Sequential Synthesis Using S1S[*]

Adnan Aziz[1] **Felice Balarin**[2] **Robert Brayton**[1] **Alberto Sangiovanni-Vincentelli**[1]

[1] Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley CA 94720

[2] Cadence Design Systems
Cadence Berkeley Laboratories
Berkeley CA 94704

Abstract

We present a mathematical framework for analyzing the synthesis of interacting finite state systems. The logic S1S is used to derive simple, rigorous, and constructive solutions to problems in sequential synthesis. We obtain exact and approximate sets of permissible FSM network behavior, and address the issue of FSM realizability. This approach is also applied to synthesizing systems with fairness and timed systems.

1 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 designs (for various measures such as area, speed, or power) specified at the gate level, has shifted to the forefront of CAD research. Another area rapidly gaining importance is design verification, the study of systematic methods for formally proving the correctness of designs.

Logic synthesis algorithms originally targeted the optimization of PLA implementations; this was followed by research in synthesizing more general multi-level logic implementations. Currently, the central thrust in logic synthesis is sequential synthesis, i.e. the automatic optimization of the entire system. This can be at the logic level (i.e. the input is a netlist of gates and latches) or the state transition graph level. Designs invariably consist of a set of interacting components. Natural questions related to such systems are what is the optimal choice of a component, and automatically deriving a component so as to satisfy given properties.

Previous work in the VLSI design automation community related to optimizing interacting state machines has tended to be ad hoc and incomplete. The constructions and proofs offered are often extremely cumbersome. Relevant papers include [1, 2, 3, 4, 5, 6]. One attempt at formal synthesis framework based on trace and automata theory is given in [7]. However, the central theorem relating flexibility in a sub-circuit to the specification and the environment is incorrect; we give the correct formulation. There is a large body of theoretical work related to the existence of efficient decision procedures for deciding logics of programs [8, 9, 10, 11]. [9, 10] take logical specifications and construct programs satisfying them. [8] uses a game theoretic formulation to show that the set of moves for a controller is an ω-regular set. [11] gives an elegant characterization of realizability.

Typically, the synthesis process has two stages: first, the set of all possible implementations is characterized (which is the topic of this paper), and then one is chosen according to some optimality criteria. Using the sequential calculus S1S as our basic tool we show in section 3.1 that the set of all implementations can always be captured by a single finite-state automaton, which can be generated automatically. However, in practise such an automaton may be prohibitively large to construct. One approach to alleviate this problem is to define easier to handle automata which capture only a subset of possible implementations. We pursue this approach in section 3.2.

The rest of this paper is structured as follows: §2 reviews the basic definitions and salient results. In §3 we apply these results to synthesizing and optimizing finite state machine networks; specifically, we derive sources of flexibility that can be exploited by synthesis. We also address the issue of hardware realizability. In §4 we describe extensions of our approach to synthesizing systems which incorporate fairness constraints and timed systems.

The analysis in this paper is of a theoretical nature. In particular, the complexity of the exact procedures described can be exponential, or even doubly exponential. From a theoretical point of view, the complexity is inherent; this simply reinforces the need for approximations and heuristics.

2 Definitions and Basic Results

This section reviews germane definitions and results. In particular, the relationship between finite state machines, languages, and S1S logic is established. For reasons that will become apparent later, we will consider both automata on both finite and infnite sequences. An excellent survey of the material covered in this section is in [12].

2.1 Finite State Automata and Machines

Given a finite set Σ, the set Σ^* is the set of all finite sequences over Σ. A $*$-language over Σ is a subset of Σ^*. Given $X \in \Sigma^*$, $|\,X\,|$ denotes the length of X. The set Σ^ω is the set of all infinite sequences over Σ, i.e. all maps $f : \omega \to \Sigma$, where $\omega = \{0, 1, 2, \ldots\}$ is the set of natural numbers. An ω-language over Σ is a subset of Σ^ω.

Notation: Lower case variables will take values from alphabets; upper case variables will be sequence valued.

Definition 1 A finite state automaton (FA) is a 5-tuple (Σ, S, s_0, T, A) where Σ is a finite set called the *alphabet*, S is a finite set of *states*, $s_0 \in S$ is the initial state, $T \subset S \times \Sigma \times S$ is the *transition relation*, and $A \subset S$ is the set of *accepting* states.

The automaton is said to be *deterministic* (variously a DFA) if $(\forall s.\forall x)[\,|\,\{t : (s, x, t) \in T\}\,| \leq 1]$; otherwise it is said to be *non-deterministic* (variously an NFA).

A string $X \in \Sigma^*$ is *accepted* by the FA if there exists a sequence of states $\sigma = \sigma_0\sigma_1 \ldots \sigma_n$ such that $n = |\,X\,|$, $\sigma_0 = s_0$, $\sigma_n \in A$, and $(\forall i)\,[(\sigma_i, X_i, \sigma_{i+1}) \in T]$. The language of the automaton is the set of strings accepted by it; this language is said to be $*$-regular if it is the language accepted by some automaton.

