DON'T CARE MINIMIZATION OF MULTI-LEVEL SEQUENTIAL LOGIC NETWORKS

by

Bill Lin and A. Richard Newton

Memorandum No. UCB/ERL M90/45

25 May 1990

COVER PAGE

DON'T CARE MINIMIZATION OF MULTI-LEVEL SEQUENTIAL LOGIC NETWORKS

by

Bill Lin and A. Richard Newton

Memorandum No. UCB/ERL M90/45

25 May 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

DON'T CARE MINIMIZATION OF MULTI-LEVEL SEQUENTIAL LOGIC NETWORKS

by

Bill Lin and A. Richard Newton

Memorandum No. UCB/ERL M90/45

25 May 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Don't Care Minimization of Multi-Level Sequential Logic Networks

Bill Lin A. Richard Newton
University of California, Berkeley, CA 94720
Correspondence: Bill Lin
Email: billlin@ic.Berkeley.EDU
Category: Logic Synthesis

May 23, 1990

Abstract

The use of don't cares in synthesis has been recognised to have a profound effect on area reduction, testability, and performance improvement. We address in this paper their use in sequential multi-level logic synthesis. Techniques for computing internal don't care sets for combinational networks are becoming mature. These don't cares can be exploited using well developed multi-level simplification techniques. When sequential networks are considered, many new possibilities exists. We describe an approach based on the extraction of sequential don't care conditions that arise in the context of sequential networks. The key to our approach is the use of efficient implicit state space enumeration techniques and multi-level combinational simplification procedures. Our contributions in this paper are: a) Application of BDD-based implicit state space enumeration proposed by Coudert, Berthet, and Madre for the extraction of sequential don't care conditions. Only the computation of invalid states is a trivial extension of Coudert's breadth first traversal algorithms. New novel algorithms are required and presented for the efficient computation of other classes of sequential don't cares. b) Re-computation of incompletely specified don't care conditions from high-level specification. c) Introduction of a new class of sequential don't care conditions called **operation don't cares** that are due to the intended operating nature of the circuit. d) Extensions of combinational multi-level simplification techniques to exploit sequential don't cares. e) Finally, we remark on the impact of this work on sequential redundancy removal and testability. Preliminary results using a subset of the proposed techniques are promising.

1 Introduction

This paper is concerned with techniques for don't care minimization of multi-level sequential logic networks. The importance of don't cares has emerged as a critical one in synthesizing combinational and sequential logic. Don't care simplification may be iterated with global restructing techniques for obtaining a highly optimized solution. In addition to reducing area, and possibly improving performance, it is well known that an intimate relationship exists between don't care exploitation and testability [2, 19, 14]. In [14], it was recognized that the optimal use of don't cares in minimizing a finite state machine will lead to a 100% fully non-scan testable machine.

Over the past few years, the problem computing efficiently and correctly internal don't care conditions that arise in combinational logic networks has been studied extensively [2, 24, 18, 25, 11]. Two classes of don't cares have been identified. The first is the Satisfiability Don't Care (SDC) set, which has to do with input conditions that can never occur due to the structure of the network. A much more difficult set to compute efficiently and correctly is the Observability Don't Care (ODC) set. These don't cares occur because the observability of the internal node at the primary outputs is limited the network output structure. When optimizing multi-level sequential logic networks composing of combinational logic gates and latches, many new possibilities exist. We are interested in answering the following question. Given an internal node of a sequential network, what are possible functions that we can use replace this node? Combinational don't cares are not enough because of the sequential nature of the circuit.

There are several viable ways of attacking this problem. One is to extend the don't care set computation algorithms developed for combinational networks into the sequential domain. This represents a *structural approach* and was explored by Damiani and De Micheli at Stanford [12]. A potential disadvantage of this approach is that well-developed combinational techniques cannot be directly reused. It is also unclear whether all the sequential don't care conditions can be derived this way.

An alternative approach is to first compute the **unreachable** or **invalid** states, and the **equivalent** states of the machine. If a state is invalid, then it is not important what the output or next state behavior of the machine is starting from this state. This represents an important sequential don't care condition that can be exploited very often. Consider for example a 1-hot encoded machine, only a very small subset of the state space is ever reached. In fact, it is standard practice to use invalid states as don't cares in state assignment. Equivalent states are also quite important. Given an input condition, the next state response can in fact be any one of the equivalent states. Both of these classes of don't care conditions can be directly utilized by combinational don't care techniques by treating them as **external don't cares**. For example, powerful techniques for computing the ODC can be used to propagate the external don't care conditions to the internal nodes [1]. A key advantage of this approach is that well-developed combinational techniques can be used directly.

Devadas *et al* [14] made use of the invalid and equivalent don't care set in their work on synthesizing sequentially irredundant machines. Invalid and equivalent don't cares were iteratively extracted and used by a two-level Boolean minimizer like ESPRESSO [4]. It was shown that the machine is 100% sequential

[1] As will be explained later, invalid input conditions may be better handled by simply incorporating them into the satisfiability don't care set.

testable if this process is carried out to convergence [2]. A basic limitation of this approach is that a state transition graph (STG) must first be extracted. A STG can easily blow up for sequential logic networks since every state must be explicitly represented and the combinational logic is represented by two-level covers. In practice, this approach is not applicable to a large class of sequential logic networks, especially those containing large number of latches and data path like components.

This problem can be partially circumvented by extracting an interacting network of finite state machines rather than just a single lumped machine. Algorithms for extracting the finite state machine networks and computing invalid and equivalent states were proposed in [1]. However, this technique is still limited by the complexity of some combinational components and the state space complexity of the machine interactions.

We also use the method of extracting invalid and equivalent don't cares, and then using well-developed combinational don't care simplification techniques to exploit them as external don't cares. However, we attempt to compute these sequential don't care sets directly from the sequential network without first explicitly extracting a state transition graph (STG). A key advantage of this general approach is that it can take full advantage of existing combinational methods.

In two recent papers, Coudert, Berthet, and Madre [9, 10] proposed novel implicit breadth first search traversal algorithms for exploring the state space using binary decision diagrams (BDDs) [7]. They were the first to develop and recognize the importance of this idea. The main power of their method is that a large set of states is traversed at a time and the logic is represented using BDDs. Their result demonstrate that state space information can be computed for state machines with large number of latches (millions of states) and data path like combinational logic (eg. adders). The use of BDDs to represent state sets has several additional attractive properties. They make set computations very efficient. To traverse the state space, two fundamental operations must be computed efficiently. The first is the range operator which computes the image of a set of function given a restricted domain. The second is the inverse operation of computing the inverse image given a restricted range. Using an efficient BDD based implementation of these operators, the entire set of reachable states of a sequential machine can be easily computed.

