Network Hierarchies and Node Minimization*

Area: Computer Hardware and Design

R. K. Brayton and E. M. Sentovich
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, CA 94720

Professor Robert K. Brayton	Ellen M. Sentovich
Room 519 Cory Hall	205 Cory Hall #117
Department of EECS	Department of EECS
University of California, Berkeley	University of California, Berkeley
Berkeley, CA 94720	Berkeley, CA 94720
(510) 643-5052 (FAX)	(510) 643-5052 (FAX)
(510) 643-9801	(510) 642-3626
brayton@ic.eecs.berkeley.edu	ellen@ic.eecs.berkeley.edu

*This work was supported by SRC, NSF/DARPA through grant MIP-8719546, the California MICRO Program, Bell Northern Research, and Fujitsu.

1 Introduction

Over the last decade there has been a great deal of research in the automatic synthesis of combinational logic for optimizing area, speed, testability and more recently power. Much of this has already found its way into the software of successful synthesis companies and large systems houses. Now automatic synthesis is used almost everywhere digital circuits are designed. The work continues with substantial and sometimes surprising advances made every year. As more progress and improved understanding is obtained, a coherent picture begins to emerge.

More recently, intense research has been focused on the automatic synthesis of sequential networks, i.e., those that contain state or memory elements. Although the picture here is not so clear, many of the paradigms for combinational logic have been able to be extended and a promising direction for future research can be at least mapped out. In this paper, we survey some of the results of combinational synthesis and some recent results for sequential synthesis and then use these to view possible avenues for future sequential synthesis research. In particular we look at two related questions,

- the problem of deriving and representing a set of permissible behaviors at a node or set of nodes in a network, and
- using an appropriate representation of these behaviors with an associated minimizer to search among these for one that best fits some optimization criteria.

We examine these two issues in increasingly complex situations starting with a single-output function, and proceeding to a single multiple-output function, a network of single-output functions, a network of multiple-output functions, and then similar questions where "function" is replaced by a finite state machine (FSM). We end with a discussion of a network of finite state machines and the problem of deriving the set of permissible FSM's and choosing a representative minimum one. In what follows, we will use the term **cover** to refer to a single completely defined representative among the set of permissible functions or behaviors.

2 Combinational Logic

The classical minimization problem in the area of combinational logic is two-level sum-of-products minimization where the objective is to find a representation of a function (or *cover*) with the least number of cubes (product terms). For two-level minimization, it is sufficient to consider only prime cubes (maximal implicants of the function) in the answer, so the search for the minimum is made much easier. The objective function of the number of product terms is appropriate for minimizing area and usually helpful in reducing delay, power and improving testability.

Usually this problem is formulated with don't cares given for the function. A don't care is an input vector for which a function's value can be either 1 or 0. This is usually given by the characteristic function of the don't care terms i.e., a function $d(x)$ where $d(x) = 1$ if for input minterm x the specified or derived value of $f(x)$ =1 or 0, i.e., either value is allowed. Thus in minimizing f, one has the option of choosing 1 or 0 for this value and in that sense, additional flexibility is given to the minimizer to make the choice which is best in meeting the minimization criteria. Only better answers can result by using don't cares.

As with all the problems we will discuss, one has a choice of the kind of minimizer to use, an exact one (which if it completes guarantees a global minimum) or a heuristic one which only guarantees a minimal answer (typically prime and irredundant). In this paper, we mainly discuss heuristic minimizers because of practical considerations.

2.1 The ESPRESSO Paradigm

The ESPRESSO paradigm [1] is a heuristic for finding a good minimal solution, given a current cover f and its don't care set d. It is based on iterating three operations on a current cover (sum-of-products). These are REDUCE, EXPAND, and IRREDUNDANT. EXPAND takes each cube at a time and expands it into a prime cube containing it. It tries to do this so that many other cubes in the present cover are totally covered by the new prime. After EXPAND is completed each cube in the cover is prime. IRREDUNDANT is then called to find a minimum subset of these cubes that is also a cover. At this point, the cover is then minimal but possibly far away from a global optimum. REDUCE is called to make some primes non-prime so that they can be expanded in a different direction to cover other reduced cubes. In this way, the paradigm allows a step away from a local optimum so that a more global one can be sought. Thus the paradigm makes small moves in the space of acceptable solutions (expanding or reducing a single cube). There is a check that each move is acceptable. In this case the check is easy; for EXPAND, the new cube p should be contained in $f + d$: $p \subseteq f + d$; for REDUCE, the present cover with the reduced cube replacing its prime counterpart p, should contain $f\overline{d}$. Thus the don't care function gives the upper and lower bounds for an acceptable solution; namely, g is permissible as a solution (cover) if and only if $f\overline{d} \subseteq g \subseteq f + d$. Consequently, the larger the don't care set d, the larger the space of permissible functions, and hence the more freedom for searching for an optimum function.

Multiple Output Functions For functions with more than one output, the method of solution extends in the natural way. Don't cares are given for each output: $d(x) = (d_1(x), \ldots, d_m(x))$. Multiple output primes are defined, and the paradigm works exactly in the same way. It is interesting to note here that the output part of a multiple-output function can be treated as just another input but with multiple values, one for each output. Thus the variable associated with the output part, x_{n+1}, takes on values in the set $V = \{v_1, \ldots, v_m\}$, and the function being minimized is a single-output function with $n + 1$ input variables. This function is the characteristic function giving all input/output pairs that are acceptable. An input $x = (x_1, \ldots, x_n, x_{n+1})$ is a don't care for this "characteristic function" if and only if $(x_{n+1} = v_i) \Rightarrow d_i(x_1, \ldots, x_n) = 1$.

