Optimizing Designs Containing Black Boxes

Tai-Hung Liu
Avanti Corp.
and
Adnan Aziz
The University of Texas at Austin
and
Vigyan Singhal
Tempus Fugit Inc.

We are concerned with optimizing gate-level netlists containing "black boxes", i.e., components whose functionality is not available to the optimization tool. We establish a notion of equivalence for gate-level netlists containing black boxes, and prove that it is sound and complete. We show that conventional approaches to optimizing such netlists fail to fully exploit the don't care flexibility available for synthesis. Based on our new notion of equivalence, we introduce a procedure which computes the complete don't care set. Experiments indicate that our procedure can achieve more minimization than conventional synthesis.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design aids—*Automatic synthesis*; D.2.4 [**Software Engineering**]: Software/Program verification—*Formal methods*

General Terms: Algorithms, Design, Verification

Additional Key Words and Phrases: Hierarchical logic synthesis, Don't cares, IP-based design

1. INTRODUCTION

For multilevel combinational circuits with single output nodes, it has been shown that all the flexibility in optimizing a specific node in the netlist can be expressed by an *incompletely specified* Boolean function [Bartlett, Brayton, Hachtel, Jacoby, Morrison, Rudell, Sangiovanni-Vincentelli, and Wang 1988]; when the nodes can have multiple outputs, the flexibility is characterized by a *Boolean relation* [Watanabe, Guerra, and Brayton 1993]. Watanabe *et al.* [Watanabe and Brayton 1993]

The support of the NSF under grant CCR-9702919 and The State of Texas Higher Education Coordinating Body under grant ARP 003658-0235-1997 is gratefully acknowledged. This paper is a revised and expanded version of [Liu, Sajid, Aziz, and Singhal 1997].
Address: Tai-Hung Liu, tai@avanticorp.com, Avanti Corp., 46871 Bayside Parkway Fremont, CA 94538; Adnan Aziz, adnan@ece.utexas.edu, ACE 6.120, University of Texas, Austin, TX 78712; Vigyan Singhal, vigyan@home.com, Tempus-Fugit, 525 Curtis St., Albany, CA 94706

prove that the complete set of feasible implementations for a component in a sequential netlist can be represented as a single nondeterministic finite state machine.

We describe algorithms for optimizing gate-level hardware netlists which contain "black boxes", i.e., components whose functionality is unavailable to the optimization tool. Knowledge of the exact functionality of the black box can only improve the quality of the synthesized logic. However, this may be impossible to obtain: the emerging trend towards the use of reusable cores, such as MPEG units, microcontrollers, PCI bus interfaces, etc. is example of such a situation [Alliance 1996]; for IP reasons, vendors will not provide a logic-level description of the core.

In the past the approach taken to synthesizing netlists with black boxes has been to make the inputs to the black boxes primary outputs, and the outputs of the black boxes primary inputs to the remainder of the netlist. In this manner, the result of synthesis is guaranteed to be a "safe replacement" for the original netlist, since the input-output functionality remains unchanged. This is the approach taken in a number of commercial tools, e.g., [Stok 1996].

However, we will show that this approach is pessimistic both in theory and in practice — the flexibility afforded by observability and controllability don't cares in the portion of the netlist to be synthesized is not fully exploited. In our work, we formulate a sound and complete synthesis procedure using the concept of an "observability netlist", and present experimental results.

For the sake of succinctness, we restrict our attention to combinational netlists. The ideas presented in this paper generalize to sequential netlists; such an exposition is available in [Aziz 2000].

2. DEFINITIONS

DEFINITION 1. A *simple netlist* is a directed acyclic graph, where the nodes correspond to *primitive circuit elements*, and the edges correspond to wires connecting these elements. Each node is labeled with a distinct Boolean-valued variable w_i. The two primitive circuit elements are *primary inputs* and *gates*. Primary input nodes have no nodes incident to them; the set of primary input nodes is assumed to be ordered. Every gate G has an associated Boolean function $f_G : \{0,1\}^n \mapsto \{0,1\}$, and an ordered list of nodes incident to it; these are referred to as the node's *fanins*. Some nodes are designated as being *primary outputs*.