We make use of these techniques to extract the invalid and equivalent don't care sets from the sequential network. In practice, it is possible to compute these don't care sets for circuits with large number of latches and complex combinational logic. The procedure for extracting invalid states is a straightforward adaptation of Coudert's breadth first traversal algorithm [3]. However, the computation of equivalent states is not obvious. A straightforward naive approach to extracting equivalent states would require $O(2^n!)$ comparisons even when using BDDs. We have developed a new novel procedure for efficiently computing the equivalent state don't care set.

Another important source of don't care conditions is due to the fact that real life designs are incompletely specified. This is especially true when the sequential logic network is compiled from a high-level hardware description language (HDL). The incompletely specified don't care conditions are lost when structural changes (eg. retiming) are applied. Re-computing these don't care sets is

[2]Note that exact two-level minimization is not required for testability, only that a locally prime and irredundant solution is obtained at each iteration.

[3]It is simply the complement of the reachable states

extremely important for achieving a globally optimal implementation with respect to an incompletely specified hardware description. An efficient algorithm is given for this extraction process.

In addition to the above sequential don't care conditions, we propose a completely new class of don't cares that has to do with the **intended operating nature** of the sequential circuit, which we refer to as **operation don't cares**. Consider a pipelined circuit. Starting from the initial state, suppose that we are not interested in the output behavior of circuit for the first α cycles while the pipeline is being loaded. This corresponds to a don't care condition that can be exploited in synthesis. We define this class of sequential don't cares as **Latency Don't Cares** (LDC). Another source of operation don't cares has to do with the **sampling rate** of the circuit. Suppose we have a sequential circuit obtained from high-level synthesis after scheduling and allocation [22]. The functional behavior of the design may be broken down several cycles to compute such that the primary outputs are only sampled every τ cycles. Then, we don't care about the values at the primary outputs during the in between cycles. We define this class of don't cares as **Multi-cycle Don't Cares** (MDC). Other more complicated operating don't care conditions can be envisioned. We give a general procedure for computing these classes of don't cares. Like invalid and equivalent state don't cares, these don't care conditions may be exploited by combinational techniques in the form of external don't cares. It is worth noting that these don't cares cannot easily be extracted from state graphs nor can they be easily computed using a structural approach.

Finally, we remark on the impact of this work on the problem of sequential redundancy removal and testability.

2 Definitions and Notation

2.1 Basic definitions

A **variable** represents a simple coordinate of the Boolean space (eg. a); a **literal** is a variable in its true (eg. a) or negated (eg. $\overline{a}$) form; a **cube** is a set C of literals such that $x \in C$ implies $\overline{x} \notin C$ (eg. $ab\overline{c}$ is a cube, and $a\overline{a}$ is not a cube); a **cover** or a **function** f is a set of cubes representing a Boolean expression. It can be written in the sum of product form. The cover f is said to be **contain** by g, written $f \preceq g$, if $x \in f$ implies $x \in g$. If the output value for every point in the domain is defined, then the function is said to be **completely specified**; otherwise, it is said to be **incompletely specified**. The **Hamming distance** $\Delta(c_i, c_j)$, between two cubes c_i and c_j, is equal to the number of bit positions where the corresponding entry of one cube is a 1 the other is a 0. Two cubes c_i and c_j are said to **intersect** if $\Delta(c_i, c_j) = 0$; otherwise, they are **disjoint**.

2.2 FSM and sequential network model

A **finite state machine** (FSM) is defined as a 6-tuple $M = (I, O, \Sigma, \delta, \lambda, \varphi)$ where $I = B^n$ represents the primary input space, $O = B^m$ represents the primary output space, $\Sigma = B^s$ represents the state space, $\delta : I \times \Sigma \rightarrow \Sigma$ is the next state function, $\lambda : I \times \Sigma \rightarrow O$ is the output function, and $\varphi : I \times \Sigma \rightarrow \Sigma \times O$ is the incomplete specification don't care function. A set of legal **reset states** $\dot{\sigma} \subseteq \Sigma$ of the machine is assumed to be initially given. The machine M is said to be **incompletely specified** if $\varphi \neq \emptyset$.

A finite state machine can be implemented in the form a **sequential logic network,** which is a multi-level representation of synchronous sequential behavior. It is specified by a directed graph $G = (V, E)$, where each $v_k \in V$ is associated with a cover f_k or a synchronous latch l_k. It is associated with a unique reference variable y_k. There is a directed edge e_{ij} from v_i to v_j if there is a direct connection from node i to node j. The variable y_i is said to be a **fanin** of f_j (l_j), and the Function (latch) f_j (l_j) is said to be a **fanout** of y_i. The **support** of f_k (l_k) is the set of variables that f_k (l_k) explicitly depends on. A special set of nodes are classified as the **primary inputs** and **primary outputs** of the network. They must be associated with an identity function. A external don't care network φ, which in itself is a multi-level combinational network, is also associated with the specification of the sequential machine.

2.3 Binary decision diagram

Bryant's binary decision diagrams (BDD) are an extremely important contribution to the synthesis and verification community [7]. Recently, Brace *et al* [6] implemented an efficient BDD package using a strong canonical form and caching schemes. It has been shown that this implementation may be an order of magnitude faster than the original implementation.

A BDD is a directed acyclic graph (DAG) representation of logic. It is a binary decision graph where each node is associated with a variable and two fanouts. One fanout corresponds to when the variable is set to 0, and the other corresponds to when the variable is set to 1. At the leaves of the graph are two constant nodes representing the constants 0 and 1. A variable ordering is imposed such that all transitive fanouts of node must have a higher ordering index, except for the constant nodes. Also, a variable may not be repeated in a single directed path. The BDD is said to be reduced if there are no isomorphic subgraphs. The interested reader is referred to [7] for more details.

2.4 Set computation and BDD operators

We now give the basic working definitions necessary to understand the algorithms developed in this paper. [4]

Definition 2.1 *Let $F : B^n \rightarrow B^m$. Let $\xi \subseteq B^n$ be a subset of the input space. Then the* **image** *of ξ, written $F(\xi)$, is the set of elements $y \in B^m$ such that $\exists x \in \xi$ where $y = F(x)$.*

Definition 2.2 *Let $F : B^n \rightarrow B^m$. Let $\psi \subseteq B^m$ be a subset of the output space. Then the* **inverse-image** *of ψ, written $F^{-1}(\psi)$, is the set of elements $x \in B^n$ such that $\exists y \in \psi$ where $y = F(x)$.*

Definition 2.3 *Let $F : B^n \rightarrow B^m$. Let $x_1, \ldots, x_n$ be the input variables and $y_1, \ldots, y_m$ be the output variables. The* **characteristic** *or* **consistency** *function of F, written χ_F is defined as follows:*

$$\chi_F = \prod_{i=1}^{m} y_i \equiv f_i(x_1, \ldots, x_n). \tag{1}$$

It is also referred to as the **transition relation.** *By definition:*

$$\chi_F \subseteq B^n \times B^m.$$

[4] In some mathematical text, the terminologies domain and co-domain, and range and co-range are often used. Also, inverse-image is sometimes referred to as reverse-image.