Definition 2 A Büchi automaton (BA) is a 5-tuple (Σ, S, s_0, T, B) where Σ is a finite set called the *alphabet*, S is a finite set of *states*, $s_0 \in S$ is the initial state, $T \subset S \times \Sigma \times S$ is the *transition relation*, and $B \subset S$ are the accepting states.

[*] Supported by SRC Grant 94-DC-324

The automaton is said to be *deterministic* if $(\forall s.\forall x)\,[\,|\,\{t : (s,x,t) \in T\}\,| \leq 1\,]$; otherwise it is said to be *non-deterministic.*

A string $x \in \Sigma^\omega$ is *accepted* by the BA if there exists a sequence of states $\sigma_0\sigma_1\ldots$ such that $\sigma_0 = s_0$, and $(\forall i)\,[(\sigma_i, x_i, \sigma_{i+1}) \in T]$, and $inf(\sigma) \cap B \neq \phi$, where $inf(\sigma)$ is the infinitary set of σ, i.e. the set of states that occur infinitely often in σ. The language of the automaton is the set of strings accepted by it; a language is said to be ω-regular if it is the language accepted by some Büchi automaton.

Definition 3 A *finite state machine* M is a 5-tuple (S, s_0, I, O, T) where S is a finite set of *states,* $s_0 \in S$ is the *initial* state, I is a finite set of *inputs,* O is a finite set of *outputs,* and $T \subseteq S \times I \times O \times S$ is the *transition relation.*

By definition, M is *deterministic* if

$$(\forall s.\forall i.\forall o)\Big[\ |\{t : (s,i,o,t) \in T\}\,| \leq 1\Big]$$

M is *complete* if

$$(\forall s.\forall i.\exists o.\exists t)\Big[(s,i,o,t) \in T\Big]$$

M is an *implementation* if, in addition to being complete, it satisfies

$$(\forall s.\forall i)\Big[\ |\{(t,o) : (s,i,o,t) \in T\}\,| \leq 1\Big]$$

M is *Moore* if, in addition to being an implementation, it satisifes $\forall s.\forall i.\forall o_1.\forall o_2.\forall t_1.\forall t_2$

$$\Big[[(s,i,o_1,t_1) \in T] \wedge [(s,i,o_2,t_2) \in T] \rightarrow (o_1 = o_2)\Big]$$

else it is said to be *Mealy.*

Given an input sequence $i = (i_1 i_2 \cdots i_n)$, and output sequence $o = (o_1 o_2 \cdots o_n)$, a *corresponding run* is a sequence of states $\sigma = (s_0 s_1 \cdots s_n)$ such that $(\forall k)\,[(s_k, i_{k+1}, o_{k+1}, s_{k+1}) \in T]$. The notion of a run readily generalizes to infinite sequences.

Our definition of deterministic finite state machine has been referred to as "pseudo non-deterministic" (PNDFSM) in the past [6]. The term deterministic was reserved for what we refer to as an implementation. A related notion is that of *incompletely specified* finite state machines (ISFSM), where given any state s and input i, either $(\exists! o.\exists! t)\,[(s,i,o,t) \in T]$ or $(\forall o.\forall t)\,[(s,i,o,t) \in T]$.

Definition 4 The $*$-language identified with an FSM $M = (S, r, I, O, T)$, denoted by $\mathcal{L}^M$ and referred to as the *behavior* of M, is the set of sequences $(i,o) \in (I \times O)^*$ such that there exists a run in M corresponding to (i,o).

Clearly the behavior $\mathcal{L}^M$ is a $*$-regular language ([13, 12]) on the alphabet $I \times O$. The automaton $\mathcal{A}^M$ defining the language is simply the FSM itself – the states of $\mathcal{A}^M$ are the states S, initial states of $\mathcal{A}^M$ are the states r, the transition relation of the automaton is $\{(s,(i,o),t) : (s,i,o,t) \in T\}$, and all states are accepting. Note that the behavior of a deterministic machine is defined by a deterministic finite automaton.

Definition 5 Given a language $L \subset (\Sigma_I \times \Sigma_O)^*$, a finite state machine M on input Σ_I, output Σ_O is said to be *compatible* with L if $\mathcal{L}^M \subset L$; M is said to be a *realization* of L if it is compatible with L and is an implementation. A language is said to be *realizable* if there exists a realization of it; similarly an FSM is realizable if its language is realizable.

Definition 6 A *finite state machine with Büchi fairness* is a tuple (M, C) where M is an FSM, and C is a subset of the states of M. Given $F = (M, C)$, an FSM with fairness F is said to be deterministic (complete) if M is deterministic (complete).

