Sequential Synthesis Using S1S

Adnan Aziz, Felice Balarin, *Member, IEEE*, Robert K. Brayton, *Fellow, IEEE*, and Alberto Sangiovanni-Vincentelli, *Fellow, IEEE*

***Abstract*—We propose the use of the logic S1S as a mathematical framework for studying the synthesis of sequential designs. We will show that this leads to simple and mathematically elegant solutions to problems arising in the synthesis and optimization of synchronous digital hardware. Specifically, we derive a logical expression which yields a single finite state automaton characterizing the set of implementations that can replace a component of a larger design. The power of our approach is demonstrated by the fact that it generalizes immediately to arbitrary interconnection topologies, and to designs containing nondeterminism and fairness. We also describe control aspects of sequential synthesis and relate controller realizability to classical work on program synthesis and tree automata.**

***Index Terms*—Automata theory, discrete control, mathematical logic, sequential logic synthesis.**

I. Introduction

THE advent of modern VLSI CAD tools has radically changed the process of designing digital systems. The first CAD tools automated the final stages of design, such as placement and routing. As the low level steps became better understood, the focus shifted to the higher stages. In particular logic synthesis, the science of optimizing gate level designs for measures such as area, speed, or power, has shifted to the forefront of CAD research.

Logic synthesis algorithms originally targeted the optimization of two-level logic; this was followed by research in synthesizing more general multilevel logic. Currently, a major thrust in logic synthesis is sequential synthesis, i.e., the automatic optimization of the entire system. This is for designs specified at the structural level in the form of netlists, or at the behavioral level, i.e., in the form of finite state machines (FSMs). De Micheli [21] gives an excellent introduction to logic synthesis.

Designs invariably consist of a set of interacting components. The environment of a particular component gives rise to a certain amount of flexibility when implementing it; this flexibility can be exploited by optimization tools. For example, a datalink controller interacting with a bus operating in single processor mode may never see requests on consecutive cycles. This may help simplify the logical circuitry associated with the datalink controller.

Typically, the synthesis process has two stages: First, the set of all possible implementations is characterized using some finite structure (which is the topic of this paper); consequently, one is chosen according to some optimality criteria (e.g., minimum state [15]). For combinational designs, the problem of determining and using the flexibility afforded by "don't care" conditions is well solved both in theory and practice [28].

We propose the use of the logic S1S as a mathematical framework for studying the synthesis of sequential designs. We will show that this leads to simple and mathematically elegant solutions to problems arising in the synthesis and optimization of synchronous digital hardware. Specifically, we derive a logical expression which yields a single finite state automaton characterizing the set of implementations that can replace a particular component which is part of a larger design. The power of our approach is seen by the fact that it can be applied to designs containing nondeterminism and fairness [8], [18], and also to arbitrary interconnection topologies.

Optimization of compositional designs may result in combinational cycles, i.e., loops consisting solely of gates. Even though such loops can sometimes be used to optimize circuits, it is considered good design practice to avoid them, because cyclic circuits are difficult to analyze, and can have undesired oscillatory behaviors [3], [19], [29]. Guided by design practice, we identify flexibility available for synthesis while ensuring that cycles of logic will not be introduced by optimization.

The term "synthesis" is used in the theoretical computer science community to describe the process of taking a logical specification, and checking if there exists a model which satisfies it. The model depends on the context; for example, it could be a Turing machine program [20], a finite state transducer [23], or a dataflow graph [1]. The issues involved in this discipline include decidability, complexity, and expressiveness of the specification language. In this paper we will be mostly concerned with the optimization problem; we will make a connection to program synthesis.

Previous work in the VLSI design automation community related to optimizing interacting sequential designs has tended to be ad hoc, incomplete, and, sometimes, simply incorrect. The constructions and proofs offered are often extremely cumbersome. This is witnessed by a number of previous papers [10], [26], [5], [17], [33]–[35].

Similar problems have been considered in the control community under the label "*model matching*" [6], in the discrete event system (DES) community under the label "*supervisory control*" [37] [25], and in concurrency theory they appear as

Manuscript received November 6, 1999; revised March 17, 2000. This work was supported in part by grants from the Semiconductor Research Corporation and the National Science Foundation. This paper was recommended by Associate Editor M. Papaefthymiou.

A. Aziz is with the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX 78712 USA (e-mail: adnan@ece.utexas.edu).

F. Balarin is with the Cadence Berkeley Laboratories, Berkeley, CA 94704 USA.

R. K. Brayton and A. Sangiovanni-Vincentelli are with the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720 USA.

Publisher Item Identifier S 0278-0070(00)09144-2.

"*scheduler synthesis*" [36] and "*equation solving*" [22]. Compared to model matching approaches [6] we limit somewhat the choice of possible controllers. The limitation is not serious in hardware context, because it rules out only those circuits that result in a loop of combinational gates when composed (as previously remarked, avoiding combinational loops is considered good design practice). On the positive side, we allow more general specifications and provide a uniform methodology that is applicable to various model matching problems. This general framework also strictly subsumes the problem considered in [35]. Compared to supervisory control of DES [25], our approach offers the advantage of being compatible with FSM techniques that have seen continuous developments in the past three decades (e.g., [16] and [35]), provides more natural model of reactive system, and allows significantly simpler development of results.

We have chosen input-output language containment as a correctness criterion because it allows loose specifications, where a range of behaviors may be acceptable. Here, we differ from most of the previous approaches in the process algebra settings, where a much stronger relation, typically some form of bisimulation equivalence is used [22]. The exception is [14] which offers a general framework where the satisfaction relation is not set a priori, but can be defined by a formula in a logic that can express, among other relations, both simulation and bisimulation. However, the procedure presented in [14] generates only a single solution. We believe that it is advantageous to separate the solution process in two stages: first, all the possible solutions are characterized, and then one is chosen according to some optimality criteria.

The rest of this paper is structured as follows: in Section II we give definitions, in particular those connected to hardware, design composition, and fairness. In Section III, we review S1S logic and finite state automata, and use these notions to assign semantics to hardware. In Section IV, we use S1S to logically characterize the flexibility that can be used to optimize components in hierarchical designs. The relationship to the more classical view of program synthesis in the form of Church's problem [24], automata on trees, and fairness is described in Section V. We summarize our contributions in Section VI and suggest a number of ways of extending our results.

II. Formal Models for Hardware

In order to be able to formally reason about hardware, we need to develop mathematical models for digital systems. In this section, we develop two formalisms for expressing designs, namely FSMs and netlists. FSMs are more abstract—they correspond to the behavioral specification as given by the designer. Netlists are "structural"—they are closer to the actual implementation.

A. Sequences

A *finite sequence* on a set Σ is a function whose range is Σ and domain is a prefix of the set of natural numbers, $\omega = \{0, 1, 2, \ldots\}$. An *infinite sequence* (which we will interchangeably refer to as an ω-sequence) on Σ is a function mapping ω to Σ. We will denote the finite sequence α by $\langle a_0, a_1, \ldots, a_{n-1}\rangle$; an infinite sequence β will be written as $\langle b_0, b_1, b_2, \ldots\rangle$. Given a sequence α (finite or infinite), we will denote by $[\alpha]_k$ the kth element in the sequence, i.e., $\alpha(k)$. The elements of the range that occur infinitely often in an infinite sequence α will be denoted by $\mathtt{inf}(\alpha)$. The *length* of a finite sequence α is the cardinality of its domain, and will be denoted by l_α.

Given any sequence α (finite or infinite) and natural number k, the kth prefix of α is the finite sequence $\langle \alpha(0), \alpha(1), \ldots, \alpha(k-1)\rangle$; it will be denoted by α^k.

The set of all finite sequences over a set Σ is denoted by Σ^*; the set of all infinite sequences over Σ will be denoted by Σ^ω. Subsets of Σ^* will be referred to as $*$-*languages*; subsets of Σ^ω will be referred to as ω-*languages*.

B. FSMs

FSMs provide a natural way of describing systems in which the output depends not only on the current input, but also on past values of the input, while possessing only a bounded amount of memory. FSMs are described in [13, p. 42]; below we develop enough theory to suffice for this paper.

Definition 1: An FSM is a six-tuple $(Q, s, I, O, \lambda, \delta)$ where Q is a finite set referred to as the *states*, $s \in Q$ is the *initial state*, I and O are finite sets referred to as the set of *inputs* and *outputs* respectively, δ: $Q \times I \mapsto Q$ is the *next-state function*, and λ: $Q \times I \mapsto O$ is the *output function*.

The next-state function can be inductively extended to yield the function δ^*: $Q \times I^* \mapsto Q$

$$\delta^*(t, \iota) = t \quad \text{when } l_\iota = 0$$
$$\delta^*(t, \iota) = \delta(\delta^*(t, \iota^{l_\iota - 1}), [\iota]_{l_\iota - 1}) \quad \text{otherwise.}$$

An FSM can be represented graphically by a directed finite graph, referred to as a *state transition graph*, where the vertices correspond to states. The edges are labeled with input-output value pairs—the input value enables the transition, and the output value is produced. The destination node of the edge represents the next state for that input value. This is illustrated in Fig. 1(b).

Given a state s_0 and sequence of inputs $\iota = \langle i_0, i_1, \ldots\rangle$, we will refer to the sequence of states $\boldsymbol{\sigma} = \langle s_0, s_1, \ldots\rangle$ as being the *run* (sometimes referred to as the *path*) starting at s_0 on input ι iff for all k, we have $\delta_M(s_k, \iota_{k+1}) = s_{k+1}$. The output sequence $o = \langle o_0, o_1, \ldots\rangle$ *corresponds* to (ι, σ) iff for all k, we have $\lambda(s_k, i_k) = o_k$.

C. Netlists

A netlist is a representation of a design at the *structural level* which is closer to the actual implementation of the design than FSMs, which can be viewed as behavioral level descriptions of the design.

Definition 2: A *netlist* is a directed graph, where the nodes correspond to elementary circuit elements, and the edges correspond to wires connecting these elements. Each node is labeled with a distinct variable w_i. The three primitive circuit elements are *primary inputs*, *latches*, and *gates*. Primary input nodes have no fanins; latches have a single input. Each latch has a designated initial value. Associated with each gate g is a Boolean

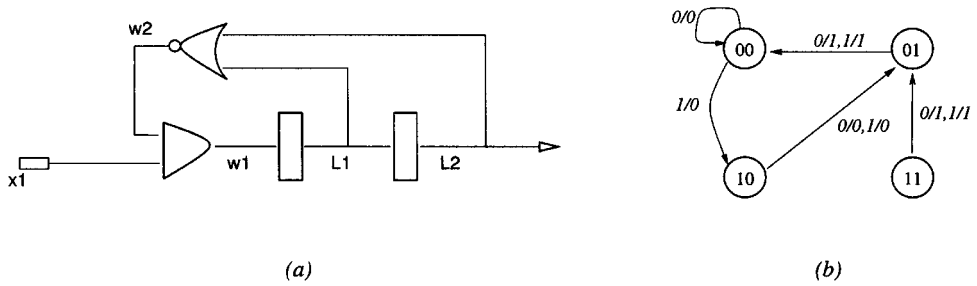


Fig. 1. Fig. 1 (a) A netlist and (b) its corresponding FSM.

