Synchronous Approach to the Functional Equivalence of Embedded System Implementations

Harry Hsieh, *Member, IEEE*, Felice Balarin, *Member, IEEE*, Luciano Lavagno, *Member, IEEE*, and Alberto Sangiovanni-Vincentelli, *Fellow, IEEE*

***Abstract*—Design space exploration is the process of analyzing several functionally equivalent alternatives to determine the most suitable one. A fundamental question is whether an implementation is consistent with the high-level specification or whether two implementations are "equivalent." The synchronous assumption has made it possible to develop efficient procedures for establishing functional equivalence between different implementations in the domains of synchronous circuits and synchronous reactive systems. We extend this notion to embedded systems that do not satisfy the synchronous assumption inside their boundaries but only at the interface with the environment. Leveraging this property, we define synchronous equivalence for embedded systems that strongly resembles the concept of functional equivalence for sequential circuits. We develop efficient synchronous equivalence analysis algorithms for embedded system designs. The efficiency comes from analyzing the behavior statically on abstract representations, at a cost that some of the negative results may be false, i.e. the analysis is conservative. We develop primitives for making the representation more/less abstract, trading off complexity of the algorithms with the conservativeness of the results. We apply our analysis algorithms to an ATM switch and demonstrate that synchronous equivalence opens design exploration avenues uncharted before.**

***Index Terms*—Embedded systems, equivalence checking, formal methods, hardware software co-design, implementation verification, symbolic techniques, symbolic verification.**

I. INTRODUCTION

CURRENT embedded system design practice is quite informal and application-specific. Designers often start with a requirement written in a natural language and break the design into hierarchy so as to better manage the complexity. They then take each part of this hierarchy, use "intuition" to pick a particular interpretation of the requirement, and write a so-called reference (or golden) model in VHDL, Verilog, C, or any other language that has an execution semantics. The reference model is executed on a computer to investigate whether it satisfies a set of requirements, including a match with the original informal specification. This is done on parts of the design and the result is extrapolated to the whole through simple logic deduction. A (candidate)[1] implementation is then generated through a combination of manual labor and tools that are often poorly connected. The correctness and optimality of the (candidate) implementations are assessed with filtered simulation traces obtained from the reference model and from the candidate implementations. The best implementation is chosen based on issues such as performance, cost, reliability, and flexibility. This contorted and highly informal design flow is obviously very error-prone and does not promote efficient design space exploration since the set of "correct" implementations cannot be precisely identified.

A fundamental point of clarification to improve the design methodology is the formal definition of correctness. Because embedded system designs are inherently complex, the principle of "separation of concerns" in specification and verification is essential. Functional correctness and timing should be verified independently. This principle is the basis of the synchronous design methodology for sequential circuits [1], where designs are verified by paying attention only to the Boolean functions computed by the circuits, irrespective of the propagation time. Timing can then be verified independently by performing a worst-case timing analysis and making sure that this bound is within the clock cycle.

We extend this approach to a model of embedded systems supported by codesign finite state machines (CFSMs) [2] while retaining the fundamental idea of separation between timing and functionality. We establish *synchronous equivalence*, a "functional" equivalence among a set of candidate implementations of embedded system specifications. Just as the sequential circuit design methodology abstracts away gate delays of the implementations and enables speed/area tradeoffs among functionally equivalent implementations, we abstract away the delays of embedded system computational resources. The synchronous equivalence relation divides the design space into synchronous equivalence classes. Within an equivalence class, different implementations represent different tradeoffs among speed, power, reliability, expandability, and cost. Exact equivalence analysis requires exhaustive simulation or reachable state analysis methods (e.g., formal verification tools [3], [4]). Since this is too complex for most practical systems, we propose conservative (but more efficient) methods that are abstract, static, and structural.

In Section II, we briefly review a formal model for control dominated embedded system design, CFSMs that provides a

Manuscript received August 7, 2000; revised January 31, 2001. This paper was recommended by Associate Editor J. Kukula.

H. Hsieh was with the Mainline Systems Laboratory, Hewlett-Packard, Cupertino, CA 95014 USA. He is now at 2001 Addison Street, Berkeley, CA 94704 USA.

F. Balarin is with the Cadence Berkeley Laboratories, Berkeley, CA 94704 USA (e-mail: felice@cadence.com).

L. Lavagno is with the Università di Udine, 33100 Udine, Italy (e-mail: lavagno@uniud.it).

A. Sangiovanni-Vincentelli is with the University of California at Berkeley, Berkeley, CA 94710 USA (e-mail: alberto@ic.eecs.berkeley.edu).

Publisher Item Identifier S 0278-0070(01)05204-6.

[1]An "implementation" generated through this process may only be considered a candidate because it may not be correct. It is not generated through formal refinement. Some *ad hoc* manual procedures are involved.

0278–0070/01$10.00 © 2001 IEEE

convenient representation of the design space. In Section III, we introduce the synchronous equivalence relation. In Section IV, we present communication analysis that is efficient and effective in telling us whether two implementations are synchronously equivalent. We also show some results of applying this methodology to a real-life industrial example. Many levels of abstraction exist between high-level specifications and low-level implementations. Communication analysis operates at one specific level of abstraction. In Section V, we provide primitives to move around this abstraction/refinement scale, trading off computational complexity of the algorithm with the conservativeness of the result. We discuss future directions in Section IV.

II. Network of CFSMs

Embedded systems can be represented as networks of interacting CFSMs [2]. CFSMs extend finite state machines (FSM) with side-effect-free computation on the transition edges. The communication entities between CFSMs are events, which may or may not carry values. A CFSM can transition only when an input event has "occurred."

Each individual CFSM operates in a "locally synchronous" fashion with its own "clock." There is no *a priori* synchronization between CFSMs and all the local clocks are completely unsynchronized. There is no *a priori* relations between the local clocks and physical time.

Implementing a CFSM network involves allocating individual CFSMs to computation resources, assigning schedulers to shared resources, and synthesizing CFSMs into code or gate-level description for the resources. Implementation has the consequence of refining the relationship between the local clock and physical time. If the CFSM is to be implemented on a very large scale integration (VLSI) synchronous hardware, its local clock will become periodic and the clock period will be equal to the synchronous hardware clock. If the CFSM is to be implemented on a processor, the local clock will not have fixed period and will run at some multiple of the processor clock, depending on the execution delay of the code implementing each transition on that processor, which in turns depend on software synthesis and scheduling.

Since local CFSM clocks are unsynchronized, a network of CFSMs is inherently nondeterministic: for a fixed input sequence, many system responses are possible (and they are all equally valid). However, an implementation is deterministic: its response is unique for any fixed input sequence. In fact, we extend the notion of implementation to include any set of rules that resolve nondeterministic choices in a CFSM network (making it deterministic). For example, simulating a CFSM network requires resolving nondeterministic choices, so we consider a simulation model (often referred to as "reference model") to be an implementation.

Next, we give a formal definition of CFSM networks. To simplify the presentation of novel proposals, we have simplified significantly the original definition [2]. Extending our approach to the original model is quite straightforward, except for three features: valued events, internal states of CFSMs, and time. In the model presented here, events are pure, i.e., they can be present or not, but cannot carry a value, while the original definition allowed for valued events. This is not a serious limitations because for every network with valued events, it is possible to find an equivalent (though larger) one with pure events by treating events with different values as different events. The more serious limitations is that in the model presented here, CFSMs can have no internal state. We will remove this limitation in Section V-B, where we extend our approach to deal with internal states. Finally, the model presented here is untimed. Consequently, execution delays of CFSMs do not appear in our models and an implementation is determined only by a scheduling policy. This means that we can model only implementations with *delay-insensitive* policies [5], i.e., implementations whose behavior does not change when CFSM delay are changed (as long as the design as a whole does not get into race conditions with the environment). An example of scheduling policies that are not delay insensitive is the class that is referred to as "time slicing." In time slicing, each enabled component is allotted a certain amount of time and if the execution has not been completed within the allotted time, the component will be preempted. For a CFSM network, changing some execution delays of CFSMs may result in different time-slicing implementations that have quite different behaviors. Fortunately, many popular scheduling policies, including static priority and cyclic executive, are delay insensitive. We have chosen an untimed model, even though the concept of synchronous equivalence is valid for delay-sensitive implementations as well. However, the synchronous equivalence checking procedure that we propose is applicable only to delay-insensitive implementation, so we will present it in the simple untimed framework.

Formally, CFSM A is a triple (I_A, O_A, T_A), where I_A is the set of input events of A, O_A is the set of output events of A, and $T_A = \{T_A^{\mathrm{Out}}\colon \{0, 1\}^{|I_A|} \mapsto \{0, 1\} \mid \mathrm{Out} \in O_A\}$ is the set of transition functions of A. For each $e \in I_A$, we say that A is the consumer of e and write $\mathrm{Cons}(e) = A$. Similarly, for each $e \in O_A$, we say that A is the producer of e and write $\mathrm{Prod}(e) = A$. We also require that each CFSM is *reactive*, i.e., it cannot emit an output event unless some input event is present. More precisely, we require $T_A^{\mathrm{Out}}(0, 0, \ldots, 0) = 0$ for all $\mathrm{Out} \in O_A$. If a CFSM is executed with no inputs present, we call such an execution empty. An empty execution consumes no input events (because there are none) and produces no output events (because CFSMs are reactive).

A *CFSM network* is a triple $(\mathrm{PI}, \mathrm{PO}, \mathrm{C})$, where PI is the set of primary input events, PO is the set of primary output events, and C is the set of CFSMs. We require that PI and PO are disjoint. We say that $E = \mathrm{PI} \cup \mathrm{PO} \cup \bigcup_{A\in \mathrm{C}} I_A \cup \bigcup_{A \in \mathrm{C}} O_A$ is the set of events of the network. We require that each event in $E - \mathrm{PI}$ has a unique producer and that each event in $E - \mathrm{PO}$ has a unique consumer.

A *state* of a CFSM network is a function that assigns to each event either one or zero, indicating event presence or absence, respectively. Given a state s and a subset of events $X = \{x_1, \ldots, x_n\}$, we use $s(X)$ to denote $(s(x_1), \ldots, s(x_n))$. Given a CFSM A and a state s, we say that A is enabled in s if there exists $\mathrm{In} \in I_A$ such that $s(\mathrm{In}) = 1$. We use $\mathrm{Enabled}(s)$ to denote a set containing all CFSMs enabled in s.

An *implementation* Select of a CFSM network is a function that takes as an argument a network state and returns a subset of CFSMs, i.e., $\mathrm{Select}(s) \subset \mathrm{C}$ for each state s. Intuitively, *Select* indicates which CFSMs are chosen to run. Since an implementation models only a scheduling policy we will use terms implementation and scheduling policy interchangeably.

Given a CFSM network $(\mathrm{PI},\ \mathrm{PO},\ \mathrm{C})$, a scheduling policy *Select* and some $\mathrm{Emit} \subseteq \mathrm{PI}$, $\mathrm{Clear} \subseteq \mathrm{PO}$, we say that a network state s' is the $(\mathrm{Emit},\ \mathrm{Clear})$ *successor* of a network state s, if[2]

$$s'(e) = \begin{cases} (e \in \mathrm{Emit}) + s(e) * \overline{\mathrm{Running}(\mathrm{Cons}(e))}, \\ \quad \text{if } e \in \mathrm{PI} \\ \overline{(e \in \mathrm{Clear})} * (\mathrm{Emitted}(e) + s(e)), \\ \quad \text{if } e \in \mathrm{PO} \\ \mathrm{Emitted}(e) + s(e) * \overline{\mathrm{Running}(\mathrm{Cons}(e))}, \\ \quad \text{otherwise} \end{cases}$$

where

$$\mathrm{Running}(A) \triangleq (A \in \mathrm{Select}(s))$$
$$\mathrm{Emitted}(e) \triangleq \mathrm{Running}(\mathrm{Prod}(e)) * T^{e}_{\mathrm{Prod}(e)}(s(I_{\mathrm{Prod}(e)})).$$

In other words, an event that is neither primary input nor primary output exists in s' if it was emitted by its producer running in s or if it already existed in s and its consumer was not running in s. Similarly, a primary output event exists in s' if it was not consumed by the environment and it was emitted by its producer running in s or it already existed in s. Finally, a primary input event exists in s' if it was emitted by the environment or if it already existed in s and its consumer was not running in s.

Given a CFSM network $(\mathrm{PI}, \mathrm{PO}, \mathrm{C})$ and its scheduling policy *Select*, we say that a sequence

$$s_0,\ (\mathrm{Emit}_1,\ \mathrm{Clear}_1),\ s_1,\ (\mathrm{Emit}_2,\ \mathrm{Clear}_2), \ldots,\ (\mathrm{Emit}_n,\ \mathrm{Clear}_n),\ s_n \qquad (1)$$

where $s_0, \ldots, s_n$ are states of the CFSM network, $\mathrm{Emit}_1, \ldots, \mathrm{Emit}_n$ are subsets of primary inputs, and $\mathrm{Clear}_1, \ldots, \mathrm{Clear}_n$ are subsets of primary outputs, is a *run* of the implementation if initially no events are present, i.e., $s_0(e) = 0$ for all $e \in E$, and s_i is the $(\mathrm{Emit}_i,\ \mathrm{Clear}_i)$ successor of s_{i-1} for all $i = 1, \ldots, n$. If $\mathrm{Select}(s_i)$ is not empty, we say that i is a *scheduling point*. If in addition some A in $\mathrm{Select}(s_i)$ has some of its inputs present, we say that i is a *true scheduling point*. All executions that start in not-true scheduling points are empty and, thus, do not affect the presence of events in the network.