The notion of a corresponding run over infinite input/output sequences is defined analogously to that for ordinary FSM's; a run σ is fair if the infinitary set of states is an element of C. Similarly the language of a finite state machine with fairness is defined to be the language of the corresponding Büchi automaton.

Composition of finite state machines is defined in the usual way [6]. In composing interacting FSM's, (sometimes referred to as a network of FSM's) some inputs of each machine may be the outputs of each machine, whereas some inputs may be external; also, not all outputs need be external. The entire systems is itself a finite state machine on the product state space, referred to as the *product machine* [6]; transitions take place synchronously, i.e. in "lock-step". When the component machines are Mealy and derived from hardware, composition can lead to combinational cycles, and the product may have undesired oscillatory behaviors [14]. These problems can be circumvented by using Moore machines; we will come back to this phenomenon in section 3.3.

2.2 The Sequential Calculus S1S

The logic S1S is a formalism for analyzing sequences over finite alphabets. It was studied in detail by Büchi in [15]; in particular it was shown to be decidable. S1S provides an extremely powerful mechanism for analyzing and manipulating sequential systems – the full expressiveness of logic (conjunction, negation, and quantification) is available.

Definition 7 The logic S1S (*second order theory of one successor)* is a second order logic [12]. Formulae are derived from the alphabet $\{0, S, =, <, \in, \wedge, \neg, \exists, x_1, x_2, \ldots, X_1, X_2, \ldots\}$. Lower case variables $x_1, x_2, \ldots$ are first order variables ranging over elements of the domain, and upper case variables $X_1, X_2, \ldots$ are second order variables ranging over subsets of the domain. The well formed formulae of the logic S1S are given by the following syntax:

- **Terms** are constructed from the constant 0 and first order variables by repeated applications of the successor function S. Examples of terms – 0, $SS0$, $SSSSx_3$.
- **Atomic formulae** are of the form $t_1 = t_2$, $t_1 < t_2$, $t \in X_k$. Examples of atomic formulas – $0 < S0, x_3 = Sx_5, Sx_7 \in X_2$.
- **S1S formulae** are constructed from atomic formulas by using the boolean connectives $\wedge, \neg$ and quantification over both kinds of variables. Examples of S1S formulas – $(0 < S0) \wedge (Sx_7 \in X_2)$, $(\exists X.\exists x)\,[(x \in X) \wedge (Sx \in X)]$. We write $\phi(X_1, X_2, \ldots, X_n)$ to denote that at most $X_1, X_2, \ldots, X_n$ occur free in ϕ (i.e. are not in the scope of any quantifier). We will routinely use the symbols $\vee, \rightarrow, \forall$, etc as logical abbreviations.

S1S formulae can be interpreted over the set of natural numbers, where Sx is simply $x + 1$. Formal semantics of S1S can be found in [12]; we informally illustrate them by means of examples.

Example 1: (Non-empty subsets of ω contain minimal elements)

$$\psi \;=\; (\forall X)\,\Big[(\exists x)(x \in X) \rightarrow (\exists y)[(y \in X) \wedge \neg((\exists z)(z \in X \wedge (z < y)))]\Big]$$

The above sentence formally states that for every subset (X) of ω, if X is non-empty $(\exists x \in X)$, then it contains a least element (y).

Example 2: (Defining subsets of ω which contain 5 whenever they contain 3.)

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

Example 3: (Defining the subset of even integers)

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

Example 4: (Defining the relation "every even number in X is in Y")

$$\phi_2(X, Y) \quad = \quad (\forall x)\big[(\exists Z)\,(\phi_1(Z) \wedge x \in Z) \rightarrow (x \in X \rightarrow x \in Y)\big]$$

Given a formula $\theta(X_1)$ in S1S, the class of subsets of ω *defined* by $\theta(X_1)$ is the set $\{\beta \subset \omega \mid \theta(\beta)$ is true$\}$. The class of subsets of ω is in a one to one correspondence with the set of ω-sequences on $\{0, 1\}$ – the 1's in the sequence can be thought of as representing the integers in the corresponding set, e.g. $010101\ldots \leftrightarrow \{1, 3, 5, \ldots\}$. In this way, an S1S formula $\theta(X_1)$ defines an ω-language over alphabet $\{0, 1\}$. More generally, formulae $\phi(X_1, X_2, \ldots, X_n)$ define subsets of $(\{0, 1\}^n)^\omega$.

The following result relates S1S formulae to ω-automata.

Theorem 2.1 (Büchi 1961 [15, 12]) *An ω-language is definable in S1S if and only if it is ω-regular.*

Proof:(*Sketch:*) The reverse direction involves a straightforward construction of a formula coding up the transition structure of the automaton.