Definition 2.4 *Let $f : B^n \to B$, and $x = \{x_1, \ldots, x_k\}$ be a subset of the input variables. The* **smoothing** *of F with respect to x is defined as follows:*

$$\begin{aligned} S_{x_i} f &= f_{x_i} + f_{\overline{x_i}}, \\ S_x f &= S_{x_1}(S_{x_2}(\ldots(S_{x_k}))) \end{aligned}$$

where f_{x_i} is the cofactor operation defined by [7] with respect to the literal x_i. It is also called the **existential** *operator.*

Definition 2.5 *Let $f : B^n \to B$, and $x = \{x_1, \ldots, x_k\}$ be a subset of the input variables. The* **consensus** *of F with respect to x is defined as follows*

$$\begin{aligned} C_{x_i} f &= f_{x_i} \, f_{\overline{x_i}}, \\ C_x f &= C_{x_1}(C_{x_2}(\ldots(C_{x_k}))) \end{aligned}$$

It is also called the **universal** *operator.*

The terminologies *smoothing* and *consensus* were introduced by [23] in his timing work. The terminologies *existential* and *universal* operators are common terminologies used in first order predicate logic and were also used by [10, 8] in their verification work.

Definition 2.6 *Let $f : B^n \to B$, and $x = \{x_1, \ldots, x_k\}$ be a subset of the input variables. The* **Boolean difference** *of f with respect to x is defined as follows*

$$\frac{\partial f}{\partial x_i} = f_{x_i} \oplus f_{\overline{x_i}}.$$

The cofactor operation with respect to a cube was previously well defined [7]. However, the solution is not unique when cofactoring with respect to another function. Coudert *et al* [9] defined an operator called the **constrained operator** which is essentially a cofactor operation with respect to functions. A definition of the (generalized) cofactor operator is given here [5].

Definition 2.7 *Let $F : B^n \to B^m$ and $\pi : B^n \to B$.* **F cofactor** *with respect to π, written F_π, is defined as follows:*

$$F_\pi(x) = \begin{cases} F(x) & \text{if } \pi(x) = 1 \\ y \in B^m & \text{where } \exists v \in B^n : \pi(v) = 1 \wedge y = F(v) \end{cases} \tag{2}$$

Given a fixed variable ordering, Coudert's implementation of the operator is deterministic. All of the above operators can be implemented efficiently in BDDs.

3 Sequential Don't Care Computation

Algorithms for computing sequential don't care sets are presented in this section. They make heavy use of BDD-based implicit symbolic computation techniques for computing the image of a function given a restricted domain, and the inverse-image given a restricted range. We shall defer to Section 6 to describe efficient implementations of these techniques. For the moment, we will assume the operations IMAGE (f, ξ) and INVERSE_IMAGE (f, ψ) exist.

[5]The notion of a generalized cofactor was introduced by Brayton and his students.

```
Algorithm EXTRACT_INVALID_STATES ($\delta$, $\dot{\sigma}$):
1.    begin
2.        $current = \dot{\sigma}$; $valid = 0$;
3.        while ($current \not\preceq valid$) do
4.            $valid = valid \vee current$;
5.            $current =$ BETWEEN ($current$, $valid$);
6.            $new =$ IMAGE ($\delta$, $current$);
7.            $current =$ CONVERT ($new$);
8.        endwhile
9.        return ($\overline{valid}$);
10.   end.
```

Figure 1: Extraction of invalid states.

3.1 Invalid state don't cares

The extraction of invalid states, states that cannot be reached from the initial state, is relatively straightforward using Coudert's breadth first traversal algorithm. Since BDDs are used to represent the reachable set of states, the set of invalid states is simply the complement. Let the sequential machine be defined by $M = (I, O, \Sigma, \delta, \lambda, \varphi)$, where $\delta : I \times \Sigma$ is the next state function. Let $\dot{\sigma} \subseteq \Sigma$ be the set of legal reset states. The procedure for computing the invalid states is outlined in Figure 1. At each iteration, the set of states reachable from the current set is computed (Line 6). The image is represented in terms of the next state variables [6]. The routine CONVERT at Line 7 simply translates the state set in terms of next state variables to present state variables. The routine BETWEEN(*current*, *valid*) at Line 5 returns a set γ such that *current* $\preceq \gamma \preceq$ *valid*. This technique was described in [9, 8] and was referred to as *frontier set simplification*. A straightforward approach would compute γ = *current* - *valid*. However, during the breadth first traversal loop, a carefully chosen γ can significantly improve the efficiency as observed in [9, 8].

3.2 Equivalent state don't cares

The extraction of equivalent states is much more complicated. Before proceeding further, we restate several basic definitions.

Definition 3.1 *Two states, σ_i and σ_j, of machine M are said to be* **compatible,** *written $\sigma_i \sim \sigma_j$, if and only if $\forall i \in I, \lambda(i, \sigma_i) = \lambda(i, \sigma_j)$.*

Definition 3.2 *Two states, σ_m and σ_n, of machine M are said to be* **implied** *by the compatible pair (σ_i, σ_j), written $(\sigma_i, \sigma_j) \Rightarrow (\sigma_m, \sigma_n)$, if $\exists i \in I : \sigma_m = \delta(i, \sigma_i)$ and $\sigma_n = \delta(i, \sigma_j)$.*

Definition 3.3 *Two states, σ_i and σ_j, of machine M are said to be* **equivalent,** *written $\sigma_i \equiv \sigma_j$, if and only if they are compatible and all their implied states pairs are also equivalent.*

[6]Using the recursive range computation method, described later, this conversion step is done already. This is inserted in the pseudo code for completeness.

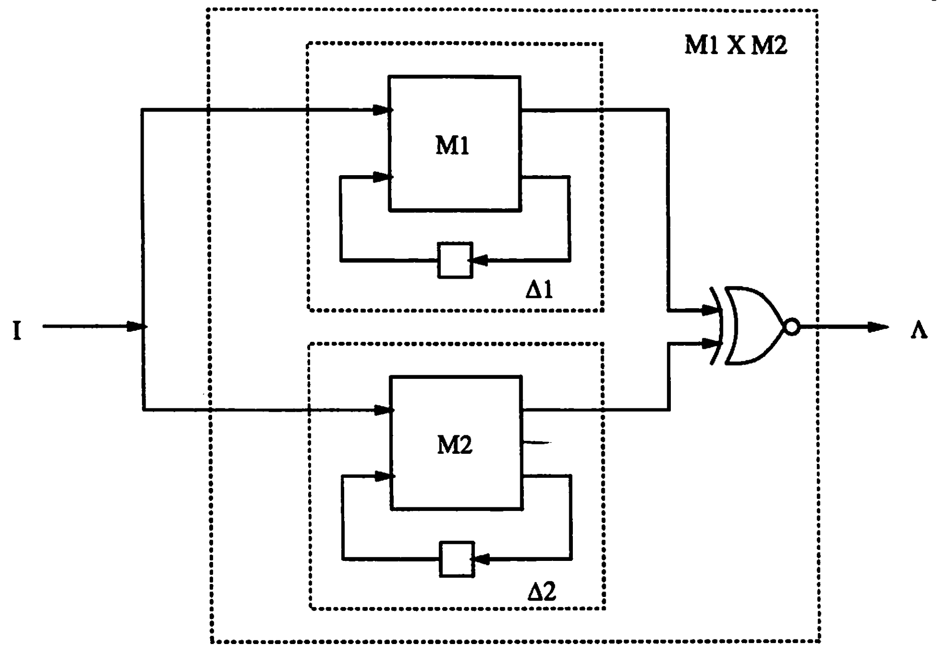

Figure 2: Pictorial view of the computed product machine.