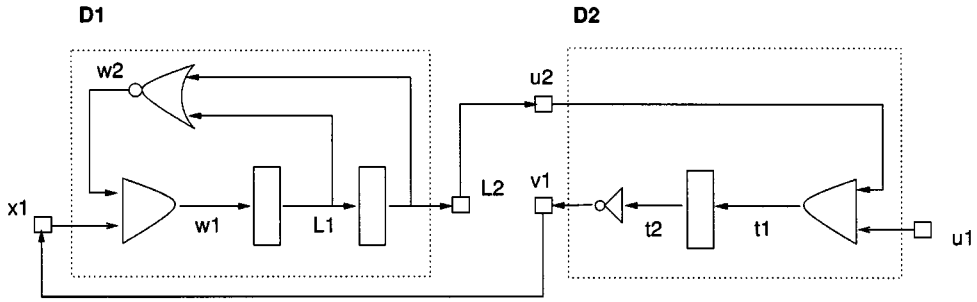


Fig. 2. Composing netlists.

function of its fanins' variables. A subset of the set of nodes is designated as being set of *primary outputs*.

For the reasons given in Section I, we require that every cycle in a netlist to include at least one latch (i.e., there are no combinational cycles).

Fig. 1(a) provides a graphical depiction of a netlist. The node x_1 is a primary input; nodes l_1 and l_2 are latches, and nodes w_1 and w_2 are gates. The node l_2 is designated a primary output. In this example, the node w_2 is driven solely by latches (i.e., there is no path from an input node to w_2 which does not pass through a latch), while the node w_1 is driven by both primary inputs and latches.

Given a set of assignments to each primary input node and a state, one can uniquely compute the values of each node in the netlist by evaluating the functions at gates. In this way, a netlist η on inputs $a_1, a_2, \ldots, a_n$, outputs $b_1, b_2, \ldots, b_m$ and latches $r_1, r_2, \ldots, r_k$ bears a natural correspondence to an FSM M_η on inputs $X = \{0,1\}^n$, outputs $Y = \{0,1\}^m$, and state-space $Q = \{0,1\}^k$, with an initial state given by the initial values for latches. An example of this correspondence is given in Fig. 1.

D. Netlist Composition

Composition of two netlists consists of placing the two netlists next to each other and connecting the nodes for primary inputs and primary outputs which are specified by the composition. The primary inputs of the composed nelist are the primary inputs of the original netlists which remain unconnected. A subset of the primary outputs of the original netlists is designated as being the primary outputs of the composed netlist; the remainder are said to be *hidden*

This is illustrated in Fig. 2, where the inputs x_1 and u_2 are "tied" to v_1 and l_2 respectively; v_1 is designated an output in the composed design. As stated in the introduction, our notion of composition is synchronous, i.e., all the latches are assumed to be driven by a single clock and, hence, change state in lockstep.

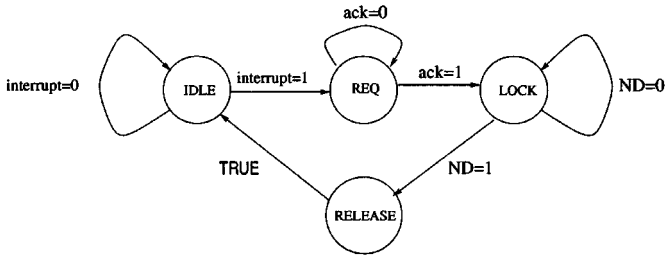


Fig. 3. A four-state abstraction of a processor.

We will only consider netlist composition when it does not result in combinational cycles.

A *Moore* netlist is a netlist where there is no path from an input to an output which does not pass through a latch; it has the property that no combinational cycles can result on composing it with any netlist. The FSM derived from such a netlist has the property that the output is purely a function of the state; such FSMs are referred to as Moore machines.

E. Fairness

There are situations when a design cannot be captured using a FSM by itself. Consider, for example, what happens when a processor is abstracted to a four state machine which cycles through *idle, request, lock,* and *release* states, with the transition out of *lock* being nondeterministic, as in Fig. 3. In order to model the processor accurately, it may be desirable to specify the condition that it does not remain in the state *lock* forever. This cannot be modeled using an FSM; a *fairness* constraint must be specified as part of the design.

In this paper, we will take a very simple approach to fairness; we will restrict our attention to *Büchi* fairness. A Büchi fairness condition is a subset of the state-space of the FSM.

Definition 3: An infinite path σ satisfies a Büchi fairness condition f iff $\texttt{inf}(\sigma)$ has a nonempty intersection with f.

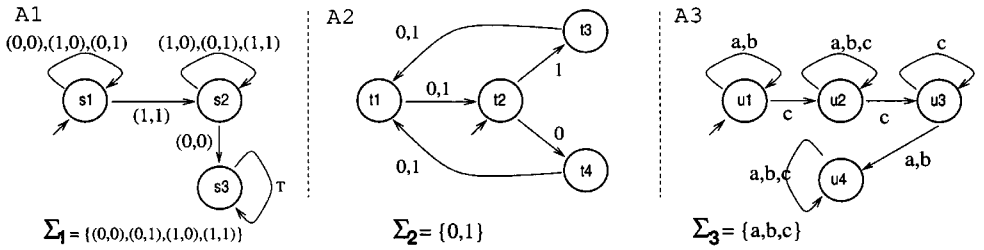


Fig. 4. Examples of finite state automata.

The path σ is fair relative to a set of Büchi fairness conditions $F = \{f_1, f_2, \ldots, f_n\}$ (collectively referred to as a *Büchi fairness constraint*), iff it is fair with respect to each fairness condition.

Fairness constraints on components of a design can be extended to fairness on the composed design: a path in the design is fair exactly when it is fair with respect to each component.

III. Finite State Automata and S1S

We start this section by defining finite state automata. We will then develop S1S which is the logical system concerned with "second order" properties of the natural numbers with the successor operation. We present a classical theorem of Büchi which shows a surprising relationship between finite state automata and S1S. Thomas [31] provides an excellent survey of the material covered in this section.

A. Finite State Automata

Definition 4: A finite state automaton is a four-tuple (Σ, S, s_0, T) where Σ is a finite set called the *alphabet* whose elements are referred to as *symbols*, S is a finite set referred to as the *states*, $s_0 \in S$ is the *initial state*, and $T \subseteq S \times \Sigma \times S$ is the *transition relation*. The relation T is required to be *completely specified*, that is for every s and i, there is some t such that $(s, i, t) \in T$.

A *run* corresponding to a finite input sequence $x \in \Sigma^*$ is a sequence $\sigma \in S^*$ starting at s_0 such that for every $i < l_x$, it is the case that $([\sigma]_i, [x]_i, [\sigma]_{i+1}) \in T$; the notion of a run extends naturally to the case when x is an ω-sequence.

One can represent a finite state automaton using a graph, as shown in Fig. 4. Vertices correspond to states, the edge (s, t) is labeled with all symbols a such that (s, a, t) is an element of the transition relation.

It is useful to classify finite state automata as being *deterministic* and *nondeterministic*. An automaton is deterministic if for all states s and for all inputs i there is exactly one state t such that $(s, i, t) \in T$; otherwise it is nondeterministic. The automata A_1 and A_2 in Fig. 4 are deterministic; A_3 is nondeterministic. Note that nondeterminism may lead to multiple runs starting at the initial state for a particular input word.

Now we describe how a finite state automaton, together with an "acceptance conditions" can be used to specify languages. This will be done for both $*$-languages and ω-languages.

1) $*$-automata:

Definition 5: A *$*$-automaton* M is a tuple $(\mathcal{A}, F)$, where $\mathcal{A} = (\Sigma, S, s, T)$ is a finite state automaton, and $F \subseteq S$ is the set of *accepting states*.

The $*$-language accepted by the $*$-automaton is the set of all sequences x in Σ^* such that there is a corresponding run σ starting at s for which $[\sigma]_{|\sigma|-1} \in F$, i.e., the last state in σ is an accepting state.

As an example, for the $*$-automaton $M_1 = (\mathcal{A}_1, \{s_1, s_2\})$, where $\mathcal{A}_1$ is as given in Fig. 4, the $*$-language accepted by M_1 is the set of all sequences in which a (0,0) never appears at any point after (1,1).

2) Properties of $*$-automata: It is easy to test whether the language accepted by a $*$-automaton is nonempty—use depth first search to see if there is an accepting state which is reachable from the initial state.

Given languages L_1 and L_2 accepted by $*$-automata $M_1 = ((\Sigma, S_1, q, T_1), F_1)$ and $M_2 = ((\Sigma, S_2, r, T_2), F_2)$, it is readily seen that there exist $*$-automata accepting the languages $L_1 \cap L_2$ and $L_1 \cup L_2$. The proof is by exhibiting the required $*$-automata by the *product construction*: the automaton for $L_1 \cup L_2$ is simply $M = ((\Sigma, S_1 \times S_2, (q, r), T), F_1 \times S_2 \cup S_1 \times F_2)$, where $T = \{((s_1, s_2), i, (t_1, t_2)) \,|\, (s_1, i, t_1) \in T_1$ and $(s_2, i, t_2) \in T_2\}$. A similar construction works for $L_1 \cap L_2$.

Given any nondeterministic $*$-automaton $\mathcal{N} = ((\Sigma, S_{\mathcal{N}}, s, T), F)$, there exists a deterministic automaton $\mathcal{D}$ which accepts exactly the same language. The proof proceeds by the subset construction [13], which build a deterministic $*$-automaton on state-space $2^{S_{\mathcal{N}}}$, which accepts the same language. The complement of a language accepted by a $*$-automaton is also accepted by a $*$-automaton. This follows from the fact that for any $*$-automaton, there exists an equivalent deterministic $*$-automaton; complementation of deterministic $*$-automata is trivial.

For a language L over $\Sigma_1 \times \Sigma_2$ accepted by a $*$-automaton M, the *projection* of L to Σ_1 consists of all sequences $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ for which there exists a sequence $\langle b_0, b_1, \ldots, b_{n-1} \rangle$ such that the sequence $\langle (a_0, b_0), (a_1, b_1), \ldots, (a_{n-1}, b_{n-1}) \rangle$ is a member of L. The projected language is also accepted by a $*$-automaton: there is a trivial construction to derive the accepting automaton from M—replace each transition label (a, b) in M by a. We refer to the resulting automaton as the projection of M to Σ_1; note that projection can result in a nondeterministic automaton, even when the original automaton was deterministic.