Multiple-Valued Inputs Once the multiple-output function is treated as a single function with its last input as a multiple-valued variable, it is natural to generalize and extend all the inputs from binary- to multiple-valued variables. Thus x_i is in general a multiple-valued variable and can take on values in the set $P_i = \{v_1, \ldots, v_{k_i}\}$. Again the notion of primes and the ESPRESSO paradigm generalizes easily [2].[1]

2.2 Networks of Single-Output Functions

We have discussed the minimization of a single function in terms of finding a minimum sum-of-products representation. This two-level representation is not what we build in practice. Most logic is implemented in multilevel form, as a network of small interconnected gates. In most modern logic synthesis programs [3, 4, 5] this is represented by a Boolean network, a directed acyclic graph where each node is a single-output logic function of arbitrary complexity and an edge exists from node i and node j if the output of function i is an input of function j. This gives a technologically independent way of representing a logic computation without committing to a particular set of library gates in which it will be built. The Boolean network is assumed to have a set of **primary inputs** $x = (x_1, \ldots, x_n)$ and a set of nodes $z = (z_1, \ldots, z_m)$ whose outputs are the **primary outputs** of the network. In addition, a set of **don't cares** $d(x) = (d_1(x), \ldots, d_m(x))$ may be given. These are called the external don't cares. Note, it is implicitly assumed that the given don't cares are independent, i.e., each can be used independently of how the others are used. These are called **compatible** don't cares. This issue will appear later in the section on Boolean relations.

Thus, a given Boolean network with several outputs has the same specification as given for a set of two-level functions which we intend to optimize simultaneously, i.e., there is an initial cover (the given Boolean network) and a set of don't cares. The only difference is that the Boolean network representation (or cover) is a multiple-level net-list, whereas in minimizing the multiple two-level functions, we seek an implementation as a two-level sum-of-products, and the initial cover is a set of multiple-output cubes. In both cases, during synthesis we manipulate the cover to try to obtain a more optimal one.

Implied Don't Cares One of the strategies used for multilevel minimization is to use a two-level minimizer as a subroutine. This proceeds as follows. We focus on a single node, say node i, of the Boolean network with the node's current representation given by a cover of a single-output function f_i. There are no don't cares given *a priori* for this node, but it is possible to derive some from information about the surrounding network. Three kinds are derived, SDC (satisfiability don't cares), ODC (observability don't cares), and EXDC (external don't cares) [6]. The SDC are obtained

[1] Historically, these generalizations sometimes were done just because they could be done, but without any obvious use at the time. However, we have often seen that the existence of an efficient tool to solve a problem usually removes inhibitions and opens up the applications.

from the part of the network in the fain of the node. Because of this fain network, there may be input patterns on the immediate inputs of f_i which can never appear. Since they never appear, the output for f_i for this input pattern can be either 0 or 1, i.e., it is a don't care. The ODC occurs because of the part of the network separating the node from the primary outputs. For a primary input minterm x it may be that toggling the output of node i can not be seen at any one of the primary outputs, i.e., no primary output is also seen to be toggling when the inputs are held at x. In this case, x is a don't care for f_i and is called an ODC or observability don't care. The EXDC is just the given external don't cares for the network. The EXDC is used to restrict the computations done for SDC and ODC so that only the care inputs are used in their computation.

Is is conjectured but not known that SDC+ODC+EXDC does precisely characterize all the permissible functions that can be allowed at node i without changing the input/output behavior of the network (modulo the EXDC). We do know that it does characterize a large set in the following sense [6]. If we minimize each node in the network so that it is prime and irredundant using the don't care set DC = SDC+ODC+EXDC, then each connection in the network is irredundant, i.e., the network is 100% single stuck-at-1 and stuck-at-0 testable. In general, one may have to iterate this minimization process over all the nodes in the network, until no further changes occur, since after minimizing node i and then node j, it may be possible to further minimize node i.

It turns out that deriving the complete ODC for node i is a difficult problem [7] so in practice, only a subset of ODC is used [8, 9]; experience shows that a large and effective subset can be derived and used on most reasonable-sized networks. An effective and practical way to use this is built into the command **`full_simplify`** [10] used in the synthesis system SIS. Even though in practice one may not derive and use this complete set, it is important to understand and characterize what is possible. Afterwards one can restrict the computation to a subset in a knowledgeable way. Recent experiments using BDD's [11] show that in some cases use of the full ODC can be more efficient than use of compatible subsets. They were also slightly more effective.

2.3 Networks of Multiple-Output Functions

At this point, following the line of making a generalization if possible, the question arises about why it is necessary to restrict the nodes in a Boolean network to each having just a single output. This becomes relevant since the Boolean network itself has multiple outputs. Also related, is that often in high-level languages a computation is most naturally expressed with enumerated types, i.e., variables with multiple values. When encoded, these give rise to multiple binary-valued outputs. Another argument is that in many designs, the initial specification is given as a hierarchy of interacting components and each component can be a multiple-output function.

In this context, the problem arises again about how to derive and compute the complete don't cares for each component. Thus for component (or node) γ_i with m outputs, we may want to derive a set of don't cares $d(x) = (d_1(x), \ldots, d_m(x))$ which provide the flexibilities that can be used in seeking a minimum cover of γ_i. Once these are derived, a heuristic minimizer (like ESPRESSO) can be used on γ_i.

Three questions arise:

- how to obtain all the flexibility allowed, i.e., how to derive and represent the set of all possible functions permissible at γ_i,
- in what form can we represent this set, and
- do we have a suitable minimizer that can use this form to seek an optimum solution?

We saw in the case of a single function $f(x)$ with don't cares $d(x)$ given, that a subset (conjectured all) of permissible functions was $f\overline{d} \subseteq g \subseteq f + d$. One might similarly conjecture that, for γ_i with m outputs, the set of all permissible functions can be captured with don't cares. However, this conjecture is not true; the set of all permissible functions at a node cannot be captured by deriving a set of don't cares $d(x) = (d_1(x), \ldots, d_m(x))$. A more general form is required, namely Boolean relations.