The basic idea of the algorithm is as follows. First, determine states that are compatible. This is the initial set of potentially equivalent sates. Then, compute the pairs of states that are implied by these potential equivalent pairs. If an implied pair is not potentially equivalent, then remove the corresponding compatible pair from consideration. Repeat this process til convergence. These computations are done using sets. Thus, the compatibility and implied conditions can be determined at once for a large portion of the state space. This is quite different from the classical equivalence identification procedure which requires the construction of merger graphs where each state must be explicitly represented [21].

To do this, we construct a product machine, as illustrated in Figure 2. The product machine is constructed by duplicating the network, connecting the same primary inputs to both networks, and connecting the primary outputs of the two networks to a set of bit-wise equivalence gates. The function

$$\Delta(\Delta_1 \times \Delta_2) : I \times \Sigma^2 \rightarrow \Sigma^2 \tag{3}$$

represents the next state behavior of the product machine. The function

$$\Lambda : I \times \Sigma^2 \rightarrow B \tag{4}$$

represents the tautologous checking condition for the primary inputs. It is true under conditions that will cause both submachines to produce identical primary outputs.

Algorithm EXTRACT_EQUIVALENT_STATES given Figure 3 can be used to compute equivalent states. At Line 2, the set of compatible states is computed, which is also the initial set of potentially equivalent states. At Line 4, the stopping criterion is when all the implied states are also potentially equivalent. At Lines 5 and 6, the implied state pairs are computed implicitly for each pair of potentially equivalent states. The set *sat_implied* represents the subset of *implied*

```
Algorithm EXTRACT_EQUIVALENT_STATES ($\Delta$, $\Lambda$):
1.  begin
2.      $compatible = C_i\Lambda$; $equiv = compatible$;
3.      $implied = 1$;
4.      while ($implied \not\leq equiv$) do
5.          $implied =$ CONVERT (IMAGE ($\Delta$, $equiv$));
6.          $sat\_implied =$ CONVERT ($implied \wedge equiv$);
7.          $new\_equiv = S_i$ ($equiv \wedge$ INVERSE_IMAGE ($\Delta$, $sat\_implied$));
8.          $equiv = new\_equiv$;
9.      endwhile
10.     return $equiv$;
11. end.
```

Figure 3: Extraction of equivalent states.

that are still potentially equivalent. We next eliminate those state pairs from the set of potentially equivalent states who has non-satisfiable implied pairs. This is done using the INVERSE_IMAGE operator to compute the subdomain corresponding to a restricted range. This process is repeated to convergence.

We next show the correctness of the above procedure.

Lemma 3.1 *Let $\Lambda : I \times \Sigma^2 \rightarrow B$ be the tautologous check function constructed according to Figure 2. Then $C_i\Lambda \subseteq \Sigma^2$ represents the complete set of compatible states, where $\forall(\sigma_i, \sigma_j) \in C_i\Lambda \subseteq \Sigma^2$, $\sigma_i \sim \sigma_j$.*

Proof. By construction, Λ effectively represents the set of input conditions, (i, σ_i, σ_j) where the primary outputs of the original machine are the same. By Definition 3.1, two states are compatible if and only if identical output response is produce under all primary input conditions. This condition, $\forall i \in I, \Lambda(i, \sigma_i, \sigma_j) = 1$, can be efficiently computed using the consensus operation $C_i\Lambda$. □

Lemma 3.2 *Algorithm* EXTRACT_EQUIVALENT_STATES *terminates.*

Proof. (Sketch) The sets *implied* and *equiv* can only get successively smaller after each succeeding iteration. Since the domain and range of the state space is successively restricted, no new state pairs can be introduced in *implied* and *equiv*. □

Theorem 3.3 *Algorithm* EXTRACT_EQUIVALENT_STATES *returns the set equiv $\subseteq \Sigma^2$ that represents the entire set of equivalent state pairs.*

Proof. (Sketch) The correctness of the initial set of potentially equivalent states was proved in Lemma 3.1. The "if" part is done. Now we must show the "only if" part, which entails showing that the implied pairs are equivalent. Lines 5-8 successively removes state pairs from consideration that are known to have non-equivalent implied pairs. Thus, we are guaranteed that no equivalent pairs are removed consideration, but it must still be shown that all non-equivalent state pairs are eventually excluded. Suppose there is a non-equivalent state pair not removed from the procedure. This implies that it must either be non-compatible or has a non-equivalent implied pair. The first part is a contradiction due to Lemma 3.1. The second part implies that there exists a required implied state pair not in the set of potentially equivalent pairs, which is also a contradiction. □

Equivalence is transitive. Thus, the computation of equivalent state pairs is sufficient to determine all equivalence conditions.

An interesting by-product of the above procedure for extracting equivalent states is that we can perform **state minimization** directly on the sequential circuit structure without state graph extraction using BDDs. We will report more on this finding in the final paper.

3.3 Incomplete specification don't cares

Most real life designs are incompletely specified. There be may be input conditions in which the next state or some primary outputs may be left unspecified. This is especially true when the sequential circuit being optimized was compiled from a HDL. Let $M = (I, O, \Sigma, \delta, \lambda, \varphi)$ be the definition of sequential behavior where $\varphi : I \times \Sigma \rightarrow \Sigma \times O$ represents the incompletely specified (external) don't care network. We shall give a procedure in this section for the re-computation of this don't care network after structural transformations.

To do this, we need to describe a few basic machinary. Let $\delta_{\varphi 1}, \delta_{\varphi 2}, \ldots, \delta_{\varphi s}$ be the next state function components of φ, and let $\lambda_{\varphi 1}, \lambda_{\varphi 2}, \ldots, \lambda_{\varphi m}$ be the primary output components of φ. The goal is to compute a new don't care network $\phi : I \times \Sigma \rightarrow \Sigma \times O$ where $\delta_{\phi 1}, \delta_{\phi 2}, \ldots, \delta_{\phi s}$ are the next state function components of ϕ, and $\lambda_{\phi 1}, \lambda_{\phi 2}, \ldots, \lambda_{\phi m}$ are the primary output components of ϕ. We define $\vec{0}$ and $\vec{1}$ to be the all 0 and all 1 vector, respectively.