3) ω-automata: Informally, an ω-automaton differs from a $*$-automaton in that it operates on infinite rather than finite sequences. Unlike $*$-automata, ω-automata come in various forms. We will concentrate on Büchi automata; details on the other brands of ω-automata can be gleaned from the survey article of Thomas [31].

Definition 6: A Büchi automaton is a tuple $(\mathcal{A}, F)$, where $\mathcal{A} = (\Sigma, S, s, T)$ is a finite state automaton, and $F \subseteq S$ is the set of *Büchi states*.

The ω-language accepted by the Büchi automaton is the set of all sequences x in Σ^ω such that there is a corresponding run σ starting at s for which $\texttt{inf}(\sigma) \cap F \neq \emptyset$, i.e., there are accepting states which occur infinitely often in σ.

For example, the Büchi language accepted by $(\mathcal{A}_2, \{t_3\})$, where $\mathcal{A}_2$ is as in Fig. 4, is the set of all sequences in which a 1 occurs infinitely often at multiples of 3.

Properties of Büchi automata: It is easy to test whether the language accepted by a Büchi-automaton is nonempty—check for the existence of an accepting state which lies on a loop and is reachable from the initial state.

Given languages L_1 and L_2 accepted by Büchi automata, there exists Büchi automata accepting the languages $L_1 \cap L_2$ and $L_1 \cup L_2$; a similar (albeit marginally more complex) construction to that for $*$-automata can be applied.

The complement of a language accepted by a Büchi automaton is also accepted by a Büchi automaton, although the proof of this fact is nontrivial. An early proof [4] proceeds by taking the (possibly nondeterministic) defining Büchi automaton and creating a deterministic finite state automaton with a "Muller" acceptance condition [31], which accepts the same language; the need for a Muller acceptance condition stems from the fact that deterministic Büchi automata are strictly less expressive than nondeterministic Büchi automata. Following this, complementation is relatively straightforward. The determinization step, while similar in spirit to the subset construction for $*$-automata [13], is extremely complex. The best known procedure [27] starts with a nondeterministic Büchi automaton on n states, and yields a Büchi automaton with $2^{O(n \cdot \log(n))}$ states in the worst case.

For an ω-language L over $\Sigma_1 \times \Sigma_2$ accepted by a Büchi automaton, L projected down to Σ_1 is also accepted by a Büchi automaton; the construction is the same as for $*$-automata.

B. S1S

S1S is a logical system concerned with "second order" properties of the set of natural numbers with the successor function; the term "second order" refers to the fact that the logic refers to both subsets as well as individual natural numbers. It was studied in detail by Büchi in [2]; in particular it was shown to be decidable. S1S provides an extremely powerful mechanism for analyzing and manipulating sequential systems—the expressiveness of logic (conjunction, negation, and quantification) is available to define sets of sequences.

Definition 7: S1S formulas are finite sequences over the following set:

$$\Sigma_{\text{S1S}} = \{(,), 0, S, =, <, \in, \wedge, \neg, \exists, x_1, x_2, \ldots, X_1, X_2, \ldots\}$$

The lower case variables $x_1, x_2, \ldots$ are first order variables ranging over elements of the natural numbers, and the upper case variables $X_1, X_2, \ldots$ are second order variables ranging over subsets of the natural numbers.

We are now ready to describe the syntax yielding the *terms* and the *well formed formulas* of S1S logic. In the interests of readability, we will abuse notation, e.g., we will refer to the formula $\langle S, S, 0\rangle$ as $SS0$.

- **Terms:** $0|x_i|St$, where t is a term.
  Examples: 0, $SS0$, $SSSSx_3$.
- **Well formed formulas:** $t = u|t < u|t \in X_k|(\neg\phi)|(\phi \wedge \psi)|(\exists x_k)\phi|(\exists X_k)\phi$, where t and u are terms, and ϕ and ψ are well-formed formulas.
  Examples: $0 < S0$, $x_3 = Sx_5$, $Sx_7 \in X_2$, $(0 < S0) \wedge (Sx_7 \in X_2)$, $(\exists X . \exists x)\,((x \in X) \wedge (Sx \in X))$.

A variable occurs *freely* in a formula, if it appears in the formula, and is not quantified [9]. We write $\phi(X_1, X_2, \ldots, X_n)$ to indicate that at most $X_1, X_2, \ldots, X_n$ occur freely in ϕ.

In the sequel, we will refer to well formed formulas simply as formulas. We will routinely use the symbols $\vee, \leftrightarrow, \rightarrow, \forall$, etc., as logical abbreviations, and drop the use of parentheses unless needed to avoid ambiguity.

We now consider the semantics of S1S. An S1S formula can be interpreted over the structure consisting of the set of natural numbers, where the successor symbol S is interpreted as the function $f(x) = x + 1$. In this way, a formula $\theta(X_1)$ in S1S *defines* a set of subsets of ω, i.e., a subset of 2^ω. The defined set contains all $\beta \subset \omega$ such that the formula $\theta(X_1)$ is true when X_1 is assigned to be β. More generally, formulas $\phi(X_1, X_2, \ldots, X_n)$ define subsets of $(2^\omega)^n$; we will denote this set by $[\![\phi(X_1, X_2, \ldots, X_n)]\!]$. Formal semantics of S1S can be found in [31]; below, we illustrate the interpretation of formulas by means of examples.

Example 1: Nonempty subsets of ω contain minimal elements

$$\psi = (\forall X)\,((\exists x)(x \in X) \rightarrow (\exists y)((y \in X) \wedge \neg((\exists z)(z \in X \wedge (z < y))))).$$

Example 2: The set of subsets of ω which contain five whenever they contain three

$$\phi_0(X) = (SSS0 \in X) \rightarrow (SSSSS0 \in X).$$

Example 3: The set containing the set of even integers

$$\phi_1(X) = (0 \in X) \wedge \neg(S0 \in X) \wedge (\forall x) \cdot (x \in X \leftrightarrow SSx \in X).$$

Example 4: The binary relation on $2^\omega \times 2^\omega$ defined by $\{(X, Y)$: every even number in X is in $Y\}$

$$\phi_2(X, Y) = (\forall x)((\exists Z)(\phi_1(Z) \wedge x \in Z) \rightarrow (x \in X \rightarrow x \in Y)).$$

The set of ω-sequences on $\{0, 1\}$ is in a natural one-to-one correspondence with the set of subsets of ω; for example the sequence $010\,101\cdots$ corresponds to the subset $\{1, 3, 5, \ldots\}$. In this way, an S1S formula $\theta(X_1)$ *defines* an ω-language over the alphabet $\{0, 1\}$.

The following result relates S1S formulas to ω-automata.

Theorem 3.1 (Büchi 1961): An ω-language on $\{0, 1\}^n$ is definable in S1S if and only if it is accepted by some Büchi automaton on alphabet $\{0, 1\}^n$.

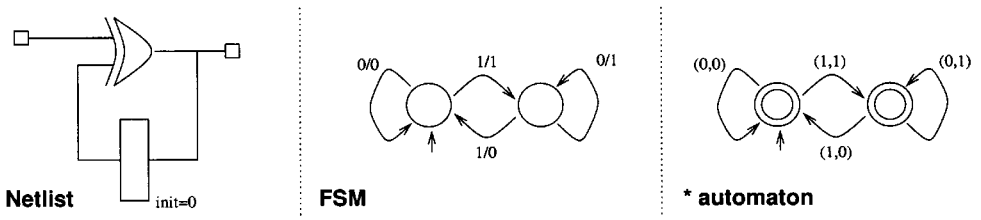


Fig. 5. Relating hardware to Büchi automata.

The right-to-left direction of the theorem follows from a straightforward construction of a formula coding up the transition structure of the automaton.

The left-to-right direction of the theorem is by induction on the length of the S1S formula. Automata for the atomic formulas are easily derived; an inductive construction is used for $\neg$, $\wedge$, and $\exists$. The case of $\exists$ is handled by automaton projection, $\wedge$ by automaton intersection, and $\neg$ by automaton complementation, as discussed in Section III-A2.

1) WS1S: With minor modifications, the formal treatment of ω-languages done in S1S can be applied to $*$-languages. In this case the resulting logic is referred to as weak S1S (WS1S), the weak referring to the fact that set variables range over finite subsets of ω. In a manner analogous to Theorem 3.1, it can be shown that a $*$-language is accepted by a $*$-automaton if and only if it is definable by a formula in WS1S.

Given the relative ease with which $*$-automata can be complemented, it is not surprising that the proof of this fact is much easier than that of Büchi's theorem; in fact it predates Büchi's result [7].

C. Netlists, FSMs, Languages, and Compositional Designs

We now make precise the relationship between designs and languages accepted by automata.

Recall that in Section II we defined formal models for hardware; these consisted of FSMs and netlists. We made the point there that given a netlist, we could derive a FSM from it. An FSM $M = (Q, s, I, O, \lambda, \delta)$ bears a natural correspondence to a $*$-automaton $B = ((Q, s, (I \times O), T), Q)$ where $(u, (i, o), v) \in T$ precisely when $\delta(u, i) = v$ and $\lambda(u, i) = o$. An example of this correspondence is shown in Fig. 5.

Observe the language of this $*$-automaton characterizes the input–output behavior of M; given any finite input sequence $\mathbf{i}$, we can construct the output sequence $\mathbf{o}$ that M would have produced on application of $\mathbf{i}$ by examining the $*$-automaton. By Theorem 3.1, it follows that we can also characterize a netlist by a formula of WS1S.

As described in Section II-D, designs are built compositionally. For designs specified as netlists, composition is specified by simply placing the two netlists next to each other and making the connections required by the composition. Inputs and outputs which are not hidden by this composition become the inputs and outputs of the composed design.

We illustrate the relationship between the WS1S formula for the $*$-language of the composed design and the WS1S formulas for the components by considering the netlist composition illustrated in Fig. 2. Let $\phi_1(X_1, L_2)$ and $\phi_2(U_1, U_2, V_1)$ be the WS1S formula defining the $*$-language of D_1 and D_2. Then the

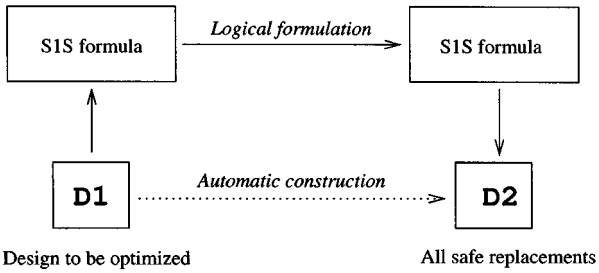


Fig. 6. A paradigm for sequential synthesis.

