent past, much less attention has been devoted to the process of software synthesis.

We informally distinguish between *software synthesis* and *software compilation*, according to the type of input specification. The term software compilation is generally associated with an input specification using C- or Pascal-like imperative, generally nonconcurrent, languages. These languages have a syntax and semantics that is very close to that of the implementation (assembly or executable code). In some sense, they already describe, at a fairly detailed level, the desired *implementation* of the software. We will use the term software synthesis to denote an optimized translation process from a high-level specification that describes the *function* that must be performed, rather than the way in which it must be implemented. Software synthesis can be, for example, the

Manuscript received April 7, 1998; revised October 21, 1998. The work of H. Hsieh was supported by Semiconductor Research Corporation (SRC) under Contract DC-324. This paper was recommended by Associate Editor R. Camposano.

F. Balarin is with Cadence Berkeley Laboratories, Berkeley, CA 94707 USA (e-mail: felice@cadence.com)

M. Chiodo and P. Giusto are with Cadence Design Systems, Inc., San Jose, CA 95134 USA.

H. Hsieh and A. Sangiovanni-Vincentelli are with the University of California at Berkeley, Berkeley, CA 94707 USA.

A. Jurecska is with Synopsys, Beaverton, OR 97006 USA.

L. Lavagno is with Politecnico Di Torino, Torino 10129, Italy.

E. Sentovich is with Cadence Berkeley Laboratories, Berkeley, CA 94707 USA.

K. Suzuki is with Central Research Laboratory, Hitachi Ltd., Kokubunji, 185 Tokyo, Japan.

Publisher Item Identifier S 0278-0070(99)03962-7.

C or assembly code generation capabilities of DSP graphical programming environments, such as Ptolemy ([11]), of graphical finite state machine (FSM) design environments, such as StateCharts ([20]), or of synchronous programming environments such as Esterel, Lustre, and Signal ([19]).

This notion of software synthesis received much attention in the early 1970's but results were mainly theoretical with little practical impact on software design practices. The lack of results of practical importance was mainly caused by the very wide range of possible applications that required heterogeneous models of computation and constructs, such as pointers, memory allocation, and recursion, that were too difficult to manipulate efficiently in an automatic fashion.

C. Compiler Technology

On the other hand, compiling a high-level language into machine instructions has been the enabling technology for the extended use of computers for all kinds of applications. The progress of compiler technology has been exciting over the past 20 years. Compilers translate high-level constructs into an optimized set of machine instructions. This translation occurs in two basic steps: mapping from high-level constructs into intermediate code that is often processor independent, and mapping of the intermediate code into the actual "architecture" (instruction set and registers) of the processor to be used. The first step is optimized by applying a set of semilocal transformations to yield an intermediate representation that can be directly translated to more compact and faster code. The following steps deal with the architecture-specific transformations that include register allocation and instruction selection.

The use of semilocal, peep-hole optimization is justified on the one hand by the need for compiling code in reasonable time, on the other by the great variety of constructs to deal with. After all, it is often reported that the bottleneck in software debugging is compilation time.

D. Software versus Hardware Compilation

The analogy between hardware and software compilation has been known and exploited for a long time [16]. Hardware compilation (or high-level synthesis) [7], [27] involves functional and register allocation and scheduling, followed by component synthesis and optimization. Software compilation [1] traditionally involves register allocation and instruction selection, followed by local optimizations.

Exploiting the link between these two continues to be a fruitful avenue for research: technologies for each are continually advancing, and the two domains have traditionally had significant differences that have prevented the application of some techniques in each domain to the other. In particular, optimization has typically been more aggressive in hardware synthesis, while for software long compilation times have prevented global techniques from being employed.

In our context, hardware and software in a single system are synthesized together. This implies that the software part is derived from the same starting point as the hardware and, thus, is "hardware-like" and contains a small set of simple constructs. For example, features like recursion, pointers and loop bounds that can be determined only at run-time, are not used, because they are hard (and in general even impossible) to implement in hardware. This software lends itself well to aggressive, global optimizations as are traditionally applied to hardware. As mentioned previously, we refer to this creation and optimization of restricted software as "software synthesis" rather than "software compilation." We apply optimization techniques derived from the hardware optimization area to optimization of embedded software.

E. Restricted Application Domains

When we deal with specialized applications, such as DSP, the range of different constructs to consider is much more restricted than in general computing. Hence, more aggressive optimization is possible, and has indeed been attempted, yielding interesting results [26].

Most of the embedded systems applications do not require a wide variety of coding constructs. We strongly believe that in this class of systems, automatic optimization and verification can be pushed to a level that is unprecedented in "standard" compiler and operating system technology.

F. Our Environment and Requirements

To really decrease development time for embedded systems, we have to select a high-level representation that is implementation independent and easy to use for a system designer, and then select a set of mathematically well-defined operations to translate and optimize this representation. Among all embedded systems applications, we chose to focus on *control-dominated* embedded systems, characterized by the importance given to the decision process that leads from a set of input events to a set of output events (reaction). The high-level representation of choice is an interconnection or network of communicating processes with FSM semantics. Today, FSM's are commonly used in embedded system design tools both explicitly (specified in graphical or textual form [20], [32]) and as an intermediate format ([8], [19], [35]).

G. The FSM Model: CFSM's

The use of FSM's for embedded control specification offers several advantages over apparently more powerful formalisms (such as unrestricted programming languages). First of all, they are easily understood and widely used even as informal specifications. Second, there are abundant theoretical and practical results concerning their manipulation (minimization, encoding, formal verification of properties, etc.).

Unfortunately "pure" FSM's do not provide a very convenient representation for systems that perform even a small amount of computation. It is then customary to extend them with the capability to perform assignments of expressions to variables, and to use relational operators to determine transition conditions. This mechanism increases the expressive power at the expense of the synthesis and verification capabilities (e.g., there is no longer a "canonical" form for such extended FSM's, verification becomes much more difficult,

etc.). In our design methodology and tools, we have selected an extended FSM model called *codesign FSM* (CFSM), defined in [12] and [13]. This representation extends classical FSM's with arithmetic and relational operators, and assumes that CFSM's interact via an asynchronous communication mechanism that allows great flexibility and expressive power. Even though throughout this paper we refer to CFSM's as representation, our results on software generation can be applied to any extended FSM-based specification, like those mentioned above.

H. Our Approach to Software Synthesis

The purpose of this paper is to describe algorithms for a software synthesis system generating C code from FSM specifications. This system includes optimization techniques that are either impossible or simply too expensive in the general compiler domain ([1]), but are very effective in our restricted domain. Moreover, unlike classical compilation algorithms, our software synthesis technique starts from a description of the *function* to be computed, rather than from an operational implementation of it. This allows the use of powerful optimization algorithms based on Boolean function manipulation methods. We tightly couple the optimization process with a fast and accurate timing and code-size estimation procedure to take into account constraints at a much finer granularity than is possible with a truly general-purpose compiler. We do not claim to have invented a new general purpose compiler, because the domain of applications is much more restricted.

Throughout this paper we make the following main assumptions.

1) The specification is given as a network of CFSM's. Note that even though such a specification is not biased toward any particular implementation, it does impose a network structure which we preserve during synthesis. This means that each CFSM is a synchronous, statically scheduled entity, while the network is asynchronous, concurrent, and dynamically scheduled. The granularity level is defined *a priori* by the designer, and the ordering of emission of output events is decided statically by our synthesis algorithm, with the objective of minimizing code size as discussed in Section III-B. A more global approach, in which the synchronous/asynchronous boundary can be chosen as part of the synthesis process, with the objective of simultaneously optimizing code size and execution time is left to future research. Note that a growth of the synchronous islands (CFSM's) typically induces:
   - an increase in code size, due to the more complex transition function that must be computed;
   - a reduction in execution time (if synthesis is performed using the techniques described in Section III-B, where the execution time of the control portion of the code depends almost entirely on the number of inputs and outputs of each CFSM), due to:
     - —the reduction of communication and scheduling overhead;
     - —(possibly) the increased utilization of processor resources, due to exposing larger statically scheduled units to the underlying C compiler.
2) A real-time operating system (RTOS) is used to activate appropriately the tasks implementing the CFSM's. Our synthesis procedure, in addition, provides execution time estimates that can be used either by a user or by an automatic RTOS generator to devise a scheduling policy that is guaranteed to meet the timing constraints.
3) An existing general-purpose C compiler is used to transform the C code that we produce into machine code. This allows us to concentrate on domain-specific transformations, while leaving general ones such as register allocation and instruction selection to the general-purpose C compiler. Note that the C code that we produce is so simple and low-level that we can keep a very tight control over the resulting machine code, and the compiler cannot "undo" our optimizations.

We use a control/data-flow diagram (called an *s-graph*, for software graph) as an intermediate data structure. The s-graph is simpler than general control/data-flow diagrams, because it needs only to represent a single function from a discrete domain (the set of input events and values) to a discrete domain (the set of output events and values). As such, it requires only conditional branch and assignment as primitives (augmented with arithmetic and relational expressions without side effects). The s-graph has a direct representation in C and can be translated with equal ease into object code by any available compiler. In this way, we can obtain good cost and performance estimates at any intermediate stage of the optimization process, without the need to compile the code and analyze the results.