Note that our model allows an arbitrary number of CFSMs to execute in every steps. In that sense, it is a generalization of both synchronous models (where all components execute in each step) and asynchronous models (where a single component is active in each step). Also note that our model can be classified as zero-delay communication: it takes one step to generate a reaction to some input events, but once the response is generated, the output events are immediately observable by all consumers.

[2]Conjunction is represented by "$*$," disjunction by "$+$," and negation by an overline.

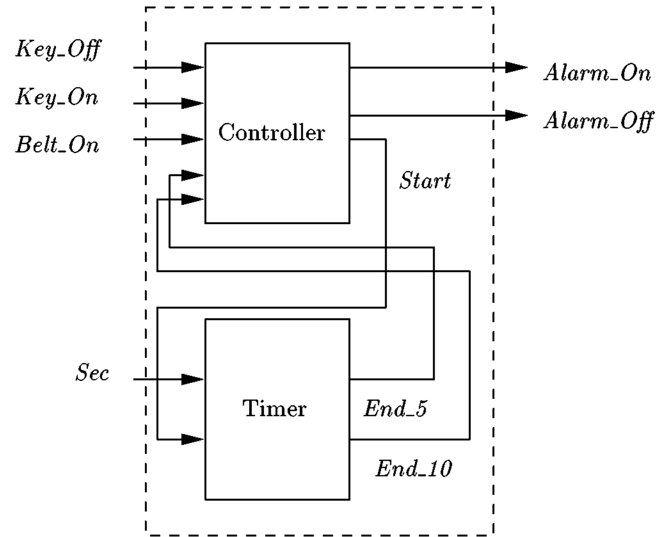

Fig. 1. Seat belt alarm controller example.

Definition II.1 (Unit Delay Parallel): An implementation follows unit delay parallel (UDP) scheduling policy if for any state s, $\mathrm{Select}(s) = \mathrm{C}$, i.e., all CFSMs are always selected.

Definition II.2 (Static Priority Serial): An implementation follows static priority serial (SPS) scheduling policy if *Select* chooses one of the enabled CFSMs according to some statically determined priority order.

Definition III.3 (Cyclic Executive Serial): An implementation follows cyclic executive serial (CES) scheduling policy if *Select* selects the first enabled CFSM according to some statically determined list. The list is searched cyclically, starting from the most recently selected CFSM.

In the case of CES, we assume that the state of network is extended to include information on which CFSM was most recently selected. That part of the state is updated each time *Select* is executed and we assume that *Select* is executed exactly once for each step i in (1).

A. Example

Suppose that we want to specify a simple safety function of an automobile: a seat belt alarm control system. A natural language specification written by a designer could be: "Five seconds after the key is turned on, if the belt has not been fastened, an alarm will beep for five seconds or until the key is turned off." The specification can be represented by two reactive components as shown in Fig. 1, consisting of two CFSMs: a controller and a timer.

State transition graphs of CFSMs representing the controller and the timer are shown in Figs. 2 and 3, respectively. Inputs and outputs for a state transition are separated by "/." To fit them in our current model, we need to abstract the state, because our model allows only memory-free transition functions. To do so, we replace the state machines in Figs. 2 and 3 with CFSMs with (memory-free) transition functions

$$\mathrm{Alarm_On} = \overline{\mathrm{Key_Off}} * \overline{\mathrm{Belt_On}} * \mathrm{End_5} \qquad (2)$$
$$\mathrm{Alarm_Off} = \mathrm{Key_Off} + \mathrm{Belt_On} + \mathrm{End_10} \qquad (3)$$
$$\mathrm{Start} = \mathrm{Key_On} * \overline{\mathrm{Key_Off}} * \overline{\mathrm{Belt_On}} \qquad (4)$$

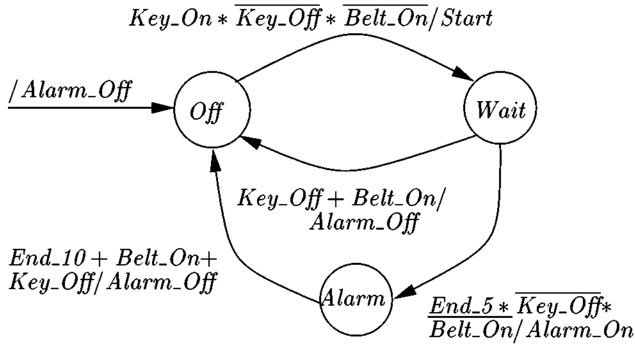

Fig. 2. Controller CFSM in seat belt example.

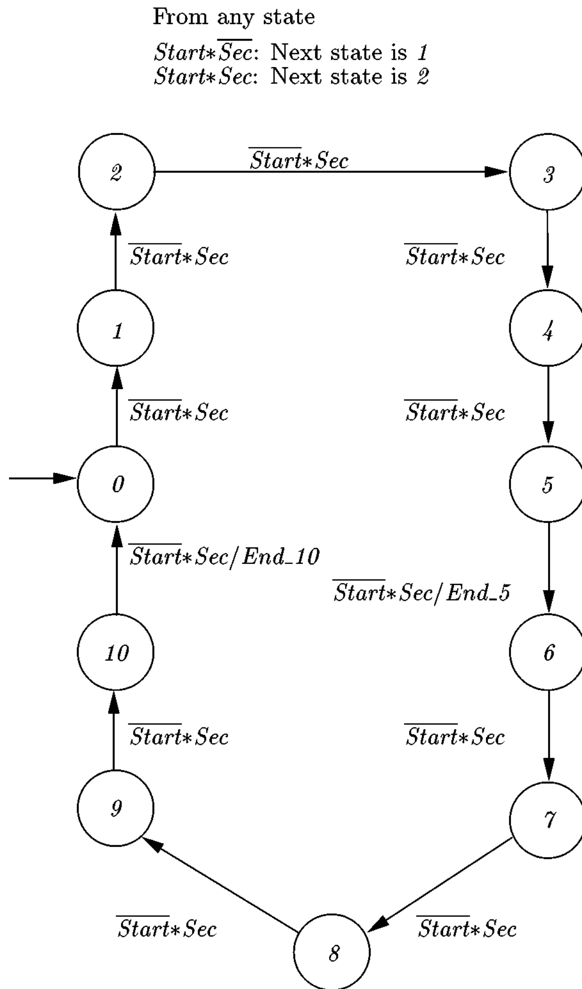

Fig. 3. Timer CFSM in seat belt example.

$$\text{End_5} = \overline{\text{Start}} * \text{Sec} \tag{5}$$
$$\text{End_10} = \overline{\text{Start}} * \text{Sec}. \tag{6}$$

These transition functions were obtained from the state machines by existentially quantifying state variables. For example, $\text{End_5} = \overline{\text{Start}} * \text{Sec}$ because there exists a state (namely, 5) in which End_5 is emitted as a response to $\overline{\text{Start}} * \text{Sec}$. It can be shown that using these functions, it is still possible to decide synchronous equivalence of two implementations of the original network. However, if state variables are kept and dealt with explicitly, it is possible to define less conservative analysis, as we will show in Section V-B.

A run of the CFSM network in Fig. 1 with transition functions (2)–(6), is shown in Fig. 4. That run is based on SPS policy with Controller having higher priority than Timer. The first row contains index i. The next two rows correspond to Emit_i and Clear_i in (1). To preserve space, we have encoded Emit_is as bit vectors. The bits (from left to right) indicate whether Key_On, Key_Off, Belt_On, and Sec are in Emit_i. For example, Emit_1 in Fig. 4 is $\{\text{Key_On}, \text{Sec}\}$. The following eight rows show s_i [i.e., the state following $(\text{Emit}_i, \text{Clear}_i)$ in (1)]. The last row, labeled *Select* is not a part of the run, but helps reading it. It indicates which CFSM is running in each step (C is for Controller and T is for Timer).

III. Synchronous Assumption and Synchronous Equivalence

A key assumption is made on the class of "acceptable" specifications in order to make it easier to specify desirable behaviors and, at the same time, make efficient analysis algorithms possible. We believe that this class includes many interesting examples and that other specifications may be respecified to satisfy this assumption.

Many popular design methodologies separate time into intervals of interaction with the environment and computation within the design. Software programs accept some input to start the computations. Under most circumstances, they finish computations before accepting new sets of input. Synchronous sequential circuits are hazard-free and race-free because interaction (propagation of state changes and signal changes) and computation (calculation of state changes and signal changes) are strictly separated. More recently, synchronous languages [6] used the zero-delay assumption, i.e., assumed that computation is always faster than the environment changes. It is, therefore, impossible for interaction and computation to overlap. We can, thus, imagine to strictly separate computation and interaction also for embedded system designs based on CFSMs. The advantage in ease of specification and analysis outweighs some suboptimality due to the restriction in design style.

Definition III.1 (Synchronous Assumption): A run

$$s_0, (\text{Emit}_1, \text{Clear}_1), s_1, (\text{Emit}_2, \text{Clear}_2), \ldots, (\text{Emit}_n, \text{Clear}_n), s_n$$

of some implementation of CFSM network (PI, PO, C) conforms to synchronous assumption if for all $i = 1, \ldots, n$:

1) either $\text{Emit}_i = \text{Clear}_i = \emptyset$ or
2) $\text{Clear}_i = \text{PO}$ and $s_{i-1}(e) = 0$ for all $e \in E - \text{PO}$.

If $\text{Clear}_i = \text{PO}$, we say that i is a *cycle boundary*. If $i_1 < i_2 < \cdots < i_j$ are all and only cycle boundaries of the run, then

$$\text{Emit}_{i_1}, \text{Emit}_{i_2}, \ldots, \text{Emit}_{i_j}$$

is the *input trace* of the run and

$$\{po \in \text{PO} \mid s_{i_1-1}(E(po)) = 1\}, \ldots, \{po \in \text{PO} \mid s_{i_j-1}(E(po)) = 1\}$$

is the *output trace* of the run.

i	0		1		2		3		4		5		6		7
Emit		1001		0000		0000		0011		0000		0000		0000	
Clear		*PO*		$\emptyset$		$\emptyset$		*PO*		$\emptyset$		$\emptyset$		$\emptyset$	
Key_On	0		1		0		0		0		0		0		0
Key_Off	0		0		0		0		0		0		0		0
Belt_On	0		0		0		0		1		0		0		0
Sec	0		1		1		0		1		1		0		0
Start	0		0		1		0		0		0		0		0
End_5	0		0		0		0		0		0		1		0
End_10	0		0		0		0		0		0		1		0
Alarm_On	0		0		0		0		0		0		0		1
Alarm_Off	0		0		0		0		0		1		1		1
Select		-		C		T		-		C		T		C	

Fig. 4. Run of the CFSM network in Fig. 1 with transition functions (2)–(6).

If a run conforms to the synchronous assumption, it can be split into two alternating nonoverlapping phases. An *interaction* phase (one step when $\mathrm{Clear}_i = \mathrm{PO}$), where the environment interacts with the design and a *computation* phase (possibly many steps when $\mathrm{Emit}_i = \mathrm{Clear}_i = \emptyset$), where the components in the design perform executions and communicate among themselves.

This notion of synchronicity is a system-level extension to the fundamental mode operation of asynchronous circuits [1]. During an interaction phase, the design "stands still" until the environment completes its receiving of outputs from and writing of inputs to the design for use during the next computation phase. During a computation phase, the design takes inputs that were generated by the environment, performs computations, and generates outputs that will be read by the environment during the next interaction phase. There are no interactions during a computation phase and no computations during an interaction phase. The interaction phase followed by its associated computation phase is called a *cycle*. We will only consider runs that conform to this synchronous assumption. The implementation process must guarantee that only such runs occur in a given environment. This can be done by a separate worst case timing analysis in the flavor of [7]. If the worst case timing delay is within the constraints imposed by the environment, the implementation can be said to conform to the synchronous assumption.

Definition III.2 (Synchronous Equivalence): Two implementations are synchronously equivalent if and only if any two synchronous assumption conforming runs of the two implementations that have the same input traces also have the same output traces.

Note how internal events play absolutely no role in the definition of synchronous equivalence. As we will see, they may or may not play a role in deciding it. As long as the primary outputs are the same at the end of every cycle, the delay of the executions of CFSMs, the order of the executions, or even the parallel/serial nature of the executions do not matter. These lead to freedom in synthesis, optimization, scheduler selection, and assignment of components to computational resources.

Our restricted CFSM model can, thus, be defined as externally synchronous, globally asynchronous, and locally synchronous. The concepts of synchronous assumption and synchronous equivalence facilitate specification. Designers can now think about the input/output reaction of the design to the environment separately from the speed of the reaction. Synchronous assumption is not too restrictive, especially for control-dominated applications. Designing with the synchronous assumption is strongly analogous to designing synchronous circuits and fundamental mode asynchronous circuits. In that domain, the ease of validation and synthesis often outweighs the decreased freedom with respect to full asynchrony. The same can be extended to embedded system design, board level, or "system on a chip," where different processors or dedicated units can be considered as different computational resources. There are other research efforts in embedded system domain that also take a synchronous approach, as will be discussed below.