$*$-language of the composed design is defined by the WS1S formula $\phi^{D_1 \times D_2}$ given below

$$\phi^{D_1 \times D_2}(U_1, V_1) = (\exists X_1.\exists L_2 \cdot \exists U_2)(\phi_1(X_1, L_2) \wedge \phi_2(U_1, U_2, V_1) \wedge (L_2 = U_2) \wedge (X_1 = V_1)).$$

D. Applications to Synthesis

Fig. 6 illustrates the approach we will be using. Given design, we will first identify a formula for it; this formula will be in WS1S or S1S, depending on the context. We will cast and solve the problem of characterizing permissible solutions in logic; essentially this amounts to writing down a system of logical constraints. This takes the form of a formula which can then be reflected back to an automaton.

In practice, it is not necessary to actually build any formulas—we can mimic the steps taken in the construction of an automaton from a formula to derive the automaton for the synthesized design directly. This corresponds to taking the dotted line in Fig. 6. The advantage of S1S is that it is much easier to come up with the characterizations. Additionally, elegant yet rigorous proofs can be given; furthermore, these proofs are constructive. Furthermore, as we will see in Section V, the approach generalizes immediately to nondeterministic designs, possibly with fairness constraints.

IV. Synthesizing Compositional Designs

As mentioned in Section I, a critical first step toward synthesizing a component in a design is characterizing the set of all valid implementations for that component. There is an obvious "operational" characterization: given a candidate implementation, plug it in, and test if there is no change in the input-output behavior observed from the external world, i.e., if the language of the composed design remains unchanged. Since equivalence of automata is decidable, this check is effective.

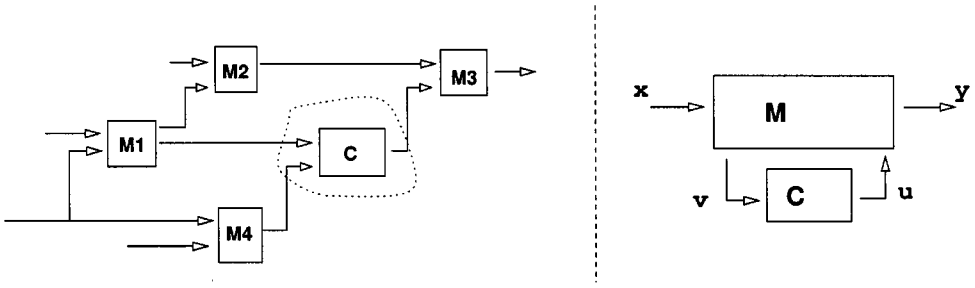


Fig. 7. A feedforward network.

This characterization is correct, since if the condition holds, there is no way the change can be determined by looking at the external inputs and outputs. Conversely, if there were some input on which the composed design had an output differing from that in the original design, there is a surrounding environment which could observe the change and as a result function incorrectly. Following the parlance of Singhal [30], we will refer to implementations satisfying this condition as being "safe replacements" for the component.

However, this characterization is not well suited for synthesis; we want a finite structure, on which some kind of algorithmic search for simple solutions can be performed. In this section, we will show that the flexibility available for sequential synthesis can be characterized using a $*$-automaton.

This result was previously shown by Watanabe and Brayton [35], who referred to this automaton as the *E-machine*, the "E" standing for environment. Their approach was based on examining the design on a state-by-state basis; we derive this result using S1S. We also derive an approximation to the set of valid implementations on which it is easier to perform optimization, and adapt the E-machine construction to a number of interconnect schemes.

We can gain some intuition as to the source of the flexibility available for optimization by considering a component C in the design. Observe that nature of the surrounding components may make it impossible for certain sequences to be input to C. Similarly, there may be input sequences for which the output from C does not affect the external outputs. Knowledge of these facts may make it possible to simplify C, while preserving the overall input-to-output behavior.

A. Feedforward Designs

In order to illustrate the principles and arguments we will be using, we start with the simple case of computing the set of permissible behaviors for feedforward networks. A feedforward network corresponds to a composition of a set of component netlists in such a way that there is no path in the composed netlist from an output of a component netlist to one of its inputs which passes only through vertices from other netlists. An example of such a netlist is given in Fig. 7.

In a feedforward network, it is possible to compose the environment around C to form a single netlist M, and have C connected to M as shown on the right of Fig. 7. The external inputs and outputs of this design are x and y. Here, v is an output of M and an input to C; similarly, u is an output from C and an input to M. Note that the variables x, y, u, and v may correspond to vectors of inputs.

Let the $*$-language of M be defined by the WS1S formula $\phi^M(X,U,V,Y)$, and the $*$-language of C be defined by the WS1S formula $\phi^C(V,U)$. Then the $*$-language of the composed design is defined by the formula $(\exists U.\exists V)\,(\phi^M(X,U,V,Y) \wedge \phi^C(V,U))$; denote this formula by $\phi^S(X,Y)$.

We now characterize all possible netlists those which can safely replace C without changing the input-to-output behavior of the overall design.

Theorem 4.1: Let $\tilde{C}$ be a netlist. Then $\tilde{C}$ is a safe replacement for C if and only if the language defined by $\phi^{\tilde{C}}(V,U)$ is included in the language defined by the formula

$$\phi^{C_{\max}}(V,U) = (\forall X.\forall Y)(\phi^M(X,U,V,Y) \rightarrow \phi^S(X,Y)).$$

Proof: Suppose $[\![\phi^{\tilde{C}}(V,U)]\!] \subseteq [\![\phi^{C_{\max}}(V,U)]\!]$. Let γ be an arbitrary finite sequence of inputs applied to the composition of M and $\tilde{C}$.

Since the composed design is a feedforward network, v is purely a function of x and the design M. Let α be the result at v of applying γ at x. The output seen at u is purely a function of the input at v and the design C; let β be the output at u corresponding to α. This fixes the output seen at y to some δ, since y is a function of the sequences u and x and the design M.

Observe that $(\gamma,\beta,\alpha,\delta) \in [\![\phi^M(X,U,V,Y)]\!]$; furthermore, $(\alpha,\beta) \in [\![\phi^{\tilde{C}}(V,U)]\!]$, which in turn is contained in $[\![\phi^{C_{\max}}(V,U)]\!]$. Hence, $(\gamma,\delta) \in [\![\phi^S(X,Y)]\!]$, i.e., the output of the composition of M with $\tilde{C}$ on input γ is the same as the output of the composition of M with C on input γ. But γ was chosen arbitrarily, and so $\tilde{C}$ is a safe replacement for C.

Conversely, suppose $[\![\phi^{\tilde{C}}(V,U)]\!] \not\subseteq [\![\phi^{C_{\max}}(V,U)]\!]$. Take $(\alpha,\beta) \in [\![\phi^{\tilde{C}}(V,U)]\!] - [\![\phi^{C_{\max}}(V,U)]\!]$; thus, (α,β) is an element of the complement of $[\![\phi^{C_{\max}}(V,U)]\!]$, i.e., (β,α) is an element of $[\![(\exists X.\exists Y)\,(\phi^M(X,U,V,Y) \wedge \neg\phi^S(X,Y))]\!]$. Hence, there exists an ordered pair (γ,δ) such that $(\gamma,\beta,\alpha,\delta) \in [\![\phi^M(X,U,V,Y)]\!]$ and $(\gamma,\delta) \notin [\![\phi^S(X,Y)]\!]$.

Since the composed design is a feedforward network, v is purely a function of x and M. Thus, on application of γ to the composition of M and $\tilde{C}$, the sequence seen at v will again be α. Since u is purely a function of v and the design $\tilde{C}$, applying α to $\tilde{C}$ will produce β at the output. This in turn uniquely determines the output seen at y to be δ.

However, $(\gamma,\delta) \in [\![\neg\phi^S(X,Y)]\!]$; thus, δ was not the output of M composed with C when γ was the input. Hence, $\tilde{C}$ is not a safe replacement for C. ■

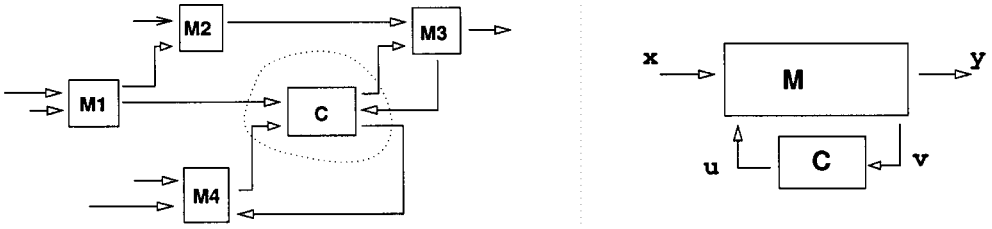


Fig. 8. A feedback network.

Thus, the formula $\phi^{C_{\max}}(V,U)$ completely characterizes the set of implementations that can replace the component C. We will see in Section IV-C how to construct a $*$-automaton that accepts $[\![\phi^{C_{\max}}(V,U)]\!]$; this is a finite representation which is suitable for constructing an optimal implementation.

B. Feedback Designs

We now consider the case of a general compositional design, as illustrated in Fig. 8. Given a component C we can coalesce its environment into a single netlist M in the topology shown on the right of Fig. 8. The external input is x and the external output is y; v is an output of M and an input to C; similarly, u is an output from C and an input to M. Again, we want to characterize all netlists which can replace the component without changing the input-to-output behavior of the overall design.

Case 1—M is Moore: In the presence of feedback, there exists the possibility of a combinational cycle resulting on composition. In order to avoid this possibility, we will first consider the case where M is a Moore netlist. (Actually, we only need there to be no combinational path from v to u.)

Let the $*$-language of M be defined by the formula $\phi^M(X,U,V,Y)$, and the $*$-language of C be defined by the formula $\phi^C(V,U)$. Then the $*$-language of the composed design is defined by the formula $(\exists U.\exists V)(\phi^M(X,U,V,Y) \wedge \phi^C(V,U))$; denote this formula by $\phi^S(X,Y)$.

We now characterize all netlists which are safe replacements for C.

Theorem 4.2: Let $\tilde{C}$ be a netlist. Then $\tilde{C}$ is a safe replacement for C if and only if the language defined by $\phi^{\tilde{C}}(V,U)$ is contained in the language defined by the formula

$$\phi^{C_{\max}}(V,U) = (\forall X.\forall Y)(\phi^M(X,U,V,Y) \rightarrow \phi^S(X,Y)). \tag{1}$$

Proof: Suppose $[\![\phi^{\tilde{C}}(V,U)]\!] \subseteq [\![\phi^{C_{\max}}(V,U)]\!]$. Let γ be an arbitrary finite sequence of inputs applied to the composition of M with $\tilde{C}$. Note that the nets v, u, and y are functions of x in the netlist consisting of M composed with $\tilde{C}$. Let α, β, and δ be the result at v, u and y respectively on applying γ at x;

Observe that $(\gamma,\beta,\alpha,\delta)$ is an element of $[\![\phi^M(X,U,V,Y)]\!]$; furthermore, $(\alpha,\beta) \in [\![\phi^{\tilde{C}}(V,U)]\!]$. By hypothesis $[\![\phi^{\tilde{C}}(V,U)]\!]$ is included in $[\![\phi^{C_{\max}}(V,U)]\!]$. Hence, $(\gamma,\delta) \in [\![\phi^S(X,Y)]\!]$, i.e., the output of the composition of M with $\tilde{C}$ on input γ is the same as the output of the composition of M with C on input γ. But γ was chosen arbitrarily, and so $\tilde{C}$ is a safe replacement for C.