The forward direction is by induction on the length of the S1S formula. Automata for the atomic formulae are easily derived; an inductive construction is used for $\neg, \wedge, \exists$. $\exists$ is handled by automaton projection, $\wedge$ by automaton intersection, and $\neg$ by automaton complementation. The latter is the non-trivial step – invariably performed by first determinizing the automaton, following which complementation is trivial. The process of complementation is inherently exponential, since the number of states in the complement can (in the worst case) be exponential in the number of states of the given automaton. ■

With minor modifications, Büchi's result also holds for sets of *finite* words i.e. when set quantification is restricted to finite sets only. In this case one speaks of the theory WS1S (weak S1S). The corresponding result for WS1S states that a $*$-language is definable in WS1S if and only if it is $*$-regular [12, 15].

Definition 8 Given a formula $\phi^{\mathcal{A}}(X_1, X_2, \ldots, X_n)$ in S1S (WS1S) we can uniquely identify a Büchi automaton (finite automaton) $\mathcal{A}$ over the alphabet $\{0, 1\}^n$.

The relationship between automata and S1S allows us to formally and succinctly express behaviors as formulae in logic, and also provides an automatic procedure to obtain automata from the formulae. Hence, elegant yet rigorous proofs can be given to a large class of solutions to problems related to finite state systems. Furthermore these proofs are constructive, i.e. given formula in S1S/WS1S it is possible to mechanically construct the corresponding automaton.

3 Synthesizing FSM networks

As mentioned in the introduction, a critical first step towards synthesizing a component in a design is characterizing the set of all valid implementations. In this section, the flexibility available for sequential synthesis is analyzed. We use S1S to formulate the "E-machine" of Watanabe [6] for a number of FSM interconnect topologies, derive a spectrum of approximations to the E-machine, and address the issue of realizability.

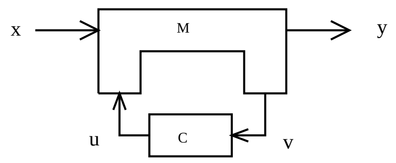

Figure 1: Example of interacting finite state machines; we will refer to C as the *controller*

3.1 The E-machine

Consider machines communicating in the configuration shown in figure 1. Suppose x and y are the observable inputs and outputs, and $\mathcal{L}^S \subset (\Sigma_X \times \Sigma_Y)^*$ is a $*$-regular specification on them, i.e. the only acceptable input–output behavior is that which is contained in $\mathcal{L}^S$. Also suppose the machine M is fixed. The following theorem states that all the flexibility available for synthesis at C is characterized by a single language $\mathcal{L}^{C^*}$ definable in S1S: for any FSM C, $M \times C$ satisfies S if and only if $\mathcal{L}^C \subset \mathcal{L}^{C^*}$

Theorem 3.1 The set of all behaviors on u, v which will yield behavior on the inputs and outputs which is compatible with the specification is given by the following expression:

$$\phi^{C^*}(U, V) \quad = \quad (\forall X.\forall Y)\left[\phi^M(X, Y, U, V) \rightarrow \phi^S(X, Y)\right] \quad (1)$$

Proof: For input sequence V, the controller should *not* generate output sequence U if and only if there exist sequences X, Y such that M produces Y, V on input X, U and (X, Y) does not lie in the specification. Mathematically, the set of behaviors that C should not produce is given by $(\exists X.\exists Y)[\phi^M(X, U, V, Y) \wedge \neg\phi^S(X, Y)]$; the complement of this set, namely the set of U's that can be generated corresponding to V is precisely the set defined by equation 1. ■

By the remarks in section 2.2, $\mathcal{L}^{C^*}$ is regular. Note that the number of states in an automaton for $\neg\phi^S(X, Y)$ is $O(2^{|S_S|})$ (automaton complementation by the power set construction [13]). Thus, the number of states in the automaton for $[\phi^M(X, U, V, Y) \wedge \neg\phi^S(X, Y)]$ is $O(|S_M| \cdot 2^{|S_S|})$ (automaton product), and so, the number of states for the automaton for $\neg(\exists X.\exists Y)[\phi^M(X, U, V, Y) \wedge \neg\phi^S(X, Y)]$ is $O(2^{|S_M| \cdot 2^{|S_S|}})$. This complexity is inherent: it can be achieved in the worst case. We illustrate the construction for C^* by means of an example, described in figure 2.

In the special case when the automaton defining $\mathcal{L}^S$ is deterministic, the corresponding bound is $2^{|S_M| \cdot |S_S|}$. This is precisely the construction of [6]; it is instructive to contrast this approach with that taken in [6] and noting the simplicity afforded by appealing to S1S.

The specification automaton could be $M \times C$; this corresponds to the *re-synthesis* problem, i.e. suppose we wish to find a replacement for the C block which is optimal (with respect to an appropriate objective function) while preserving the observed behavior. Then the behavior of the replacement must be contained in the behavior $\mathcal{L}^{C^*}$.

In the most general setting M and the specification automaton are non-deterministic and incompletely specified. In this case, simply deciding if an implementation (in the sense of definition 5, section 2) exists for the block C which is compatible with the specification is non-trivial; realizability is discussed in § 3.3.