A *complex netlist* (or simply, a *netlist*) is a similar to a simple netlist, with the addition of black boxes to the set of primitive circuit elements. Each black box has an associated *variety*, and an ordered list of nodes incident to it. Black boxes of the same variety are required to have the same number of inputs.

A complex netlist is shown in Figure 1; this netlist has two black boxes of variety G and one black box of variety F.

For a simple netlist, given an *input*, i.e., a Boolean-valued assignment to the set of primary input nodes, one can uniquely compute the value resulting at each node in the netlist by evaluating the gate functions in topological order. In this way, a netlist D on input nodes $a_1, a_2, \ldots, a_n$ and output nodes $b_1, b_2, \ldots, b_m$ bears a natural correspondence to a Boolean function on domain $\{0,1\}^n$ and range $\{0,1\}^m$.

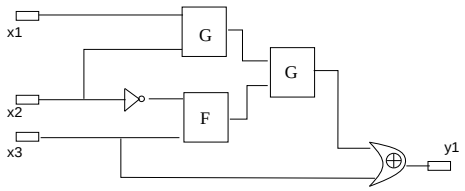

Fig. 1. A complex netlist containing two black boxes of variety G and one of variety F.

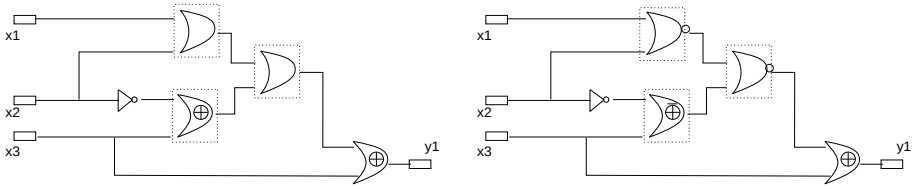

Fig. 2. Instantiations for the complex netlist in Figure 1.

2.1 Composing netlists

Composition of two netlists simply consists of placing the two netlists next to each other and connecting the nodes for primary inputs and primary outputs which are specified by the composition. We will refer to the composition of netlists C and D by $C \otimes D$; for notational convenience we do not include the pairs of nodes which are connected together.

The result of composing netlists may not be a netlist, since cycles may be introduced; we will consider netlist composition only when it results in a netlist. We will revisit this issue in Section 3.1.

2.2 Instantiating netlists

An *instantiation* μ is a mapping from varieties to simple netlists; a variety with n inputs is mapped to a simple netlist with n inputs and 1 output. For a variety V, the simple netlist it is mapped to by μ is denoted by $\mu(V)$. Thus black boxes of the same variety are mapped to the same netlist. Given a complex netlist D, the instantiation μ of D, denoted by $D[\mu]$, is the simple netlist resulting from the composition of the μ-instantiations of all the black boxes with the remainder of the complex netlist. In this way, a complex netlist D corresponds to a whole *family* of simple netlists. In Figure 2, we present two instantiations of the complex netlist in Figure 1.

3. DESIGN EQUIVALENCE

We now address the following fundamental question – when are netlists equivalent? Clearly, any meaningful notion of equivalence must be *compositional*, i.e., if netlist D_1 is equivalent to netlist D_0 then the composition of D_1 with any other netlist should be equivalent to the composition of D_0 with the same netlist.

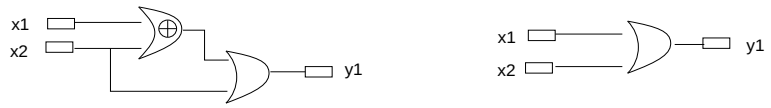

Fig. 3. Simple netlists which are equivalent: in both cases $y_1 = x_1 + x_2$.