Our software synthesis procedure is composed of the following main steps:

1) optimized translation of the transition function of a given CFSM into an s-graph;
2) s-graph optimization and code-size estimation;
3) translation of the s-graph into a target language;
4) scheduling of the CFSM's and generation of the RTOS;
5) compilation into machine code to be run on the target processor.

Step 1 consists of building an optimized binary decision diagram (BDD, [10]) for the transition function as an intermediate representation, to generate an initial s-graph corresponding to code that executes *very fast*, potentially at the expense of code size. It is based on a new result, described in this paper, that states the equivalence between:

- a multioutput multivalued function f;
- an s-graph computing f, that is directly obtained from a BDD representing f.

Step 2 is similar to standard software optimization techniques based on control/data-flow diagrams (Section II contains a discussion of its relation to previous work). Thus far, we have generated C code in Step 3, though any target language is possible. Step 4 uses the software performance estimation package and classical real-time scheduling algorithms [24], [18] to schedule the CFSM's while meeting the given timing

constraints. The compilation in Step 5 is done using existing C compilers for the target embedded processors.

The paper is organized as follows. Section II contains background information and a summary of the CFSM network model. Section III contains the s-graph structure definition, its synthesis and optimization from CFSM's, C code generation, and software cost and performance estimation based on s-graphs. Section IV describes the functionality of the automatically generated RTOS. Section V shows some experimental results demonstrating the effectiveness of the approach.

II. Preliminaries

A. Previous Work

1) Software Synthesis: Previous approaches to automated software synthesis for reactive real-time systems have started either from synchronous programming languages (e.g., Esterel, [8]), or from other high-level languages ([14] and [17]).

In the first case, the main problem is the identification of a *single* FSM equivalent to the Esterel specification, and its efficient implementation as a software program. Previous versions of the Esterel compiler (v3) produced a single FSM, which resulted in a very *fast* implementation (as all the internal communication between modules disappears when the single FSM is produced), at the expense of code size. The versions from v4 on ([34]; see also [9]) maintain a multi-FSM representation, while ensuring that the global behavior is equivalent to that of a single FSM.[1] Thus, the composition is never computed explicitly. This results in a code size that is usually linear in the size of the specification (in the worst case, it is proportional to the square of the size of the specification). The translation is done via the intermediate form of Boolean circuits, enabling logic optimization techniques to be used to reduce the final code size. However, these optimization techniques are applied to an abstract representation of the final code, and no low-level or target-specific optimizations are available. Furthermore, the optimization is applied globally to the entire system, with no opportunity for optimizing on a module-by-module basis. Our approach, on the other hand, allows a finer tradeoff between size and speed:

- the designer can choose the granularity of the generated CFSM's, even if they are produced from an Esterel specification ([36]);
- the designer can manipulate the CFSM hierarchy during synthesis;
- the optimizations are done at a level closer to the final C-code implementation;
- optimizations include both Boolean-circuit based algorithms and decision-tree based algorithms; thus far the decision-tree based algorithms have been more successful in producing compact code.

In the second case, the main emphasis is on the *scheduling* of operations derived from a concurrent high-level specification (e.g., hardware-C, [21]). The problem is that of choosing an order for potentially concurrent operations that satisfies the given timing constraints. In our case, we decompose the problem of satisfying timing constraints into two (possibly iterated) steps:

a) software generation for each CFSM;
b) scheduling of CFSM transitions to satisfy timing constraints.

Thus, we can take advantage of the large body of research about scheduling for real-time systems (e.g., [24]) for the second step. On the other hand, some of the fine-grained scheduling algorithms described in [17] and [14], for example, can also be used to perform a preliminary optimization before our synthesis algorithm. This would allow an easier satisfaction of "short term" timing constraints (e.g., those dictated by a specific interface protocol implemented directly in software) which may be more difficult to satisfy with classical scheduling techniques (designed for "long term" response and input rate constraints).

2) Hardware High-Level Synthesis: Hardware high-level synthesis can roughly be divided into two stages: *behavioral* and *register-transfer level* (RTL). The input to the behavioral synthesis is a sequential specification, where timing of actions is not fixed. Its output is a collection of registers and a cycle-by-cycle specification of how the registers change. This is then the input to RTL synthesis, which mostly deals with the combinational specification of register transfers, and builds an optimized circuit for it.

Similarly to classical compilers, behavioral synthesis usually operates on a description that is structurally very similar to the original specification (e.g., CFG graph). However, the *combinational* specification at its output no longer bears this resemblance, but it is rather in a form that facilitates powerful combinational logic optimizations (e.g., sum-of-products or BDD's [7]). In our approach (enabled by domain restrictions), we transform the original *sequential* specification into such a form. This enables us to extend combinational logic optimization techniques (BDD's, to be more precise) to the optimization of sequential programs. In general, these optimizations are more powerful than local transformations used by compilers and behavioral synthesis systems.

Perhaps not surprisingly, the relative value of optimization techniques changes in the software area. For example, while BDD's are used extensively in combinational logic synthesis to represent and manipulate Boolean functions (e.g., [7], [25], and [31]), it is generally accepted that they are not a very good structure for circuit implementation (except for low-power [22]). In contrast, we will show that BDD-like structures are very efficient (though quite unreadable) program implementations.

The approach of [7] used BDD's to represent control functions in a high-level synthesis system. In their work, the application is purely to hardware; the authors did not make special considerations for software such as optimization based on estimations of timing of instruction execution. They derive the control functions and build the BDD for them on-the-fly. Several ordering methods are used, but they are all static heuristics for obtaining a good initial BDD. BDD optimization based on reordering is not applied. Furthermore, that work

[1] See [7] for more on composition and causality.

does not exploit interleaving of input and output variables (or at least, does not mention this).

3) Hardware Simulation: Surprisingly enough, the closest relatives of our software synthesis techniques come from the area of cycle-based hardware simulation. Both [28] and [3] are aimed at solving a problem that is quite similar to ours: *efficiently computing on a sequential machine the transition function of an* FSM. Both approaches rely on a BDD representation of the transition function, and exploit the fact that, given an input and present state assignment, there exists a unique BDD path that can be used to compute the value of the next state and output functions. This means that a single-threaded sequential execution of this BDD can be efficiently implemented on a standard workstation.

Even though the basic problems are related, there are a number of differences.

1) Their starting point is a large synchronous circuit represented at the gate level, hence, their main problem is to efficiently represent the FSM without a blow-up of the BDD sizes (mostly due to data computations represented in FSM form).
   Our starting point is an explicit representation of an *Extended* FSM, in which data computation are represented using explicit arithmetic operators. Hence, we do not suffer from blow-up problems due to the data part. The BDD's representing the control part suffer from exponential growth less often, and the designer can control this phenomenon directly, by changing the CFSM granularity.
2) Their target is execution on high-end workstations. Hence, they can afford to use very large BDD's, and even to use multivalued variables to represent sets of binary variables, thus, causing large lookup tables to be generated. Their main problem is cache and translation lookaside buffer thrashing due to the huge size of the generated tables. They also use a simple and fast table lookup algorithm to implement the BDD's using the *data* section of the object code.
   Our target is execution on often very small embedded controllers, in which memory is a very expensive and scarce resource. Hence, we implement the BDD's directly in executable code, i.e., in the *text* section of the object code. In this way, we can use the efficient encoding of the BDD branching structure provided by the instruction set encoding of the target processor (often using fewer bits of address for near jumps).

For this reason, our techniques and results are quite different from those of [28] and [3].

B. Binary Decision Diagrams

The BDD is a key data structure as an intermediate representation for our software optimization techniques. A *binary-decision diagram* (BDD [2], [10]) is an efficient representation for storing and manipulating Boolean functions. A BDD is a directed acyclic graph with a root node for each output function and leaf nodes representing the value of each output function for each input minterm. Each nonleaf node is associated with an input variable, and each of the two out-edges of the node is associated with the value of the variable (zero or one) along that branch. The representation is made compact (*reduced*) by sharing common functional subgraphs. Given a function f and an *ordering* of the input variables, the reduced-ordered BDD (simply called BDD in the following) is a canonical form for f.

While the size of the BDD may be exponential in the number of inputs for any ordering, in many practical cases a good ordering can be found that produces a BDD of manageable size. Functional operations on the BDD take at most n^2 space and time (n is the number of nodes in the BDD); equivalence checking between two BDD's requires only a graph isomorphism check. The canonicity property of BDD's, efficient BDD package implementation, and recent improvements in variable ordering strategies have made BDD-based algorithms efficient and effective for a variety of problems involving Boolean function manipulation.

C. Characteristic Functions

Multioutput functions (or, equivalently, sets of functions defined on the same domain) can be represented by their *characteristic functions*. A single-output binary-valued function $\chi^f : (X \times Y) \rightarrow \{0,1\}$, where $X = X_1 \times \cdots X_m$ and $Y = Y_1 \times \cdots Y_l$, represents the multioutput multivalued function $f : X \rightarrow Y$ if $\chi^f(x,y) \Leftrightarrow (y = f(x))$. The same notation can also be used to describe a *relation* R, as $\chi^R(x,y) \Leftrightarrow (y \in R(x))$.

The function resulting when some argument x_j of a function f is replaced by a constant b is called a *restriction* (or *cofactor*) and is denoted $f_{x_j=b}$. The projection of a function f onto a space orthogonal to x_j (or *smoothing* of f by x_j, or existential quantification of x_j in f) is denoted $S_{x_j}f$. That is, if $x_j \in \{0,1\}$, then $S_{x_j}f = f_{x_j=1} \vee f_{x_j=0}$.