A. Related Work

Synchronous languages are a group of languages proposed for automatic synthesis of embedded software [6], [8]. They describe complex systems consisting of interconnected components each represented by an FSM model. The communication among the components and the executions of the components all take "zero time" to perform. In practice, this means that the interaction with the environment has to be "much slower" than all the communication and execution time required for the reaction to the environment. In addition, the zero communication and execution times of the components imply an execution order that must respect causal relationships among the component executions. This order is assured by "synchronous schedulers," the class of schedulers that defines the correct behavior. Our synchronous assumption, on the other hand, is related only to the "external" interaction of the design with the environment and we put no *a priori* constraints on the scheduler.

Synchronous data flow is a powerful formalism for data-dominated embedded systems geared toward simulation and code synthesis for digital signal processors [9]. It also exploits the synchronous assumption at the interface between the network and the environment, but "blocking read" is required of all components in the design to ensure that the behavior is the same (in Kahn's sense [10]) independent of allocation and scheduling.

Time-triggered architecture [11] is specifically proposed for safety critical applications. It achieves determinacy by having a common notion of time for all components. The communication among the components can be carried out only at predefined

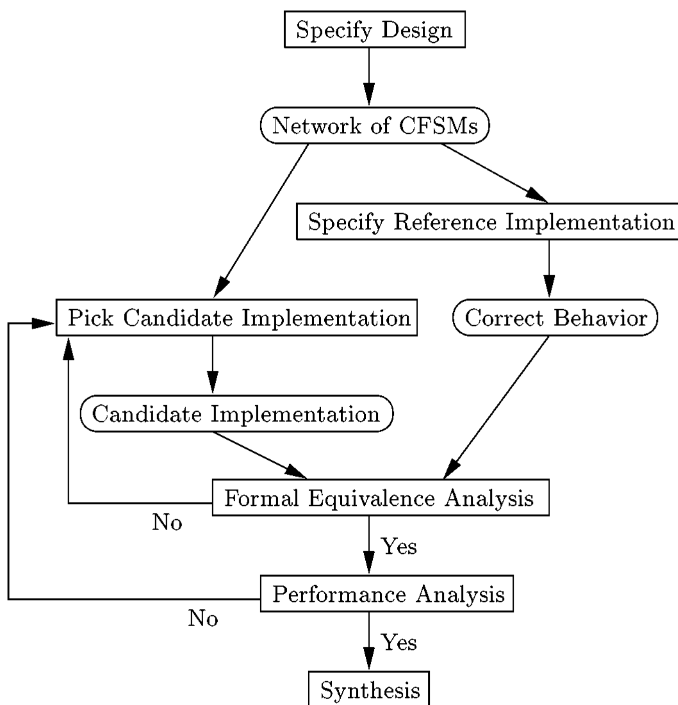

Fig. 5. Synchronous approach to design exploration.

time slots in time division multiple access (TDMA) fashion. The cost of this determinacy is the severe restriction in the communication architecture that can be used in the design.

Our work does not restrict the implementation choices to those utilizing a synchronous scheduler, nor does it require communications to have the "blocking read" property, nor does it restrict the designers to a single communication architecture. Therefore, many different functional behaviors are possible depending on the choice of implementation parameters. We use equivalence analysis to tell us whether *any* two implementations are equivalent to each other.

Javatime [12] is inspired similarly by the fundamental mode operation of asynchronous circuits and the concept of refinement. Their emphasis is on the specification and simulation using the language Java. Our concentration, on the other hand, is on the development of efficient formal algorithms for equivalence analysis.

B. Design Exploration Methodology

Fig. 5 illustrates a possible design methodology using the synchronous assumption and synchronous equivalence. The designer specifies the functionalities of the components and the structure of the design as a network of CFSMs. One or more behaviors among those allowed by the nondeterminism in the network are chosen as reference model(s). They are represented by reference implementations that produce those functional behaviors and only those functional behaviors. The functional behaviors under synchronous assumption of these implementations are considered "correct" and different implementations with the same functional behavior are all functionally equivalent. Separate analysis, in the flavor of [7] is performed to make sure that nonfunctional constraints are satisfied and to decide whether one implementation is superior to others given some design metrics. The best one is chosen to be synthesized.

For example, the reference implementation may be a simulation following UDP policy and intended implementation may be an SPS policy. If some high priority CFSM in the initial SPS implementation is enabled by many other CFSMs, it will be executed many times with very few inputs present each time. Lowering the priority of such CFSM can reduce the total number of context switches and improve performance. Synchronous equivalence may be used to show that functional behavior is preserved through all these transformations (from the UDP to an SPS to a different SPS policy).

C. Analyzing Synchronous Equivalence

In the next section, we will propose to compare two implementations of the same CFSM network by simulating them abstractly and then comparing the resulting traces using a trivial graph-isomorphism comparison. Because the simulation is abstract, the results may be conservative in the sense that two equivalent implementations may be declared not equivalent.

In Section V, we will show how to make our analysis less conservative by case-splitting on abstract values of binary signals and creating two less abstract traces (one for each concrete signal value) instead of a single more abstract one. We will show that if all signals are refined in this way, then the resulting set of traces is equal to exhaustive simulation traces.

Besides the approaches described in Sections IV and V, synchronous equivalence can be checked in many other ways. Exhaustive simulation is one way to check it exactly. A conservative approach based on property called *delay insensitivity* is described in [5]. An exact check could also be made using formal verification techniques. For example, each CFSM could be modeled as a process in Milner's synchronous CCS [13]. The scheduling policy defines which of the processes perform empty actions and which perform observable actions. Synchronous equivalence is similar to trace equivalence, except that the latter is concerned with all observable actions, while the former deals only with the action that are observable at the end of the cycle. It is possible to define an algorithm that hides all actions of no interest to synchronous equivalence and synchronizes all relevant actions in the same cycle. After such a transformation, the synchronous equivalence of the original networks can then be checked by checking trace equivalence on the transformed ones.

The analysis techniques span a continuum on computation time and conservativeness of the analysis, as shown in Fig. 6. Communication analysis denotes the method proposed in Section IV, while refined communication analysis denotes the method proposed in Section V. Conservativeness is measured by the number of false negative results produced by the analysis algorithm, given a set of implementations of a CFSM network. Exact analysis of any form produces no false negative results, but it often has very long computation time. Conservative analysis methods require less computation time at a cost of false negative results. On the other extreme from exhaustive simulation, one could therefore reach a conclusion of "don't know" without performing any analysis at all. In the

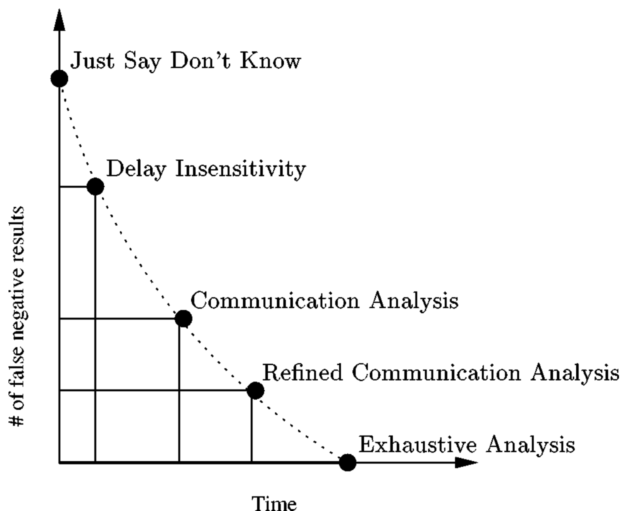

Fig. 6. Tradeoff in analysis methods.

design methodology proposed in the previous section, false negative results show up as candidate implementations that are falsely declared to be functionally incorrect. This may lead to suboptimal final implementation.

IV. Communication Analysis

In this section, we introduce communication analysis, which can be used to check the equivalence between two different implementations. The analysis is conservative in the sense that it is correct when it says that two implementations are equivalent, but it is inconclusive otherwise.

There is a simple intuition for looking at the communication between components. Since the corresponding components in the two implementations have the same functionality and the connectivity among the components is also the same, it should be possible to deduce the equivalence of two implementations from the behavior of the communication. It should be in fact possible to establish a *communication signature* in the flavor of worst case analysis in real-time scheduling [14]. If two implementations have the same communication signature, they should be synchronously equivalent to each other. By looking at only the worst case communication characteristics and not at the details, the analysis can be efficient, though at a cost of being conservative.

A. Motivating Example

Consider the seat belt example in Fig. 7 and the abstract transition functions (2)–(6). We propose to check for synchronous equivalence of two implementations of the seat belt network by comparing their communication signatures. The particular communication signature we define is called execution cover (EC). ECs represent all executions of an implementation and if two implementations have the same EC, they are synchronously equivalent. Fig. 7 shows the ECs for the seat belt implementations. EC nodes can be thought of as "containers" representing *possible* events in the flavor of event graphs representing partially ordered histories [3], [15], but using the additional notion of possibility to further abstract it. Thus, a container can contain

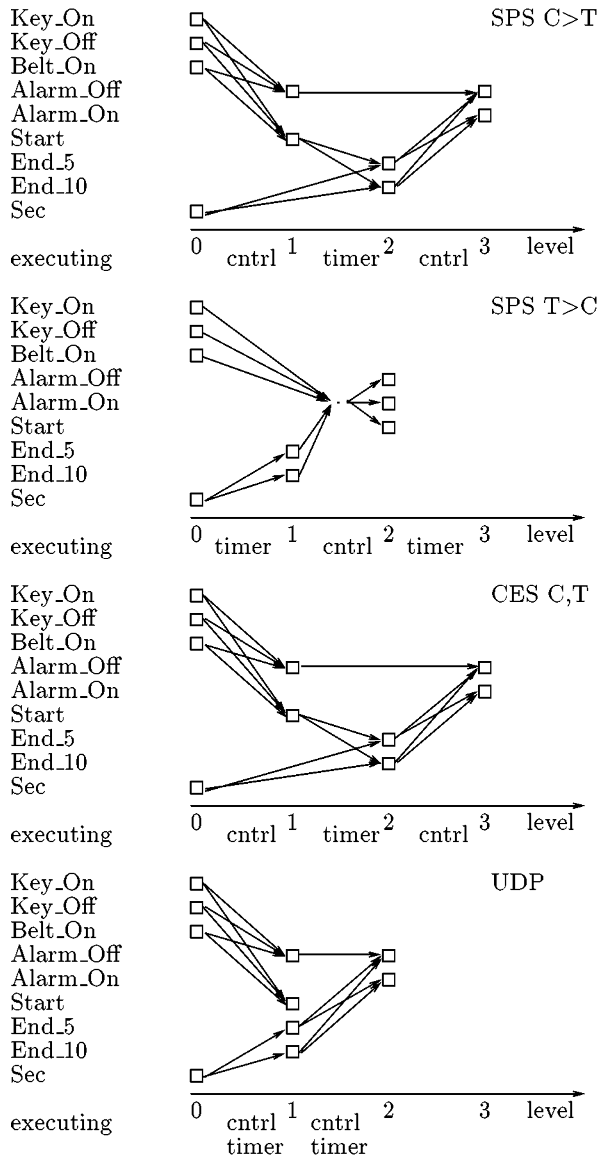

Fig. 7. LEC for seat belt example.

either a "0" (event absence) or a "1" (event presence). Edges in the EC represent dependencies between input and output events.

To construct the topmost EC in Fig. 7 (SPS policy with controller having the higher priority), we start with a container at each primary input, indicating that there could be primary input events present at the beginning of the cycle (nodes at level 0 in Fig. 7). Since both controller and timer could be enabled we choose to "execute" the controller. We use "execute" in quotation marks because we do not know precisely what is at the inputs, so we generate a container at an output if there exists some assignment of zeros and ones to containers that would cause the controller to emit that output. More precisely, we generate a container for Alarm_Off and Start, but not for Alarm_On because Alarm_On is emitted only if End_5 is present and there is no container at that input. Next, we draw an edge between newly generated containers and containers used in their generation. Since the controller can no longer be enabled, we "execute" the timer next. This will result in two new nodes (at level 2)

and four new edges pointing to those two nodes. Now, the controller could be enabled again with $\mathrm{End_5}$ or $\mathrm{End_10}$ possibly at its inputs. Therefore, we create containers for $\mathrm{Alarm_Off}$ and $\mathrm{Alarm_On}$ and draw appropriate dependency edges. We also draw an edge between two nodes for $\mathrm{Alarm_Off}$, indicating the precedence between them.

We invite the reader to check that the implementation using CES policy with the controller being first followed by the timer has the same EC as above. Our main result (stated in Theorem 2) implies that because their ECs are the same, the two implementations are synchronously equivalent. Intuitively, actual execution of either SPS or CES implementations fills each container with zeros or ones for a given set of primary inputs. The corresponding nodes at the first level of two ECs will be filled with the same values because the same transition functions execute with the same inputs (but not necessarily in the same order) in both implementations. We can repeat this argument inductively to argue that all the corresponding containers in two ECs are filled the same way. Therefore, the value of each event at the end of the cycle will also be the same.