Boolean Relations A Boolean relation is just a relation between inputs and outputs. Suppose x is the vector of inputs of component γ_i and y is the vector of its outputs. Then a Boolean relation $B(x,y)$ gives a set of allowable outputs y for each input vector x, i.e., the set $\{y|B(x,y)=1\}$ gives a set of allowed outputs for input x. Note that the situation $f(x)=01$ or $f(x)=10$ can be represented with a Boolean relation but cannot be represented with don't cares, since don't cares can be used only to represent sets of the form $f(x)=-1$,[2] $f(x)=-0$, $f(x)=1-$, $f(x)=0-$, none of which represent the set $f(x)=\{01,10\}$.

It was shown that for a network of components where each is a combinational function, the set of permissible functions for a component must in general be represented by a Boolean relation, and that a single Boolean relation is sufficient for this situation [12].

The next question is how to derive and represent the Boolean relation representing all permissible functions allowed at node i. With don't cares, it was possible to represent them with a cover of cubes. One way to represent a Boolean relation would be to give for each input minterm x, the list of acceptable output values $\{y_i\}$. Of course this is unacceptably complex. Fortunately, at this juncture in the development, binary-decision diagrams BDD's were coming into broad use and they provide the means of efficient derivation and representation of Boolean relations. BDD's are a canonical representation of a logic function. By using the notion of a characteristic function, a logic function can be used to represent any set. Thus for a Boolean relation, we use its characteristic function $f(x,y)$ to represent the set of all acceptable input/output pairs, i.e., input x with output y is allowed if and only if $f(x,y)=1$. Then we represent f by its BDD.

Another point is the initial statement of the problem. In dealing with don't cares, we assumed that the network specification was given by an initial cover with an associated set of external don't cares, one for each output. Now, it is more appropriate to have the network specification take the same form as the one representing the specification for the internal multiple-output node. In this way, the forms and representations used are closed under composition and decomposition. We can start with a Boolean network of multiple-output nodes with a Boolean relation giving the network specification, and then derive a new Boolean relation for the complete permissible functionality of an internal node. Each internal node may be in fact another Boolean network and so the theory works recursively on a hierarchy.

Deriving the Boolean Relation Consider the characteristic function of the Boolean relation of the given network η with n inputs $x=(x_1,\ldots,x_n)$ and m outputs $z=(z_1,\ldots,z_m)$. Denote this by $\mathcal{O}(x,z)$ and call it the *observability* function. The problem posed is to find the observability relation $\mathcal{O}_j(u,v)$ of an internal node η_j of the network η in terms of its local inputs u and outputs v. Thus we want to propagate the flexibility given for the entire network η to a flexibility for the internal node η_j. These flexibilities should be the set of all permissible functions that can be implemented at η and η_j respectively. A function $f: B^n \to B^m$ is permissible or *compatible* with $\mathcal{O}(x,z)$ if and only if for all x, $\mathcal{O}(x,f(x))=1$. This is written as $f \prec \mathcal{O}$. Typically, for practical purposes, $\mathcal{O}(x,z)$ is stored as a BDD and the derivation for $\mathcal{O}_j$ is made using efficient BDD operations. For the purpose of this paper we just mention one such formula to give the flavor of the derivation.

Let $L_1(x,u)$ give the relation of the mapping from the primary inputs x to the inputs u of node j. Let the outputs of node j be v, and $L_2(x,v,z)$ be the mapping from (x,v) to the primary outputs z. Then the formula for $\mathcal{O}_j(u,v)$ is

$$\mathcal{O}_j(u,v) = \mathcal{C}_x[\mathcal{C}_{\tilde{u},z}[\overline{L}_1(x,\tilde{u}) + \overline{L}_2(x,v,z) + \mathcal{O}(x,z)] + \overline{L}_1(x,u)]$$

Here $\mathcal{C}_x$ denotes the universal quantification of the variables x. For example, $\mathcal{C}_{x_i} = f_{x_i} f_{\overline{x}_i}$ where $f_{x_i} = f|_{x_i=1}, f_{\overline{x}_i} = f|_{x_i=0}$. In general, $\mathcal{C}_x f = \mathcal{C}_{x_1}(\mathcal{C}_{x_2}(\ldots(\mathcal{C}_{x_n} f)\ldots)$.

It is useful to think of $\mathcal{O}(x,z)$ as an automaton with inputs x,z and one binary output. In this case there are only two internal states, one designating that only acceptable inputs have been seen, and the other that a bad input has occurred. We transition to the bad state only under a bad input and stay there forever. Otherwise we stay at the good state. The automaton is the representation or specification of what is to be implemented. We shall see that this interpretation has a natural extension to sequential synthesis. The output of $\mathcal{O}$ is 0 or 1 and is just an indication of the occurrence or absence of a bad signal.

[2] '-' means don't care so this represents the set $f(x)=\{01,11\}$

The relation of $\mathcal{O}(x,z)$ with don't cares is enlightening at this point. Define for a function f its Boolean difference $\frac{\partial f}{\partial x_i} = f_{x_i} \oplus f_{\overline{x}_i}$. Then the don't care $d_i(x,z)$ at output z_i is

$$\overline{\frac{\partial \mathcal{O}(x,z)}{\partial z_i}}.$$

Note that this function depends on the other values of the outputs $z_j, j \neq i$. Thus the set

$$\left\{ \overline{\frac{\partial \mathcal{O}}{\partial z_j}} \right\}$$

of different don't cares for the outputs z_j are not independent or "compatible". In [13], a technique was given to derive a set of compatible don't cares from $\mathcal{O}$. These are easier to compute and to propagate through a Boolean network, and since they are compatible, can be used independently as required by two-level minimizers.