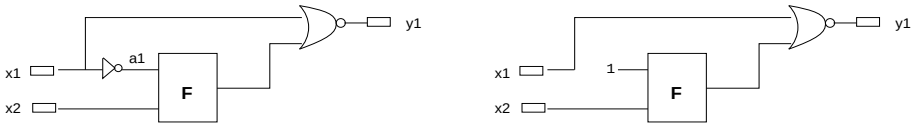

Fig. 4. Equivalent complex netlists: for both netlists, when x_1 is 1, y_1 is zero and when x_2 is 0, $y_1 = F(1, x_2)$.

3.1 Combinational loops

Since we require that there be no cycles in a netlist, we also require that both composition and instantiation not create any loops as well. Thus, we will require the following as a necessary condition for two netlists D_0 and D_1 to be equivalent: for every netlist E and for every instantiation μ, $D_0[\mu] \otimes E$ contains a cycle *if and only if* $D_1[\mu] \otimes E$ contains a cycle. From now on, when discussing equivalence between two netlists we will ignore the possibility of exactly one of the two netlists causing a cycle when composed with another netlist.

3.2 Equivalences for netlists

Consider the netlists in Figure 3; intuitively, they can be safely interchanged in any larger netlist without affecting the logical functionality of the overall netlist.

More formally, let D_1 and D_2 two simple netlists. Let the primary inputs of D_1 be $x_1, \ldots, x_n$ and the primary outputs be $y_1, \ldots, y_m$ similarly, let the primary inputs and outputs of D_2 be $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$. As in Section 2, we can construct Boolean functions $f_{D_1} : \{0,1\}^n \mapsto \{0,1\}^m$ for D_1, and $f_{D_2} : \{0,1\}^n \mapsto \{0,1\}^m$ for D_2.

DEFINITION 2. Two simple netlists D_1 and D_2 are equivalent if and only if $f_{D_1} = f_{D_2}$.

We can now define equivalence for complex netlists:

DEFINITION 3. Two complex netlists D_1 and D_2 are equivalent if and only if for each instantiation μ of the black boxes appearing in D_1 and D_2, we have $f_{D_1[\mu]} = f_{D_2[\mu]}$.

An example of complex netlists which are equivalent is given in Figure 4. Note that the inputs to the black boxes are not the same in the two netlists, and yet they are equivalent. Furthermore, it is not necessary for the number of black boxes to be equal for netlists to be equivalent; consider the example of Figure 5. These two examples show that our definition of equivalence differs from the traditional criterion of matching the black boxes for the two netlists, and ensuring that in addition to the Boolean functions of the primary outputs being equal, the Boolean functions at the black box inputs are equal [Stok 1996].

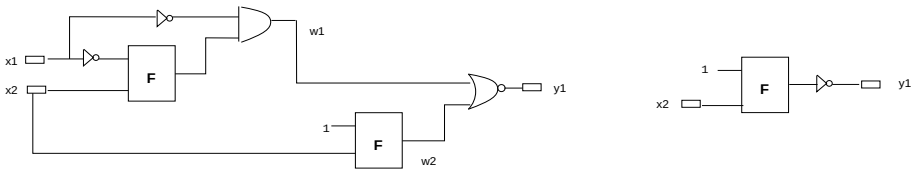

Fig. 5. Equivalent complex netlists with no correspondence between black boxes. When x_1 is 0, $y_1 = \bar{F}(1, x_2) = \bar{w_1} = \bar{w_2}$; when x_1 is 1, $w_1 = 0$, and so $y_1 = \bar{w_2}$. Thus w_1 can be replaced by the constant 0; propagating this constant results in the netlist on the right.

We state, without the easy but tedious formal proofs, that our notions of equivalence given in Definitions 2 and 3 are compositional as long as the necessary condition in Section 3.1 is met.

4. SYNTHESIS

As mentioned in the introduction, netlists with black boxes have traditionally been synthesized by making the inputs to the black boxes primary outputs, and output of the black boxes primary inputs, and synthesizing the simple logic.

However, this approach is pessimistic; the flexibility afforded by observability and controllability don't cares in the portion of the netlist to be synthesized is not fully used. This is illustrated in Figure 4; in which the gate a_1 can be replaced by the constant 1, even though it is an input to the black box. Additionally, the fact that certain components may be instantiations of the same black box, and thus when presented with the same input will produce the same output, is not used in the traditional approach. This is illustrated in Figure 5.