There are variations on the interconnect structures to which an approach similar to that of theorem 3.1 can be applied to derive formulae expressing the set of permissible behaviors at a specified

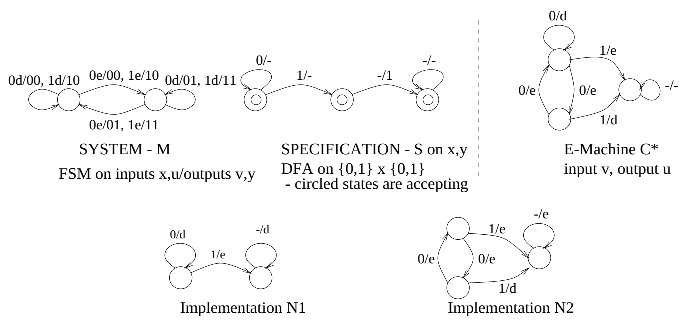

Figure 2: We are given an FSM on inputs x, u and outputs y, v, and the specification that "if x goes high, then in the next state, y should be high" formalized by the DFA S. We obtain C^* by the construction corresponding to equation 1. Any controller which on composition with M yields a machine compatible with S is contained in C^*; in particular N1 and N2 are valid controllers.

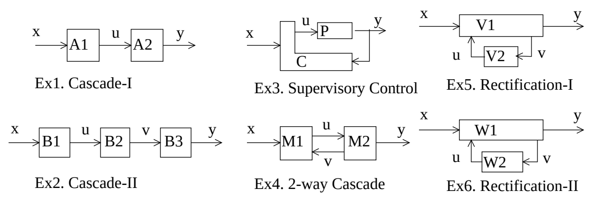

Figure 3: A variety of FSM network topologies; the name suggests applications.

machine. In figure 3 we describe a set of FSM network topologies; below we give the S1S formulae defining the corresponding E-machines. The ease with which we derive the flexibility is a measure of the power of S1S logic – in the past, individual topologies have been considered individually, and the flexibility has been laboriously derived. It is noteworthy that when all signals are observable at a machine, and the specification is deterministic, then the machine defining the set of permissible behaviors is polynomial sized.

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

2-way Cascade (a) $\phi^{{M_1}^*}(X, V, U) = (\forall Y)[\phi^{M_2}(V, U, Y) \rightarrow \phi^S(X, Y)]$

2-way Cascade (b) $\phi^{{M_2}^*}(U, V, Y) = (\forall X)[\phi^{M_1}(X, V, U) \rightarrow \phi^S(X, Y)]$

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

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

3.2 Optimization of FSM networks

In this section we describe procedures for optimizing networks of finite state machines. In particular, we are interested in the *re-synthesis* problem, i.e. the specification is the functionality of the original FSM network. For such systems, the full range of admissible behavior at a node is described by the E-machine, and the original machine trivially satisfies the specification.

Deriving Optimal Implementations

Given the E-machine, one would like to derive an implementation that is optimal. One criterion for optimality is state minimality. In practice, deriving state minimal realizations from ISFSM's is easier than from general deterministic FSM's [6] (although both are NP-complete). Our interest in input don't care sequences and satisfiability don't care sequences defined below partly stems from the fact that the flexibility afforded by them can be captured by ISFSM's rather than general deterministic FSM's.

We provide a spectrum of approximations to the flexibility at a component, starting from more conservative approximations, and leading up to the E-machine.

Let x, y be the input and output of the FSM network. Consider a component machine C on inputs v and outputs u. Let M be the rest of the network.

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

$$\phi^{SDC_0^C}(V) \quad = \quad \neg(\exists\tilde{X}.\exists\tilde{U}.\exists\tilde{Y})[\phi^M(\tilde{X}, \tilde{U}, V, \tilde{Y})]$$

It is precisely the set of sequences over v which can never be generated, no matter what replacement is used for C.

This set gives a certain amount of flexibility in choosing implementations for C; namely any behavior in the machine C_0 defined below is acceptable.

$$\phi^{C_0}(V, U) \quad = \quad \neg\phi^{SDC_0^C}(V) \rightarrow \phi^C(V, U)$$

In [16] we prove the above claim, show that C_0 is an ISFSM, and also demonstrate that C_0 does not provide all the flexibility available in optimizing C.

Definition 10 The *weak satisfiability don't care set* for C is given by the following expression:

$$\phi^{SDC_1^C}(V) \quad = \quad \neg(\exists\tilde{X}.\exists\tilde{U}.\exists\tilde{Y})\,[\phi^M(\tilde{X}, \tilde{U}, V, \tilde{Y}) \wedge \phi^C(\tilde{U}, V)]$$

It is precisely the set of sequences over v which can never be generated in the product machine $M \times C$, and corresponds to the *input don't care* sequences of [4].