Minimizing the Boolean Relation Since a Boolean relation is more general than an incompletely specified function, the existing minimizers for minimizing functions with don't cares are not appropriate. Thus work was begun on a Boolean relation minimizer and several heuristic and exact minimizers have been reported [14, 15, 12, 16, 17].

One such is GYOCRO [15], a heuristic minimizer patterned on the ESPRESSO paradigm. The EXPAND, IRREDUNDANT, REDUCE iteration was examined and several interesting differences required that these basic operations be altered. The first was that during the expansion of a cube, intermediate cubes between the initial cube and a prime cube are not necessarily acceptable. This is unlike incomplete function minimization where if c is a cube in the current cover and p a prime covering c, then all $\tilde{c}$, $c \subseteq \tilde{c} \subseteq p$ are acceptable. However, for a Boolean relation it might be that c and p are acceptable but $\tilde{c}$, some $c \subseteq \tilde{c} \subseteq p$, is not, i.e., $\tilde{c}$ cannot be used to replace c in the current cover and still have a function compatible with the Boolean relation. Of course this makes the expansion process more difficult, but it can be done reasonably efficiently using BDD's to represent the functions involved. In GYOCRO, a computation is made which constructs a BDD for the set of all cubes $\tilde{c}$ that can replace c in the current cover. Then the one that can replace the most other cubes is chosen. Because of this same phenomenon, there is also no notion of a prime in the absolute sense. It might be that p is a maximal cube that can replace c in the current cover, but as the cover changes, another cube $\tilde{p}$, such that $\tilde{p} \supset p$ may then exist for replacing p.

The REDUCE operation is likewise more complicated. In taking a relative prime cube and replacing it with a reduced cube in the current cover, there is no unique maximally reduced cube like for functions. Similarly the set of reduced cubes $\tilde{c}$ that can replace p is not a connected set. For standard function minimization, it is possible to simply insert a literal in a cube and test if the resulting function is still a cover. If so, it is kept and another literal attempted. This is done until no more literals can be inserted. The result is the unique maximally reduced cube. The order of insertion of literals does not affect the final answer. For the REDUCE operation in GYOCRO the set of all cubes that can replace p in the current cover is computed and a heuristic is used to choose the best one. Again, BDD's come to the rescue in making these computations feasible. Experiments showed that GYOCRO is effective in that the minimizer can "jump" between different incompletely specified functions contained in the Boolean relation. Functions of medium size (say 10 to 30 variables) can be handled in reasonable time. This makes it possible to consider Boolean networks with multiple-output nodes where each node is small/medium-sized, and to use GYOCRO to minimize each internal node. Experiments were done on Boolean networks starting with single-output internal nodes. Multiple output nodes were formed by suitably clustering compatible nodes. Comparison with state-of-the-art multilevel minimizers showed roughly a 10% improvement [18].

An interesting compromise between using the more complex Boolean relation minimizer and a two-level function minimizer consists of grouping together a set of "compatible" gates to form a multiple-output node [19]. They show how to use an efficient function minimizer (not a relation minimizer) in this case and that this can lead to better results than normal methods. Thus one can use some of the flexibility provided by the Boolean relation to obtain a better answer without having to invoke a more complex relation minimizer.

2.4 Multiple Boolean Relations

An interesting classification of various ways to represent permissible behaviors for combinational logic is given in [20]. The most general set of functions is just an arbitrary subset of functions. Such sets can be compacted into multiple Boolean relations (MBR's) (or into multiple incompletely specified functions) such that a function is in the subset if and only if it is compatible with one of the relations (or compatible with one of the incompletely specified functions). Finding the smallest such sets can be related to finding a smallest cover of primes of a derived function. It was observed [20] that the set of permissible functions for some problems cannot be represented by a single Boolean relation. So indeed, multiple Boolean relations are required. The reason is that a Boolean relation or an incompletely specified function is a set of functions of a special type. For example, a Boolean relation can be said to be a set of functions that are "output correlated" but input uncorrelated. This means that the choice made for one of the outputs for a particular input affects what the other outputs can be for that same input, but does not affect any of the choices allowed for the other inputs. An example with input correlation is the set of two 2-output functions $\{f_1, f_2\}$ where $f_1(0) = 01, f_1(1) = 10$, $f_2(0) = 10, f_2(1) = 01$. This set cannot be represented with a single Boolean relation since the choice of 01 for input 0 forces the choice of 10 for input 1. Thus the input choices are correlated.

A multiple Boolean relation can be thought of as an automaton analogous to the observability relation automaton described in Section 2.3: the automaton has inputs x, z and one binary output, but in this case there are many internal states. All but one state indicate that only permissible input/output pairs or combinations of input/output pairs have occurred. The single non-accepting state indicates that a non-permissible input/output pair or combination of input/output pairs has occurred. A path in the automaton keeps track of input/output pair combinations and thus insures that the input correlations in the specification are satisfied. The resulting automaton is in general a non-deterministic finite automaton, under-scoring the expressibility of multiple Boolean relations for capturing sequential behaviors.

In [21], synchronous relations were defined for minimizing sequential circuits. A synchronous relation is derived by "unrolling" a circuit in time, and relating the time shifted variables. Thus $S(x^1, x^2, x^3, y^1, y^2, y^3)$ gives a relation between inputs x and outputs y where $x^1 = x(t), x^2 = x(t-1)$ etc. In this problem we seek a cover f such that $S(x^1, x^2, x^3, f(x^1), f(x^2), f(x^3)) = 1$. It was shown [20] that this problem can be formulated using MBR's but in general not with a single Boolean relation. Special minimizers for synchronous relations have been reported [22, 21], where the objective is to find a cover for f with a minimum number of cubes.

3 Sequential Logic

Sequential logic just means a combination of pure logic elements and storage (or state) elements, which we can think of as generic latches. Each generic latch is a state variable which may take on several values. In this setting, we use non-binary variables since internal state variables more naturally are given names and not necessarily a binary encoding. Since symbolic variables (i.e., multiple-valued variables) are effectively multiple binary variables, the parallel with the previous discussion on combinational logic is with functions with multiple outputs. This is one reason why the computational model for combinational logic with multiple outputs at each node is of special interest.