The *support* of an output variable y_i of a multioutput function is the set of inputs upon which y_i *essentially depends*. More precisely, an input variable x_j belongs to the support of y_i if $S_{x_j}y_i(x_1, \cdots x_n) \neq y_i(x_1, \cdots x_n)$.

D. Network of Codesign FSM's

Our model of a control-dominated reactive system (originally proposed in [12] and [13]) is globally asynchronous locally synchronous (GALS) *network* of CFSM's communicating via *events*. The interested reader is referred to [5] for a more complete description of the CFSM model and of the implementation choices that we have made.

An input or output CFSM event occurs at some point in time and *may* carry a value which is represented by a discrete-valued variable (cfr. the notion of *signal* in Esterel). An example of a valued event is a temperature sample, or a key hit on a keyboard; an example of a value-less (also called "pure") event is an excessive pressure alarm, or a reset button. Each CFSM receives atomically (*locally synchronous*), or *detects*, a snapshot of its input events, and performs its calculations independently and asynchronously of other CFSM's (*globally asynchronous*). As a result of the computation, it may sometimes later change state and/or

emit output events. Each event, with or without a value, is associated with a Boolean flag, indicating the *presence* of the event, which is true in the time interval between its emission and its detection. The value of an event is updated by the emitter:

- ideally, at the same time the presence flag is set;
- in practice, especially in a software implementation, "some small amount of time before" the presence flag is set[2];

Each input event may be detected at most once at any time after its emission: once detected, it is no longer present at the input of the CFSM. A CFSM's reaction to an event occurrence is defined by the *transition function* of the CFSM. The transition function synchronously maps the set of input events and values onto the set of output events and values, possibly based on its internal state: $f : X_1 \times \cdots X_m \rightarrow Y_1 \times \cdots Y_l$. A CFSM begins its reaction (computation of the transition function) to an input event after a *sensing delay (sd)* that is greater than or equal to zero. It completes its reaction by emitting some events after a *reaction delay (rd)* that is strictly greater than zero.

The model of communication is, in general, *asynchronous*: the emission of events by the CFSM's in the network may happen at any time and independently. Because of the asynchrony, there is no guarantee that a CFSM will detect an event before it is emitted again. Hence, it is possible that an event and its value are overwritten and lost. This is equivalent to the assumption that there is a buffer of length one between each CFSM and for each event.

Synchrony and Asynchrony: In our framework, the CFSM model is used throughout the design process and, in particular, as a mechanism to capture the design intent. We strongly believe that at the highest levels of abstraction, restrictive hypotheses have to be carefully evaluated to make sure that designs of practical relevance can be appropriately modeled. For this reason, we have chosen the communication mechanism to be asynchronous since this model does not overly restrict the implementation domain to be considered.

We also believe that the restriction imposed by synchronous languages such as Esterel, although very convenient from the analysis point of view, is too strong at the highest level of abstraction. The *synchronous hypothesis* assumes that the reaction to external stimuli of the network of concurrent processes occurs in *zero time*, i.e., all internal communication can be abstracted away. As a result, it is theoretically possible to extract a single, albeit possibly very large, FSM equivalent to the network of concurrent processes. This implies that formal verification and other analysis techniques based on standard FSM's are possible. On the other hand, this also may imply a costly final implementation (where one must ensure that the synchrony hypothesis is indeed valid, that is, that the system reacts much faster than its environment) and a small solution space (where one must force the validity of the hypothesis). In particular, the synchronous hypothesis implies that the behavior of a correct implementation of a synchronous program must be analogous to that of a *combinational circuit* (except for waiting statements, which are equivalent to registers). As such, a correct heterogeneous implementation (mixed hardware and software) requires the solution of a very complex run-time scheduling problem, equivalent basically to hardware simulation.

Our asynchronous communication model is inherently nondeterministic. This feature certainly makes the design and verification process more complex, because *all* possible resolutions need to be considered. On the other hand, nondeterminism enables us to easily model the unpredictability of the reaction delay of a CFSM both at the specification level, where we need to represent different implementation styles that imply quite different reaction delays, and at the implementation level, where delays may still remain intrinsically unpredictable. Specifically, a software implementation has a delay that may be *much larger* than a hardware one. Moreover, if software is based on a RTOS supporting preemption, it has reaction delays that depend on the context and as such, in our opinion, cannot be represented accurately with a model based on a synchronous hypothesis. Note that our model does allow one to *choose a synchronous implementation* that fits into the synchronous hypothesis by modeling the embedded system as a single CFSM, but of course does not force one to choose such an implementation from the very beginning.

The rationale for the introduction of this model is based on the following considerations.

- A *network* of components can express a complex behavior while keeping the complexity of each component at a reasonable level.
- The behavior specification is extended with the use of arithmetic (or other) operations to be able to represent embedded systems where *real-valued variables* are controlled.
- The reaction and sensing delays are useful in modeling and constraining the *timing behavior* of heterogeneous implementations (software may take an *a priori* unknown number of clock cycles to execute a task represented by a CFSM while a straightforward synchronous hardware implementation takes only one cycle).
- An *asynchronous communication mechanism* is more efficient for representing the interaction among tasks in an embedded system, where timing constraints are tight and synchronous implementations may cause unnecessary delays because the common pace of the system must be slow enough to accommodate the slowest communication link.

III. Software Graphs

This section begins with the definition of an s-graph in Section III-A. The synthesis and optimization of an s-graph from a CFSM transition function is described in Section III-B. Finally, software cost estimation based on the s-graph is described in Section III-C.

[2]This choice was made in order to allow an efficient implementation in software, without the need to disable interrupts for long periods of time to implement emission atomicity.

A. S-Graph Definition

In this section, we define more precisely the control-flow graph that we use as internal representation of the CFSM transition function.

Definition 1: An s-graph G is a *directed acyclic graph* (DAG) with one source and one sink. Its vertex set V contains four types of vertices: BEGIN, END, TEST, and ASSIGN. The source has type BEGIN, the sink has type END. All other vertices are of type TEST or ASSIGN. Each TEST vertex v has two children true(v) and false(v).[3] Each BEGIN or ASSIGN vertex u has one child *next*(u) only. Any nonsource vertex can have one or more parents. An s-graph is associated with a set of m input and l output variables $z_1, \cdots, z_{m+l}$, ranging over finite domains $\mathcal{D}(z_1), \cdots, \mathcal{D}(z_{m+l})$, respectively.

- Each ASSIGN vertex v is labeled with an output variable z_v, and a $\mathcal{D}(z_v)$-valued function $a_v(z_1, \cdots, z_{m+l})$. In the graphical representation of s-graphs, e.g., in Fig. 1, we label such a vertex with $z_v := a_v(z_1, \cdots, z_{m+l})$ to indicate the intuitive meaning of ASSIGN vertices.
- Each TEST vertex v is labeled with a predicate $p_v(z_1, \cdots, z_{m+l})$, whose truth value determines which child will be executed.

A simple s-graph is shown in Fig. 1. It corresponds to the reactive behavior, represented in Esterel, as shown at the bottom of the page.

1) To each valued input signal we associate two input s-graph variables, one Boolean and one with the same domain as the signal, while a pure input is associated only with the Boolean input. For example, input signal `c` is associated with:
   - a Boolean variable `present_c`, indicating that `c` is present in the current CFSM input snapshot;
   - an integer variable `?c`, holding its value.
2) Similarly, to each valued output signal we associate two output s-graph variables. For example, output signal `y` is associated with Boolean variable `emit_y` indicating whether `y` is being emitted in the current synchronous reaction. If `y` were not pure, it would also be associated with an output variable holding the value to be emitted.
3) To each state variable we associate one input and one output s-graph variable (no presence information is associated with state variables, of course). For example, input variable `a` and output variable `a'` correspond to the values of state variable `a` in the current and the next reaction, respectively.

[3] The implementation described in Section V allows more than two children. The extension of the definitions and theorems to the more general case is trivial.

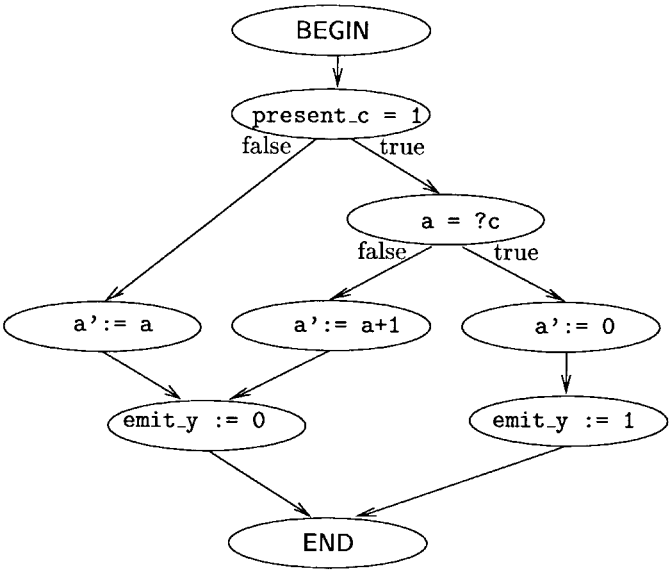


Fig. 1. A simple s-graph.