In the following sections, we will define EC precisely by proposing a general procedure for EC generation and formally state our main result.

B. Abstracting Communication

We want to abstract or summarize communication between the components for a particular implementation. To find a signature that is "worst case," over-approximating, conservative, but correct, we utilize the concept of a container space function as a *monotonic cover* for an event space function. The word "container" is chosen to denote an entity that is necessary, but not sufficient, to contain an actual event.

1) Containers: Intuitively, a container is an entity that may or may not contain an event. The presence of a container implies the possibility of an event. The absence of a container means that definitely there is not an event. We use containers to represent an arbitrary set of primary inputs, because our goal is an equivalence analysis that is independent of actual inputs.

Definition IV.1 (Container and Event Spaces, Minterms): An n-dimensional container space is defined as $\mathrm{C}^n = \{0, \mathtt{x}\}^n$. An n-dimensional event space is defined as $E^n = \{0, 1\}^n$. Elements of E^n and C^n are called minterms.

Value 0 indicates event absence and value 1 indicates event presence. Value $\mathtt{x}$ indicates presence of a container, i.e., possible presence of the event. Thus, there is a natural *information ordering* between these values, namely, $0 \preceq \mathtt{x}$ and $1 \preceq \mathtt{x}$ ($\mathtt{x}$ is at least as abstract as 0 and 1) and of course $0 \preceq 0$, $1 \preceq 1$, $\mathtt{x} \preceq \mathtt{x}$. We extend this ordering to minterms component-wise, i.e., $(x_1, \ldots, x_n) \preceq (y_1, \ldots, y_n)$ if and only if $x_i \preceq y_i$ for all $i = 1, \ldots, n$. In words of lattice theory [16], $\{0, 1, \mathtt{x}\}$ equipped with $\preceq$ is a partially ordered set and $\mathtt{x}$ is its top. Similarly, $\mathrm{C}^n \cup E^n$ is a partially ordered set and $(\mathtt{x}, \ldots, \mathtt{x})$ is its top.

In the examples that follow, we use the notation from multivalued logic synthesis [17] to represent functions in the container space. We represent a container space function $\mathbf{f}\colon \{0, \mathtt{x}\}^n \rightarrow \{0, \mathtt{x}\}$ with two characteristic functions $\mathbf{f}^{\{\mathtt{x}\}}, \mathbf{f}^{\{0\}}\colon \{0, \mathtt{x}\}^n \rightarrow \{1, 0\}$ such that $\mathbf{f}^{\{\mathtt{x}\}}$ ($\mathbf{f}^{\{0\}}$) evaluates to one for all minterms for which $\mathbf{f}$ evaluates to $\mathtt{x}(0)$. The same notation is used for literals, i.e., $\mathbf{a}^{\{\mathtt{x}\}}$ ($\mathbf{a}^{\{0\}}$) is one if and only if $\mathbf{a}$ is $\mathtt{x}(0)$.

Given some event space function f, we want to represent it abstractly with some container space function $\mathbf{f}$. The key property of $\mathbf{f}$ is that for each container space minterm $\mathbf{m}$, it should evaluate to $\mathtt{x}$ ("may-be-one") if f evaluates to one for some minterm abstractly represented by $\mathbf{m}$. For example, $\mathbf{f}(0\mathtt{xx}0)$ should be $\mathtt{x}$ if at least one of $f(0110)$, $f(0100)$, $f(0010)$, or $f(0000)$ is one. We call such $\mathbf{f}$ a *monotonic cover* of f.

Definition IV.2 (Monotonic Cover): A container space function $\mathbf{f}\colon \{0, \mathtt{x}\}^n \rightarrow \{0, \mathtt{x}\}$ is a monotonic cover of the event space function $f\colon \{0, 1\}^n \rightarrow \{0, 1\}$ if for any container space minterm $\mathbf{m}$

$$(\exists\, m \preceq \mathbf{m}\colon f(m) = 1) \implies (\mathbf{f}(\mathbf{m}) = \mathtt{x}).$$

For example, function $\mathbf{f1}^{\{\mathtt{x}\}} = \mathbf{b}^{\{\mathtt{x}\}}$ is a monotonic cover of $f = \overline{a} * b * \overline{c}$. So are $\mathbf{f2}^{\{\mathtt{x}\}} = \mathbf{b}^{\{\mathtt{x}\}} + \mathbf{c}^{\{\mathtt{x}\}}$ and $\mathbf{f3}^{\{\mathtt{x}\}} = \mathbf{b}^{\{\mathtt{x}\}} + \mathbf{a}^{\{\mathtt{x}\}}$. $\mathbf{f4}^{\{\mathtt{x}\}} = \mathbf{a}^{\{\mathtt{x}\}} + \mathbf{c}^{\{\mathtt{x}\}}$ is not a monotonic cover of f because the minterm $a = 0$, $b = 1$, $c = 0$ evaluates f to one, but its abstraction $\mathbf{a} = 0$, $\mathbf{b} = \mathtt{x}$, $\mathbf{c} = 0$ evaluates $\mathbf{f4}$ to zero.

Now that we have precisely defined the functions used in abstract executions, we are almost ready to define precisely EC generation procedure, but first we need to define a class of scheduling policies to which our approach applies.

2) Well-Behaved Scheduling Policy: Checking synchronous equivalence for arbitrary implementations is difficult. We concentrate on algorithms that can be applied to implementations that utilize a "well-behaved" subset of scheduling policies.

Definition IV.3 (Well-Behaved Scheduling Policy): A scheduling policy *Select* is well-behaved if for every CFSM network state s and any state s' such that $\mathrm{Enabled}(s) \subseteq \mathrm{Enabled}(s')$, either

$$\mathrm{Select}(s') = \mathrm{Select}(s) \tag{7}$$

or

$$\mathrm{Select}(s') \cap \mathrm{Enabled}(s) = \emptyset. \tag{8}$$

In other words, if a well-behaved scheduling policy is presented with a superset of enabled CFSMs $\mathrm{Enabled}(s')$, rather the exact set $\mathrm{Enabled}(s)$, then it may choose to perform empty executions on some CFSMs not enabled in s, but it cannot change the relative execution order of CFSMs actually enabled in s. Whether or not a given scheduling policy is well-behaved needs to be established separately through formal proofs. The proofs will only need to be done once since the property is not design dependent. Many common scheduling policies can be proven to be well behaved.

Theorem 1: The UDP, SPS, and CES scheduling policies are all well behaved.

Proof: UDP satisfies (7) by definition. SPS and CES always select at most one CFSM. If the selected one is not in $\mathrm{Enabled}(s)$, then (8) is obviously satisfied. If SPS selects from $\mathrm{Enabled}(s')$ a CFSM that is also in $\mathrm{Enabled}(s)$, then no CFSM in $\mathrm{Enabled}(s')$ has higher priority than the selected one. Since $\mathrm{Enabled}(s) \subseteq \mathrm{Enabled}(s')$, the selected CFSM has the highest

priority in $\mathrm{Enabled}(s)$ as well and (7) holds. The argument for CES is similar. ■

An example of a policy that is not well behaved is the following (quite unnatural) Select rule: "If CFSM C is enabled, the (dynamic) priority is $\mathrm{C} > B > A$. Otherwise, the priority is $A > B$." At some point in time when B and A are enabled, additionally enabling C (which will cause an empty execution on C) can actually change the execution order of B and A.

3) ECs: Given an implementation *Select* of some CFSM network (PI, PO, C) and given some monotonic cover of each transition function of each CFSM in C, the EC is the directed acyclic graph generated by the following EC generation procedure:

Step 1) Create a container (an $\mathtt{x}$) for each primary input and label it with level 0. Set the current level to zero.

Step 2) Determine the set of *active* events. An event is active if there exists a corresponding container with a level that is larger than the level at which its consumer was previously executed.

Step 3) Let s_{active} be a network state such that $s_{\mathrm{active}}(e) = 1$ if and only if e is active.

Step 4) If $\mathrm{Enabled}(s_{\mathrm{active}})$ is empty, then remove all level labels and STOP.

Step 5) Abstractly execute CFSMs in $\mathrm{Select}(s_{\mathrm{active}})$ by increasing the current level by one, and evaluating the given monotonic cover of each CFSM transition function with all active inputs set to $\mathtt{x}$ and other inputs set to 0. If the monotonic cover evaluates to $\mathtt{x}$, then:

1) create a new container and label it with the current level and the name of that output;
2) for every active input, create an edge from the most recent container corresponding with that input to the newly created container;
3) create an edge from the previous container labeled with the same name (if any exists) to the new container.

Step 6) Go to Step 2.

We can (conservatively) check two implementations for synchronous equivalence by comparing their ECs as stated by the following result. The theorem applies to any two ECs no matter what monotonic covers are used for Step 5 above.

Theorem 2: If two implementations of a given CFSM network with well-behaved scheduling policies have isomorphic ECs, then they are synchronously equivalent.

To prove the theorem, we first argue that any actual execution of the two implementations follows the execution ordering implied by the two ECs. Then we argue by induction on a topological order of the ECs that corresponding executions in two ECs must have the same input events present. This holds because actual transition functions in corresponding executions are the same (since the two networks differ only in scheduling). The details of the proof are given in the Appendix.

Checking whether two ECs are isomorphic is quite simple. Each node is labeled with some event and nodes labeled with the same event form a chain. Therefore, isomorphism can be checked by a simple linear sweep over two graphs.

The EC generation procedure may not terminate even if the CFSM network stabilizes for every input pattern. This nontermination is due to looping through the same patterns and, thus, can be easily identified in the algorithm implementation. In this case, the algorithm can be aborted and the communication analysis returns an inconclusive result. However, if the procedure terminates successfully, then identical ECs imply synchronous equivalence. The abstract nature of EC gives it efficiency, but also makes it reach many inconclusive results due to the false negatives or nontermination. We will show the usefulness of EC on industrial designs in Section IV-C.

The abstract execution in Step 5 is performed by evaluating a *monotonic cover* of the CFSM transition functions. There are many monotonic covers of a function so there is a whole set of different ECs that can be used as communication signatures. For example, it was suggested in [5] that $\bigvee_{in \in I_A} in^{\{\mathtt{x}\}}$ be used as a monotonic cover for all transition functions. This is a very simple monotonic cover to compute, but unfortunately it is also very conservative in the sense that it evaluates to $\mathtt{x}$ for all the minterms except $(0, 0, \ldots, 0)$ [recall that CFSMs are reactive so no transition function can evaluate to one at $(0, 0, \ldots, 0)$]. In the next section, we will show how to compute the least monotonic cover, i.e. the one that evaluates to $\mathtt{x}$ for the fewest number of minterms. We call EC obtained by using it the least EC (LEC). LEC is the best EC, i.e., it produces the least number of inconclusive results. While least monotonic covers clearly lead to the strongest results, there may be designs where they are too expensive to compute and a cheaper, but more conservative covers still provide satisfactory result.

4) Least Monotonic Cover: To compute least monotonic covers of transition functions, we first manipulate in the event space each function T_k^A to obtain an auxiliary function $\mathcal{T}_k^A$ also in the event space and then translate $\mathcal{T}_k^A$ into the container space.

We say that a container space minterm $(\mathbf{c_1}, \ldots, \mathbf{c_n}) \in \{0, \mathtt{x}\}^n$ is the translation of an event space minterm $(e_1, \ldots, e_n) \in \{0, 1\}^n$ if for all $i = 1, \ldots, n$

$$e_i = 1 \iff \mathbf{c_i} = \mathtt{x}.$$

Similarly, we say that a container space function $\mathbf{f}$ is the translation of an event space function f if for all $m \in \{0, 1\}^n$ and all $\mathbf{m} \in \{0, \mathtt{x}\}^n$ such that $\mathbf{m}$ is the translation of m

$$f(m) = 1 \iff \mathbf{f}(\mathbf{m}) = \mathtt{x}.$$

Translation is just renaming 1 with $\mathtt{x}$. In practice, it is a free operation because both f and $\mathbf{f}$ can be represented with the same data structure.

Given some transition function T (we are dropping subscripts for brevity), we want an auxiliary function $\mathcal{T}$ that has the following property: every minterm in the on set of T must also be in the on set of $\mathcal{T}$. In addition, if some minterm whose ith component is zero is in the on set of $\mathcal{T}$, then the same minterm, but with ith component 1 must also be in the on set of $\mathcal{T}$. This can be expressed by the following formulae for functions with 1, 2, or 3 input variables:

$$\mathcal{T}(v_1) = T + T_{\overline{v_1}} \tag{9}$$

$$\mathcal{T}(v_1, v_2) = T + T_{\overline{v_1}} + T_{\overline{v_2}} + T_{\overline{v_1 v_2}} \tag{10}$$

$$\mathcal{T}(v_1, v_2, v_3) = T + T_{\overline{v_1}} + T_{\overline{v_2}} + T_{\overline{v_3}} + T_{\overline{v_1 v_2}} + T_{\overline{v_1 v_3}} + T_{\overline{v_2 v_3}} + T_{\overline{v_1 v_2 v_3}}. \quad (11)$$

The computation of $\mathcal{T}$ can be simplified drastically with the following recursive formulation for a function with n input variables thanks to the commutativity of cofactors

$$\mathcal{T}^0 = T \quad (12)$$

$$\mathcal{T}^i = \mathcal{T}^{i-1} + \mathcal{T}^{i-1}_{\overline{v_i}}, \qquad \text{for } i = 1, \ldots, n. \quad (13)$$

Theorem 3: The least monotonic cover of some event space function T is the translation of $\mathcal{T}^n$ [defined by (12) and (13)] into the container space.