The computational model for sequential synthesis is one of a finite state machine network (FSM network). The concept is the same as a Boolean network, except that the network need not be acyclic, and each node in the network is a FSM instead of a pure logic function. Each FSM can have multiple symbolic inputs, outputs, and state variables. Each FSM is in general a Mealy machine.

3.1 Representing Behaviors by Nondeterministic Finite Automata

In place of a Boolean relation as the specification, we can use a finite (possibly nondeterministic) automaton (NFA). Similar to don't cares and Boolean relations, the specification automaton provides the flexibility for the FSM network to be synthesized.

An NFA, A, has a set of accepting states $\mathcal{F}$; the other states are non-accepting. To make things similar to the previous discussion, suppose there is one binary output with value 1 when a state is in $\mathcal{F}$ and 0 otherwise. The **language** of A (denoted by $\mathcal{L}(A)$) contains any string of inputs that never outputs a 0. Suppose the input to the automaton is partitioned into two parts $I = (x, z)$, where x and z represent the input and output of an FSM implementation of the automaton. The language of an FSM, M, (assume it is completely specified and deterministic, i.e., an "implementation") contains any sequence of inputs x and outputs z starting from its initial state s_0. Thus it is a string of the form $\sigma = ((x^0, z^0), (x^1, z^1), \ldots, (x^k, z^k))$. If σ can be produced by M, then we write $\sigma \in \mathcal{L}(M)$. If $\mathcal{L}(M) \subseteq \mathcal{L}(A)$, we say that M is compatible with A, and write $M \prec A$. In this sense A can compactly represent an entire set of implementable FSM's, and we can use such to represent a set of flexibilities allowed in implementing an FSM network. By allowing A to have non-deterministic next states, we enhance the ability to compactly represent even larger sets of FSM's. An NFA is the most general way we know for doing this.

3.2 Deriving the Specification Automaton for an FSM

We pose now a problem similar to what was done for combinational logic. Consider an FSM network η with a specification automaton A. Consider an internal node η_j. We want to derive the local automaton specification A_j for η_j given in terms of the local inputs u and outputs v of η_j. The derived automaton A_j should give precisely the set of all permissible behaviors allowed to be implemented at node η_j in network η, i.e., the implemented M_j is permissible if and only if its language is contained in the language of A_j, i.e., $M_j \prec A_j$. A_j contains the entire set of permissible FSM behaviors.

Some History - Don't Care Sequences Historically, this problem was attacked by trying to derive a set of *don't care sequences*, in analogy to what was done for multilevel logic networks. Similar to the SDC and ODC in a Boolean network, input and output don't care sequences were attempted [23, 24, 25, 26, 27]. Input don't care sequences are sequences of inputs that never appear at the inputs to the node η_j. Output don't care sequences are pairs of sequences for which the behavior of the network η is not altered if the output of η_j is changed from one of the pair to the other. An elegant solution to the input don't care sequences problem was given by Kim and Newborn [23] in terms of automata which were used to derive an incompletely specified machine for the internal machine. In that derivation, it is assumed that there are just two machines in η, M_1 and M_2 in cascade, with the outputs of M_1 feeding into M_2, but not vice versa. Kim and Newborn derived the set of permissible behaviors as an automaton which is incompletely specified but deterministic.

In a recent paper, Wang [27] also studied input don't care sequences but allowed an output of M_2 to feed back into M_1 so that general two-way communication is allowed. He showed that the Kim and Newborn construction can be generalized to this case and derived again an incompletely specified deterministic automaton which characterized precisely all permissible behaviors when input don't care sequences are taken into account.

Since in these cases, the derived machine is incompletely specified, one can use standard state minimization methods [28, 29], using the notion of compatible pairs of states to "minimize the machine": a minimum state machine for M_2 is derived. Of course a minimum state machine does not necessarily lead to the best implementation; state minimization is simply a good heuristic. Other techniques can also be used to search for a good implementation. However, the availability of good minimizers for incompletely specified FSM's is decidedly inferior to that for combinational logic.

The situation for characterizing output don't care sequences is even worse; in general it seems that MBR's are needed. It is not known in what simpler form these can be characterized, for example, by a single NFA. The paper by Wang [26] provides a good discussion about the state of this problem and some new heuristics to derive a subset of permissible behaviors.

It is quite surprising then that some recent work gave an answer for capturing all the input and output don't care sequences using a single NFA [30].

General Problem Consider the general situation. A is the NFA representing the specification automaton for machine η. η is an FSM network which is a "cover" for some machine $M \prec A$. We focus on a particular submachine of η, say η_j, and want to derive a specification automaton A_j characterizing any and all machines that are permissible, given that the rest of the network η is fixed. For this purpose, M_1 represents the machine at η_j and M_2 the rest of the machines of η.

We want to derive A_j such that $M_1 \prec A_j$ if and only if M_1 is permissible in η at node j, i.e., the product machine $M_1 \times M_2 \prec A$. The advantage of deriving A_j, if possible, is that, as for combinational logic, the number of inputs u, outputs v and internal states of A_j can be considerably smaller than for A and M_2 and hence more manageable in terms of searching for a small implementation of M_1.