4.1 Sound and complete synthesis

We examine the problem of optimizing complex netlists, subject to the requirement that the resultant netlist be equivalent to the starting netlist. Our approach will be to first identify all the flexibility available for synthesizing the simple portion of the netlists. This is then used to minimize the simple logic using existing logic optimization techniques. In particular, the notion of "don't cares" sets, i.e., inputs for which a gate can output any value carries over from logic synthesis on simple combinational netlists [Bartlett, Brayton, Hachtel, Jacoby, Morrison, Rudell, Sangiovanni-Vincentelli, and Wang 1988].

DEFINITION 4. Let D be a simple netlist with n primary inputs. For a gate G, the set $\Delta \subset \{0,1\}^n$ is a *don't care* set if the gate function f_G can be replaced by any function f_G^* and the resulting netlist produces the same output as D on all inputs from Δ.

It is readily seen that the class of don't care sets of a gate G is closed under union; hence there exists a *maximal* don't care set for each gate.

We have defined don't care sets for G in terms of the space of primary inputs. A don't care set for G can be "projected" into the space of fanins of G. Let X be the set of primary input variables, and Y the set of G's fanin variables. Define the Boolean relation $R \subset 2^X \times 2^Y$ by $(\alpha, \beta) \in R$ exactly when applying α to the primary inputs results in β at the fanins of G. Then the *projection* of a set of inputs A to

the space of fanins of G is defined to be $\{\beta \mid \text{ for each } \alpha \in 2^X\ (\alpha, \beta) \in R \Rightarrow \alpha \in A\}$. Savoj *et al.* [Savoj, Brayton, and Touati 1991] have shown that the projection of the maximal don't care set for G to the space of local fanins for G is the maximal set of minterms over which f_G can be changed without affecting the overall input–output behavior of D. They then use these don't cares to minimize the sum-of-product representation of the gate functions.

The notion of a don't care set generalizes to complex netlists as follows:

DEFINITION 5. Let D be a complex netlist with n primary inputs. For a gate G the set $\Delta \subset \{0,1\}^n$ is a *don't care* set if the gate function f_G can be replaced by any function f_G^* and the resulting netlist produces the same output as D on all inputs from Δ for all possible instantiations of the black box varieties.

As before, don't care sets are closed under union, and hence there exists a maximal don't care set for each gate.

By treating the inputs and outputs of the black boxes as primary inputs and primary outputs, we can use conventional methods to compute don't care sets for the gates of the netlist. However, as illustrated by the examples in Figures 4 and 5, this approach is suboptimal, i.e., fails to compute the maximal don't care set.

When the netlist contains only one black box per variety, then the maximal don't care sets for gates can be easily computed. Specifically, let the gate G be a fanin to a black box H; for simplicity, assume that G is not a fanin to any other gate or black box. Then we claim that the maximal don't care set for G is simply the the maximal don't care set for H. The reasoning is as follows — certainly the maximal don't care set for G is a subset of the the maximal don't care set of H, since G has no other fanouts. Conversely, consider an input ι in a don't care set for H. Since one instantiation for H is simply to connect G directly to the output of H, the input ι must lie in the maximal don't care set for G. Hence the claim follows.

In the more general case, where G may be a fanin to other gates and black boxes, the don't care set for G is further constrained by its other fanouts. However, assuming the netlist contains only one black box per variety, the maximal don't care set for G is given by a construction entirely analogous to that in the previous paragraph.

When the netlist contains more than one black box of a given variety, the maximal don't care computation becomes significantly more difficult. Consider for example the netlist on the left of Figure 5. Suppose that the black box whose fanins are $x1'$ and $x2$ was of variety G rather than F. Then the maximal don't care set for the gate labelled by $w1$ is empty, since there exists an instantiation for F for which $w2$ is always 0, and hence the output $y1$ is directly driven by $w1$. However, we argued that in the original netlist, where both black boxes are of variety F, the gate labelled by $w1$ can safely be replaced by the constant 0.