This set gives additional flexibility over SDC_0^C in choosing implementations for C; namely any behavior in the machine C_1 defined below is acceptable.

$$\phi^{C_1}(V, U) \quad = \quad \neg\phi^{SDC_1^C}(V) \rightarrow \phi^C(V, U)$$

In [16] we prove the above claim, show that C_1 is an ISFSM, and also demonstrate that C_1 does not provide all the flexibility available in optimizing C.

Definition 11 The *strong observability equivalence relation* for C is given by the following expression:

$$\phi^{\mathcal{O}_0^C}(V) \quad = \quad (\forall\tilde{X}.\forall\tilde{Y}.\forall\tilde{U})\,[\phi^M(\tilde{X}, \tilde{U}, V, \tilde{Y}) \rightarrow \phi^{M\times C}(\tilde{X}, \tilde{Y})]$$

It is precisely the set of sequences over v for which any output sequence over u is permissible. Clearly, for input sequences which are never generated any output is acceptable, so $\phi^{SDC_1^C}(V) \Rightarrow \phi^{\mathcal{O}_0^C}(V)$.

This set gives additional flexibility over SDC_1^C in choosing implementations for C; namely any behavior in the machine C_2 defined below is acceptable.

$$\phi^{C_2}(V, U) \quad = \quad \neg\phi^{\mathcal{O}_0^C}(V) \rightarrow \phi^C(U, V)$$

In [16] we prove the above claim, show that C_2 is an ISFSM, and also demonstrate that C_2 does not provide all the flexibility available in optimizing C.

```
Realizable_States: (DFA D : $(S_D, s_0, \Sigma_V \times \Sigma_U, T_D, A_D)$){
$S_C = A_D$;
        while ( TRUE ) {
            remove states $s$ from $S_C$ such that
                $\neg[(\forall v.\exists u.\exists t)\,[(s,v,u,t) \in T_D \wedge (t \in S_C)]$
            if (no states were removed)
                break;
        }
return $S_C$;
}
```

Figure 4: Algorithm for deciding if a realization exists in a $*$-regular specification

Definition 12 The *true observability equivalence relation* for C is given by the following expression:

$$\begin{aligned}\phi^{\mathcal{O}_2^C}(V,U) \quad &= \quad (\forall \tilde{X})\,[(\exists \tilde{Y})\,\phi^M(\tilde{X},U,V,\tilde{Y}) \rightarrow \\ &\qquad (\forall \tilde{Y})\,(\phi^M(\tilde{X},U,V,\tilde{Y}) \rightarrow \phi^{M\times C}(\tilde{X},\tilde{Y}))]\end{aligned}$$

In [16], we prove that the true observability equivalence relation is logically equivalent to ϕ^{C^*} which defines the E-machine, and hence captures all the flexibility possible for synthesizing C.

3.3 Realizability

The set $\mathcal{L}^{C^*}$ defined by $\phi^{C^*}(U,V)$ is the set of all acceptable controller behavior. In general, $\mathcal{L}^{C^*}$ may not be realizable (§ 2). This can happen in two ways. There may be *blocking* input sequences $\tilde{V}$, i.e. sequences for which there is no $\tilde{U}$ such that $(\tilde{U},\tilde{V}) \in \mathcal{L}^{C^*}$. However, even if there are no blocking input sequences, a realization may not exist because of *causality*, viz the output may depend on future values of the input.

In [11] it is argued that a necessary and sufficient condition for realizability of a language $\mathcal{L}$ over $\Sigma_V \times \Sigma_U$ is that a *strategy tree* must exist for a player observing inputs over v and producing outputs over u while ensuring that the input-output behavior is compatible with the relation $C^*(U,V)$. This can be checked using known algorithms for emptiness of tree automaton [12]; we have developed an alternative algorithm which does not require tree automata. This algorithm extends to the similar problem for finding Moore realizations.

Given $\mathcal{L}^{C^*} \subset (\Sigma_V \times \Sigma_U)^*$, we wish to determine the existence of an implementable (cf section 2.1) finite state machine C which is compatible with $\mathcal{L}^{C^*}$ in the sense $(\forall V.\forall U)\,[\phi^C(V,U) \rightarrow \phi^{C^*}(V,U)]$. This can be done by calling the algorithm in figure 4 on a DFA for the language $\mathcal{L}^{C^*}$.

Lemma 3.1 C^* is realizable if and only if the set of states S_C remaining after convergence contains the initial state.

It should also be pointed out that when both M and the controller C are Mealy machines, there exists the possibility of combinational cycles. In this case, hardware implementations may be erroneous [14]. Since it is sufficient for the controller to be a Moore machine to avoid combinational cycles, one approach is to look for Moore realizations of $\mathcal{C}^*$. The algorithm of figure 4 can easily be adapted to search for Moore realizations by removing states s from S_C such that $\neg[(\exists u.\forall v.\exists t)\,((s,v,u,t) \in T_D \wedge (t \in S_C))]$. Of course, if M is a Moore machine, then a Mealy realization of $\mathcal{L}^{C^*}$ will operate correctly.