The Set of Permissible Behaviors for η_j We illustrate the construction of the specification automaton A_j for node subnetwork η_j in the simpler case where A is deterministic. The transition relation $T_j(s_p, u, v, s_n)$ for A_j relating present states s_p, input u, output v, and next states s_n is obtained by a fixed point iteration. States of A_j correspond to pairs (s_2, s) where s_2 is a state of M_2 and s a state of A. There are two additional states of A_j, ϕ (the empty state) and κ (the non-accepting state). We start with the initial state of $A_j = \{(r_2, r)\}$ the initial state of $M_2 \times A$. T_j^0 is initialized with only self loops for states ϕ and κ. On iteration $t+1$, we compute $T_j^{(t+1)}$ by adding edges and states as follows. $T_j^{(t+1)}(s_p, u, v, s_n) = 1$ if and only if, with $s_p = (\tilde{s}_2, \tilde{s})$,[3]

1. $[\forall x, u \neq u_2(\tilde{s}_2, x, v)] \Rightarrow s_n = \phi$, or
2. $[\exists x, u = u_2(\tilde{s}_2, x, v), z_2(\tilde{s}_2, x, v) \neq z_A(\tilde{s}, x)] \Rightarrow s_n = \kappa$, or
3. $\forall x, (u = u_2(\tilde{s}_2, x, v)) \Rightarrow (z_2(\tilde{s}_2, x, v) = z_A(\tilde{s}, x))$ then for each x where $(u = u_2(\tilde{s}_2, x, v)), s_n = (s_2(\tilde{s}_2, x, v), s_A(\tilde{s}, x))$.

Thus, a new next state s_n is constructed for each state of $M_2 \times A$ that can be reached from s_p under u, v and any x (condition 3) provided there does not exist an x for u, v such that the z outputs do not agree but where x is such that $u = u_2(\tilde{s}_2, x, v)$. In that case the transition is made to the non-accepting state κ (condition 2). Note that the automaton A_j is non-deterministic because of 3 above, since there may be many next states s_n associated with s_p, u, v; each possible x may generate a different next state s_n.

The above equations for $T_j^{(t+1)}$ can be translated easily into a set of equivalent BDD computations for the characteristic function of $T_j^{(t+1)}$. The iteration stops when no new next states s_n are reached. The final automaton A_j has one non-accepting state, κ, and the language of A_j bounds the set of permissible machines at η_j, i.e., a machine M_1 is permissible at η_j only if $\mathcal{L}(M_1) \subseteq \mathcal{L}(A_j)$. However, A_j may contain behaviors that are not permissible. Thus another construction is required to trim A_j down to where it is precisely the set of permissible behaviors. This requires a technique which is similar to a subset construction for determinizing a nondeterministic machine. States of the new $\tilde{A}_j$ are now subsets of states of A_j where subsets containing the state κ have been deleted. This construction is given in [30] and results in a machine called the E-machine. Also there, a similar construction is given for A_j when A, the specification automaton, is allowed to be nondeterministic.

Example: *The construction of the E-machine is illustrated with an example. The interconnection structure is shown in Figure 1.*

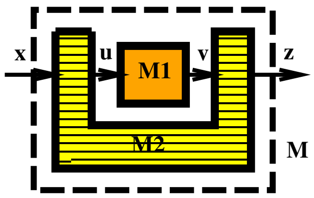

Figure 1: M is viewed as two interconnected machines.

Two deterministic machines, M and M_2 are given (whose transition structures are shown in Figure 2).

We want to derive a specification E containing the permissible behaviors for M_1. The construction is done as one fixed point computation with the subset construction included. Thus each state added is, in general, a subset of pairs of states (s_2, s). Figure 3 shows the first step from the initial state $(1, A)$. Consider the transition from $\{(1, A)\}$ under $u/v = 1/0$. M_2 can either go from 1 to 2 with $x/z = 0/0$ or from 1 to 1 with $x/z = 1/1$. In the first case, $x/z = 0/0$ moves M from A to B (while $M2$ goes to 2). In the second case $x/z = 1/1$ moves M from A to A (while $M2$ goes to 1).

[3] We use the notation $u_2(\tilde{s}_2, x, v)$ to denote the output u of M_2 and $z_2(\tilde{s}_2, x, v)$ to denote the output z of M_2. Similarly $z_A(s, x)$ denotes the z value related to s, x in A. Finally, the functions s_2 and s_A give the next state functions of M_2 and A respectively.

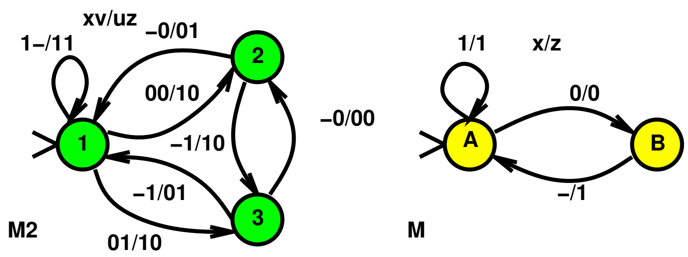

Figure 2: Specification for $M2$ and M.

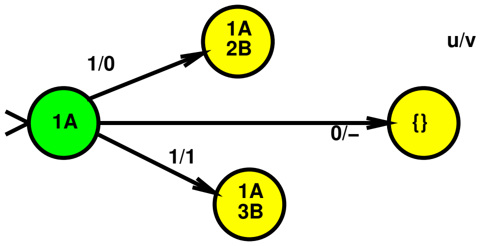

Figure 3: The first set of transitions from the initial state.

Thus the new state of the E-machine under input $u/v = 1/0$ is the set $\{(1, A), (2, B)\}$. The other transitions are arrived at similarly.

Figure 4 shows the second step, which is the final one, since no new states are added. Note that a node with κ is not shown, since it is automatically eliminated as it would be if the subset construction were to be done explicitly.

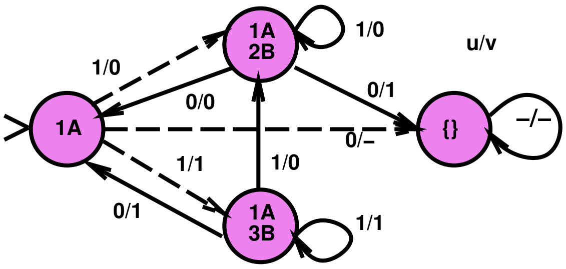

Figure 4: The second set of transitions away from the new states.