The Esterel statement `await c` is implemented by the TEST node checking if `present_c` is `1`. The outermost loop is implemented by the RTOS, that will call the C code synthesized from the s-graph every time it must execute a reaction (a CFSM transition), i.e., every time it has at least one present input event.

The s-graph model resembles *branching programs* ([23], [29]) and *binary decision diagrams*. Both branching programs and BDD's are different from s-graphs because they allow:

- only single-variable predicates on TEST nodes;
- assignments to only a single output variable (as the last level of nodes).

```
module simple:   % CFSM name
input c:integer; % integer input signal
output y;        % pure output signal
var a:integer in % local state variable
  loop           % loop forever
    await c;     % wait for c to be present
    if a = ?c then % if a is equal to the value of c
      a := 0;
      emit y;
    else
      a := a + 1;
    end if
  end loop
end var
end module
```

We will see in Section III-B, though, that there is a close connection between a BDD representation of a CFSM transition function and an s-graph computing it.

The evaluation of the multioutput function computed by an s-graph with BEGIN node v, m input variables, and l output variables uses the following algorithm. Let z denote a vector of temporary variables, each uniquely associated with an input or an output variable,[4] and ϵ denote an uninitialized value.

```
procedure evaluate (v: vertex; x_1, ..., x_m:variable)
begin
    for 1 ≤ j ≤ m + l
      if z_j is an input then assign to it the corresponding x_i
      else z_j ⇐ ε
    eval_step (next(v), z_1, ..., z_{m+l})
    for 1 ≤ j ≤ m + l
      if z_j is an output then assign it to the corresponding y_i
    return (y_1, ..., y_l)
end

procedure eval_step (v:vertex; z_1, ..., z_{m+l}:variable)
begin
    if v is a TEST then
        if p_v(z_1, ..., z_{m+l}) then
            eval_step (true(v), z_1, ..., z_{m+l})
        else eval_step (false(v), z_1, ..., z_{m+l})
    else if v is an ASSIGN then
        z_v ⇐ a_v(z_1, ..., z_{m+l})
        eval_step (next(v), z_1, ..., z_{m+l})
end
```

Definition 2: Let G be an s-graph, and let z be partitioned among input and output variables as assumed by procedure **evaluate**.

G is *functional* if every output variable z_j:

1) is assigned by **eval_step** at least one defined (i.e., different from ϵ) value for each combination of values of the input variables;
2) has a defined value whenever a predicate or a function depending on z_j is visited by **eval_step**.

It is easy to show that for a functional s-graph, **evaluate** defines a completely specified I/O function, i.e., that

$$\textbf{evaluate}(c_1, \cdots, c_m) \in \mathcal{D}(y_1) \times \cdots \times \mathcal{D}(y_l)$$
$$\text{for all } (c_1, \cdots, c_m) \in \mathcal{D}(x_1) \times \cdots \times \mathcal{D}(x_m).$$

It is also easy to check that a *nonfunctional* s-graph denotes:

- either an incompletely specified function, if condition 1) in Definition 2 is violated;
- or a *relation* between the input and the output variables, if condition 2) in Definition 2 is violated. In this case, we consider a *nondeterministic* execution of **evaluate**, in which the undefined value can mean any value in the domain of the corresponding variable. Then an element of $X_1 \times \cdots \times X_m \times Y_1 \times \cdots \times Y_l$ belongs to the relation if one possible choice in the evaluation of a TEST or ASSIGN node with an undefined predicate or function can yield it.

[4]In Section III–B2, we will show how to compute a heuristically "good" association in order to minimize the s-graph size.

This fact can be used in optimizing the s-graph, as briefly explained in Section III-B2.

B. S-Graph Implementation and Optimization

Software generation for a given CFSM proceeds by generating the initial s-graph from the transition function, optimizing the s-graph, and translating it into C code.

1) Handling of Extended FSM's: The transition function of a CFSM in general involves both Boolean (or symbolic multivalued) and (bounded) integer variables. The former are used in the reactive control part, while the latter are used in the computational data part. This paper focuses on optimizing control-dominated specifications and, hence, the discussion in the remainder of this section is mostly devoted to an efficient implementation of the reactive control component. However, real specifications seldom consist completely of reactive control. Hence, we adopt a mixed representation of a CFSM.

In this paper, we represent the CFSM transition function as a composition of the following:

- set T of *tests* on input and state variables;
- set A of *actions*, which can be either output emissions or assignments to state variables;
- the *reactive function* mapping subsets of T to subsets of A, represented by its characteristic function $F : \{0,1\}^{|T|+|A|} \mapsto \{0,1\}$.

For example, for the ESTEREL module in Section I tests are `present_c` and `a=?c`, actions are `a'`$:=$`a`, `a'`$:=$`0`, `a'`$:=$`a+1`, and `emit_y`, and the reactive function is

present_c	a′ =?c	a′ := a	a′ := 0	a′ := a + 1	emit_y
0	0	1	0	0	0
0	1	1	0	0	0
1	0	0	0	1	0
1	1	0	1	0	1.

Tests and actions will be implemented as expressions in the target language. The reactive function is just a Boolean function, for which we construct an s-graph, as described in the next section. The procedure produces an initial implementation, that can be optimized using a variety of techniques.

Conceptually, the CFSM transition function is executed in three phases.

a) Tests are evaluated to determine the values of input variables of the reactive function.
b) The s-graph of the reactive function is evaluated to determine the values of its output variables.
c) Actions corresponding to output variables with value one, are executed.

In practice, these three phases are interleaved. Test are evaluated as they are needed during s-graph evaluation, and actions are executed as soon as the corresponding variable is known to be one.

Throughout this paper, we assume that expressions do not have side effects and, hence, that their execution can be reordered at will in order to optimize, e.g., code size as described later. For the basic CFSM model as described

previously, the only expression that possibly has a side effect is a division by zero, that may cause an arithmetic exception. We will, hence, assume that division is implemented safely (by first checking if the divisor is zero) and that a correct CFSM never uses the result of a division by zero (even though it may perform it as part of its evaluation).

2) Initial S-Graph Implementation: The initial s-graph is built recursively starting from the reactive function, as follows. The input and output variables $z_1, \cdots, z_{m+l}$ (where $m = |T|$ and $l = |A|$) are visited based on an initial *arbitrary* ordering. The assignment functions are computed based on the Shannon decomposition ($f = x_i f_{x_i} + \bar{x}_i f_{\bar{x}_i}$). Procedure **build** is called with an initial variable index of zero, and the function F set to the reactive function. As we will see below, the choice of the ordering influences the form of the final s-graph (specifically the relative mix of TEST and ASSIGN nodes).

procedure **build**($z_1, \cdots, z_{m+l}$: variable; i: index; F: function)
begin
  if $i = 0$ then
    create a BEGIN vertex v
    next(v) $\Leftarrow$ **build**$(z_1, \cdots, z_{m+l}, 1, F)$;
  else if $F = \mathbf{1}$ then
    create the END vertex v;
  else if z_i is an input, then
    create a TEST vertex v with $p_v = (z_i)$;
    true(v) $\Leftarrow$ **build**$(z_1, \cdots, z_{m+l}, i+1, F_{z_i=1})$
    false(v) $\Leftarrow$ **build**$(z_1, \cdots, z_{m+l}, i+1, F_{z_i=0})$
  else if z_i is an output, then
    create an ASSIGN vertex v labeled with z_i
    and $a_v = S_{z_j | i<j\leq m+l, z_j \text{ is an output}} F_{z_i=1}$
    next(v) $\Leftarrow$ **build**$(z_1, \cdots, z_{m+l}, i+1, S_{z_i} F)$
  return **reduce**(v)
end

We assume that the **reduce** function, called in the last step, ensures that a graph with root v has no isomorphic subgraphs, exactly as in BDD construction [10]. In particular, **reduce** should eliminate all but one END node.

We can now show the correctness of this algorithm.

Theorem 1: Let $\chi^f(x_1, \cdots, x_m, y_1, \cdots, y_l)$ be the characteristic function of multioutput function f, such that $y_k = f_k(x_1, \cdots, x_m)$, and let v be the BEGIN vertex of the s-graph G returned by procedure **build**$(x_1, \cdots, x_m, y_1, \cdots, y_l; 0; \chi^f)$. Then, for all $(c_1, \cdots, c_m) \in \mathcal{D}(x_1) \times \cdots \times \mathcal{D}(x_m)$

$$\mathbf{evaluate}(v, c_1, \cdots, c_m) = f(c_1, \cdots, c_m).$$

Proof: Consider an assignment of values $(c_1, \cdots, c_m)$ to the input variables $x_1, \cdots, x_m$ of f, and an arbitrary output $y_k = f_k(x_1, \cdots, x_m)$.