Let $\Delta : I \times \Sigma^2 \rightarrow \Sigma^2$ be the next state behavior of the product machine as defined before (*Cf.* Figure 2). However, we now construct the product machine with the original specification network and the current network. Let $\vec{\sigma\sigma}$ be the set of legal initial state pairs of the product machine.

The main idea of the recomputation procedure is as follows. First, we established the state correspondence between the states in the original machine and the new machine. It is important to note that, through structural transformations like retiming or re-encoding, the states in the original machine may be splitted or combined. Thus, there need not be a one-to-one state correspondence between the original machine and the new machine. Then, the incomplete specification don't cares for the new machine can be recomputed by identifying the corresponding input conditions that would cause the don't care.

The overall algorithm, RECOMPUTE_INCOMPLETE_SPECIFICATION_DC, is outlined in Figure 4. Lines 2-8 establishes the state pair correspondence between the new machine and the original machine. This portion of the procedure is similar to the procedure for computing the set of reachable states. The main difference is that the set reachable states is being computed for the product machine. At Lines 9-12, the corresponding incomplete specification don't cares for the primary outputs are computed. Essentially, $\lambda_{\varphi i}$ is the set of primary input and present state conditions where the primary output λ_i is left unspecified. Then, the corresponding primary input and present state condition for the new machine is determined. As noted earlier, splitting and merging may occur through structural transformations. The problem arises when multiple states are merged into one. For example, suppose the states σ_m and σ_n of the original machine were combined to one state σ_p in the new machine. Suppose

```
Algorithm RECOMPUTE_INCOMPLETE_SPECFICATION_DC ($\Delta$, $\varphi$, $\ddot{\sigma\sigma}$):
1.  begin
2.          $current = \ddot{\sigma\sigma}$; $valid = 0$;
3.          while ($current \not\leq valid$) do
4.                  $valid = valid \vee current$;
5.                  $current$ = BETWEEN ($current$, $valid$);
6.                  $new$ = IMAGE ($\delta$, $current$);
7.                  $current$ = CONVERT ($new$);
8.          endwhile
9.          foreach ($i = 1$; $i \leq m$; $i = i + 1$) do
10.                 $\omega = \lambda_{\varphi i} \wedge valid$;
11.                 $\lambda_{\phi i}$ = DETERMINE_CONSISTENT ($\omega$);
12.         endfor
13.         $\omega$ = DETERMINE_CONSISTENT ($\prod_{i=1}^{s}$ ($\sigma_{\varphi i} \wedge valid$));
14.         foreach ($i = 1$; $i \leq s$; $i = i + 1$) do
15.                 $\delta_{\phi i} = \omega$;
16.         endfor
17.         return $\phi$;
18. end.
```

Figure 4: Re-computation of incomplete specification don't cares.

there exists a primary input condition i such that $\lambda_i(i, \sigma_m)$ is unspecified (don't care), but $\lambda_i(i, \sigma_n)$ is specified. Then $\lambda_i(i, \sigma_p)$ of the new machine must remain specified. The procedure DETERMINE_CONSISTENT is used to resolve this consistency. We shall not describe now how the routine DETERMINE_CONSISTENT is being implemented, but the main idea is to resolve conflicts when multiple states are merged into one.

4 Latency and Multi-Cycle Don't Care Sets

We have thus far focussed on sequential don't care conditions that arise in normal operation of the sequential circuit. However, depending on the intended use of the sequential circuit, one may not be interested in the output behavior of the machine after every cycle. This is, for example, the case with pipeline circuits. Suppose we have a α-stage pipeline. Then it may not be of interests to examine the output of the pipeline during the first α cycles while it is being loaded. We refer this class of don't care operating condition as **Latency Don't Cares** (LDC). Another source of operation don't cares has to do with the **sampling rate** of the circuit. Suppose we have sequential circuit that is only sampled every τ cycles. Here, the primary outputs need only be valid during every τ cycles. This is not uncommon, for example, in signal processing applications. We refer this class of don't care operating condition as **Multi-cycle Don't Cares** (MDC). Other more complicated operating don't care conditions can be envisioned.

This don't care conditions can be treated in a similar manner as invalid state don't cares. The argument goes as follows. Suppose the primary output response from state s is never examined due to latency or the sampling rate.

```
Algorithm LATENCY_AND_MULTICYLE_DC ($\delta$, $\dot{\sigma}$, $\alpha$, $\tau$):
1.  begin
2.      current = $\dot{\sigma}$;
3.      foreach ($i = 1$; $i \leq \alpha$; $i = i + 1$) do
4.          new = IMAGE ($\delta$, current);
5.          current = CONVERT (new);
6.      endfor
7.      active = 0; c = 0;
8.      while (current $\not\leq$ active) do
9.          c = c + 1;
10.         if (c = $\tau$) do
11.             active = active $\vee$ current;
12.             c = 0;
13.         endif
14.         new = IMAGE ($\delta$, current);
15.         current = CONVERT (new);
16.     endwhile
17.     return ($\overline{active}$);
18. end.
```

Figure 5: Computation of latency and multi-cycle don't cares.

Then the primary output response from state s can be set to don't care. However, the next state response must be maintained. Thus, this don't care set is only associated with the primary outputs of the network, and not the next state lines.

We give a general procedure for efficiently computing these classes of don't cares. Like invalid and equivalent state don't cares, these don't care conditions may be exploited by combinational techniques in the form of external don't cares. It is worth noting that these don't cares cannot easily be extracted from state graphs nor can they be easily computed using a structural approach.

A general procedure called LATENCY_AND_MULTICYCLE_DC is outlined in Figure 5. The latency α and the sample value τ are provided to the procedure. A set of inactive states is returned that can used in combinational simplification. Specifically, the primary outputs from these states can be left unspecified.

5 Extensions to Combinational Logic Simplification

The sequential don't care sets (eg. invalid and equivalent states) computed in the previous sections may be exploited by known multi-level combinational logic simplification techniques in the form of external don't cares. We review two broad classes of simplification techniques, namely logic minimization based techniques and ATPG based techniques, and outlined the necessary extensions. Redundancy removal via ATPG techniques is a special form of simplification that does not usually involve local restructuring. Logic minimization based techniques can result in local restructuring via a process of Boolean resubstitution.