Essentially, what is happening is that the fact that different black boxes of the same variety must produce the same output on the same input constrains the set of values that can be seen in the netlist. This is analogous to the "satisfiability" don't cares that arise in simple netlists.

4.2 Maximal don't care computation for complex netlists

We now address the problem of computing maximal don't cares for gates in a general complex netlist.

One way in which to optimize a simple netlist S is to form two copies S_1 and S_2 of S and then construct the product machine $S^{\mathtt{PM}}$ from S_1 and S_2 by merging corresponding primary input nodes, and creating a single primary output which is the OR of the XOR's of pairs corresponding primary outputs from S_1 and S_2.

Savoj and Brayton [Savoj and Brayton 1991] showed that the function f_G at a node G in the netlist S can be changed to the function f_G^* without changing the Boolean function computed by S if and only if replacing f_G by f_G^* in S_1's copy of G in $S^{\mathtt{PM}}$ results in a circuit whose output is still always 0. Conceptually, S_2 acts as the specification; S_1 is then optimized while ensuring that the Boolean function it computes remains unchanged.

In spirit, our strategy for optimizing complex netlists is similar to the above. However, we need to take into account the fact that the functionality of the black boxes is unknown. In order to compute the full set of don't cares at a gate we use the concept of a *consistency netlist*.

4.3 The consistency netlist

Given a complex netlist D, we construct the consistency netlist for D as follows:

Step 1. Duplicate D to obtain $D_{\mathtt{dup}}$. Combine D and $D_{\mathtt{dup}}$ by merging corresponding primary inputs, and creating a single primary output which is the OR of the XOR's of pairs of corresponding primary outputs from D and $D_{\mathtt{dup}}$. Call this new netlist $D^{\mathtt{PM}}$.

Step 2. Replace all black boxes in $D^{\mathtt{PM}}$ by new primary input nodes; call the resulting netlist $D^{\mathtt{SIMP}}$. (Note that $D^{\mathtt{SIMP}}$ is a simple netlist.)

Step 3. For each pair of primary inputs in $D^{\mathtt{SIMP}}$ which correspond to black boxes of the same variety in $D^{\mathtt{PM}}$, add to $D^{\mathtt{SIMP}}$ a "consistency" logic gate, which produces 1 exactly when the pair of primary inputs have the same value **or** the inputs to the corresponding black boxes from $D^{\mathtt{PM}}$ have different values. Call the resulting netlist $D^{\mathtt{CON}}$.

Step 4. Form the gate $G_{\mathtt{CON}}$ by taking the conjunction of all consistency logic gates and the output of $D^{\mathtt{CON}}$. Add this gate to $D^{\mathtt{CON}}$; designate $G_{\mathtt{CON}}$ to be the only primary output. Call the resulting netlist $D^{\mathtt{CHECKER}}$.

The netlist $D^{\mathtt{CHECKER}}$ is defined to be the *consistency netlist* for D.

We illustrate the concept of the consistency netlist by means of Figure 6. This is the consistency netlist corresponding to the complex netlist on the left half of Figure 4.

4.4 Logic optimization

We claim that the consistency netlist embodies all the flexibility available for synthesis. In order to illustrate this claim, we first consider a simple yet surprisingly powerful global optimization technique known as *redundancy removal*. This consists of identifying gates which can be replaced by constants while ensuring that the re-

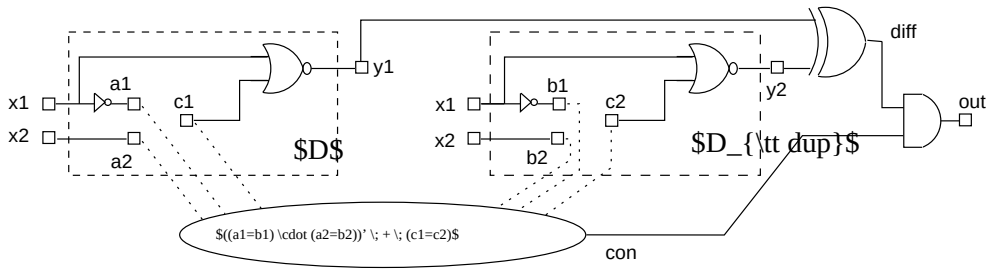

Fig. 6. Example of a consistency netlist.

sulting design is equivalent to the original netlist. These constants are subsequently use to simplify the netlist.