Traverse the s-graph up to an ASSIGN node v labeled with y_k (from the definition of **build**, it follows there exists exactly one such node on any path from BEGIN to END). Suppose, without loss of generality, that in this traversal, TEST nodes labeled with the first n input variables and ASSIGN nodes labeled with the first $k-1$ output variables have been visited. When **build** is called for node v, $F = S_{y_1, \cdots, y_{k-1}} \chi^f_{x_1=c_1, \cdots, x_n=c_n}$. Hence, $a_v = S_{y_1, \cdots, y_{k-1}, y_{k+1}, \cdots, y_m} \chi^f_{x_1=c_1, \cdots, x_n=c_n, y_k=1}$ (a function which depends on the unvisited inputs $x_{n+1}, \cdots, x_m$ only). The value of a_v is one if and only if $f_k(c_1, \cdots, c_m) = 1$ (i.e., the value assigned to y_k is correct), because 1) by definition the characteristic function is $\chi^f = \prod_{k=1}^{l} (y_k \equiv f_k(x_1, \cdots, x_m))$; 2) the smooth function distributes over functions that are independent of the smoothing variable: $\mathcal{S}_x(f \cdot g) = f \cdot (\mathcal{S}_x g)$ if f is independent of x. 1) and 2) imply that $a_v = S_{y_1, \cdots, y_{k-1}, y_{k+1}, \cdots, y_m} \chi^f_{x_1=c_1, \cdots, x_n=c_n, y_k=1} = f_k(x_1 = c_1, \cdots, x_n = c_n, x_{n+1}, \cdots, x_m)$. □

Note that the mapping from the transition function to the s-graph is not unique since it is based on the ordering of the variables. Section III-B3 discusses the influence of this choice over the characteristics of the generated code.

Moreover, the input to this algorithm need not always be a *function*, but could also be a *relation* (e.g., when nondeterminism is used to describe design freedom). In that case, with a modification to the ASSIGN function a_v as indicated below, there may be cases in which the value assigned to z_i still depends on z_i (as well as on any input still to be tested). The simplest case, when $S_{z_{i+1} \cdots z_{m+l}} \chi^f = z_i$, corresponds to a classical "don't care", because z_i can be assigned any value (including the cheapest option of no assignment) and still be compatible with the characteristic function.

This flexibility can be exploited to minimize the size of the s-graph, because the ASSIGN label a_v could in fact be any function which satisfies the conditions shown at the bottom of the page. Note that the two cases do not cover all possible input combinations and, hence, we have some flexibility in the implementation.

3) S-Graph Optimization:

a) Optimization by reordering: An s-graph can be optimized by reordering the nodes to minimize size and/or depth. In practice, it is more efficient to consider the ordering before

---

1) is one whenever

$$\left(S_{z_j | i<j\leq m+l, z_j \text{ is an output}} \chi^f_{z_i=1}\right) \wedge \overline{\left(S_{z_j | i<j\leq m+l, z_j \text{ is an output}} \chi^f_{z_i=0}\right)}$$

is one

and

2) is zero whenever

$$\left(S_{z_j | i<j\leq m+l, z_j \text{ is an output}} \chi^f_{z_i=1}\right) \vee \overline{\left(S_{z_j | i<j\leq m+l, z_j \text{ is an output}} \chi^f_{z_i=0}\right)}$$

is zero.

building the complete graph. There are three major classes of variable orderings.

i) Ordering each output *after* its support yields an s-graph where all the decision computation is done by TEST's. ASSIGN nodes are labeled only with actions on output variables (i.e., only with output variables and constants in the reactive function representation).
ii) Ordering each output *before* its support yields an s-graph without TEST nodes.
iii) All other orderings yield an s-graph with a mix of TEST and ASSIGN nodes.

b) Ordering outputs after their support: Our current implementation uses the first ordering scheme. In that case, it is easy to show that the structure of the s-graph corresponds exactly to that of a BDD representing CFSM's reactive function. Informally, TEST nodes correspond to BDD nodes associated with inputs of the reactive function, while ASSIGN nodes correspond to the outputs. Therefore, *optimization of the s-graph can be done directly on the* BDD representing the CFSM characteristic function.

We heuristically optimize the size of this BDD by dynamic variable reordering, using the "sift" algorithm [31]. Sifting moves one variable at a time up and down in the ordering, and freezes it in the position where the BDD size is minimized. In our case we must add the constraint that *no output can sift before any input in its support*.

The s-graph is built using the same ordering as the sifted transition function BDD and, thus, it has the following.

- Each input variable is tested only once per path. This provides the minimum depth s-graph, and thus implies a heuristically *minimum execution time*.
- The ordering of the variable tests is heuristically *optimal for code size*, in the sense that no single variable can be moved in the ordering while decreasing the size of the BDD and, hence, of the s-graph.

c) Ordering outputs before their support: The second ordering scheme, which is implemented in the current version of the Esterel compiler ([9]), can be implemented by directly building a Boolean circuit implementing the reactive function. The Boolean circuit is optimized using, e.g., the logic synthesis algorithms described in [33]. The s-graph can now be constructed as a string of ASSIGN vertices, one for each action. For example, the s-graph in Fig. 1 would be reduced to four ASSIGN vertices with the following labels[5]:

$$\begin{aligned}
&\texttt{emit_y} := \texttt{present_c} \wedge (\texttt{a} = ?\texttt{c});\\
&\qquad \texttt{a}' := \texttt{ITE}(\neg\texttt{present_c}, \texttt{a}, \texttt{a}');\\
&\qquad \texttt{a}' := \texttt{ITE}(\texttt{present_c} \wedge \neg(\texttt{a} = ?\texttt{c}), \texttt{a}+1, \texttt{a}');\\
&\qquad \texttt{a}' := \texttt{ITE}(\texttt{present_c} \wedge (\texttt{a} = ?\texttt{c}), 0, \texttt{a}').
\end{aligned}$$

The s-graph obtained in this way has no TEST vertices. Hence, all its executions take exactly the same time, if we ignore the effects of the memory hierarchy and of different execution times for the same arithmetic operation with different input data. This property is very important for highly critical real-time systems where absolute exactness in execution time *prediction* is a key for safe operation.

Experimentally we have seen that this method of s-graph construction, even though it could in principle offer better opportunities than the first one due to the more general sharing properties of Boolean expressions with respect to BDD's, in practice yields larger and slower code. We did not experiment with intermediate orderings, and we leave this exploration to future work.

d) Optimization by collapsing test nodes: We have also experimented with optimization of TEST nodes with respect to the first ordering scheme, by allowing each TEST node function to depend on more than one variable. Just as Boolean logic can be made more efficient by factoring out common subexpressions, both the size and speed of code generated from the s-graph can potentially be improved by judiciously combining TEST nodes and, thereby, factoring out common test expressions.

The algorithm performs a depth-first search from the BEGIN node to generate closed subgraphs. A closed subgraph is one in which all incoming edges share a common parent; a closed subgraph of TEST nodes can be collapsed without changing the functionality of the s-graph. Once a set of subgraphs is determined, each is replaced by a single TEST node. The C-code is generated from the Boolean function associated with each collapsed TEST nodes in two different ways:

i) by generating an if-then-else statement based on the truth table;
ii) by generating a Boolean network implementing the required function. In this case, we create a temporary variable in the C-code for each internal variable in the Boolean network, and an if-then-else statement in the C-code for each node function in the Boolean network.

In a series of experiments including Boolean network optimization and two-level and multilevel C-code generation, we never observed an improvement in the final running time or size of the generated code. As a result, we do not currently use TEST node collapsing.

4) S-Graph to C Translation: The s-graph obtained according to one of the procedures described in the previous section can now be translated into C code to be compiled on the target machine. There are a number of optimization operations that can be performed on the s-graph before final code generation. Some of them, such as common subexpression factoring, etc., are common to general-purpose compilers and will not be described in detail here.

The final translation of the s-graph into C (or any other high-level language) is straightforward due to the direct correspondence between s-graph node types and basic C primitives. The fact that the code is so unstructured may hinder its readability, but allows greater efficiency. In our codesign methodology the designer should not see this low-level, just like the user of a general-purpose compiler should never have to look at the assembly code. The generated C code contains compiler directives that relate directly the object code with the *source language* files that were used for the CFSM specification (e.g., using Esterel or Verilog). This allows any

[5]The value of `ITE(x,y,z)` is `y` if `x` is `1` and `z`, otherwise.

source level debugger used for the embedded software to display directly the original code and variables.

A TEST node is translated to a string of *if* or *switch* statements and an appropriate number of *goto*s. A target-dependent parameter can be used to specify how many children a TEST node must have in order to make an *if*-based implementation more convenient than a *switch*-based one.[6] An ASSIGN node is translated to an assignment. Appropriate declarations of local and global variables are also inserted into the output code.

Note that this code is very different from standard hand-written structured code, and is almost like a *portable assembly code*. We could also produce true assembly code from an s-graph, but this would require to solve too many processor-dependent issues, like instruction selection, register assignment, and so on.

C. Software Cost and Performance Estimation

Hardware/software partitioning and software synthesis for real-time embedded systems require accurate and quick estimates of code size and of minimum and maximum execution time.

The following two aspects of the problem must be considered:

1) the structure of the code, e.g., loops and false paths[7];
2) the system on which the program will run, including the CPU (instruction set, interrupts, etc.), the hardware architecture (cache, etc.), the operating system, and the compiler.

The s-graph structure is very similar (as shown in Section III-B4) to the final code structure and, hence, helps in solving the problem of cost and performance estimation as follows:

- each vertex in an s-graph is in one-to-one correspondence with a statement of the synthesized C code;
- the form of each statement is determined by the type of the corresponding vertex.

This means that the resulting C code is poorly structured from a user's point of view, but is simple enough that the effects of the target system on the execution time and code size of each vertex type can be easily determined.

Timing analysis is simple because s-graphs are acyclic: looping is dealt with at the operating system level. Moreover, false paths can be determined with a good degree of accuracy from the structure of the CFSM network, e.g., by computing event incompatibility relations.