Conversely, suppose $[\![\phi^{\tilde{C}}(V,U)]\!] \not\subseteq [\![\phi^{C_{\max}}(V,U)]\!]$. Let (α,β) be an element of $[\![\phi^{\tilde{C}}(V,U)]\!]$ which is not in $[\![\phi^{C_{\max}}(V,U)]\!]$; thus, $(\beta,\alpha) \in [\![(\exists X.\exists Y)\ [\phi^M(X,U,V,Y) \wedge \neg\phi^S(X,Y)]\!]$. Hence, there exists an ordered pair (γ,δ) such that $(\gamma,\beta,\alpha,\delta) \in [\![\phi^M(X,U,V,Y)]\!]$ and $(\gamma,\delta) \notin [\![\phi^S(X,Y)]\!]$.

It now suffices to show that applying γ to the composition of M and $\tilde{C}$ will result in δ as an output. Let $\hat{\alpha}$ and $\hat{\beta}$ be the outputs produced at v and u on applying γ. We now prove that $\hat{\alpha}$ and $\hat{\beta}$ are equal to α and β, respectively. We do this by using the fact that the output of a Moore netlist at step k is uniquely determined by its inputs at steps $0,1,\ldots,k-1$ to inductively show that for all k we have $[\hat{\alpha}]_k = [\alpha]_k$ and $[\hat{\beta}]_k = [\beta]_k$.

The base case is direct—the initial output of M at v is uniquely determined by the initial state since M is a Moore netlist, so $[\hat{\alpha}]_0 = [\alpha]_0$. The initial output of $\tilde{C}$ is purely a function of the initial state and the input at v, and so $[\hat{\beta}]_0 = [\beta]_0$.

Now for the induction step, consider $[\hat{\alpha}]_{k+1}$; it is uniquely determined by the values of $[\gamma]_0, [\gamma]_1, \ldots, [\gamma]_k$ and $[\hat{\beta}]_0, [\hat{\beta}]_1, \ldots, [\hat{\beta}]_k$. But by the induction hypothesis, $[\hat{\beta}]_i = \beta_i$ for all $i \leq k$. This determines $[\hat{\alpha}]_{k+1} = \alpha_{k+1}$. Since $[\hat{\beta}]_{k+1}$ is a function of $[\hat{\alpha}]_0, [\hat{\alpha}]_1, \ldots, [\hat{\alpha}]_{k+1}$, it follows that $[\hat{\beta}]_{k+1} = [\beta]_{k+1}$. Hence, the induction step goes through.

The output $\hat{\delta}$ at y is uniquely determined by γ and $\hat{\beta}$; since $\hat{\beta} = \beta$, it follows that $\hat{\delta} = \delta$. But $(\gamma,\delta) \notin [\![\phi^S(X,Y)]\!]$; thus, $\tilde{C}$ is not a safe replacement for C. ■

Case 2—General M: Now we consider the case when M is not a Moore netlist. Observe that if we pick a $\tilde{C}$ which is Moore, then its composition with M will still be guaranteed to have no combinational cycles. In order to characterize the Moore netlists which can replace C, we need the concept of a Moore language.

Definition 8: Let $L \subseteq (\Sigma_I \times \Sigma_O)^*$ be a $*$-language. The language L is a Moore language if whenever we have $\langle (i_0,o_0),(i_1,o_1),\ldots,(i_{n-1},o_{n-1})\rangle \in L$, then for any $\alpha \in \Sigma_O$, we have $\langle (i_0,o_0),(i_1,o_1),\ldots,(\alpha,o_{n-1})\rangle \in L$.

Intuitively, a Moore language is a language with the property that for any string x in the language, the second component of the last symbol in x is independent of the first component. The $*$-language corresponding to the input–output behavior of a Moore netlist is a Moore language, since the output at time k does not depend on the input at time k.

The following proposition is a consequence of the fact that the set of Moore languages is closed under union.

Proposition 4.3: Given an arbitrary $*$-language L defined over $\Sigma_I \times \Sigma_O$, there exists a unique maximal Moore language L_{Moore} contained in it.

We will refer to L_{Moore} as the *Moore restriction* of L.

We are now ready to characterize the set of Moore netlists which can safely replace C; unlike the previous case, this argument does not require that M be Moore.

```
function Moore_States(DSA D: (S_D, s_0, Σ_V × Σ_U, T_D, A_D) ) {
  S_C = A_D;
  while ( TRUE ) {
      remove states s from S_C and T_C such that
          (∃u) ((∃v ∃t) ((s,(v,u),t) ∈ T_C) ∧ ¬(∀v' ∃t') ((s,(v',u),t') ∈ T_C))
      if (no states were removed)
          break;
  }
  return S_C;
}
```

Fig. 9. Computing the Moore restriction for a language accepted by a $*$-automaton.

Theorem 4.4: Let $\tilde{C}$ be a Moore netlist. Then $\tilde{C}$ is a safe replacement for C if and only if the language defined by $\phi^{\tilde{C}}(V,U)$ is contained in the Moore restriction of the language defined by the formula

$$\phi^{C_{\max}}(V,U) = (\forall X.\forall Y)(\phi^M(X,U,V,Y) \rightarrow \phi^S(X,Y)).$$

Proof: The first stage of the proof, namely demonstrating that $\tilde{C}$ can be safely substituted for C when its language is included in the Moore restriction of $[\![\phi^{C_{\max}}(V,U)]\!]$ is identical to that for Theorem 4.2.

Now suppose $[\![\phi^{\tilde{C}}(V,U)]\!]$ is not contained in the Moore restriction of $[\![\phi^{C_{\max}}(V,U)]\!]$. Observe that $[\![\phi^{\tilde{C}}(V,U)]\!]$ is a Moore language (by hypothesis $\tilde{C}$ is a Moore netlist); by Proposition 4.3, this implies that it is not contained in $[\![\phi^{C_{\max}}(V,U)]\!]$. The rest of the proof can be completed as in Theorem 4.2. ■

Let A be a $*$-automaton on alphabet $\Sigma_I \times \Sigma_O$, accepting the language L. Using the subset construction, one can construct from A a deterministic $*$-automaton D accepting L. Given D, it is straightforward to construct a deterministic automaton D' for the Moore restriction of L: recursively remove from D edges $(s,(i,o),t)$ whenever for some i' applying (i',o) to s leads to a nonaccepting state t'. An algorithm which returns exactly the set of states in the DFA is given in Fig. 9.

C. Constructing an Automaton Accepting $[\![(\forall X.\forall Y)(\phi^M(X,U,V,Y) \rightarrow \phi^S(X,Y))]\!]$

In Section IV-B, we saw the set of replacements for a component C in a compositional design is characterized by a formula of the form $(\forall X.\forall Y)(\phi^M(X,U,V,Y) \rightarrow \phi^S(X,Y))$. This formula can be rewritten as follows: $\neg(\exists X.\exists Y)(\phi^M(X,U,V,Y) \wedge \neg\phi^S(X,Y))$. This formula suggests the following four-step construction for constructing an automaton $\mathcal{A}_{C_{\max}}$ accepting $[\![(\forall X.\forall Y)(\phi^M(X,U,V,Y) \rightarrow \phi^S(X,Y))]\!]$.

- Step 1) Complement the automaton $\mathcal{A}_S$ accepting $[\![\phi^S(X,Y)]\!]$ to obtain an automaton $\mathcal{A}_{\overline{S}}$ which accepts $[\![\neg\phi^S(X,Y)]\!]$.
- Step 2) Form an automaton $\mathcal{A}_P$ for the intersection of $\mathcal{A}_M$ and $\mathcal{A}_{\overline{S}}$.
- Step 3) Project out the inputs x and y from $\mathcal{A}_P$ to obtain an automaton $\mathcal{A}_{\tilde{P}}$ accepting $[\![(\exists X.\exists Y)(\phi^M(X,U,V,Y) \wedge \neg\phi^S(X,Y))]\!]$.
- Step 4) Complement $\mathcal{A}_{\tilde{P}}$ to obtain an automaton $\mathcal{A}_{C_{\max}}$ for $\neg(\exists X.\exists Y)(\phi^M(X,U,V,Y) \wedge \neg\phi^S(X,Y))$.

We illustrate the construction for $\mathcal{A}_{C_{\max}}$ by means of an example. Consider the design specified in Fig. 10. In order to optimize the component C (shown with a dotted outline), we first characterize all safe replacements for C. The construction for each step is shown in Fig. 11; by inspection, we can see that C can be replaced by an inverter.

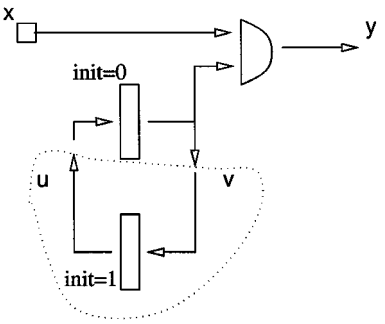


Fig. 10. Design to be optimized.

1) Complexity issues: It is straightforward to build a $*$-automata corresponding to M, C, and S (cf. Section III-C). Since the automaton for S is deterministic, an automaton for its complement, constructed in Step 1, is trivially obtained; it has $|S_M| \cdot |S_C|$ states. The product automaton in Step 2 has a state-space whose cardinality is the size of the product of the state-spaces of the automata for M, and S. The projection of the signals x and y in Step 3 is also easy to achieve.

The complexity comes in the complementation performed in Step 4. Even though the product automaton resulting in Step 2 is deterministic, the projection of Step 3 makes it nondeterministic. The complementation in Step 4 is performed by first determinizing the nondeterministic automaton, which, in the worst case, can lead to an automaton on $2^{|S_M| \cdot |S_S|} = 2^{|S_M|^2 \cdot |S_C|}$ states.

By virtue of Theorems 4.1, 4.2, and 4.4, the automaton capturing the entire set of replacements for a component C interacting with an environment M accepts exactly the language defined by an S1S formula of the form $(\forall X.\forall Y)(\phi^M(X,U,V,Y) \rightarrow \phi^S(X,Y))$. It follows that if we want to capture all the flexibility available for optimizing C by an automaton, then the automaton is obliged to accept $[\![(\forall X.\forall Y)(\phi^M(X,U,V,Y) \rightarrow \phi^S(X,Y))]\!]$, and it may be very large.