Proof: Let relations $\leq_0, \leq_1, \ldots, \leq_n$ on n-dimensional event space, be defined by[3]

$$m \leq_i m', \qquad \text{if and only if } \begin{cases} m_j \leq m'_j, & \forall j \leq i \\ m_j = m'_j, & \forall j > i \end{cases}$$

$$\text{for all } i = 0, \ldots, n \quad (14)$$

where $m = (m_1, \ldots, m_n)$ and $m' = (m'_1, \ldots, m'_n)$. Note that $m \leq_{i-1} m'$ implies $m \leq_i m'$, but not vice versa. We show (by induction on i) that for all $i = 0, \ldots, n$

$$\forall m' : (\exists\, m \leq_i m' : T(m) = 1) \iff \mathcal{T}^i(m') = 1. \quad (15)$$

The desired result follows easily from the case $i = n$.

Base case ($i = 0$): Trivial because $\leq_0$ is equality and $\mathcal{T}^0$ is T.

Inductive step ($\Longrightarrow$ direction): Assume $m \leq_i m'$ is such that $T(m) = 1$. Consider the case $m_i = m'_i$. In this case, $m \leq_{i-1} m'$, so $\mathcal{T}^{i-1}(m') = 1$ by inductive assumption, implying (by definition) that $\mathcal{T}^i(m') = 1$.

In the case $m_i \neq m'_i$, $m_i = 0$ must hold. Let m'' be such that $m''_j = m'_j$ for all $j \neq i$ and $m''_i = 0$. We have

$$\begin{aligned} \mathcal{T}^i(m') &\geq \text{(by definition)} \\ \mathcal{T}^{i-1}_{\overline{v_i}}(m') &= \text{(because } \mathcal{T}^{i-1}_{\overline{v_i}} \text{ does not depend on } \overline{v_i}) \\ \mathcal{T}^{i-1}_{\overline{v_i}}(m'') &= \text{(because } m''_i = 0) \\ \mathcal{T}^{i-1}(m'') &= \text{(by inductive assumption, using the fact that } m \leq_{i-1} m''). \end{aligned}$$

(**$\Longleftarrow$ direction**): Assume $\mathcal{T}^i(m') = 1$. By definition, at least one of $\mathcal{T}^{i-1}(m')$ and $\mathcal{T}^{i-1}_{\overline{v_i}}(m')$ must be one. If $\mathcal{T}^{i-1}(m') = 1$, then (by inductive assumption) there exists $m \leq_{i-1} m'$ (hence, also $m \leq_i m'$) such that $T(m) = 1$.

If $\mathcal{T}^{i-1}_{\overline{v_i}}(m') = 1$, let m'' be such that $m''_j = m'_j$ for all $j \neq i$ and $m''_i = 0$ (which may or may not be different from m'). We have

$$\begin{aligned} 1 = \mathcal{T}^{i-1}_{\overline{v_i}}(m') &= \text{(because } \mathcal{T}^{i-1}_{\overline{v_i}} \text{ doesn't depend on } \overline{v_i}) \\ \mathcal{T}^{i-1}_{\overline{v_i}}(m'') &= \text{because } m''_i = 0) \\ \mathcal{T}^{i-1}(m''). \end{aligned}$$

By inductive assumption, there exists $m \leq_{i-1} m''$ such that $T(m) = 1$. However, such m must satisfy $m \leq_i m''$ and also $m \leq_i m'$ (because $m''_i = 0$). ■

Function $\mathbf{f1}^{\{x\}} = \mathbf{b}^{\{x\}}$ is a least monotonic cover of $f = \overline{a} * b * \overline{c}$, while $\mathbf{f2}^{\{x\}} = \mathbf{b}^{\{x\}} + \mathbf{c}^{\{x\}}$, $\mathbf{f3}^{\{x\}} = \mathbf{b}^{\{x\}} + \mathbf{a}^{\{x\}}$, and $\mathbf{f5}^{\{x\}} = \mathbf{b}^{\{x\}} + \mathbf{c}^{\{x\}} * \mathbf{a}^{\{0\}}$ are monotonic covers of f, but are not the least monotonic cover.

[3]In $\{0, 1\}$, we use $\leq$ with the usual, arithmetic semantics, i.e., $0 \leq 0$, $0 \leq 1$, $1 \leq 1$.

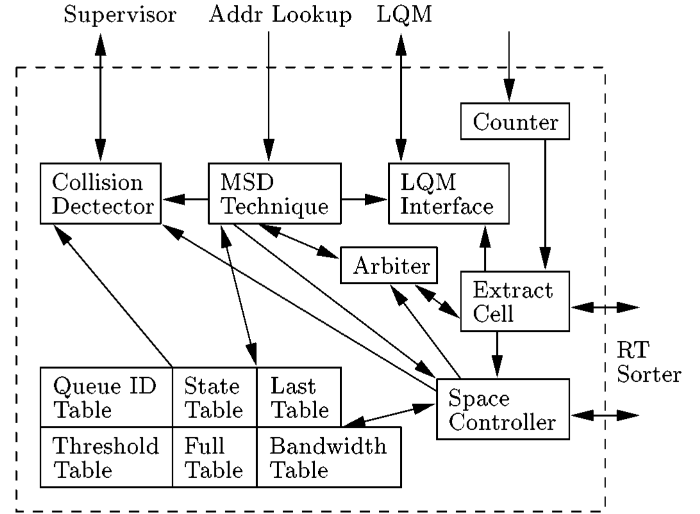

Fig. 8. Algorithm block of a ATM server.

The least monotonic covers for transition functions (2)–(6) are

$$\begin{aligned} \mathbf{Alarm_On}^{\{x\}} &= \mathbf{End_5}^{\{x\}} \\ \mathbf{Alarm_Off}^{\{x\}} &= \mathbf{Key_Off}^{\{x\}} + \mathbf{Belt_On}^{\{x\}} + \mathbf{End_10}^{\{x\}} \\ \mathbf{Start}^{\{x\}} &= \mathbf{Key_on}^{\{x\}} \\ \\ \mathbf{End_5}^{\{x\}} &= \mathbf{Sec}^{\{x\}} \\ \mathbf{End_10}^{\{x\}} &= \mathbf{Sec}^{\{x\}}. \end{aligned}$$

The LECs for the seat belt example are shown in Fig. 7 for several different scheduling policies. For the sake of readability, the LEC for SPS $T >$ C (Timer at higher priority than controller) has been compressed so that there is an edge from all (5) inputs to all (3) outputs.

From the LECs in Fig. 7, we can conclude that implementations with SPS C $> T$ are synchronously equivalent to implementations with CES C, T. For the four scheduling policies considered, LEC analysis returns the same result as exhaustive simulation. It does so with negligible time and memory requirement.

C. Case Study

We have applied LEC communication analysis to a real-life industrial design, the algorithm block of a server that supports asynchronous transfer mode (ATM)-based virtual private networks. The complete server [18], [19] required a design effort of approximately three man-years. The algorithm block, as shown in Fig. 8, was respecified as about 1200 lines of Esterel code in 13 different CFSMs. If we were to represent even just the control portion of the system (excluding tables) as a Boolean network, it would have required more than 500 binary latches. Without extensive manual abstraction, verifying this design is clearly beyond the capability of existing formal verification tools [4].

The algorithm block decides which input cells must be accepted or discarded to avoid node congestion, and implements

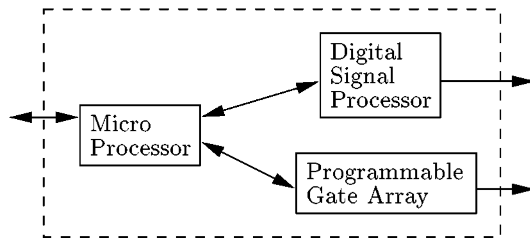

Fig. 9. Example of coprocessor architecture.

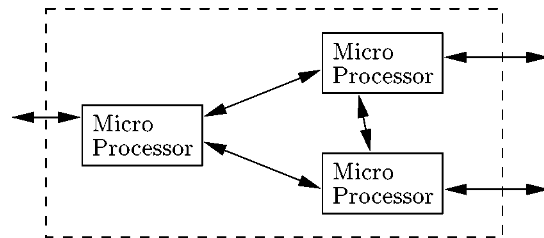

Fig. 10. Example of synchronous parallel architecture.

the shaping and bandwidth partition functions among ATM virtual path connections. Upon arrival of a new cell, it receives the cell ID from the address-lookup module. If the cell is accepted, the message selective discarding technique sends instructions to the logic queue manager (LQM) about the shared buffer queue in which the cell must be stored. Communication between the algorithm block and the LQM is handled by the LQM interface, which performs the required protocol adaptations.

We used LEC communication analysis to decide synchronous equivalence among many different implementations, including ones with UDP and various SPS and CES scheduling policies. The LECs were generated using POLIS [2] on a network where CFSMs were replaced with their least monotonic covers. In all cases, the generation of the LEC took less than 1 s of CPU time. The LEC associated with the SPS policy chosen by an expert designer consists of 314 containers. Even with such a complex LEC, we have found a CES scheduling policy that is synchronously equivalent to the designer-specified SPS, therefore resulting in an implementation with less scheduling overhead.

D. Analysis of Heterogeneous Architectures

One of the salient characteristics of embedded systems is that they may be implemented on heterogeneous architectures. Different parts of the system may be better suited for mapping to computational resources with very different performance and cost factors. Here, we consider two very common heterogeneous architectures, the coprocessor architecture and the synchronous parallel architecture, and show how they can be modeled for efficient synchronous equivalence analysis.

An example of the coprocessor architecture is shown in Fig. 9. The microprocessor acts as the master of the communication to and from the coprocessors. Whenever there is some computation to be done on the coprocessors, the microprocessor emits the events and associated data to the coprocessors and waits for the operation on the coprocessors to be completed. The operations of the master and the coprocessors, thus, become serialized. A coprocessor architecture can be modeled as and is indeed synchronously equivalent to a SPS scheduling policy on a single processor architecture with all the components mapped to the coprocessors having higher priority than the ones that are mapped to the master.

The synchronous parallel architecture is characterized by processors executing in parallel, but communication is allowed only at a time when the execution on each processor has been completed. An example is shown in Fig. 10. A synchronous parallel architecture can be hierarchically modeled as having a UDP scheduling policy on top of the original single processor scheduling policies for the individual processors. Exactly how various segments of ECs should be compared is left as future work.

V. Refining Communication Analysis

Communication analysis is efficient in deciding the equivalence between two different implementations of the same CFSM network. When two implementations have communication signatures (i.e., ECs) that are identical and finite, they are guaranteed to be synchronously equivalent to each other. Unfortunately, when two ECs are different or when one or both of the ECs are infinite, the result of the analysis is inconclusive. In the context of design exploration, the inconclusive result in equivalence analysis means that an implementation must be declared, possibly falsely, to be functionally different from the reference. If that implementation has a better performance characteristic than all the implementations that were declared to be functionally correct, it will still not be selected. If the negatives do turn out to be false, the final implementation is suboptimal.

In this section, we identify and remove, bit by bit, the sources of these false negatives through the process of refinement and pruning. Refinement simply means case-splitting on the value of a container, i.e., replacing a single EC with two, one in which the container is replaced with zero, and the other in which it is replaced one. Pruning means removing irrelevant parts of EC. We will show that if all primary input containers are refined and all irrelevant parts of ECs are removed, then ECs are exactly equal to simulation traces. Thus, we establish a smooth path to exact simulation, where the analysis result is precise and no false negative is possible. When two implementations are actually synchronously equivalent to each other, communication analysis, perhaps with some refinement and pruning, can prove this positive result without going all the way down to exhaustive simulation. Figuring out precisely how much refinement and pruning are needed and on which containers is as hard as the verification problem itself. Heuristic algorithms can be developed, but that is left as a possible topic of future research.

A. Container Refinement

In communication analysis, containers are used to represent the possibility of an event. If there is no container at some input, it is interpreted as event absence. If there is a container present at some input, the event may be actually present or it may not.

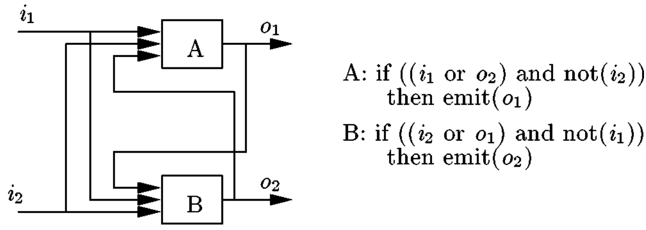

Fig. 11. Example with infinite LEC.