Cost estimation can, hence, be done with a simple traversal of the s-graph. Costs are assigned to every vertex, representing its estimated execution cycle requirements and code size (including most effects of the target system).

1) Cost Estimation on the S-Graph: Our estimation method consists of first determining the cost parameters for the target system and then applying those parameters to the s-graph to compute the minimum and maximum number of execution cycles and the code size.

Each vertex is assigned two cost parameters, one for timing and one for size, which depend on the type of the vertex and the type of the input and/or output variables of the vertex. Edges may also have an associated cost, as the *then* and *else* branches of an *if* statement generally take different times to execute. Currently, we use 17 cost parameters for calculating execution cycles, 15 for code size, and four for characterizing the system (e.g., the size of a pointer).

The parameters for execution time or code size correspond to the kind of statements generated from a node in the s-graph. These are as follows:

- a TEST node detecting the presence of a signal (which yields an RTOS function call);
- a TEST node branching on a multivalued expression (which yields an *if* or *switch* statement);
- an ASSIGN node emitting a signal (which yields an RTOS call),
- an ASSIGN node which assigns an expression to a variable (which yields an assignment).

In the case of a TEST node with two outgoing edges, the cost parameters for each edge (i.e., the true case and the false case) are stored explicitly. For a TEST node which has more than three edges, the execution time for the k-th edge is represented as $T_{\text{switch}} = C_{\text{base}} + kC_{\text{case}}$, by using two parameters C_{base} and C_{case}.

The other parameters for the execution time and code size are defined for:

- calling and returning from a C function with a given number of local variables and parameters;
- a branch operation (generated from a *goto* statement);
- initialization of a local variable;
- average execution time and size for predefined software library functions (currently about 30 arithmetic, relational and logical functions, such as `ADD(x1,x2)`, `OR(x1,x2)`, `EQ(x1,x2)`, $\cdots$);
- the size of pointers;
- the size of integer variables.

The functions used in the data-flow graph portion of a CFSM can also be user-defined. A user-derived cost for those function should be given by hand to the estimation algorithm.

The cost parameters are determined for each target system (CPU, memory/bus architecture, compiler) with a set of sample benchmark programs. These programs are written in C and consist of about 20 functions, each with 10–40 statements. Each *if* or assignment statement which is contained in these functions has the same style as one of the statements generated from a TEST or ASSIGN vertex. The value of each parameter is determined by examining the execution cycles and the code size of each function. A profiler or an assembly-level code analysis tool, if available, can be used for this examination. We are currently using an internally developed cycle calculator for the Motorola 68HC11, the *pixie* tool for MIPS R3000 CPU's, and the profiling tool provided with a commercial in-circuit emulator for the Motorola 68 332.

[6] This is done for very simple compilers that do not have the capability of performing this optimization. Such compilers are indeed encountered with standard micro-controllers.

[7] A path in an s-graph is false if it can never be executed, e.g., due to conflicting Boolean conditions.

The calculation of software performance can be done dynamically or statically after tagging each line of code with its estimated execution time. Dynamic calculation can be done with realistic inputs by using the simulation environment described in [30], where both the structure of the synthesized code (e.g., false or seldom executed paths) and the architecture of the target system (e.g., preemptive scheduling policy and interrupts) can be considered. Static calculation, useful for example for worst case execution time analysis in the context of real-time scheduling, can be done by using graph traversal algorithms. Assume that E is the number of edges in the s-graph and N the number of nodes. The minimum execution cycles can be calculated by finding a minimum-cost path based on Dijkstra's shortest path algorithm from the BEGIN to the END vertex of the s-graph $(O(E \log N))$. The maximum execution cycles can be calculated by finding a maximum-cost path based on the PERT longest path algorithm $(O(E))$. The code size, useful for ROM cost estimation, can be calculated simply by summing the code size parameters for all the vertices of the s-graph $(O(E))$.

IV. Generation of the Real-Time Operating System

In Section III we described the software generation process for individual CFSM's. To implement a valid behavior of a *network* of CFSM's, additional code is needed to perform the following functions:

- schedule individual CFSM's implemented in software (sw-CFSM's) such that each one is executed in a timely manner;
- provide a mechanism for event emission and detection between sw-CFSM's;
- provide a mechanism for transferring events between CFSM's implemented in hardware (hw-CFSM) and those implemented in software;
- ensure that consumption of input events by a sw-CFSM is consistent with the semantics described in Section II-D.

We propose to synthesize this code automatically. We call this code *real-time operating system* (RTOS), because it performs communication and scheduling functions traditionally performed by an operation system.

A. Scheduling of sw-CFSM's

Every sw-CFSM's can be in one of two states: *disabled* (when there are no events at its inputs) or *enabled* (when such events exist). A sw-CFSM's becomes enabled when any of its input events occur. An enabled sw-CFSM needs to be executed. Once it finishes its execution, a sw-CFSM is disabled.

The RTOS must keep track of the state of every sw-CFSM. Moreover, it must decide which one of the (possibly many) enabled sw-CFSM's to execute. The set of rules used to make this decision is called the *scheduling policy*. In the current implementation, a user chooses off-line one of the several available scheduling policies (round-robin, static-priority based, with or without preemption). The user can also instruct the system to bypass the RTOS and "chain" certain executions of CFSM's into a single task, thus reducing scheduling and communication overhead. We expect that eventually it will be possible to automatically select a scheduling policy which provably meets all the timing constraints, based on the frequency of events in the environment and on the estimated execution times of the sw-CFSM's and of the RTOS ([4]). In any case, once a scheduling policy is chosen, C (and some assembly) code that implements that policy at run-time is automatically generated [15].

B. Communicating Events Between sw-CFSMs

When a sw-CFSM emits an event, every other sw-CFSM sensitive to that event must be informed of it and enabled. To every CFSM we assign a set of private flags, one for each input, to indicate whether that event has occurred since the previous transition. A CFSM is scheduled to run by the RTOS whenever it has at least one input flag set (its actual execution may be delayed by CFSM's with higher priority, according to the scheduling policy). Once it is run, the CFSM checks its input flags to decide (using the s-graph) which one (if any) of its transitions to execute. Thus, the emission of an event consists of setting all the appropriate flags and enabling all the appropriate tasks, and the detection of an event is a simple check on the status of a flag.

C. Communicating Events Between hw- and sw-CFSM's

Events emitted by a sw-CFSM and consumed by a hw-CFSM are communicated through a memory mapped input–output (I/O) port of the micro-controller. Events emitted by a hw-CFSM are delivered to a sw-CFSM by one of the following mechanisms.

Polling: In this case, a hw-CFSM only sets an appropriate bit on an I/O port of the micro-controller. An automatically generated polling routine is periodically scheduled to execute and if it finds the bit set, it will execute the event emission routine. This solution has minimal hardware requirements, but does introduce an additional delay because an event cannot be detected by a sw-CFSM before the polling routine is executed.

Interrupts: In this case, if a hw-CFSM wants to emit an event, it requests an interrupt. When the interrupt is serviced, the corresponding interrupt-service routine (ISR) is executed. By default, an ISR contains only an event emission routine. However, the user has the option to specify that for designated events, all sw-CFSM's sensitive to that event are also to be executed inside the ISR. In this way, the most critical tasks can be given immediate attention.

By default, all events are communicated through interrupts, but a user may specify any number of events to be polled. This will typically depend on the interrupt handling capabilities of the processor used in the implementation.

For completeness, let us note that communicating events between hw-CFSM's is easily accomplished via buffer registers (one bit for each input event).

D. Consumption of Events

A CFSM is enabled whenever any of its input events occur. Thus, it may happen that the software routine implementing a CFSM is executed, but no transitions are enabled, and thus none is executed (i.e., no ASSIGN nodes are visited while traversing the s-graph from the BEGIN to the END node). The RTOS ensures that in this case input events are not consumed, but rather preserved for the next execution.

As described in Section II, a CFSM may execute a transition if at some moment in time the set of input events matches one of those specified by the transition relation. However, since a CFSM checks input events in sequence (determined by the ordering of TEST nodes), it may happen that a CFSM will detect a particular set of events at its inputs that does not correspond to any single time point. Consider, for example, a CFSM with two input events A and B, and assume that the CFSM first checks its A presence flag and then its B presence flag. In a straightforward (and incorrect) implementation, the following sequence may occur:

1) the CFSM checks the A flag and finds that A has not occurred,
2) the CFSM is interrupted,
3) A occurs,
4) B occurs,
5) the CFSM continues the execution, finds that B has occurred and executes a transition which is enabled only if B has occurred and A has not.

Such a behavior is erroneous because at no point in time was it true that B had occurred and A had not. To avoid this problem, the generated RTOS ensures that once a CFSM starts reading its input event flags, no new flags can be set until the CFSM finishes its execution. However, any events occurring in that time period are remembered and can be consumed in the following execution.

E. Comparison with Commercial RTOS's

Instead of automatically generating an RTOS, it is also possible to use a commercially available one. For this we only need to implement the event emission and detection mechanisms using the event flag services provided by the RTOS, and provide the RTOS scheduler with enough information (usually task execution times and deadlines) to enable it to perform its duties. However, we believe that our approach has several advantages. First, since the RTOS has a fixed communication structure (neither the number of tasks nor the sensitivity of tasks to events changes over the lifetime of the generated RTOS), the emission and detection of events can be extremely efficiently implemented, and in some cases (when a task is sensitive to a single event) completely avoided. Second, since only the necessary functionality is generated, the size of the generated RTOS is often much smaller than the size of commercial ones. Finally, in our approach one can easily experiment with tradeoffs, e.g., between scheduling policies or different event input mechanisms (polling versus interrupts). Commercial RTOS's typically do not provide such a flexibility.