5.1 Combinational don't care sets

Two classes of don't cares have been identified. The first is the Satisfiability Don't Care (SDC) set, which has to do with input conditions that can never occur due to the structure of the network. These don't cares are a result of the extended Boolean space, which includes both primary input and internal variables. It is succinctly defined as:

$$SDC_i = \sum_{i \neq j} y_j \oplus f_j(x) \tag{5}$$

A much more difficult set to compute efficiently and correctly is the Observability Don't Care (ODC) set. These don't cares occur because the observability of the internal node at the primary outputs is limited the network output structure and is defined as follows:

$$ODC_j \equiv \prod_{k \in PO} (\overline{\frac{\partial f_k}{\partial y_j}} + D_{k_{ext}}) \tag{6}$$

The term $D_{k_{ext}}$ is the external don't care set at primary output k. The main problem in computing ODC is reconvergent fanouts. Various researchers have attempted to tackle this problem in different ways [2, 24, 18, 25, 11].

For sequential don't care minimization, there are two basic types of external don't care conditions, namely input conditions that never occur (eg. invalid states) and equivalent output responses (eg. equivalent states and incomplete specification). Invalid don't care input conditions may be propagated into the internal nodes of the logic networks in two different ways. The first is to simply express them in terms of traditional external don't cares and use the observability don't care equation (6) to propagate them inwards. However, since these invalid input conditions are don't cares for *all* (primary and state) outputs, they can be directly incorporated into the satisfiability don't care set since these conditions are indeed not satisfiable:

$$SDC_i = \sum_{i \neq j} (y_j \oplus f_j(x)) + D_{inv} \tag{7}$$

The D_{inv} term is the set of invalid input conditions.

The exploitation of equivalent output conditions is more difficult. Equivalent output conditions may be Boolean relations [5] rather than traditional external don't cares. Currently, ODC computation methods cannot handle directly external don't care conditions due to Boolean relations. In principle, an internal node may be set to either 1 or 0 under some input condition if doing so will cause it to produce a valid output pattern under an equivalence class even though the pattern may be different from the original pattern. ODC under Boolean relation don't cares can be computed in a naive way by performing 2^n fault simulations. However, this is extremely inefficient. Alternatively, we can temporarily construct a dummy network and attach to the end of the original network in a cascade manner for computing the internal don't cares. The dummy network would map equivalent output patterns to the same output values. In essence, combinational don't care algorithms are tricked into handling Boolean relations output. However, this is not easy to compute due to reconvergent fanouts. This is related to the problem of computing internal don't cares corresponding to multiple faults in the circuit. Fortunately, a large part of Boolean relations can be expressed using classical external don't cares. Note that the handling of Boolean relations in a multi-level context does not require a two-level Boolean relations minimizer since we are only simplifying a single internal node at a time. Once the internal don't cares are computed, standard two-level Boolean minimizers [4] can be invoked. Currently, a maximal subset of Boolean relation that can be expressed in terms of classical external don't cares are used.

5.2 Using ATPG for combinational logic simplification

It is well known that ATPG techniques can be harnessed to perform the task of redundancy removal, a restricted form of simplification. A signal or a gate is considered redundant if removing it will not effect the overall combinational behavior. Many different well-developed algorithms exist for this purpose. The reader can refer to [17, 15, 26, 27] for some representative approaches. For some circuits, simplification via ATPG is extremely efficient and effective. Although redundancy removal via ATPG is NP-complete, it can usually performed quite fast using state-of-the-art pruning strategies. In [20], it was reported that all the ISCAS combinational benchmarks can be made prime and irredundant in less than one hour on a 10MIPS class machine. The simplified results were comparable to those obtained using logic minimization.

One interesting property of ATPG based simplification is that it can be easily extended to handle complicated external don't care conditions and even Boolean relations. The two types of external don't care conditions, invalid input and equivalent output conditions, can be accommodated in the ATPG process. In the normal redundancy process, a signal or a gate is declared redundant if no test vector can be found to produce a different vector at the primary outputs. Handling invalid input conditions and equivalent output responses can be done with some slight modification to the ATPG process as follows: If the test generated for a node corresponds to an invalid input condition, or the faulty output corresponds to an equivalent output vector, then the test pattern generator can be tricked to reject the test. If no valid test can be found to produce a non-equivalent output, then the signal or gate can be declared redundant and removed. This extended combinational ATPG process can in fact be harnessed to remove sequential redundancies, as described in Section 8.

6 Efficient Symbolic Computation Techniques

6.1 Transition relations

Coudert, Berthet, and Madre [9, 10] propose an extremely elegant method of implicitly enumerating through state space using BDDs to represent the state space. Unlike explicit enumeration on state transition graphs, their breadth first traversal algorithm examines a large set of states at each step, thus making it extremely powerful for large state space exploration. In [9], they proposed the use of transition relations, or consistency functions, to perform the range and domain operations used in implicit traversal. Burch *et al* [8] used a similar idea in their work on model checking and design verification. The overall method can be succinctly stated as follows. Let

$$F : B^n \rightarrow B^m$$

x denote the set of input variables, and y denote the set of output variables. Then

$$\chi_F = \prod_{i=1}^{m} y_i \equiv f_i(x),$$

$$\chi_F \subseteq B^n \times B^m$$
$$\textit{where} \quad (x,y) \in \chi_F \;\Rightarrow\; y = F(x).$$

The transition relation captures all ***satisfiable*** input output conditions. Once this is constructed, both range and domain computation can be done quite elegantly using known BDD operators. Concisely, let

$$\xi : B^n \;\rightarrow\; B$$

be the subdomain under consideration. Then the image of F under ξ is simply

$$F(x) = \mathcal{S}_x(\chi_F(x,y) \wedge \xi(x)). \tag{8}$$

The interpretation is the subset of output elements that satisfies the following condition:

$$\{y \mid \exists x \in \xi(x) \;\;:\;\; \chi_F(x,y) = 1 \wedge \xi(x) \equiv 1\}.$$

Similarly, the inverse-image of F under a restricted subrange of $\psi : B^m \;\rightarrow\; B$ is

$$F^{-1}(y) = \mathcal{S}_y(\chi_F(x,y) \wedge \psi(y)). \tag{9}$$

6.2 Recursive range computation

Range computation can be easily performed on the transition relation of the function. However, it requires the construction of the consistency function, which may be quite time consuming for some circuits. Alternatively, Coudert *et al* [10] described a recursive method using the cofactor operation that in practice performs much faster. Let

$$F : B^n \;\rightarrow\; B^m$$

be a multiple output function and $f_1, \ldots, f_m$ be the corresponding individual functions. Then the image computation can be efficiently computed using the following recursive equation:

$$F(x) = y_1(f_2, \ldots, f_m)_{f_1}(x) + \overline{y_1}(f_2, \ldots, f_m)_{\overline{f_1}}(x) \tag{10}$$

In the worst case, this basic method would still require exponential computation. The efficiency depends heavily on the selection of output function at each level of recursion. Also, bounding techniques play a significant role in pruning unnecessary computation. For example, disjoint support information may be used to advantage in bounding the recursion.