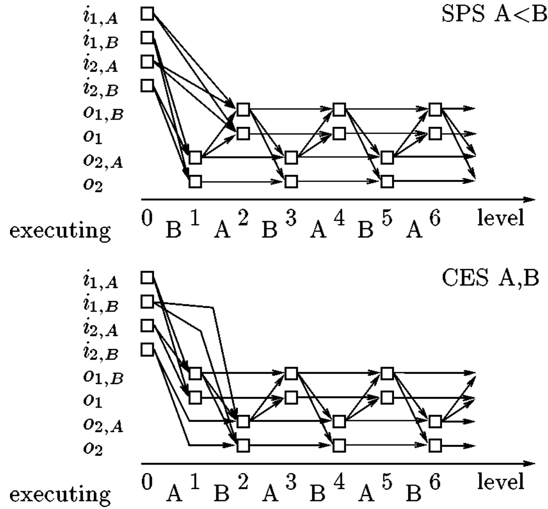

Fig. 12. Infinite LEC for example of Fig. 11.

Consider the example in Fig. 11. The output functions are

$$\begin{aligned} o_1 &= i_1 * \overline{i_2} + o_2 * \overline{i_2} \\ o_2 &= i_2 * \overline{i_1} + o_1 * \overline{i_1}. \end{aligned} \tag{16}$$

The least monotonic covers for these functions are

$$\begin{aligned} \mathbf{o}_1^{\{x\}} &= \mathbf{i}_1^{\{x\}} + \mathbf{o}_2^{\{x\}} \\ \mathbf{o}_2^{\{x\}} &= \mathbf{i}_2^{\{x\}} + \mathbf{o}_1^{\{x\}}. \end{aligned} \tag{17}$$

The LECs for two different implementations are shown in Fig. 12. In both cases, the LECs are infinite and communication analysis returns an inconclusive result.

It can be established independently through exhaustive simulation that not only these two implementations produce finite output traces for any finite input trace, but also that they are actually synchronously equivalent to each other. The infinite LECs in Fig. 12 are due to the abstraction of event containers. To make the abstract simulation finite, containers need to be "refined" to take on more precise information.

With the definition of container given in Section IV, there is no way to represent the precise presence of an event, while it is possible to represent the precise absence of an event by the absence of a container at the input of a component. ECs in the previous section are computed with two values for each input variable of a component function: event absence represented by 0 and event unknown represented by $\mathtt{x}$. For the analysis to be more precise, we need to represent the case where the event is definitely present.

We can define an n-dimensional ternary space as follows.

Definition V.1 (Ternary Container Space): An n-dimensional ternary container space is $\mathbf{C}^n = \{0, 1, \mathtt{x}\}^n$.

We use the same definition of the information ordering as in Section IV, i.e., $0 \preceq \mathtt{x}$, $1 \preceq \mathtt{x}$, $0 \preceq 0$, $1 \preceq 1$, $\mathtt{x} \preceq \mathtt{x}$, extended componentwise to minterms. The definition of a monotonic cover needs to be changed slightly. We say that a ternary container space function $\mathbf{f}$ is a monotonic cover of the event space function f if for any ternary container space minterm $\mathbf{m}$

$$\forall (m \preceq \mathbf{m}) \implies (f(m) \preceq \mathbf{f}(\mathbf{m})).$$

It is not hard to check that for the binary container space, the condition above is equivalent to the condition in Definition IV.2. While we find the form in Definition IV.2 more intuitive and easier to apply in proofs, the form above reveals the origin of the name "monotonic."

Ternary ECs for an implementation with a given scheduling policy can be obtained by the EC generation procedure described in Section IV-B-4 with the following modifications.

Step 1 Modified: Create containers for primary inputs and label them with level 0. Set the current level to zero. The input container may be filled, left with content unknown, or not be created at all, depending on what is known about the inputs. For example, if an input is known to be present for all cycles, then it can be set to be a 1-container.

Step 5 Modified: Abstractly execute the selected CFSMs by increasing the current level by one and evaluating a ternary monotonic cover of each CFSM output function with all active inputs with a 1-container set to 1, all active inputs with an $\mathtt{x}$-container set to $\mathtt{x}$, and all other inputs set to 0. If the ternary monotonic cover evaluates to one or $\mathtt{x}$, then:

1) create a new 1-container or $\mathtt{x}$-container, respectively, and label it with the current level and the name of that output;
2) for every active input, create an edge from the most recent container corresponding with that input to the newly created container;
3) create an edge from the previous container labeled with the same name (if any exists) to the new container.

The ternary monotonic covers in Step 5 modified are computed by converting the output functions from the event space to the ternary container space. Unfortunately, the iterative technique for binary container space described in Section IV cannot be applied here. We only note that similar ternary functions are computed also for symbolic simulation in the logic domain [20].

The ternary least monotonic covers for the example in Fig. 11 are

$$\begin{aligned} \mathbf{o}_1^{\{1\}} &= \mathbf{i}_1^{\{1\}} * \mathbf{i}_2^{\{0\}} + \mathbf{o}_2^{\{1\}} * \mathbf{i}_2^{\{0\}} \\ \mathbf{o}_1^{\{0\}} &= \mathbf{i}_2^{\{1\}} + \mathbf{i}_1^{\{0\}} * \mathbf{o}_2^{\{0\}} \\ \mathbf{o}_1^{\{x\}} &= \overline{\mathbf{o}_1^{\{1\}} + \mathbf{o}_1^{\{0\}}} \\ \mathbf{o}_2^{\{1\}} &= \mathbf{i}_2^{\{1\}} * \mathbf{i}_1^{\{0\}} + \mathbf{o}_1^{\{1\}} * \mathbf{i}_1^{\{0\}} \\ \mathbf{o}_2^{\{0\}} &= \mathbf{i}_1^{\{1\}} + \mathbf{i}_2^{\{0\}} * \mathbf{o}_1^{\{0\}} \\ \mathbf{o}_2^{\{x\}} &= \overline{\mathbf{o}_2^{\{1\}} + \mathbf{o}_2^{\{0\}}}. \end{aligned} \tag{18}$$

Computing the ternary EC with primary inputs set to $\mathtt{x}$ at Step 1 modified produces an infinite EC exactly like that in Fig. 12. This is expected because we have not really done any refinement

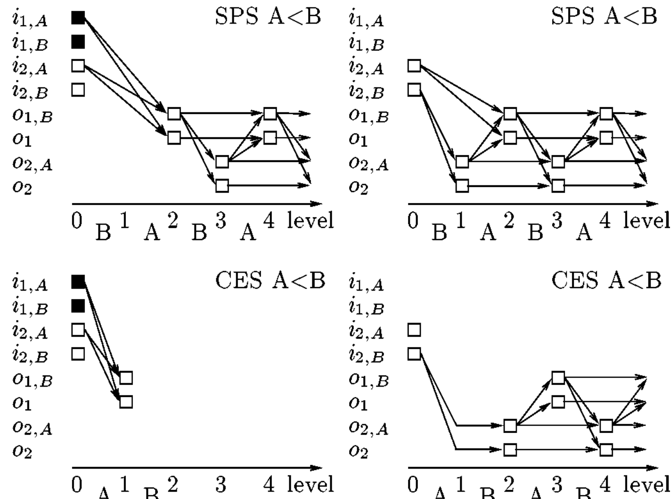

Fig. 13. Infinite LEC for example in Fig. 11 with $\mathbf{i_1}$ refined.

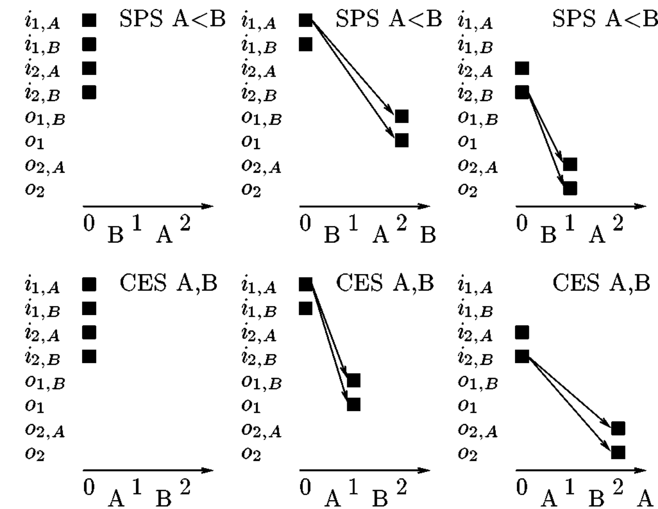

Fig. 14. Finite LEC for examples in Fig. 11 with both $\mathbf{i_1}$ and $\mathbf{i_2}$ refined.

on the containers at all. We only changed the cover functions so that they can handle the 1-containers. Abstract execution with $\mathbf{i_1}$ refined at the primary input produces the ECs shown in Fig. 13. We now have a pair of "split" ECs. One on the left side of the figure with $\mathbf{i_1} = 1$ denoted by a solid box. The other on the right side of the figure with $\mathbf{i_1} = 0$ denoted by no box at $\mathbf{i_1}$. If the corresponding portions of the ECs are identical, we can say that the two implementations are synchronously equivalent to each other, since i_1 can only be zero or one in the original event space.

Unfortunately, the refined LEC is still infinite. Further refinement is needed. The ECs with both $\mathbf{i_1}$ and $\mathbf{i_2}$ refined is shown in Fig. 14. Now the corresponding LECs are indeed identical. The implementations are, therefore, synchronously equivalent to each other. We do not have to consider the case where both $\mathbf{i_1}$ and $\mathbf{i_2}$ are refined to zero, since the system is reactive by definition. The absence of all inputs cannot cause any reaction.

We are able to (conservatively) check two implementations for synchronous equivalence by comparing their ternary ECs as stated by the following result.

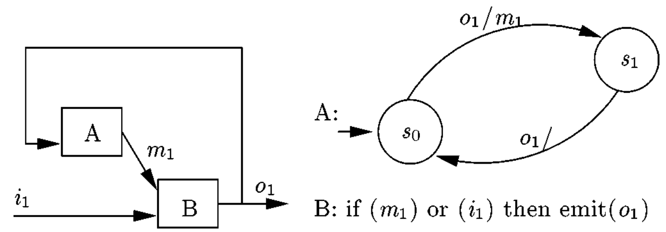

Fig. 15. Example that has infinite LEC if state is abstracted.

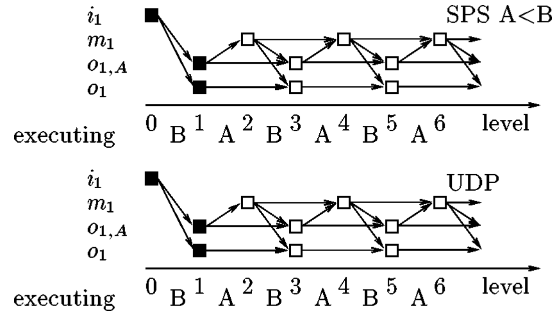

Fig. 16. Infinite LEC for example in Fig. 15 with state abstracted.

Theorem 4: If two implementations of a given CFSM network with well-behaved scheduling policies have identical ternary ECs, then they are synchronously equivalent.

This theorem is an extension of Theorem 2. Since the ternary abstract execution is carried out using a monotonic cover of the original output function, the proof proceeds in a similar fashion.

B. State Refinement

For some designs, the false negatives from communication analysis cannot be removed without explicitly representing states. Consider the example in Fig. 15. The ternary least monotonic covers of transition functions obtained by existentially quantifying out state variables are

$$\begin{aligned}
\mathbf{m_1^{\{1\}}} &= 0\\
\mathbf{m_1^{\{0\}}} &= \mathbf{o_1^{\{0\}}}\\
\mathbf{m_1^{\{x\}}} &= \mathbf{o_1^{\{x\}}} + \mathbf{o_1^{\{1\}}}\\
\mathbf{o_1^{\{1\}}} &= \mathbf{m_1^{\{1\}}} + \mathbf{i_1^{\{1\}}}\\
\mathbf{o_1^{\{0\}}} &= \mathbf{m_1^{\{0\}}} * \mathbf{i_1^{\{0\}}}\\
\mathbf{o_1^{\{x\}}} &= \mathbf{m_1^{\{x\}}} * \mathbf{i_1^{\{0\}}} + \mathbf{m_1^{\{0\}}} * \mathbf{i_1^{\{x\}}} + \mathbf{m_1^{\{x\}}} * \mathbf{i_1^{\{x\}}}.
\end{aligned} \tag{19}$$

The ternary LECs for two selected scheduling policies are shown in Fig. 16. One cannot conclude whether or not the two implementations are synchronously equivalent because the LECs are infinite. Even though the infinite LECs "look" similar, one cannot draw any conclusion about the finite execution traces from the infinite ECs. The infinite EC prevents the analysis algorithm from analyzing the entire execution trace. It can also be shown that no amount of event container refinement can make the EC finite in this case *unless we take the CFSM state into account*.

Without loss of generality, we assume that there is only one state variable per CFSM.[4] The single state variable is one-hot

[4]Multiple state variables can be mapped into a single state variable by taking the Cartesian product.

encoded into a set of state events. A 1-container at a state event means that the CFSM is at that state. An x-container at more than one state event means that the CFSM may be in any one of those states. State events have the following property.