TABLE I
RESULTS OF THE COST/PERFORMANCE ESTIMATIONPROCEDURE

function	estimated		measured		perc. difference	
	timing	size	timing	size	timing	size
BELT	353	433	270	392	30	10
ODOMETER	379	287	380	266	0	7
FUEL	541	657	555	631	-2	4
SPEEDOMETER	851	601	872	621	-2	-3
NORMALIZE	920	479	999	458	-7	4
CROSS_DISP	3795	4169	4040	5182	-6	-19
DETECT_EDGE	850	511	810	484	4	5
QUAD2SIGN	919	509	928	509	0	0
COIL_SWITCH	1038	677	912	712	13	-4
TIMER	1005	1417	859	1137	16	24

TABLE II
EFFECT OF DIFFERENT TEST VARIABLE ORDERINGS

function	measured size (bytes)		
	in before out	support	two-level
BELT	396	392	1029
ODOMETER	350	266	365
FUEL	1148	631	872
SPEEDOMETER	724	621	714
NORMALIZE	458	458	516
CROSS_DISP	6275	5182	6274
DETECT_EDGE	519	484	612
QUAD2SIGN	799	509	931
COIL_SWITCH	1699	712	1136
TIMER	32447	1137	1206

V. EXPERIMENTAL RESULTS

We first report the results of the cost/performance estimation procedure and of the s-graph synthesis procedure applied to a subset of a car dashboard control system. We then compare a manual design and the results of software synthesis for a real industrial example, a shock absorber controller.

In all cases, the numbers are given for a Motorola 68HC11 micro-controller. They are obtained using our estimation package, as well as by actual measurements done on the output of the INTROL C compiler for the 68HC11. The timing columns are given in terms of the maximum number of clock cycles for a *single transition* of each CFSM and the code size columns are given in terms of bytes. All the results include both the control and the data part.

A. The Dashboard Controller

The example considered here is a subset of the functionality of a dashboard controller, that implements the computational chain from the wheel and engine speed sensors to the pulse width-modulated outputs controlling the gauges.

Table I summarizes the result of the cost estimation procedure, and compares it against an exact measurement of the code size and timing (maximum number of clock cycles), performed by analyzing the compiled object code.

Table II shows the effect of the different orderings in procedure **build** on the software size. The timing remains

TABLE III
COMPARISON OF SOFTWARE SYNTHESIS WITH ESTEREL

Program	Time	Size		Synthesis time
		Text	Data	
POLIS	41,920,700	15,008	3168	87.0
ESTEREL	103,299,313	40,112	11,312	199.5
ESTEREL_OPT	45,371,358	29,040	11,056	299.8

approximately the same, since only the order of the tests is changed. In both cases, the computed function is exactly the same. The only difference is the order of the variables, which affects the number of TEST nodes. In both cases we use dynamic reordering by sifting [31] (which is known to be more efficient than the static methods used, for example, in [6]). In the first case we restrict sifting so that all outputs appear after all inputs in the BDD. In the second case the constraint is relaxed as discussed in case 1 in Section III-B3, forcing each output to appear only after its own support. The difference in size is due to the sharing among subgraphs, which can be performed better in the second case. As a reference, we also compare the result with an implementation which uses a two-level multiway jump structure. The first jump is done based on the current state, the second jump is done based on the concatenation of all the decision variable into a single integer. The jumps are followed by an appropriate sequence of ASSIGN's. This simple implementation (similar to what is often done during structured hand-coding of reactive systems) performs better than the naive ordering, but worse than the optimized decision graph. The results are in bytes of code, after compiling with the -O option.

We have also tried to compile the same code using the MIPS compiler, which has much better optimization capabilities than the INTROL compiler, and the results are similar. This demonstrates that our BDD-based code restructuring optimizations are beyond the optimization capabilities of general-purpose compilers.

Finally, we compared our software implementation to that produced by ESTEREL v5 for the dashboard. These last experiments were done by compiling all the code on a DEC ALPHA and running a large simulation file. The results are shown in Table III. The software simulation time is given in cycles as reported by `pixie`, and the software size in bytes as reported by `size`. Only the reactive core code is compared (the simulation interface code is excluded), and the numbers for the dashboard modules have been summed to obtain the results shown. The final column gives the total elapsed time to generate the software implementation. Note that POLIS uses ESTEREL to process the CFSM's individually, while the ESTEREL compiler (shown in the last two lines) processes the whole design into a single FSM. Moreover, the majority of the time for the ESTEREL synthesis was taken by the C-compiler.[8]

The ESTEREL_OPT row shows the results using the Boolean circuit optimization inside the v5 compiler. This technique corresponds to ordering outputs before inputs in the approach described in Section III-B3, and shows that the possible saving in code size due to the better sharing opportunities offered by Boolean functions in this case does not help (nor it does in any of the practical cases that we have examined so far).

Our synthesis used the default variable ordering scheme, with single-pass dynamic variable ordering (sift) under the restriction that each output appear after its support.

[8]The C-code was always compiled with the -O option, and the -O limit flag had to be significantly increased in order to utilize this option on the ESTEREL code.

B. The Shock Absorber Controller

We have also performed a complete redesign of a real example, a shock absorber controller. No detailed module-by-module comparison with the manual design size is possible, due to a different functional level organization chosen for the redesign.

The code size of the synthesized implementation is 46 639 bytes of ROM and 10 229 bytes of RAM, including the RTOS (round-robin scheduler and I/O drivers), on a 68HC11. The hand-designed implementation had a ROM size of 32 Kbytes and a RAM size of 8 Kbytes.

The performance of the synthesized implementation was comparable to that of the manual implementation, since both satisfied the 12 μs I/O latency required by the specification.

The increase in ROM and RAM size is due mostly to the fact that all variables used by an s-graph are copied upon entry in the corresponding routine, to provide a safe implementation of the update of their next-state values. We are working on a data flow analysis step that will allow us to detect *write-before-read* cases that require such buffering, and reduce ROM and RAM, as well as CPU time, when no such buffering is needed to correctly implement the CFSM semantics.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a new methodology for the synthesis of software for embedded real-time control-dominated systems. The methodology exploits the use of a FSM specification, and unlike classical compilation algorithms starts from a description of the *function* to be computed, rather than from one operational implementation of it. This allows the use of powerful optimization algorithms based on Boolean function manipulation methods.

The internal representation that we use is also the basis of a quick but fairly precise cost- and performance-estimation procedure. The procedure is based on assigning cost parameters to the control/data-flow graph, and can be easily customized for different CPU's and runtime environments.

In the future we plan to exploit the cost-estimation procedure to perform global optimizations aimed at satisfying timing and size constraints, with a much finer tuning than is currently possible. Moreover, the current code size minimization algorithm uses a single order for variables along all s-graph paths. While this is required in BDD's in order to ensure canonicity of representation, it is not clear whether it helps in the software synthesis case. We are thus planning to explore unordered variants of decision diagrams for our software optimization [29]. We are also exploring the coupling between scheduling algorithms and code synthesis, to allow

the scheduling procedure to transmit user-defined constraints to the compilation steps.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Reading, MA: Addison-Wesley, 1988.

[2] S. B. Akers, "Binary decision diagrams," in *IEEE Trans. Comput.*, vol. C-27, pp. 509–516, June 1978.

[3] P. Ashar and S. Malik, "Fast functional simulation using branching programs, presented at *Int. Conf. Computer-Aided Design*, Nov. 1995.

[4] F. Balarin and A. Sangiovanni-Vincentelli, "Schedule validation for embedded reactive real-time systems," presented at *Design Automation Conf.*, June 1997.

[5] F. Balarin, E. Sentovich, M. Chiodo, P. Giusto, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli, *Hardware-Software Co-Design of Embedded Systems—The POLIS Approach*. Norwell, MA: Kluwer Academic, 1997.

[6] R. Bergamaschi, R. Camposano, and M. Payer, "Allocation algorithms based on paths," *Integration, the VLSI J.*, vol. 13, pp. 283–299, 1992.

[7] G. Berry, *The Constructive Semantics of Pure Esterel*, submitted for publication; Available: FTP://www.inria.fr/meije/esterel/papers/ constructiveness.ps.gz.

[8] G. Berry, P. Couronné, and G. Gonthier, "The synchronous approach to reactive and real-time systems," *Proc. IEEE*, vol. 79, Sept. 1991.

[9] G. Berry, 1996, see http://cma.cma.fr/Esterel.

[10] R. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, pp. 677–691, Aug. 1986.

[11] J. Buck, S. Ha, E. A. Lee, and D. G. Masserschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. J. Comput. Simulation*, special issue on Simulation Software Development, Jan. 1990.

[12] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli, "A formal specification model for hardware/software codesign," Univ. Calif. Berkeley, CA, Tech. Rep. UCB/ERL M93/48, June 1993.

[13] ———, "A formal specification model for hardware/software codesign," presented at Int. Workshop Hardware-Software Codesign, 1993.

[14] P. Chou, E. Walkup, and G. Borriello, "Scheduling for reactive real-time systems," *IEEE Micro*, vol. 14, pp. 37–47, Aug. 1994.