4 Synthesizing General Finite State Systems

In this section we sketch extensions of the technique developed in the previous section to more general systems – specifically systems with fairness, and real time systems.

4.1 Synthesizing Fairness

The analysis of systems with *fairness* is of growing importance with the advent of formal verification [17]. In order to verify large systems, they have to be simplified; in practice this is often done by abstraction, i.e. adding behaviors (possibly through non-determinism) that the original system did not have in order to obtain a more compact representation. Verification performed on the abstract system is usually conservative. To get a more accurate representation of the system, fairness constraints, which are restrictions on the infinitary behavior of the system, are added.

Fairness constraints can be used to model specifications that formalize notions of progress, eventuality, justice, liveness, etc [18]. Fairness is inherently required to model such properties.

Since fairness is a restriction on the infinitary behavior of the system, the defining relations for languages derived from FSM's with fairness are formulae in S1S rather than WS1S. The definition for the E-machine continues to give the range of permissible behaviors at the controller in the context of ω-languages. The construction of the Büchi automaton C^* proceeds as before – complementation of S, followed by conjunction with M, projection down to u, v and complementation again.

Complementation of non-deterministic Büchi automata is done by a (highly non-trivial) generalization of the powerset construction for NFA [19].

Testing the existence of an implementation requires checking for the existence of a strategy tree (cf 3.3) which can be done using tree automaton. Given $L^{C^*} \subset (\Sigma_V \times \Sigma_U)^\omega$, defined by a non-deterministic Büchi automaton over the alphabet $\Sigma_V \times \Sigma_U$ the following is a procedure for determining if a finite state machine C exists which realizes $\mathcal{L}^{C^*}$:

1. Determinize the automaton C^* to obtain a deterministic Rabin automaton [19].

2. In this Rabin automaton, project the symbols of the alphabet $\Sigma_V \times \Sigma_U$ down to Σ_V. Interpret the new structure as a Rabin automaton on *trees* and check for tree emptiness;

As is shown in [11], an implementable controller exists if and only if the tree emptiness check is negative. The algorithm of given in [11] derives an implementation if one exists.

The complexity of this procedure is very high – the construction of the deterministic Rabin automaton potentially yields of the order of $2^{|S_M|\cdot 2^{|S_S|}}$ states. Furthermore, the tree emptiness check is NP-complete; the algorithm of [11] has complexity polynomial in the number of states and exponential in the number of accepting pairs.

We illustrate this procedure by means of an example, as shown in figure 4.1. Contrast this with the example on finite sequences in figure 2 – in particular note the inherent need for a Büchi automaton to capture the eventuality condition in the specification. Similarly fairness is needed to define the set of permissible behavior.

4.2 Synthesizing Time

The formal analysis of *real time systems* is an area of active research [20]. The behavior of a timed system is now a map from $\mathbb{R}$ rather than ω as was the case for discrete time systems. Languages can be defined in terms of sets of maps from $\mathbb{R}$ to the output, a finite

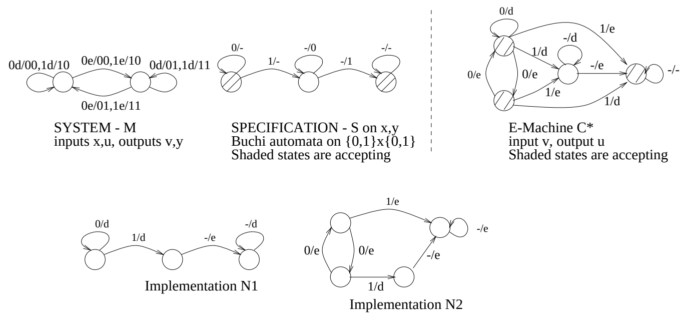

Figure 5: We are given an FSM on inputs x, u and outputs y, v, and the specification that "if x goes high, then *eventually* y should be high" formalized by the Büchi automaton S. We obtain the Büchi automaton C^* by the construction corresponding to section 4.1. Any controller which on composition with M yields a machine compatible with S is contained in C^*; in particular N1 and N2 are valid controllers.

set of scalars. The real time control/synthesis problem is defined in a manner analogous to that for discrete time.

Let S be a timed automaton whose language describes an acceptable relationship between the input timed trace X and the output timed trace Y, and M a timed automaton on inputs x, u and outputs y, v. The formulation and derivation of the E-machine continues to hold – the set $\mathcal{L}^{C^*}$ of strategies for a controller which can observe v and control u which yields acceptable behavior is still given by $\neg(\exists X.\exists Y)\,[\,\phi^M(X,Y,U,V) \wedge \neg(\phi^S(X,Y))]$, where ϕ^A is a formula defining the language of the timed system A in an appropriate logic.