6.3 Domain computation via composition

Domain computation using the consistency method is also limited by the ability to construct the consistency function. Instead, domain computation can be done using the *compose* operator [7]. Let

$$F : B^n \;\rightarrow\; B^m$$

be the function and

$$\psi : B^m \;\rightarrow\; B$$

be the subrange under consideration. Then

$$F^{-1}(\psi) = \psi \circ F \tag{11}$$

This can be directly implemented using BDDs without first constructing a consistency function.

circuit	inputs	outputs	latches	gates	literals
s27	4	1	3	10	18
s298	3	6	14	119	244
s382	3	6	21	158	306
s386	7	7	6	159	347
s400	3	6	21	164	322
s420	19	2	16	196	336
s444	3	6	21	181	352
s526	3	6	21	193	445
planet	7	19	6	606	1346
sse	7	7	6	130	318

Table 1: Statistics of some ISCAS and MCNC examples.

7 Preliminary Experimental Results

We give in this section some preliminary results on our propose ideas and algorithms. At this time, we have only implemented a small subset of the don't cares described in this paper. Thus, the results do not fully reflect the power of our propose techniques. Nonetheless, we were able to observe substantial gains over combinational simplification alone.

Our algorithms were developed on top of misII [3], version 2.2, which provided extensive support for experimentation. For preliminary experiment, we tested our algorithms on a set of multi-level sequential network benchmarks; all but two are from the ISCAS sequential logic benchmark set. The examples *planet* and *sse* were from the MCNC benchmark set. Some vital statistics of the examples tested are given in Table 1. The most number of latches tested so far is 21. We should be able to handle significantly larger examples than the ones tested thus far. We plan to report results on them in the final paper. To illustrate the effects of sequential don't cares, we compare results with pure combinational logic simplification, as shown in Table 2. The column labeled *initial* is the number literals initially prior to simplification. We then ran the SIMPLIFY and CSPF_SIMPLIFY commands in misII to minimize the network. The command SIMPLIFY makes use of the satisfiability don't care sets and CSPF_SIMPLIFY makes use of Muroga's compatible set of permissible functions (CSPFs) and observability don't cares. The results after one pass of SIMPLIFY and CSPF_SIMPLIFY are reported in the column labeled *comb-dc*. Note that substantial reduction was already possible without considering sequential don't cares. For the ISCAS examples, it is well known that they contain a number of combinational redundancies, which would explain the significant reduction using only combinational simplification. We than proceeded to compute the invalid and equivalent don't care sets using the BDD based algorithms described in the paper. These don't care sets were used as external don't cares to further minimize the network. We used the CSPF_SIMPLIFY command with the external don't care sets. Only a single pass simplification was carried out [7]. Depending on the example, significant further reduction was possible when these don't cares are exploited. The results after the use of sequential don't cares are reported in the column labeled *seq-dc*. In the fifth column labeled *ratio*, we indicate the

[7]To check the correctness of the minimization procedure, we actually performed sequential verification using a BDD based method [28].

circuit	initial	comb-dc	seq-dc	ratio
s27	18	17	17	1.06/1.00
s298	244	179	149	1.64/1.20
s382	306	238	204	1.50/1.17
s386	347	259	233	1.49/1.11
s400	322	253	207	1.56/1.17
s420	336	275	209	1.61/1.32
s444	352	275	223	1.58/1.23
s526	445	340	244	1.82/1.39
planet	1346	1159	1153	1.17/1.01
sse	318	239	212	1.50/1.13

Table 2: Preliminary results using a subset of the don't cares.

relative improvements of the methods. The first ratio is sequentially simplified result versus the initial circuit size. The second ratio is the sequentially simplified result versus the result via only combinational simplification. To check the correctness of the algorithms and the intermediate steps, the optimized results were sequentially verified against the initial specification. For all the examples reported, the optimized results were correct, as indicated in the last column.

For the final paper, we expect to have results for larger examples. Also, we believe that much greater simplification can be achieved when iterated with global restructuring techniques. We are currently implementing the algorithms for computing the latency and multi-cycle don't cares. Experimental results for these don't care sets will also be reported.

8 Sequential Redundancy Removal

Sequential redundancy removal is required if the circuit being manufactured is to be tested using non-scan testing procedures. A fault is said to be redundant if the removal of the associated circuitry will not effect the overall behavior of the circuit. Combinationally redundant faults can be removed quite efficiently today using state of the art combinational ATPG techniques such as PODEM [17], FAN [15], and SOCRATES [26, 27]. However, sequentially redundancies, those that cannot be detected at the primary outputs with a single test, are substantially more difficult to eliminate.

One approach to sequential redundancy removal is to use sequential ATPG technology to identify redundancies. Particularly, ATPG techniques are very good at identifying when a fault is *not* redundant. But effectively, the faulty machine must be shown to be *equivalent* to the fault-free machine before we can be certain that the fault under test is sequentially redundant. This can be very time consuming since a large part of the state space must be traversed. Sequential testers such as STEED [16], developed by Ghosh and Devadas, can be harnessed for sequential redundancy removal. The basic STEED procedure can be improved upon, possibly quite significantly, by incorporating the BDD-based symbolic implicit enumeration techniques. Specifically, the state justification and differentiation steps of STEED can be replaced by BDD operations. This represents a viable approach for sequential redundancy removal, but may require time consuming justification and differentiation steps to remove each redundancy.

An alternative approach is use combinational techniques in conjunction with sequential don't cares to remove sequential redundancies. This is very similar to the general concept introduced by Devadas, Ma, Newton, and Sangiovanni-Vincentelli in [14], and Devadas and Keutzer in [13]. Their general method can be summarized as follows: [8]

1. Compute invalid and equivalent states by extracting a state transition graph for the state machine.

2. Perform optimal two-level Boolean minimization using a minimizer like ESPRESSO [4] to obtain a prime and irredundant cover using the invalid and equivalent states as external don't cares. Repeat Step 1 and 2 until convergence. At this point, the two-level implementation is guaranteed to be sequentially irredundant.

3. Algebraically factorize the network to produce a multi-level logic network. The fully sequential testable property is retained.

The above procedure is limited in applicability to those circuits whose state space and combinational logic can be represented efficiently in two-level form. However, a large class of sequential logic networks has large numbers of latches and complicated data path like logic components. A further shortcoming is that an area penalty may be incurred due to the restriction of algebraic operations in obtaining the multi-level implementation. The severity of this shortcoming is dependent on the problem instance.

We improve on the above procedure by the use of BDD-based implicit state space enumeration techniques to compute the sequential don't care sets, as described in the previous sections, and well developed combinational ATPG algorithms (or multi-level combinational don't care based simplification algorithms) to obtain a prime and irredundant multi-level network under the sequential don't care sets. Our new procedure is outlined as follows:

1. Begin with a (potentially optimized) multi-level sequential logic network. No restriction is placed on the type of optimizations that can be applied to the network prior to this redundancy removal procedure. Thus, sequential logic optimization can be carried out without regards for testability considerations. Transformations like Boolean factorization and retiming can be applied freely (assuming the overall sequential behavior is preserved).