We complemented the automaton $\mathcal{A}_{\tilde{P}}$ by first determinizing it. It may be the case that the final automaton, $\mathcal{A}_{C_{\max}}$, after merging equivalent states, is much smaller than the determinization of $\mathcal{A}_{\tilde{P}}$, i.e., generating the complement by determinizing as a first step leads to an intermediate blow-up. However, complementing a nondeterministic finite automaton is inherently computationally expensive. This is due to the fact that the problem of deciding if a nondeterministic finite automaton $\mathcal{N}$ is *universal*, i.e., accepts all sequences, is PSPACE-complete [11]. Once an automaton $\mathcal{M}$ (deterministic or nondeterministic) accepting the complement of $\mathcal{N}$ is constructed, checking the emptiness of $\mathcal{M}$ is trivial; hence, performing complementation is as difficult as checking universality.

Watanabe and Brayton [35] have successfully constructed the automaton $\mathcal{A}_{C_{\max}}$ accepting $[\![\phi^{C_{\max}}(V,U)]\!]$ on some examples. However, the designs they used were synthetic—they consisted of randomly composed MCNC benchmarks. Fur-

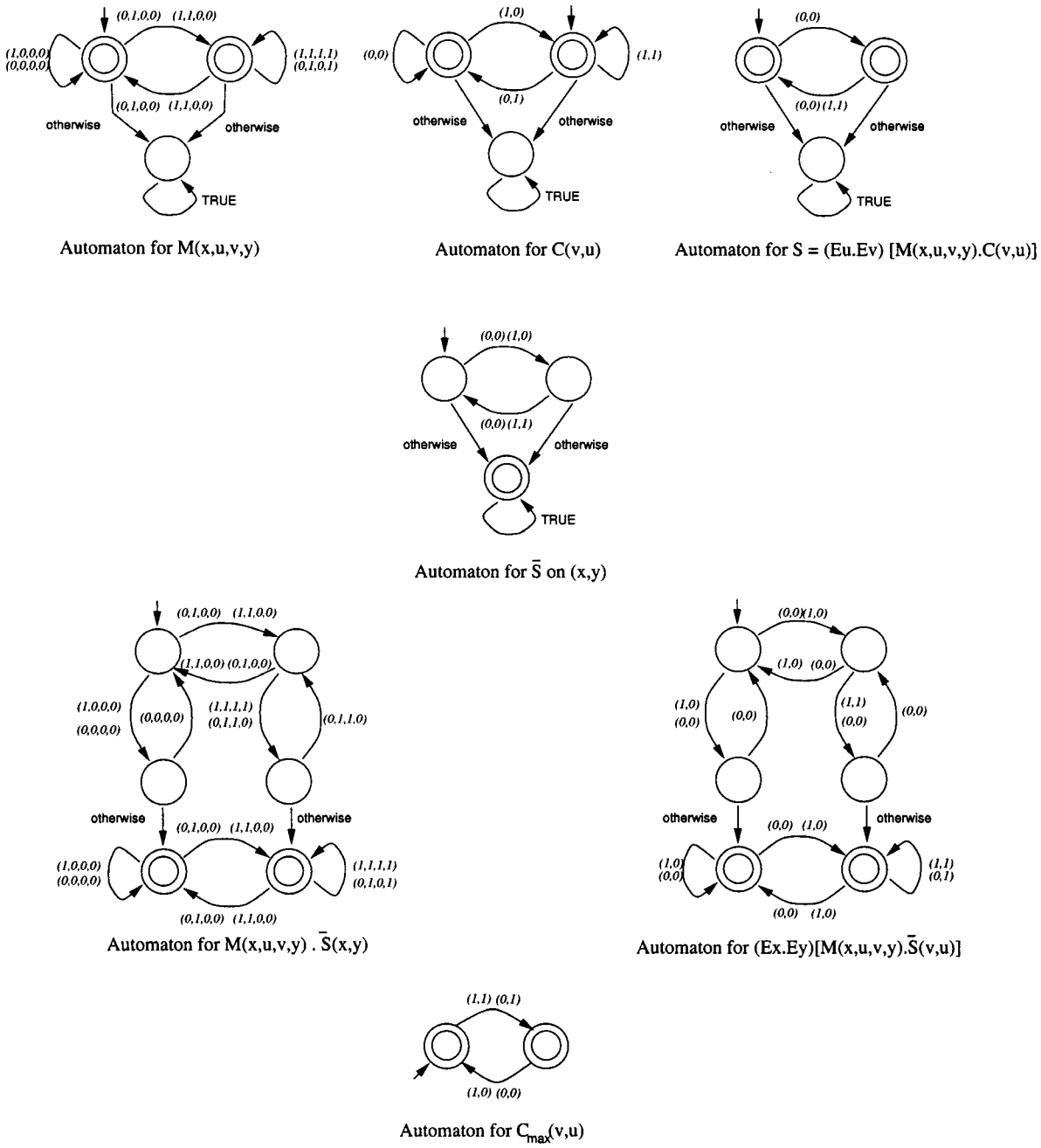


Fig. 11. Constructing the E-machine—circled states are accepting.

thermore, they were small—the component to be synthesized contained at most 18 states, and the entire design contained at most 336 states. Their results suggest that the final size of $\mathcal{A}_{C_{\max}}$ is much smaller than the upper bound we derived above. The run times they report vary by orders of magnitude, and can be very large. In view of the fact that their experiments were performed on small and synthetic examples, a definitive statement about the average case time and space complexity of constructing $\mathcal{A}_{C_{\max}}$ cannot be made at this time.

One reason for the high complexity of constructing the $\mathcal{A}_{C_{\max}}$ is the fact that we chose language containment as our criterion for conformance %(cf. the remarks in Section I). Testing language containment for nondeterministic finite state automata is PSPACE complete; we could have used a stronger notion for conformance, e.g., *simulation* [32] which can be tested in polynomial time. If we did so, the development of the E-machine would be quite different. Such an approach could reduce complexity, at the cost of completeness.

D. Optimization from Automata Specifications

Once a $*$-automaton characterizing the set of safe replacements possible for a component is available, the next step is to find an optimal replacement. There are many criteria for optimality such as area, timing, power consumption, etc. One starting point is determining a replacement whose underlying FSM is minimum state.

Not surprisingly, this is closely related to the problem of minimizing an *incompletely specified finite state machine* (ICFSM)

[12]. However, there is a subtle distinction: for an incompletely specified FSM, at a given state, for a specific input, either the next-state and output is fixed, or any output and next-state is allowed. In the context of the E-machine, at a given state, for a specific input, a subset of all possible outputs and next-states may be allowed; this is referred to as *pseudo-nondeterminism* [35]. Watanabe and Brayton [35] explain why the problem of finding a minimum state FSM compatible with a specification given as a pseudo-nondeterministic automaton is more difficult than when the specification is given as an ICFSM.

E. An Approximation to the Full Set of Safe Replacements

We now again consider optimization of a compositional design with feedback as in Fig. 8. It is of some interest to study a particular subset of the set of safe replacements for C, namely that corresponding to the *input don't care set*. This will help us better understand previous work; furthermore, we will see that this subset in certain respects is better suited for optimization.

Input don't care sequences for C are those sequences at v which can never be generated in the composition of C and M; intuitively, we are free to change the behavior of C on such sequences, leading to flexibility which can be exploited by optimization tools.

We assume M is a Moore netlist. As before, let the $*$-language of M be defined by the formula $\phi^M(X,U,V,Y)$, and the $*$-language of C be defined by the formula $\phi^C(V,U)$.

Definition 9: The *input don't care set for* C is the set defined by the formula

$$\phi^{IDC}(V) = \neg(\exists U.\exists X.\exists Y)(\phi^M(X,U,V,Y) \wedge \phi^C(V,U)).$$

This formula defines precisely the set of finite sequences which can never arise at v when M is composed with C; as a consequence, any component which seeks to replace C is free to produce any output for inputs which lie in the set defined by $\phi^{IDC}(V)$.

More formally, we have the following theorem:

Theorem 4.5: Let $\tilde{C}$ be a netlist. Then $\tilde{C}$ can be safely substituted for C if the language defined by $\phi^{\tilde{C}}(V,U)$ is contained in the $*$-language defined by the formula

$$\phi^{C_{\mathrm{IDC}}}(V,U) = \neg\phi^{IDC}(V) \rightarrow \phi^C(V,U). \tag{2}$$

Proof: Let γ be some sequence of inputs to the composition of M and $\tilde{C}$. Note that v, u, and y are uniquely determined by γ; call the resulting sequences α, β, and δ. It suffices to show that applying γ to the composition of M and C also results in α, β, and δ.

Let the result of applying γ to the composition of M and C be $\hat{\alpha}, \hat{\beta}$, and $\hat{\delta}$. The construction used in Theorem 4.2 to show that $\hat{\alpha} = \alpha$, $\hat{\beta} = \beta$, and $\hat{\delta} = \delta$ can be applied in this case also, and the result follows immediately. ∎

A closer analysis of the formula $\phi^{C_{\mathrm{IDC}}}(V,U)$ demonstrates that the corresponding automaton has, in the worst case, $|S_C| \cdot 2^{|S_M|\cdot|S_C|}$ states; contrast this with the automaton for $\phi^{C_{\max}}(V,U)$ which, as shown in Section IV-C, has $2^{|S_M|^2\cdot|S_C|}$ states in the worst case.

Furthermore, the automaton accepting $[\![\phi^{C_{\mathrm{IDC}}}(V,U)]\!]$ corresponds to an incompletely specified FSM, rather than a pseudo-nondeterministic automaton as is the case for the automaton accepting $[\![\phi^{C_{\max}}(V,U)]\!]$. This follows from the fact that the automaton accepting $[\![\phi^C(V,U)]\!]$ is completely specified and deterministic, and for $v \in [\![\phi^{IDC}(V)]\!]$, any sequence u will do. Thus, for any sequence v, either it is in $[\![\phi^{IDC}(V)]\!]$, and then any sequence of outputs is allowed (implying that the next-state and output of the FSM is not specified), or u is uniquely determined. Hence, the set $[\![\phi^{C_{\mathrm{IDC}}}(V,U)]\!]$ can be characterized by an ICFSM, which, as Watanabe and Brayton [35] show, is easier to perform optimization on than a pseudo-nondeterministic automaton.

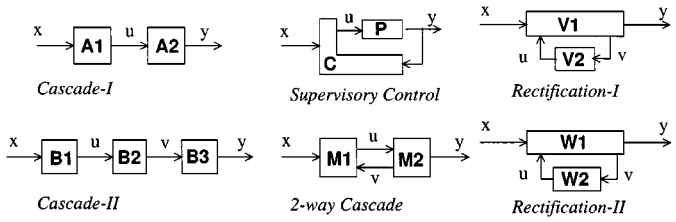


Fig. 12. A variety of FSM interconnection schemes—the names suggest applications.

Wang and Brayton [33] report results on computing an automaton accepting $[\![\phi^{C_{\mathrm{IDC}}}(V,U)]\!]$. On comparing their results with those in [35], we see that an automaton accepting $[\![\phi^{C_{\mathrm{IDC}}}(V,U)]\!]$ can be constructed far more efficiently than an automaton accepting $[\![\phi^{C_{\max}}(V,U)]\!]$; this is in concordance with the reasoning above. Again, their examples are small and synthetic, so no definitive claims can be made about the practical applicability of their approach.