Different formulations of timed automaton yield different classes of definable timed languages. [21] has identified a class closed under both quantification and complementation; thus in theory this class is synthesizable.

5 Conclusions

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

Firstly, for any S1S formula it is possible to automatically generate an automaton describing the same behaviors as the formula. Thus, a 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 the state-of-the-art optimization algorithms. Nevertheless, S1S provides a rigorous framework in which one can prove that 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. In this paper we have also suggested three other points, analogous to sets of don't cares used in combinational synthesis. S1S provides a systematic and simple way of reducing the problem of optimizing interacting FSM's to optimizing a single FSM, with different methods generating FSM's of different sizes. Thus, any future improvement in FSM optimization algorithms will provide immediate benefits to optimization of interacting FSM's.

Secondly, 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 previously been investigated. By observing specifications for different topologies we were able to formulate the following general property: if an FSM can observe values of all the signal in the system, then the size of its E-machine is polynomial; otherwise it is exponential.

Finally, our approach can also be extended to more general systems. We have sketched the extension to systems with fairness and real-time systems.

References

[1] J.-K. Rho, G. Hachtel, and F. Somenzi, "Don't Care Sequences and the Optimization of Interacting Finite State Machines," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 418–421, Nov. 1991.

[2] S. Devadas, " Optimizing Interacting Finite State Machines Using Sequential Don't Cares," *IEEE Trans. Computer-Aided Design*, pp. 1473–1484, Dec. 1991.

[3] J. Kim and M. M. Newborne, " The Simplification of Sequential Machines With Input Restrictions," *IRE Transactions on Electronic Computers*, pp. 1440–1443, Dec. 1972.

[4] H.-Y. Wang and R. K. Brayton, "Input Don't Care Sequences in FSM Networks," in *Proc. Intl. Conf. on Computer-Aided Design*, 1993.

[5] H.-Y. Wang and R. K. Brayton, "Permissible Observability Relations in FSM Networks," in *Proc. of the Design Automation Conf.*, June 1994.

[6] Y. Watanabe and R. K. Brayton, "The Maximum Set of Permissible Behaviors for FSM Networks," in *Proc. Intl. Conf. on Computer-Aided Design*, 1993.

[7] J. Fron, J. C.-Y. Yang, M. Damiani, and G. D. Micheli, "A Synthesis Framework Based on Trace and Automata Theory," in *Workshop Notes of Intl. Workshop on Logic Synthesis*, (Tahoe City, CA), May 1993.

[8] H. Wong-Toi and D. L. Dill, "Synthesizing Processes and Schedulers from Temporal Specifications," in *Proc. of the Second Workshop on Computer-Aided Verification*, 1990.

[9] Z. Manna and P. Wolper, "Synthesis of Communicating Processes from Temporal Logic Specifications," *ACM Trans. Programming Languages and Systems*, vol. 6, no. 1, pp. 68–93, 1984.

[10] E. M. Clarke and E. A. Emerson, "Design and Synthesis of Synchronization Skeletons Using Branching Time Logic," in *Proc. Workshop on Logic of Programs*, vol. 131 of *Lecture Notes in Computer Science*, pp. 52–71, Springer-Verlag, 1981.

[11] A. Pnueli and R. Rosner, "On the Synthesis of a Reactive Module," in *Proc. ACM Symposium on Principles of Programming Languages*, pp. 179–190, 1989.

[12] W. Thomas, "Automata on Infinite Objects," in *Formal Models and Semantics* (J. van Leeuwen, ed.), vol. B of *Handbook of Theoretical Computer Science*, pp. 133–191, Elsevier Science, 1990.

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

[14] J. R. Burch, D. L. Dill, E. Wolf, and G. D. Micheli, "Modeling Hierarchical Combinational Circuits," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 612–617, Nov. 1993.

[15] J. R. Buchi, " On a Decision Method in Restricted Second Order Arithmetic," in *International Congress on Logic, Methodology, and Philosophy of Science*, pp. 1–11, 1960.

[16] A. Aziz, F. Balarin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Synthesizing Interacting Finite State Machines," Tech. Rep. UCB/ERL M94/96, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Dec. 1994.

[17] K. L. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[18] E. A. Emerson, "Temporal and Modal Logic," in *Formal Models and Semantics* (J. van Leeuwen, ed.), vol. B of *Handbook of Theoretical Computer Science*, pp. 996–1072, Elsevier Science, 1990.

[19] S. Safra, "Exponential Determinization for ω-Automata with Strong-Fairness Acceptance Condition," in *Proc. ACM Symposium on the Theory of Computing*, 1992.

[20] R. Alur and D. L. Dill, "Automata for Modelling Real Time Systems," in *International Colloquium on Automata, Languages and Programming*, 1990.

[21] R. Alur, L. Fix, and T. Henzinger, "A Determinizable Class of Timed Automata," in *Proc. of the Computer Aided Verification Conf.*, 1994.