2. Use the implicit enumeration algorithms EXTRACT_INVALID_STATES and EXTRACT_EQUIVALENT_STATES (*Cf.* Section 3) to compute the invalidity and equivalence don't care conditions.

3. Use combinational ATPG to remove redundancies under the invalid and equivalent state don't cares. A test is rejected if it corresponds to an invalid input condition or the faulty next state corresponds to an equivalent state (primary outputs the same). If no valid test can be found under the external don't care conditions, then the corresponding signal or gate is declared redundant. This step can be replaced, or applied in conjunction, by multi-level combinational don't care based minimization techniques, such as those proposed by [2, 24] (*Cf.* Section 5). If the only concern is redundancy removal, the ATPG process is typically much faster. Repeat Step 2 and 3 until convergence.

[8] constrained synthesis procedures from state transition graphs have also been proposed.

We conjecture that current combinational ATPG technology and implicit state space enumeration technology can be harnessed to handle very large sequential networks.

It remains to be proved that the above procedure will, upon convergence, guarantee the same degree of sequential testability as the method proposed by Devadas in [14]. We are now in the process of verifying this assertion. However, following the lines of argument from [14], *under the same assumptions*, we are optimistic that similar theoretical results can be achieved.

9 Concluding Remarks

We have described a set of algorithms for the efficient computation of sequential don't cares using BDD-based symbolic implicit state space enumeration techniques. This permits the application of our algorithms to a large class of sequential networks, including those with data path like components and large number of storage elements. We have shown how sequential don't care conditions such as invalid states and equivalent states can be exploited with well developed combinational multi-level simplification procedures by treating them as external don't cares. We also describe how incompletely specified don't care conditions, obtained from high-level descriptions, can be efficiently re-computed once structural transformations such as retiming are applied.

A new class of sequential don't cares, called operation don't cares, was also introduced to consider the intended operating nature of the circuit. Specifically, circuits involving pipeline like and multi-cycle operations can benefit tremendously from the proper use of these don't cares.

At present, we have only implemented a subset of the techniques proposed in this paper. Early results indicate the feasibility and effectiveness of our proposed approach. For the final paper, we plan on experimenting with much larger example (eg. other examples in the ISCAS benchmark set). We are currently implementing the latency and multi-cycle don't care procedures. Results are forthcoming.

As a final point, we note that the use of the sequential don't cares in a repeated combinational ATPG process can be used to remove sequential redundancies, with the end goal of producing a fully sequentially testable network. We conjecture that this process of sequential redundancy removal is more efficient than currently known sequential ATPG process involving justification and differentiation.

10 Acknowledgements

This research was supported in part by Defense Advanced Research Projects Agency under contract number N00039-87-C-0182. The algorithms described in this paper were implemented using the facilities provided by misII. Specifically, we made use of the new BDD package in misII, maintained by Sharad Malik, and some of the implicit state space traversal routines implemented by Herve Touati and Hamid Savoj. We most gratefully acknowledge their contributions as well as others who have contributed to the misII system. We would like to thank Abhijit Ghosh, Alex Saldanha, and Srinivas Devadas for several interesting discussions on the don't care minimization problem. We are also grateful to Fabio Somenzi, Gary Hachtel, and Xuejun Du for their help.

References

[1] P. Ashar, S. Devadas, and A.R. Newton. A unified approach to the decomposition and re-decomposition of sequential machines. In *Proceedings of the Design Automation Conference*, June 1990.

[2] K. Barlett, R.K. Brayton, G.D. Hachtel, R.M. Jacoby, C.R. Morrison, R.L. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang. Multi-level logic minimization using implicit don't cares. *IEEE Transactions on Computer-aided design*, June 1988.

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

[5] R.K. Brayton and F. Somenzi. An exact minimizer for boolean relations. In *Proceedings of the International Conference on Computer-Aided Design*, November 1989.

[6] K.L. Brace R.E. Bryant and R.L. Rudell. Efficient implementation of a bdd package. In *Proceedings of the Design Automation Conference*, June 1990.

[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 1986.

[8] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the Design Automation Conference*, June 1990.

[9] O. Coudert and C. Berthet J.C. Madre. Verification of sequential machines based on symbolic execution. In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, 1989.

[10] O. Coudert, J.C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In *submitted to Worshop on Computer-Aided Verification, Rutgers*, June 1990.

[11] M. Damiani and G. De Micheli. Efficient computation of exact and simplified observability don't care sets fo multiple-level combinational networks. In *IFIP International Working Conference on Logic and Architecture Synthesis*, May 1990.

[12] M. Damiani and G. De Micheli. Synchronous logic synthesis: Circuit specifications and optimization algorithms. In *Proceedings of the International Symposium on Circuits and Systems*, May 1990.

[13] S. Devadas and K. Keutzer. Boolean minimization and algebraic factorization procedures for fully testable sequential machines. In *Proceedings of the International Conference on Computer-Aided Design*, November 1989.

[14] 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*, pages 8–18, January 1990.

[15] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Transactions on Computers*, pages 1137–1144, December 1983.

[16] A. Ghosh, S. Devadas, and A.R. Newton. Test generation for highly sequential circuits. In *Proceedings of the International Conference on Computer-Aided Design*, November 1989.

[17] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE Transactions on Computers*, pages 215–222, March 1981.

[18] G. Hachtel, R. Jacoby, and P. Moceyunas. On computing and approximating the observability don't care set. In *MCNC Logic Synthesis Workshop*, May 1989.

[19] G.D. Hachtel, R.M. Jacoby, K. Keutzer, and C.R. Morrison. On the relationship between area and optimization and multi-fault testability of multilevel logic. In *Proceedings of the International Conference on Computer-Aided Design*, November 1989.

[20] R. Jacoby, P. Moceyunas, H. Cho, and G. Hachtel. New atpg techniques for logic optimization. In *Proceedings of the International Conference on Computer-Aided Design*, November 1989.

[21] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw Hill Publishing Company, 1978.

[22] M.C. McFarland, A.C. Parker, and R. Camposano. Tutorial on high-level synthesis. In *Proceedings of the Design Automation Conference*, June 1988.

[23] R. McGeer. On the interaction of functional and timing behavior of combinational logic circuits. In *UC Berkeley Electronics Research Laboratory*, November 1989.

[24] 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 Computer-aided design*, 10(10):1404–1424, October 1989.

[25] 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 Design Automation Conference*, June 1990.

[26] M.H. Schulz and E. Auth. Advanced automatic test pattern generation and redundancy identification techniques. In *18th Symposium on Fault Tolerant Computing*, 1988.

[27] M.H. Schulz, E. Trischler, and T. Sarfert. Socrates: A highly efficient atpg system. *IEEE Transactions on Computer-aided design*, 7(1):126–137, January 1988.

[28] H.J. Touati, H. Savoj, B. Lin, R.K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdd's. In *Submitted to ICCAD-90*, November 1990.