Lemma V.1: A set of one-hot encoded state events for a CFSM has a 1-container if and only if it is the only container present for that set of state events.

Proof: By definition, any CFSM must be in a single state at any given time. Only one container present means that the CFSM can be only in one state. The event is therefore definitely present. Conversely, if there is a 1-container at some state event, then the CFSM cannot possibly be in any other state so no other state event container can be present. ■

Due to the lemma, we will convert state events with a single x-container immediately into a 1-container. Ternary ECs with state refinement for an implementation can be obtained by the EC generation procedure of Section IV-B-4 with the following modifications.

Step 1 Modified: Create containers for primary inputs and state values of the CFSMs and label them with level 0. Set the current level to zero. The input containers may be filled, left with content unknown, or not be created at all, depending on what is known about the inputs and reachable states.

Step 3 Modified: Let all CFSMs with at least one nonstate active input be enabled. If no CFSM is enabled, then STOP.

Step 5 Modified: Abstractly execute the selected CFSMs by increasing the current level by 1 and evaluating a ternary monotonic cover of each CFSM transition function, with all active inputs with a 1-container set to 1, all active inputs with an x-container set to x, and all other inputs set to 0. If the ternary monotonic cover evaluates to 1 or x then:

1) create a new 1-container or x-container, respectively, and label it with the current level and the name of that output;
2) for every active input, create an edge from the most recent container corresponding with that input to the newly created container;
3) create an edge from the previous container labeled with the same name (if any exists) to the new container;
4) for state event containers, additionally create an edge from all previous state event containers to all new state event containers.

The ternary monotonic cover is computed by converting the output function from the event space to the ternary container space, taking into consideration the state values. The LEC of two equivalent implementations for Fig. 15 with state present is shown in Fig. 17.

C. Eliminating Irrelevant Containers

Synchronous equivalence, as defined in Section III, relates to the primary inputs and outputs of a design. Communication analysis infers this global property from the communication between components, which is not a strictly global characteristic. Inferring a global property from characteristics of local elements can be conservative. Consider the example in Fig. 18. LECs for three different scheduling policies are shown in Fig. 19. It can be established through exhaustive analysis or simple observation that the three implementations are actually synchronously equivalent to each other, though there are two different sets of LECs. It is equally apparent that many containers in the LECs have nothing to do with the primary output o_1 and are indeed *unobservable* from the primary outputs.

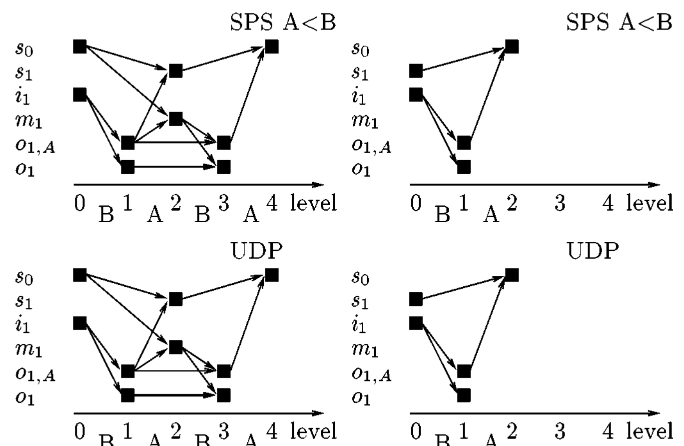

Fig. 17. LECs for example in Fig. 15 with state not abstracted.

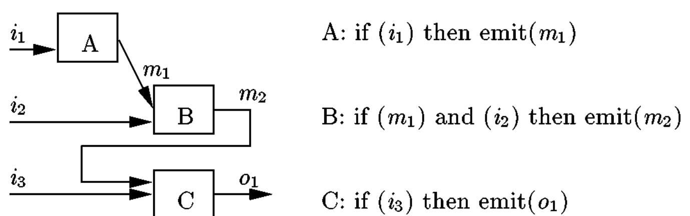

Fig. 18. Example demonstrates the need for pruning.

Definition V.2 (Observable Container): A container **c** in an EC is observable if there is an assignment of x-containers consistent with transition and output relations such that assigning zero or one to **c** produces different primary outputs for the actual implementation. Otherwise, **c** is unobservable.

The next lemma clarifies the relationship between the primary output containers in an EC and the production of a primary output event by an implementation.

Lemma V.2: If there are one or more 1-containers at some primary output of an EC, the output is emitted by the corresponding implementation.

Proof: By the definition of synchronous equivalence (Definition III.2), primary outputs matter only at the end of a cycle. By the definition of CFSM, an emitted output cannot be "cancelled." ■

A container that is not observable can be removed from the EC.

Theorem 5: Removing unobservable containers, along with all their edges, results in an EC that remains a communication signature of the implementation.

Proof: Consider an implementation A, its EC $\mathcal{C}$, and an EC $\mathcal{C}^{\mathrm{Pruned}}$, where some unobservable containers have been removed. Let A^{Pruned} be an implementation that will produce the EC $\mathcal{C}^{\mathrm{Pruned}}$. By the definition of unobservable containers and synchronous equivalence, A^{Pruned} is synchronously equivalent

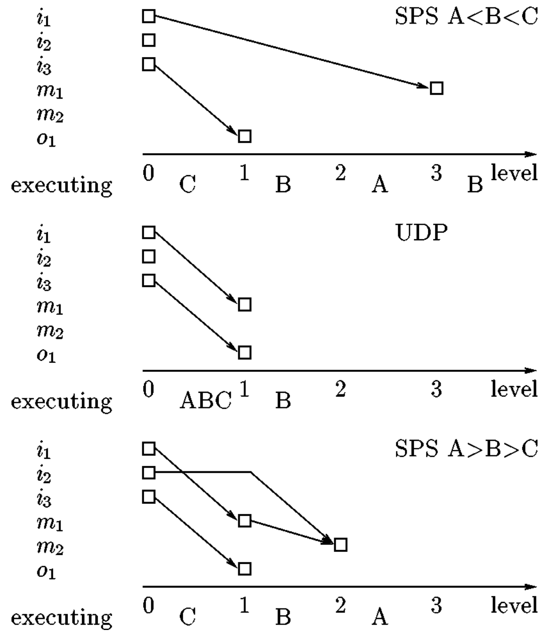

Fig. 19. LECs for example in Fig. 18.

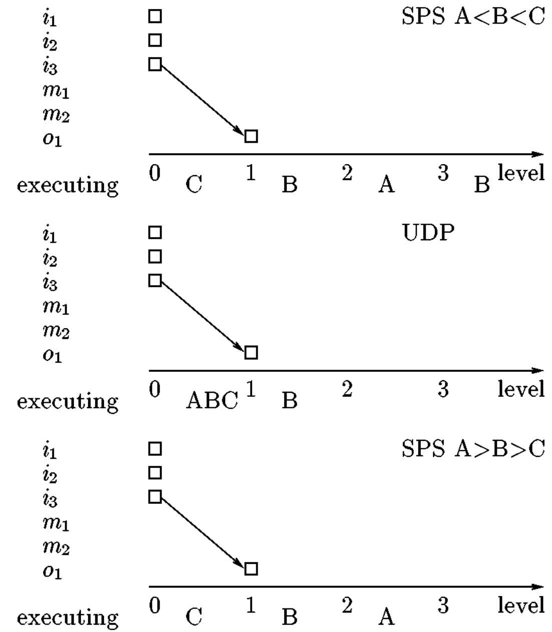

Fig. 20. Pruned LECs for example in Fig. 18.

to A. By transitivity of equivalence relations, any implementation that is synchronously equivalent to A^{Pruned} is also synchronously equivalent to implementation A. ■

Figuring out whether a container is observable is analogous to the classical observability problem in sequential circuits [21]. Finding an exact solution could require an exponential computation, though simple pruning can go a long way in removing a large number of these unobservable containers.

Corollary V.1: If there are one or more 1-containers at some primary output of an EC, all but one of the 1-containers of that primary output can be removed and the EC remains a communication signature of the implementation.

Proof: By Lemma V.2, the output will be emitted if one or more 1-containers exist at some primary output of the EC. This will remain true if all but one of these 1-containers are removed. By Theorem 5, the pruned EC is a communication signature of the implementation. ■

Corollary V.2: Any nonprimary-output container that has no directed edge to an x-container can be removed and the EC remains a communication signature of the implementation.

Proof: Assigning one or zero to a nonprimary-output container with no directed edge to an x-container clearly cannot affect the primary output, so the container is unobservable and can be removed. Removing a nonprimary-output 1-container that has no directed edge to an x-container also cannot affect the primary output. By Theorem 5, the pruned EC is a communication signature of the implementation. ■

A large number of unobservable containers can be removed by the following procedure.

1) For all primary outputs that have two or more containers and at least one 1-container, remove all containers of that primary output except any one single 1-container.
2) Iteratively remove all nonprimary-output x-containers and 1-containers that do not have an edge to an x-container.

The pruned ECs for the example in Fig. 18 (derived by pruning the original EC in Fig. 19) are shown in Fig. 20. The original primary input containers are retained for clarity. The implementations are indeed synchronously equivalent to each other.

D. Relationship with Exact Simulation

With container refinement and pruning, ECs can be related to exact simulation. Exact simulation requires deterministic transition and output relations as well as a deterministic input trace. A deterministic input trace for an actual implementation corresponds to an "abstract" execution with an input trace consisting of only 1-containers.

Theorem 6: Given deterministic transition relations and output relations, a well-behaved scheduling policy, and a deterministic input trace, the output trace from simulation is identical to a trace of the pruned EC from the corresponding abstract simulation of the cover functions.

Proof: The transition and output relations are deterministic and the primary input of the abstract simulation consists of only 1-containers. By Step 5 modified of the ternary EC generation algorithm, only 1-containers will exist in the EC. By Corollaries V.1 and V.2, only a single 1-container at each primary output will remain. The trace of the EC is therefore a trace of 1-containers at the primary output. In addition, a container will be present for every event present in the output trace of the corresponding simulation run. ■

The theorem relates a simulation run of the implementation to an abstract execution of its monotonic covers. However, not all ECs have corresponding simulation runs, since not all combinations of the states of the components are reachable, and the given pruning algorithm does not identify all unobservable containers. The former can be solved by existing state reachability techniques on the global state and the latter can be solved by further refining the remaining x-containers, both at a cost of higher computational complexity.

VI. Conclusion and Future Directions

In this paper, we have introduced the concept of synchronous equivalence among embedded system implementations satisfying the synchronous assumption. This property is analogous to functional equivalence for sequential circuits and takes full advantage of the separation of timing and functionality. One powerful result of this equivalence criterion is the identification of a set of delay insensitive implementations. For delay insensitive implementations, any variation in delay does not affect the functional behavior. Static analysis of this sort is very efficient and effective in proving equivalence, though it may at times be conservative. To check for equivalence between different delay insensitive implementations, we proposed algorithms based on worst case analysis of the communication among components. The events communicated between components are abstracted into a signature that is maximal in the sense that it represents all possible communication patterns of that implementation. By comparing the signatures of different delay insensitive implementations for a given specification, we were able to determine equivalence conservatively, but efficiently. We demonstrated with real-life examples that synchronous equivalence opens design exploration avenues uncharted before. We also related communication analysis to exact simulation through a series of refinement and pruning operations on the communication signatures. An algorithm can choose to work at any abstraction level, trading off computational efficiency with the possibility of inconclusive result due to false negatives. We provided primitives to move amongst different abstraction levels that exist between abstract communication analysis and exact simulation.

An interesting open question is what happens if we violate the assumption that the component functionalities and the component connectivity are the same among all possible implementations. General repartitioning can be thought of as a series of decompositions and compositions of the CFSMs. It is not hard to compare two implementations where the difference is only that one component is decomposed into two or that two components are combined into one through synchronous decomposition and composition. If the "subnetwork" represented by the two decomposed components for the given implementation satisfies the synchronous assumption locally, then it must also behave the same as the implementation with a single, synchronously composed component. How this check can be done efficiently and locally will be crucial for an efficient general repartitioning methodology.

Appendix
Proof of Theorem 2

We prove the theorem in several steps. Given two policies Select_1 and Select_2, we first construct auxiliary policies $\mathrm{Select}_1^{\mathrm{EC}}$ and $\mathrm{Select}_2^{\mathrm{EC}}$. The policy $\mathrm{Select}_1^{\mathrm{EC}}$ ($\mathrm{Select}_2^{\mathrm{EC}}$) is similar to Select_1 (Select_2), but use a different criterion to decide which CFSMs are enabled based on the EC of Select_1 (Select_2 respectively). Then, we prove that Select_1 and $\mathrm{Select}_1^{\mathrm{EC}}$ are synchronously equivalent (the same proof applies to the equivalence of Select_2 and $\mathrm{Select}_2^{\mathrm{EC}}$). Finally, we show that $\mathrm{Select}_1^{\mathrm{EC}}$ and $\mathrm{Select}_2^{\mathrm{EC}}$ are equivalent and the desired result follows by transitivity.