[15] D. Engels, "Real-time task level scheduling in the POLIS co-design environment," Master's thesis, Univ. California, Berkeley, 1995.

[16] D. D. Gajski, Ed., *Silicon Compilation*, Reading, MA: Addison-Wesley, 1988.

[17] R. K. Gupta, C. N. Coelho Jr., and G. De Micheli, "Program implementation schemes for hardware-software systems," *IEEE Comput.*, vol. 27, pp. 48–55, Jan. 1994.

[18] W. A. Halang and A. D. Stoyenko, *Constructing Predictable Real Time Systems*. Norwell, MA: Kluwer Academic, 1991.

[19] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Norwell, MA: Kluwer Academic, 1993.

[20] D. Har'el, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: A working environment for the development of complex reactive systems," *IEEE Trans. Software Eng.*, vol. 16, no. 4, Apr. 1990.

[21] D. Ku and G. De Micheli, *High Level Synthesis of ASIC's Under Timing and Synchronization Constraints*. Norwell, MA: Kluwer Academic, 1992.

[22] L. Lavagno, P. McGeer, A. Saldanha, and A. Sangiovanni-Vincentelli, "Timed Shannon circuits: A power efficient design style and synthesis tool," in *Proc. Design Automation Conf.*, June 1995, pp. 254–260.

[23] C. Y. Lee, "Representation of switching functions by binary decision programs," *Bell Syst. Tech. J.*, vol. 38, pp. 985–999, 1959.

[24] C. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. ACM*, vol. 20. no. 1, pp. 44–61, Jan. 1973.

[25] S. Malik, A. R. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic verification using binary decision diagrams in a logic synthesis environment," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1988, pp. 6–9.

[26] P. Marwedel and G. Goossens, Eds., *Code Generation for Embedded Processors*. Norwell, MA: Kluwer Academic, 1995.

[27] M. C. McFarland, A. C. Parker, and R. Camposano, "Tutorial on high-level synthesis," in *Proc. 25th ACM/IEEE Design Automation Conf.*, Anaheim, CA, 12–15 June 1988, pp. 330–336.

[28] P. McGeer, K. McMillan, A. Saldanha, A. Sangiovanni-Vincentelli, and P. Scaglia, "Fast discrete function evaluation using decision diagrams," presented at *Int. Conf. Computer-Aided Design*, Nov. 1995.

[29] C. Meinel, *Modified Branching Programs and Their Computational Power*. New York: Springer-Verlag, 1989.

[30] C. Passerone, L. Lavagno, M. Chiodo, and A. Sangiovanni-Vincentelli, "Fast hardware/software co-simulation for virtual prototyping and trade-off analysis," presented at *Design Automation Conf.*, Anaheim, CA, June 1997.

[31] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," presented at *Int. Conf. Computer-Aided Design*, Santa Clara, CA, Nov. 1993.

[32] R. Saracco, J. R. W. Smith, and R. Reed, *Telecommunications Systems Engineering Using SDL*. Amsterdam, The Netherlands: North-Holland/Elsevier, 1989.

[33] 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," Univ. California. Berkeley, Tech. Rep. UCB/ERL M92/41, May 1992.

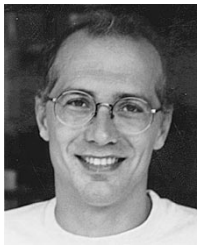
[34] T. Shiple, G. Berry, and H. Touati, "Constructive analysis of cyclic circuits," presented at *European Design and Test Conf.*, Paris, France, Mar. 1996.

[35] W. Wolf, A. Takach, C.-Y. Huang, and R. Manno, "The princeton university behavioral synthesis system," presented at *Design Automation Conf.*, Anaheim, CA, June 1992.

[36] S. Y. Yee, "An esterel to SHIFT compiler for a hardware/software codesign environment," Master's thesis, Univ. Claifornia, Berkeley, 1994.

Felice Balarin (S'90–M'95) received the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley in 1994.

Since then, he has been a Research Scientist at the Cadence Berkeley Laboratories, Berkeley, CA. His research is focused on development and application of formal methods to design, verification and timing analysis of systems consisting of both hardware and software.

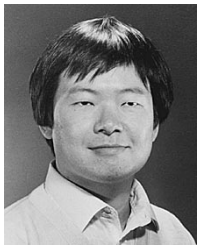


Massimiliano Chiodo (M'91) received the degree in physics from the Universita' degli Studi, Milan, Italy.

He worked for various companies in the embedded software field until 1989 when he joined Magneti Marelli, Turin, Italy. While with Magneti Marelli he spent several months as Visiting Industrial Fellow at the University of California at Berkeley's CAD Group where he worked on formal verification and hardware/software co-design. In the summer of 1995 he joined Cadence Design System, San Jose, CA, where he works in the software/hardware co-design area.

Paolo Giusto received the Doctor of Information Sciences from the Universita' di Torino, Italy, and received his MSc degree in hardware engineering from CEFRIEL in Milan.

He works for Cadence Design Systems, Sunnyvale, CA, as a member of the Design Methodology Team for the Co-Design Technology Group. He was a Visiting Industrial Fellow for Magneti Marelli, at the University of California, Berkeley from 1992 to 1994 where he worked on hardware-software co-design.



Harry C. Hsieh (S'93) received the B.S. degree from University of Wisconsin, Madison, and the M.S. degree from Stanford University, Stanford, CA. He is currently a Ph.D. degree candidate 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, Cupertino, CA, and has worked at IBM's Federal System Division and T.J. Watson Research Center, Hawthrone, NY. His primary research interests include system-level design methodologies, logic synthesis, and design of embedded systems.

Attila Jurecska received the M.S. degree in electrical engineering from the Technical University of Budapest, Hungary, and the M.S. degree in information technology from CEFRIEL, Milan, Italy, in 1990 and 1992, respectively.

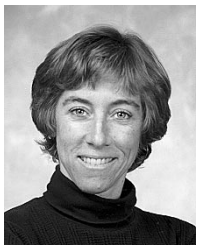
From September 1992 to December 1997, he was employed by Magneti Marelli, Turin, Italy as a Software Engineer working on hardware-software co-design of embedded controllers. From August 1994 to December 1996, he was a Visiting Industrial Fellow for Magneti Marelli at the University of California, Berkeley. He worked on the development of Polis hardware-software co-design environment with the hardware-software co-design group of the University of California, Berkeley. Since March 1998, he has worked for Synopsys, Inc., Beaverton, OR, as a Senior R&D Engineer in the Eagle Technology Group. His present research interest includes hardware-software co-design and co-verification.

Luciano Lavagno (S'88–M93) graduated *magna cum laude* in electrical engineering from Politecnico di Torino, Torino, Italy in 1983. In 1992, he received the Ph.D. degree in electrical engineering and Computer Science from the University of California at Berkeley (U.C. Berkeley).

From 1984 to 1988, he was with CSELT Laboratories, Torino, Italy, where he was involved in an ESPRIT project that developed a complete high-level synthesis system. In 1988, he joined the Department of Electrical Engineering and Computer Science of the University of California at Berkeley, where he worked on logic synthesis and testing of synchronous and asynchronous circuits. Between 1993 and 1998, he was the architect of the POLIS project, developing a complete hardare/software co-design environment for control-dominated embedded systems. He has also been a consultant for various EDA companies, such as Synopsys and Cadence. He is currently an Associate Professor at the University of Udine, Udine, Italy, and a Research Scientist at Cadence Berkeley Laboratories. His research interests include the synthesis of asynchronous and low-power circuits, the concurrent design of mixed hardware and software systems, and the formal verification of digital systems. He is the author of a book on asynchronous circuit design and the co-author of a book on hardware/software co-design of embedded systems. He has published over 80 journal and conference papers.

In 1991 Dr. Lavagno received the Best Paper award at the Design Automation Conference in San Francisco, CA. He has served on the technical committees of several international conferences in his field (the Design Automation Conference, the International Conference on Computer Aided Design, and the European Design Automation Conference).

Alberto Sangiovanni-Vincentelli, (M'74–SM'81–F'83) for a photograph and biography, see p. 190 of the February 1999 issue of this TRANSACTIONS.



Ellen M. Sentovich received the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley, in 1993.

Since 1994, she has been with Cadence Berkeley Laboratories, Berkeley, CA. From March 1995 to October 1996 she was on sabbatical working with the MEIJE project, which is shared between the Ecole des Mines de Paris' Centre de Mathematiques Appliquees and INRIA, the French national computer science research lab. While there, she worked with the Esterel team on specification, analysis, and synthesis of reactive systems. Since returning to Cadence, her research has focused on system-level design issues, including specification and languages, models of computation, semantics, synthesis, and optimization.

Kei Suzuki received the B.S., M.S., and Dr. Eng. degrees from Waseda University, Tokyo, Japan, in 1984, 1986, and 1989, respectively all in electrical engineering.

In 1989, he joined the Central Research Laboratory, Hitachi, Ltd., and is now a Senior Researcher. From 1993 to 1994, he was a Visiting Industrial Fellow at the department of Electrical Engineering and Computer Sciences, University of California, Berkeley. His research interests include system-level design methodology and hardware/software codesign.

Dr. Suzuki is a member of ACM, the Institute of Electronics, Information and Communication Engineers of Japan, and of the Information Processing Society of Japan.