3.3 Minimizing A_j

We now have the difficult problem of finding a "minimum" permissible machine. Usually this means that we want to find a completely specified deterministic FSM, M_1, (an implementation) with the least number of states whose language $\mathcal{L}(M_1) \subseteq \mathcal{L}(A_j)$. Good algorithms for this are basically nonexistent. The problem of finding a minimum state automaton equivalent to a given NFA is known to be P-space hard. However, this is not our problem here. Instead we want a minimum state machine contained in a given NFA. The distinction is that in the former we want to keep all behaviors while in the latter, we are looking for only one behavior contained in the NFA.

A key observation, made in [31], is that the E-machine is a pseudo non-deterministic FSM (PNDFSM), meaning that its corresponding automaton, constructed by merging input and output symbols into single input symbols on the edges, is deterministic. For instance, in the example, the corresponding automaton for the E-machine in Figure 4 is shown in Figure 5.

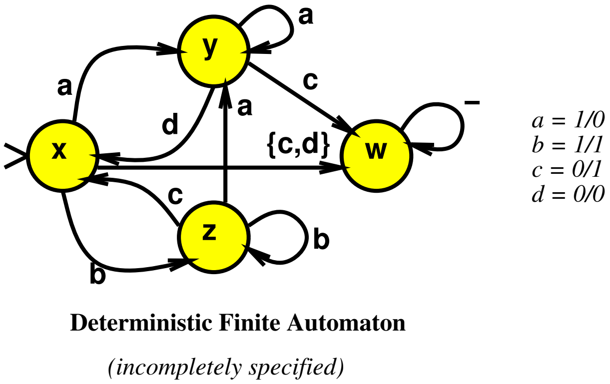

Figure 5: DFA derived from NDFSM in Figure 4.

PNDFSM's are as expressive as NDFSM's but with possibly exponentially many more states. However, they have some nice properties which make them more easily minimized, so that the minimization of a contained behavior can proceed in a way similar for incompletely specified machines. Thus the notion of a compatible set of states is defined. This is a set of states that are output compatible, plus an output function that makes them compatible. A feasible machine is defined as any machine composed of compatible states, with a transition between a present state compatible and a next state compatible if for all states contained in the present state compatible, there is a next state in the E-machine contained the next state compatible. It is proved that every behavior in a PNDFSM can be represented by a feasible machine. Further, a minimum state machine contained in a PNDFSM is a feasible one. This implies that we need only search among the feasible machines for a minimum contained machine. Figure 6 shows a set of compatibles of the E-machine of our example. Some of the compatibles are listed on the right. The compatible $(\{y, w\}, -/0)$ is a compatible because the output generated at states y and w can be determinized in such a way that states y and w can be merged to a single state with output function $-/0$; this means that the output function at that merged state is $f(u) \equiv 0$. Thus all of the compatibles shown could be part of a Moore machine because each output function is independent of the input u.

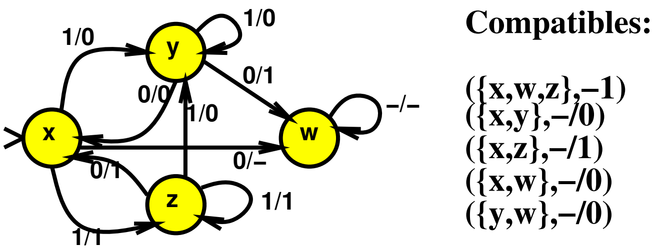

Figure 6: The E-machine and some of its compatibles.

Figure 7 shows a set of feasible machines composed of sets of these compatibles. Each feasible machine is a deterministic machine (possibly incompletely specified) and represents behaviors which are permissible, and hence acceptable implementations of M_1.

In [31], both exact and heuristic minimization procedures are presented for generating a solution, and again the ESPRESSO paradigm is employed in a heuristic search procedure. Still, these techniques produce only a minimum-state

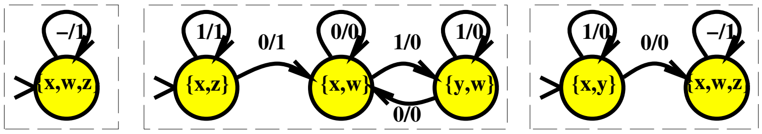

Figure 7: Machines composed of sets of compatibles.

machine, which does not necessarily lead to a minimum-sized implementation. The number of states of the STG is only a crude measure of the size of the final implementation, and algorithms for state minimization are thus only rough heuristics. Obviously, much more research needs to be done in this area.

4 Conclusions

The paradigm of having a hierarchy of functionalities and then summarizing the set of all permissible behaviors at a component of this hierarchy was discussed starting with pure combinational logic networks and ending with FSM networks. Methods were given for extracting and representing the set of all permissible behaviors. In some cases, minimization algorithms exist for operating on this set to obtain a good implementation. In other cases, no such minimizers exist. Future research should deal with either obtaining new minimization methods for the most general form of this set, or reasonable heuristics which effectively restrict the set of behaviors to a form so that existing minimizers are applicable to the set.

Even so, the story is not over here. The "ultimate" synthesis problem is one associated with design refinement. In this, a specification automaton is given as an ω-automaton with specified general "fairness" constraints. Such automata can represent certain properties that should hold for the final design. An example property is a sequencing constraint such as "if r_1 occurs before r_2 then always s_1 occurs before s_2", i.e., the order of requests and services is never mixed up. Then the problem is given a property automaton A_i, to refine an abstract version of component η_j so that it satisfies the properties desired. One wants to derive all such permissible behaviors by deriving an ω-automaton A_j for this. Then the problem is to synthesize one (the best one) of the acceptable behaviors (these are the ones that satisfy the properties) contained in this automaton.