F. General Topologies

One of the benefits of the S1S approach is its generality. For example, in the past different topologies (schemes for interconnecting networks) have been studied separately. Using the style of reasoning given previously, one can easily characterize safe replacements for components for the topologies in Fig. 12. In all cases, the techniques described in Section IV-B2 to avoid combinational cycles must be used.

Cascade—I(a) $\phi^{A_1^*}(X,U) = (\forall Y)(\phi^{A_2}(U,Y) \rightarrow \phi^S(X,Y))$.

Cascade—I(b) $\phi^{A_2^*}(U,Y) = (\forall X)(\phi^{A_1}(X,U) \rightarrow \phi^S(X,Y))$.

Cascade—II $\phi^{B_2^*}(U,V) = (\forall X.\forall Y)(\phi^{B_1}(X,U) \wedge \phi^{B_3}(V,Y) \rightarrow \phi^S(X,Y))$.

Supervisory Control $\phi^{C^*}(X,Y,U) = \phi^P(U,X) \rightarrow \phi^S(X,Y)$.

Bidirectional Cascade—(a) $\phi^{M_1^*}(X,V,U) = (\forall Y)(\phi^{M_2}(V,U,Y) \rightarrow \phi^S(X,Y))$.

Bidirectional Cascade (b) $\phi^{M_2^*}(U,V,Y) = (\forall X)(\phi^{M_1}(X,V,U) \rightarrow \phi^S(X,Y))$.

Rectification—I $\phi^{V_2^*}(V,U) = (\forall X.\forall Y)(\phi^{V_1}(X,V,U,Y) \rightarrow \phi^S(X,Y))$.

Rectification—II $\phi^{W_1^*}(X,V,U,Y) = \phi^{W_2}(V,U) \rightarrow \phi^S(X,Y)$.

It is worth noting that when there is no "hiding" of signals, i.e., all inputs and outputs of the components are visible in the composed design, the size of the corresponding automaton is

polynomial in the number of states in the component FSMs. This is because we begin with deterministic components, and, since no signals are hidden, there are no projected variables in the formula for the automaton (cf. Section III-A1—projection can make a deterministic automaton nondeterministic). This is the case for the Supervisory Control and Rectification-II examples.

V. Synthesizing Properties

Up to this point we have addressed the problem of optimizing components of larger designs. We now examine the problem of selecting a component so that the larger design meets user-specified properties.

The scenario is as follows: Let M be a design on primary input x, auxiliary input u, primary output y and auxiliary output v, exactly as in Fig. 8, and let S be some specification on acceptable primary input–output for M. It is natural to ask: does there exist a design $\tilde{C}$ which when composed with M results in the primary input–output behavior conforming to S?

In order to answer this question, we need to formalize the notions of specification and conformance. Let Σ_X and Σ_Y be the sets of values that x and y can take. A natural way of specifying acceptable input–output behaviors on x and y is by specifying a Büchi automaton accepting an ω-language $L^S \subseteq (\Sigma_X \times \Sigma_Y)^\omega$, i.e., for an infinite input sequence $\gamma \in \Sigma_X^\omega$, exactly those $\delta \in \Sigma_Y^\omega$ should be produced for which $(([\gamma]_0,[\delta]_0),([\gamma]_1,[\delta]_1),([\gamma]_2,[\delta]_2),\ldots) \in L^S$. Similarly, it is natural to say that the composition of the composed design *conforms to* L^S if its language is included in L^S.

It is preferable to specify the input–output behavior using ω-sequences rather than finite sequences (as we used in optimization). This is because the use of Büchi automata allows the specification of fairness. This is because whenever we want to specify a liveness property, it is invariably necessary to include a fairness constraint in the description of the system (M in this case). Use of a fairness constraint makes it possible to ignore behaviors that correspond to extreme execution scenarios which would not occur in any reasonable system.

Let the ω-language of C and M be defined by the S1S formulas $\phi^C(V,U)$ and $\phi^M(X,U,V,Y)$. Let us re-examine the expression characterizing the set of safe replacements for a component C interacting with design M

$$\phi^{C_{\max}}(V,U) = (\forall X.\forall Y)(\phi^M(X,U,V,Y) \rightarrow \phi^S(X,Y))$$

where $\phi^S(X,Y) = (\exists X.\exists Y)(\phi^M(X,U,V,Y) \wedge \phi^C(V,U))$.

We argued that any netlist $\tilde{C}$ whose language was included in $[\![\phi^{C_{\max}}(V,U)]\!]$ would be a safe replacement for C, i.e., composition of C with M would result a netlist whose language was contained in $[\![\phi^S(X,Y)]\!]$.

Now suppose $\phi^S(X,Y)$ was some arbitrary specification on the input-output behavior of M, as discussed above. Exactly the same arguments as were used in proving Theorem 4.2 can be applied to prove the following:

Theorem 5.1: The composition of netlist $\tilde{C}$ with netlist M conforms to the Büchi specification $\phi^S(X,Y)$ if and only if the language of $\tilde{C}$ is included in the language defined by the formula

$$\phi^{C_{\max}}(V,U) = (\forall X.\forall Y)(\phi^M(X,U,V,Y) \rightarrow \phi^S(X,Y)).$$

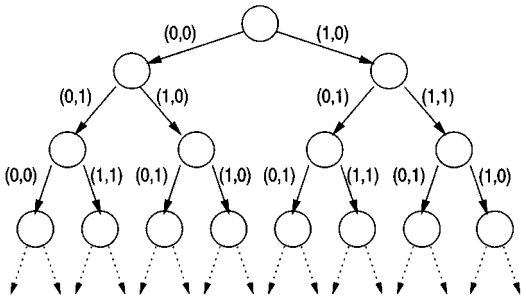


Fig. 13. Infinite tree for realizability.

Again, the caveats about introducing combinational cycles must be taken.

1) Realizability: Computing the Büchi automaton for the formula $\phi^{C_{\max}}(V,U)$ does not directly answer the existence question we posed at the beginning of this section. Our question is closely linked to the problem of *realizability*. Given an ω-language $L \subseteq (\Sigma_V \times \Sigma_U)^\omega$ accepted by a Büchi automata B, it is natural to ask if there exists a netlist whose corresponding language is contained in L.

Note that this can be trivially answered in the affirmative when dealing with the optimization problem, since it would suffice to use the original component. However, when the specification is given by an arbitrary Büchi automaton, it may be the case that there is no netlist whose language is contained in the specification. A necessary condition for the existence of a netlist is that for any $\alpha \in \Sigma_V^\omega$, there exists a $\beta \in \Sigma_U^\omega$ such that $(([\alpha]_0,[\beta]_0),([\alpha]_1,[\beta]_1),([\alpha]_2,[\beta]_2),\ldots) \in L$. However, this condition is not sufficient, because it does not guarantee *causality*: the netlist realizing L must produce $[\beta]_k$ based only on the values $([\alpha]_0,[\alpha]_1,[\alpha]_2,\ldots,[\alpha]_k)$.

Pnueli and Rosner [23] argue that a necessary and sufficient condition for a language $L \subset (\Sigma_V \times \Sigma_U)^\omega$ to be realizable by a netlist is that $|\Sigma_V|$–branching infinite tree must exist, whose edges are labeled with pairs (v,u) such that:

1) at each vertex t, for every $v \in \Sigma_V$, there is a $u \in \Sigma_U$ such that (v,u) labels some edge coming out of t;
2) for every infinite path from the root of the tree, the sequence of (u,v) pairs is an element of L.

An example of such a tree, where $\Sigma_V = \{0,1\}$ and $\Sigma_U = \{0,1\}$, is shown in Fig. 13.

Given $L^{C^*} \subseteq (\Sigma_V \times \Sigma_U)^\omega$ accepted by a nondeterministic *Büchi* automaton over the alphabet $\Sigma_V \times \Sigma_U$, the following is a procedure for determining if a netlist C exists whose language is contained in L^{C^*}.

1) Use the construction of [27] to determinize the automaton C^* to obtain a deterministic Streett automaton.
2) In this Streett automaton, project the symbols of the alphabet $\Sigma_V \times \Sigma_U$ down to Σ_V. Interpret the new structure as a Streett automaton on *trees* and check for tree emptiness [24].

As is shown in [23], an implementable controller (a netlist in our context) exists if and only if the tree emptiness check is negative; this approach will produce an implementation if one exists.

The complexity of this procedure is very high—the construction of the deterministic Streett automaton potentially yields an

automaton whose state-space is exponential in S_M, and doubly exponential in S_S. Furthermore, the tree-emptiness check is NP-complete; the algorithm of [23] has complexity polynomial in the number of states and exponential in the number of accepting pairs of the Streett automaton.

VI. Summary

We have proposed the logic S1S as a formalism to describe permissible behaviors of an FSM interacting with other FSMs. We believe that this framework offers several advantages.

First, for any S1S formula it is possible to generate automatically an automaton describing the same behaviors as the formula. Thus, fully automatic synthesis is possible that takes into account all available degrees of freedom. In practice, the generated automaton is often too large to handle with state-of-the-art optimization algorithms. Nevertheless, S1S provides a rigorous framework in which one can prove that a set of behaviors used as a don't-care condition indeed represents permissible behaviors of the system. This allows easy development of a spectrum of methods that explore trade-offs between flexibility provided by the information about the environment, and the price of storing and using this information—on one side of the spectrum is the optimization of a component in isolation, and on the other side is the construction of the E-machine. A concrete example of this trade-off was presented in Section IV-D, where we saw that by restricting our attention to the flexibility afforded by input don't care sequences, we arrived at an approximation which was significantly more tractable. The formalism S1S provides a systematic and simple way of reducing the problem of optimizing interacting FSMs to optimizing a single FSM, with different methods generating FSMs of different sizes. Thus, any future improvement in FSM optimization algorithms will provide immediate benefits to optimization of interacting FSMs.

Second, in contrast to previous approaches, our approach is easily extended to different interconnection topologies. In this paper we have derived specifications of permissible behaviors for several topologies, some of which have not been previously investigated. By observing specifications for different topologies we were able to formulate the following general principle: if a component FSM can observe values of all the signals in the system, then the size of its E-machine is polynomial; otherwise it is exponential.

Finally, our approach immediately generalizes to the synthesis of properties, such as safety and liveness. In doing so, we have also shed some light on the relationship between interpretations of the term "synthesis" in different communities.

Future Work: There are a number of ways in which this work can be extended. Experiments need to be performed on a meaningful set of examples to see how the proposed procedures perform in the average case. Additionally, studies can be made on the use of partitioning and peephole optimization techniques (as are used in combinational logic synthesis) to reduce complexity when dealing with large designs.