To define $\mathrm{Select}_1^{\mathrm{EC}}$ and $\mathrm{Select}_2^{\mathrm{EC}}$, we assume that the network state is extended to include a (partially constructed) EC. Under this assumption, $\mathrm{Select}_1^{\mathrm{EC}}$ is defined as follows.

1) Examine the EC portion of the current state and construct s_{active} as in Step 3 of the EC generation procedure.
2) Abstractly execute all CFSMs in $\mathrm{Select}_1(s_{\mathrm{active}})$ as in Step 5 of the EC generation procedure (modifying the EC portion of the current state accordingly).
3) Return $\mathrm{Select}_1(s_{\mathrm{active}})$. $\mathrm{Select}_2^{\mathrm{EC}}$ is defined similarly.

Lemma A.1: If s is a network state that appears in some synchronous assumption conforming run of a CFSM network (PI, PO, C) and s_{active} is the state constructed while evaluating $\mathrm{Select}_1^{\mathrm{EC}}$ in s, then for all events $e \in E$

$$s(e) = 1 \implies s_{\mathrm{active}}(e) = 1. \tag{20}$$

Proof: By induction on the number of scheduling points in the run and in the base case $s_{\mathrm{active}}(pi) = 1$ for all primary inputs, these are the only events that could be present in s. For the inductive step, we consider some output out of some $A \in \mathrm{Select}_1^{\mathrm{EC}}(s) = \mathrm{Select}_1(s_{\mathrm{active}})$ and use $\mathbf{m}$ to denote the translation of $s_{\mathrm{active}}|_{I_A}$ into container space. By inductive assumption, $s|_{I_A} \preceq \mathbf{m}$. Output *out* will be present in the state in the next step of the run if $T_A^{\mathrm{out}}(s|_{I_A}) = 1$ and it will be present in the s_{active} evaluated at that state if a monotonic cover of T_A^{out} evaluates to $\mathbf{x}$ at $\mathbf{m}$. The implication between the two holds by inductive assumption and the definition of monotonic cover. ■

Lemma A.2: Select_1 and $\mathrm{Select}_1^{\mathrm{EC}}$ are synchronously equivalent.

Proof: By Lemma A.1, $\mathrm{Enabled}(s) \subseteq \mathrm{Enabled}(s_{\mathrm{active}})$ at every scheduling point. Since Select_1 is well behaved, it follows that at each true scheduling point i of $\mathrm{Select}_1^{\mathrm{EC}}$

$$\mathrm{Select}_1^{\mathrm{EC}}(s_i) = \mathrm{Select}_1(s_i).$$

We can now use the induction on true scheduling points to prove that the number of true scheduling points in two implementations is the same and that the same events are present in both implementations after corresponding true scheduling points. ■

By Lemma A.1, whenever $\mathrm{Select}_1^{\mathrm{EC}}$ generates an event, it also generates a container for it in EC. In other words Select_1^{EC} can be seen as determining the contents of the containers in the EC, one if the event is present and zero if it is absent.

Corollary A.1: The content of a container determined by $\mathrm{Select}_1^{\mathrm{EC}}$ depends only on the contents of its immediate predecessors in the EC and on the CFSM output function.

Proof: By definition, all active inputs are immediate predecessors and by Lemma A.1, all other inputs are zero. ■

The following result states that the EC contains sufficient information to decide synchronous equivalence.

Lemma A.3: If two scheduling policies Select_1 and Select_2 have identical ECs and if $\mathrm{Select}_1^{\mathrm{EC}}$ and $\mathrm{Select}_2^{\mathrm{EC}}$ assign to corresponding nodes the same contents for every primary input assignment, then Select_1 and Select_2 are synchronously equivalent.

Proof: We first show that $\mathrm{Select}_1^{\mathrm{EC}}$ and $\mathrm{Select}_2^{\mathrm{EC}}$ are synchronously equivalent. Indeed, by definition of EC, all containers corresponding to some primary output form a chain. By assumption, the contents of the last containers are the same for $\mathrm{Select}_1^{\mathrm{EC}}$ and Select_2^{EC}, implying that $\mathrm{Select}_1^{\mathrm{EC}}$ and $\mathrm{Select}_2^{\mathrm{EC}}$

are synchronously equivalent. It follows then by Lemma A.2 that Select_1 and Select_2 are also synchronously equivalent. ■

Now we have all the pieces to prove the theorem.

Proof [of Theorem 2]: By Lemma A.3, we only need to show that given two different well behaved policies Select_1 and Select_2, their modifications $\mathrm{Select}_1^{\mathrm{EC}}$ and $\mathrm{Select}_2^{\mathrm{EC}}$ assign the same values to corresponding containers. This is shown by induction, using Corollary A.1, and the fact that the CFSM transition functions are the same in the two implementations. ■

References

[1] S. Unger, *Asynchronous Sequential Switching Circuits*. New York: Wiley, 1969.

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

[3] K. McMillan, *Symbolic Model Checking*. Norwell, MA: Kluwer, 1997.

[4] R. Brayton, A. Sangiovanni-Vincentelli, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, S. Qadeer, R. Ranjan, T. Shiple, G. Swamy, T. Villa, G. Hachtel, F. Somenzi, A. Pardo, and S. Sarwary, "VIS: A system for verification and synthesis," in *Proc. Int. Conf. Computer-Aided Verification*, July 1996, pp. 428–432.

[5] H. Hsieh, F. Balarin, L. Lavagno, and A. Sangiovanni-Vincentelli, "Synchronous equivalence for embedded systems: A tool for design exploration," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1999, pp. 505–509.

[6] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proc. IEEE*, vol. 79, pp. 1270–1282, Sept. 1991.

[7] F. Balarin, "Worst-case analysis of discrete systems," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1999, pp. 347–352.

[8] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proc. IEEE*, vol. 79, pp. 1305–1320, Sept. 1991.

[9] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, pp. 1235–1245, Sept. 1987.

[10] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. Int. Federation for Information Processing Congress*, Aug. 1974, pp. 471–475.

[11] H. Kopetz, "The time-triggered architecture," in *Proc. Int. Symp. Object-Oriented Real-Time Distributed Computing*, Apr. 1998, pp. 22–29.

[12] J. Young, J. MacDonald, M. Shilman, A. Tabbara, P. Hilfinger, and A. Newton, "Design and specification of embedded systems in Java using successive, formal refinement," in *Proc. Design Automation Conf.*, June 1998, pp. 70–75.

[13] R. Milner, "A calculus of communication systems," in *Lecture Notes in Computer Science*. New York: Springer-Verlag, 1980, vol. 92.

[14] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. Assoc. Comput. Mach.*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[15] M. Nielsen, G. Plotkin, and G. Winskel, "Petri nets, event structures and domains—Part I," *Theor. Comput. Sci.*, vol. 13, pp. 85–108, 1981.

[16] B. Davey and H. Priestley, *Introduction to Lattices and Order*. Cambridge, U.K.: Cambridge Univ. Press, 1990.

[17] A. Malik, R. Brayton, A. Newton, and A. Sangiovanni-Vincentelli, "Two-level minimization of multivalued functions with large offsets," *IEEE Trans. Comput.*, vol. 42, pp. 1325–1342, Nov. 1993.

[18] P. Coppo, M. D'Ambrosio, and V. Vercellone, "The A-VPN server, a solution for ATM virtual private networks," in *Proc. Int. Conf. Conceptual Structures*, Nov. 1994, pp. 298–304.

[19] E. Filippi, L. Lavagno, L. Licciardi, A. Montanaro, M. Paolini, R. Passerone, M. Sgroi, and A. Sangiovanni-Vincentelli, "Intellectual property re-use in embedded system co-design: An industrial case study," in *Proc. Int. Symp. System Synthesis*, Dec. 1998, pp. 37–42.

[20] C. Seger and J. Brzozowski, "Generalized ternary simulation of sequential circuits," *Informatique Theorique et Applications*, vol. 28, no. 3–4, pp. 159–86, 1994.

[21] H. Wang, "Hierarchical sequential synthesis: Logic synthesis of FSM networks," Ph.D. dissertation, Univ. California, Berkeley, CA, 1996.

Harry Hsieh (S'93–M'00) received the B.S. degree from the University of Wisconsin, Madison, in 1988, the M.S. degree from Stanford University, Stanford, CA, in 1991, and the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley, in 2000.

He was a Member of Technical Staff at Hewlett-Packard's Mainline Systems Laboratory, Cupertino, CA. He was with IBM's Federal System Division and the T.J. Watson Research Center. His current research interests include computer and embedded systems, architecture, algorithm, design methodology, and computer-aided design of VLSI systems.

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

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

Luciano Lavagno (S'88–M'93) received the Dottore in Ingegneria degree (*magna cum laude*) in electrical engineering from Politecnico di Torino, Torino, Italy, in 1983 and the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley, in 1992.

From 1984 to 1988, he was with CSELT Laboratories,Torino, Italy, where he was involved in the ESPRIT 802 CVS project that developed a complete high-level synthesis system. In 1988, he joined the Department of Electrical Engineering and Computer Science at the University of California, Berkeley, where he worked on logic synthesis and testing of synchronous and asynchronous circuits. From 1993 to 1998, he was an Assistant Professor with the Department of Electronics, Politecnico di Torino. Since 1993, he has been an Architect of the POLIS project, a cooperation between University of California, Berkeley, Cadence Design Systems, Magneti Marelli, and Politecnico di Torino, developing a complete hardare/software codesign environment for control-dominated embedded systems. Since 1994, he has been a Research Scientist at Cadence Berkeley Laboratories. Since 1997, he has participated in the ESPRIT 25 443 COSY project, developing a methodology for software synthesis and performance analysis for embedded systems. Since 1998, he has been an Associate Professor with the Department of Electrical, Management, and Mechanical Engineering (DIEGM) with the University of Udine, Udine, Italy. He has also been a consultant for various EDA companies, such as Synopsys and Cadence. He has authored or coauthored a book on asynchronous circuit design, a book on hardware/software codesign of embedded systems, and over 80 journal and conference papers. 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.

Dr. Lavagno received a Best Paper Award at the Design Automation Conference in 1991. He has served on the technical committees of several international conferences in his field (namely, the Design Automation Conference, the International Conference on Computer-Aided Design, Design Automation and Test in Europe) and as a technical committee member or chair of several workshops and symposia (such as the International Symposium on Asynchronous Circuits and Systems, the International Workshop on Hardware-Software Co-Design, and the International Workshop on Logic Synthesis).

Alberto Sangiovanni-Vincentelli (F'83) received the Dottore in Ingegneria degree (*summa cum laude*) in electrical engineering and computer science degree from the Politecnico di Milano, Milano, Italy, in 1971.

From 1980 to 1981, he was a Visiting Scientist at the Mathematical Sciences Department of the IBM T.J. Watson Research Center. In 1987, he was Visiting Professor at the Massachusetts Institute of Technology. He has held a number of Visiting Professor positions at the University of Torino, University of Bologna, University of Pavia, University of Pisa, and University of Rome. He currently holds the Edgar L. and Harold H. Buttner Chair of Electrical Engineering and Computer Sciences and the Vice-Chair position for Industrial Relations at the University of California, Berkeley, where he has been on the Faculty since 1976. He is a Cofounder of Cadence and Synopsys, the two leading companies in the area of Electronic Design Automation. He was previously Director of ViewLogic and Pie Design System and Chair of the Technical Advisory Board of Synopsys and is currenlty the Chief Technology Advisor of Cadence Design System. He is also a Member of the Board of Directors of Cadence, where he is the Chairman of the Nominating Committee, Sonics Inc., Accent, an ST Microelectronics-Cadence joint venture, and Cofounder and Chairman of the Board of ComSilica, a startup in the Wireless communication domain. He is Founder of the Cadence Berkeley Laboratories, the Cadence European Laboratories, and the Kawasaki Berkeley Concept Research Center, where he is Chairman of the Board. He is on the Advisory Board of the Lester Center of the Haas School of Business and of the Center for Western European Studies and a Member of the Berkeley Roundtable of the International Economy. He is the Scientific Director of the Project on Advanced Research on Architectures and Design of Electronic Systems, a European group of economic interest supported by Cadence, Magneti-Marelli, and ST Microelectronics. He has consulted for a number of U.S. companies including IBM, Intel, AT&T, Actel, GTE, GE, Harris, Nynex, Teknekron, DEC, HP; Japanese companies including Kawasaki Steel, where he holds the title of Chief Technology Advisor, Fujitsu, Sony, and Hitachi; and European companies including SGS-Thomson Microelectronics, Alcatel, Daimler-Benz, Ericsson, Magneti-Marelli, BMW, and Bull. He has authored or coathored over 520 papers and 11 books in the area of design methodologies, large-scale systems, embedded controllers, hybrid systems, and tools.

Dr. Sangiovanni-Vincentelli is a Member of the National Academy of Engineering. He received the IEEE Circuits and Systems Society Golden Jubilee Medals in 1999, the Distinguished Teaching Award of the University of California in 1999, the worldwide IEEE Graduate Teaching Award in 1995, and numerous awards including the Guillemin-Cauer Award (1982 to 1983) and the Darlington Award (1987 to 1988). He was the Technical Program Chairperson and General Chair of the International Conference on Computer-Aided Design. He was the Executive Vice-President of the IEEE Circuits and Systems Society.