References

[1] R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A.L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[2] R. Rudell and A. Sangiovanni-Vincentelli. Exact Minimization of Multiple-Valued Functions for PLA Optimization. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 352–355, November 1986.

[3] R.K. Brayton, R. Rudell, A.L. Sangiovanni-Vincentelli, and A.R. Wang. MIS: A Multiple-Level Logic Optimization System. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1062–1081, November 1987.

[4] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report Memorandum No. UCB/ERL M92/41, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, May 1992.

[5] E.M. Sentovich, K.J. Singh, C. Moon, H. Savoj, R.K. Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *Proceedings of the International Conference on Computer Design*, pages 328–333, October 1992.

[6] K.A. Bartlett, R.K. Brayton, G.D. Hachtel, R.M. Jacoby, C.R. Morrison, R.L. Rudell, A.L. Sangiovanni-Vincentelli, and A.R. Wang. Multi-level Logic Minimization Using Implicit Don't Cares. *IEEE Transactions on Computer-Aided Design*, CAD-7(6):723–740, June 1988.

[7] M. Damiani and G. DeMicheli. Derivation of Don't Care Conditions by Perturbation Analysis of Combinational Multiple-Level Logic Circuits. In *Proceedings of the International Workshop on Logic Synthesis*, North Carolina, May 1991.

[8] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The Transduction Method - Design of Logic Networks Based on Permissible Functions. *IEEE Transactions on Computers*, C-38(10):1404–1424, October 1989.

[9] H. Savoj and R.K. Brayton. The Use of Observability and External Don't Cares for the Simplification of Multi-Level Networks. In *Proceedings of the* 27^{th} *Design Automation Conference*, pages 297–301, June 1990.

[10] H. Savoj, H.-Y. Wang, and R.K. Brayton. Improved Scripts in MIS-II for Logic Minimizaton of Combinational Circuits. In *Proceedings of the International Workshop on Logic Synthesis*, North Carolina, May 1991.

[11] Yosinori Watanabe. *Logic Optimization of Interacting Components in Synchronous Digital Systems.* PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, April 1994.

[12] R.K. Brayton and F. Somenzi. Boolean Relations and the Incomplete Specification of Logic Networks. In *Proceedings of the International Conference on VLSI*, pages 231–240, August 1989.

[13] H. Savoj, R.K. Brayton, and H.J. Touati. Extracting Local Don't Cares for Network Optimization. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 514–517, November 1991.

[14] R.K. Brayton and F. Somenzi. An Exact Minimizer for Boolean Relations. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 316–319, November 1989.

[15] Y. Watanabe and R.K. Brayton. Heuristic Minimization of Multiple-Valued Relations. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 126–129, November 1991.

[16] A. Ghosh, S. Devadas, and A.R. Newton. Heuristic Minimization of Boolean Relations using Testing Techniques. In *Proceedings of the International Conference on Computer Design*, September 1990.

[17] B. Lin and F. Somenzi. Minimization of Symbolic Relations. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 88–91, November 1990.

[18] Y. Watanabe, L. Guerra, and R.K. Brayton. Logic Optimization with Multioutput Gates. In *Proceedings of the International Conference on Computer Design*, pages 416–420, October 1993.

[19] M. Damiani, J. Yang, and G. DeMicheli. Optimization of Combinational Logic Circuits Based on Compatible Gates. In *Proceedings of the* 30^{th} *Design Automation Conference*, pages 631–636, June 1993.

[20] E.M. Sentovich, V. Singhal, and R.K. Brayton. Multiple Boolean Relations. In *Proceedings of the International Workshop on Logic Synthesis*, Tahoe City, California, May 1993.

[21] M. Damiani and G. De Micheli. Recurrence Equations and the Optimization of Synchronous Logic Circuits. In *Proceedings of the* 29^{th} *Design Automation Conference*, pages 556–561, June 1992.

[22] V. Singhal, Y. Watanabe, and R.K. Brayton. Heuristic Minimization of Synchronous Relations. In *Proceedings of the International Conference on Computer Design*, pages 428–433, October 1993.

[23] J. Kim and M. Newborn. The Simplification of Sequential Machines with Input Restrictions. *IEEE Transactions on Computers*, C-21:1440–1443, December 1972.

[24] S. Devadas, H.K. Ma, A.R. Newton, and A. Sangiovanni-Vincentelli. Irredundant Sequential Machines via Optimal Logic Synthesis. *IEEE Transactions on Computer-Aided Design*, 9(1):8–18, January 1990.

[25] J.-K. Rho, G. Hachtel, and F. Somenzi. Don't Care Sequences and the Optimization of Interacting Finite State Machines. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 418–421, November 1991.

[26] H.Y. Wang and R.K. Brayton. Permissible Observability Relations in Interacting Finite State Machines. In *Proceedings of the International Workshop on Logic Synthesis*, Tahoe City, California, May 1993.

[27] H.-Y. Wang and R.K. Brayton. Input Don't Care Sequences in FSM Networks. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 321–328, November 1993.

[28] M. Avedillo, J. Quintana, and L. Huertas. New Approach to the State Reduction in Incompletely Specified Sequential Machines. In *Proceedings of the International Symposium on Circuits and Systems*, 1990.

[29] J.-K. Rho, G.D. Hachtel, F. Somenzi, and R.M. Jacoby. Exact and Heuristic Algorithms for the Minimization of Incompletely Specified State Machines. *IEEE Transactions on Computer-Aided Design*, 13(2):167–177, February 1994.

[30] Y. Watanabe and R.K. Brayon. The Maximum Set of Permissible Behaviors for FSM Networks. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 316–320, November 1993.

[31] Y. Watanabe and R.K. Brayton. State Minimization of Pseudo Non-Deterministic FSM's. In *Proceedings of the European Conference on Design Automation*, 1994.