Our approach should be applicable to software synthesis, applications of which include optimizing embedded controllers, and hardware-software co-design. Similarly, the synthesis of richer systems, such as those which include timing functionality and statistical behavior, can be studied in our framework.

In a broader context, the ideas brought forward in this paper demonstrate the power and elegance of employing mathematical logic to solve problems in design automation. We hope this paper will motivate researchers in EDA to learn more about mathematical logic; we recommend the excellent textbook of Enderton [9] to interested readers.

Acknowledgment

The authors would like to thank V. Singhal for his help with developing the notation used in this paper, and the reviewers for their detailed feedback.

References

[1] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*. Norwell, MA: Kluwer Academic, 1996.

[2] J. R. Buchi, "On a decision method in restricted second order arithmetic," in *Proc. Int. Congress Logic, Methodology, and Philosophy of Science*, 1960, pp. 1–11.

[3] J. R. Burch, D. L. Dill, E. Wolf, and G. D. Micheli, "Modeling hierarchical combinational circuits," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1993, pp. 612–617.

[4] Y. Choueka, "Theories of automata on omega-tapes: A simplified approach," *JCSS*, vol. 8, no. 2, pp. 117–141, 1974.

[5] S. Devadas, "Optimizing interacting finite state machines using sequential don't cares," *IEEE Trans. Computer-Aided Design*, pp. 1473–1484, Dec. 1991.

[6] M. DiBenedetto, A. Saldanha, and A. Sangiovanni-Vincentelli, "Model matching for finite state machines," presented at the IEEE Conf. Decision and Control, Dec. 1994.

[7] C. C. Elgot, "Decision problems of finite automation design and related decision problems," *Trans. Amer. Math. Soc.*, vol. 98, pp. 21–52, 1961.

[8] *Formal models and semantics (Handbook of Theoretical Computer Science)*, vol. B, J. van Leeuwen, Ed., Elsevier Science, Amsterdam, The Netherlands, 1990, pp. 996–1072.

[9] H. Enderton, *A Mathematical Introduction to Logic*. New York: Academic, 1972.

[10] J. Fron, J. C.-Y. Yang, M. Damiani, and G. De Micheli, "A synthesis framework based on trace and automata theory," in *Proc. Int. Symp. Circuits and Systems*, May 1994, pp. 291–294.

[11] M. R. Garey and D. S. Johnson, *Computers and Intractability*. San Francisco, CA: Freeman, 1979.

[12] G. D. Hachtel, J.-K. Rho, F. Somenzi, and R. Jacoby, "Exact and heuristic algorithms for the minimization of incompletely specified state machines," in *Proc. Eur. Conf. Design Automation*, Amsterdam, The Netherlands, Feb. 1991, pp. 184–191.

[13] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*. Reading, MA: Addison-Wesley, 1979.

[14] O. H. Jensen, J. T. Lang, C. Jeppesen, and K. G. Larsen, "Model construction for implicit specifications in modal logic," in *Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 1993, vol. 715.

[15] T. Kam, "State minimization of finite state machines using implicit techniques," Ph D dissertation, Electron. Res. Lab., College Eng., Univ. California, Berkeley, May 1995.

[16] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, "A fully implicit algorithm for exact state minimization," in *Proc. Design Automation Conf.*, June 1994, pp. 684–690.

[17] J. Kim and M. M. Newborne, "The simplification of sequential machines with input restrictions," *IRE Trans. Electron. Comput.*, pp. 1440–1443, Dec. 1972.

[18] R. P. Kurshan, *Automata-Theoretic Verification of Coordinating Processes*. Princeton, NJ: Princeton Univ. Press, 1993.

[19] S. Malik, "Analysis of cyclic combinational circuits," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 950–956, July 1994.

[20] Z. Manna and J. Waldinger, "Toward automatic program synthesis," *Commun. ACM*, vol. 14, no. 3, pp. 151–165, Mar. 1971.

[21] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw Hill, 1994.

[22] J. Parrow, "Submodule construction and equation solving in CCS," *Theoretical Comput. Sci.*, vol. 68, 1989.

[23] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proc. ACM Symp. Principles of Programming Languages*, 1989, pp. 179–190.
[24] M. O. Rabin, *Automata on Infinite Objects and Church's Problem*. Providence, RI: Amer. Math. Soc., 1971.
[25] P. Ramadge and W. Wonham, "The control of discrete event systems," *Proc. IEEE*, vol. 77, pp. 81–98, 1989.
[26] J.-K. Rho, G. Hachtel, and F. Somenzi, "Don't care sequences and the optimization of interacting finite state machines," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1991, pp. 418–421.
[27] S. Safra, "Complexity of Automata on Infinite Objects," Ph.D. dissetation, The Weizmann Inst. Sci., Rehovot, Israel, Mar. 1989.
[28] H. Savoj, "Don't Cares in Multi-Level Network Optimization," PhD thesis, The University of California at Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA, May 1992.
[29] T. R. Shiple, "Formal Analysis of Synchronous Hardware," Ph.D. dissertation, Electron. Res. Lab., College Eng., Univ. California, Berkeley, CA, 1996.
[30] V. Singhal, "Design Replacements for Sequential Circuits," Ph.D. dissertation, Electron. Res. Lab., College Eng., Univ. California, Berkeley, CA, 1996.
[31] *Formal models and semantics (Handbook of Theoretical Computer Science)*, vol. B, J. van Leeuwen, Ed., Elsevier Science, Amsterdam, The Netherlands, 1990, pp. 133–191.
[32] R. J. van Glabbeek, "Comparative Concurrency Semantics and Refinement of Actions," PhD thesis, Centrum voor Wiskunde en Informatica, Vrije Universiteit te Amsterdam, Amsterdam, The Netherlands, May 1990.
[33] H.-Y. Wang and R. K. Brayton, "Input don't care sequences in FSM networks," in *Proc. Int. Conf. Computer-Aided Design*, 1993, pp. 321–328.
[34] ——, "Permissible observability relations in FSM networks," in *Proc. Design Automation Conf.*, June 1994, pp. 677–683.
[35] Y. Watanabe and R. K. Brayton, "The maximum set of permissible behaviors for FSM networks," in *Proc. Int. Conf. Computer-Aided Design*, 1993, pp. 316–320.
[36] H. Wong-Toi and D. L. Dill, "Synthesizing processes and schedulers from temporal specifications," in *Proc. 2nd Workshop Computer-Aided Verification*, 1990, pp. 272–281.
[37] W. Wonham and P. Ramadge, "On the supremal controllable language of a given language," *SIAM J. Contr. Optimization*, vol. 25, pp. 637–659, 1988.

Adnan Aziz received the undergraduate degree from the Indian Institute of Technology, Kanpurthe, India. He received the Ph.D. degree in electrical engineering and computer sciences from The University of California at Berkeley in 1996.

He joined The University of Texas, Austin, in the Spring of 1996. His research interests lie in algorithms for design and verification, particularly in the area of VLSI; he has made contributions to both the theory and practice of synthesizing and verifying digital systems. More specifically, he has written a number of papers on design verification and sequential synthesis. Additionally, he is one of the architects of the VIS system, a software tool that is widely used for formal verification. His current interests include enhancing simulation with symbolic algorithms, and integration of logic synthesis with physical design.

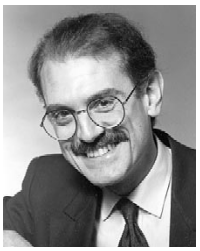
Felice Balarin (S'90–M'95) received the Ph.D. degree in electrical engineering and computer science from the University of California at Berkeley in 1994.

He has been a Research Scientist with the Cadence Berkeley Labs., Berkeley, CA, since 1994. His research is focused on development and application of formal methods to design, verification and timing analysis of systems consisting of both hardware and software.

Robert K. Brayton (M'75–SM'78–F'81) received the B.S.E.E. degree from Iowa State University, Ames, in 1956 and the Ph.D. degree in mathematics from Massachusetts Institute of Technology, Cambridge, in 1961.

From 1961 to 1987 he was a member of the Mathematical Sciences Department of the IBM T. J. Watson Research Center, Yorktown Heights, NY. In 1987, he joined the Electrical Engineering and Computer Science Department at the University of California at Berkeley, where he is the Cadence Distinguished Professor of Engineering and the director of the SRC Center of Excellence for Design Sciences. He has authored over 400 technical papers, and eight books. His past contributions have been in analysis of nonlinear networks, electrical simulation and optimization of circuits, and asynchronous synthesis. His current research involves combinational and sequential logic synthesis for area/performance/testability, formal design verification and synthesis for DSM designs.

Dr. Brayton held the Edgar L. and Harold H. Buttner Endowed Chair in Electrical Engineering at Berkeley. He is a member of the National Academy of Engineering, and a Fellow the AAAS. He received the 1991 IEEE CAS Technical Achievement Award, the IEEE CAS Golden Jubilee Medal, and five best paper awards, including the 1971 IEEE Guilleman-Cauer award, and the 1987 ISCAS Darlington award. He was the editor of the *Journal on Formal Methods in Systems Design* from 1992–1996. He received the CAS Golden Jubilee Medal and the IEEE Millennium Medal in 2000.



Alberto Sangiovanni-Vincentelli (M'74–SM'81–F'83) received the electrical engineering and computer science degree ("Dottore in Ingegneria") *summa cum laude* from the Politecnico di Milano, Milano, Italy in 1971.

He holds the Edgar L. and Harold H. Buttner Chair of Electrical Engineering and Computer Sciences at the University of California at Berkeley, where he has been on the Faculty since 1976. In 1980–1981, he spent a year as a Visiting Scientist at the Mathematical Sciences Department of the IBM T.J. Watson Research Center, Yorktown Heights, NY. In 1987, he was Visiting Professor at Massachusetts Institute of Technology, Cambridge. He was a co-founder of Cadence and Synopsys, two leading companies in the area of electronic design automation. He was a Director of ViewLogic and Pie Design System and Chair of the Technical Advisory Board of Synopsys. He is the Chief Technology Advisor of Cadence Design System. He is a member of the Board of Directors of Cadence, where he is the Chairman of the Nominating Committee, Sonics Inc., and Accent, an ST-Cadence joint venture. He is the founder of the Cadence Berkeley Laboratories and of the Cadence European laboratories. He is the Scientific Director of the Project on Advanced Research on Architectures and Design of Electronic Systems (PARADES), a European Group of Economic Interest supported by Cadence, Magneti-Marelli and ST Microelectronics. He is an author of over 520 papers and ten books in the area of design methodologies, large-scale systems, embedded controllers, hybrid systems and tools.

In 1981 Dr. Sangiovanni-Vincentelli received the Distinguished Teaching Award of the University of California. He received the worldwide 1995 Graduate Teaching Award of the IEEE (a Technical Field award for "inspirational teaching of graduate students"). He has received numerous awards including the Guillemin-Cauer Award (1982–1983) and the Darlington Award (1987–1988). He is a Member of the National Academy of Engineering.