The concept of redundancy removal can be extended to complex netlists:

Definition 6. A gate is stuck-at-1(0) redundant in a complex netlist when it can be replaced by a gate taking the constant value 1(0) and the resulting netlist is equivalent to the original netlist, where the notion of equivalence is that for complex netlists, as given in Definition 3.

We illustrate how to use the consistency network for redundancy removal by means of the example in Figure 6. We claim that $a1$ is stuck-at-1 redundant in the netlist in Figure 6. We reason as follows: fix the value of $a1$ to 1. Now we consider two cases — when the input $x1$ is 1 and when it is 0.

(1) $x1 = 1$ — this forces $y1$ and $y2$ to both be 0, regardless of the value of $a1$, and so the node *diff* must be 0; hence *out* is 0.
(2) $x1 = 0$ — we have $b1 = 1$, so $a1 = b1$. Furthermore, we see $a2 = b2 = x2$. Note that *diff* is 1 iff $c1 \neq c2$. The nodes $c1$ and $c2$ are primary inputs, and can take any values. However, $((a1 = b1) \cdot (a2 = b2)) = 1$, so if $c1 \neq c2$, then *con* is 0. Thus either *diff* is 1 and *con* is 0, or *diff* is 0 and *con* is 1; in both cases *out* is 0.

Hence $a1$ is stuck-at-1 redundant.

More formally, the following theorem demonstrates that we can perform redundancy removal on the D by performing redundancy removal on the nodes corresponding to D in the consistency netlist.

Theorem 4.1. Let α be any gate in a complex netlist D. Then α is stuck-at-1(0) redundant *if and only if* the gate corresponding to α in the simple netlist $D^{\texttt{CHECKER}}$ is stuck-at-1(0) redundant.

Proof. (**IF:**) Suppose α is not stuck-at-1 redundant, that is the complex netlist $D_{\text{s-a-1}}$ resulting on setting α to 1 in D is not equivalent to D.

This means that there exists an instantiation μ for the black boxes so that $D[\mu]$ and $D_{\text{s-a-1}}[\mu]$ are inequivalent, i.e., compute distinct Boolean functions. Let the input differentiating $D[\mu]$ and $D_{\text{s-a-1}}[\mu]$ be u.

Set the gate corresponding to α in $D^{\texttt{CHECKER}}$ to 1; call this new netlist $D^{\texttt{CHECKER}}_{\text{s-a-1}}$. Now we construct an input $\tilde{u}$ which causes $D^{\texttt{CHECKER}}_{\text{s-a-1}}$ to produce 1 as follows:

(1) Set the primary inputs of $D^{\texttt{CHECKER}}_{\text{s-a-1}}$ corresponding to the primary inputs of D to u.
(2) Set the primary inputs of $D^{\texttt{CHECKER}}_{\text{s-a-1}}$ corresponding to black boxes in D to the values taken by the outputs of the black boxes in $D[\mu]$ on input u.
(3) Set the primary inputs of $D^{\texttt{CHECKER}}_{\text{s-a-1}}$ corresponding to black boxes in $D_{\texttt{dup}}$ to the value taken by the output of the corresponding instantiation of the black box in $D_{\text{s-a-1}}[\mu]$.

It is readily seen that for input $\tilde{u}$, the values taken at the nodes of $D^{\texttt{CHECKER}}_{\text{s-a-1}}$ corresponding to the outputs of D and $D_{\texttt{dup}}$ are the same as the outputs of $D[\mu]$ and $D_{\text{s-a-1}}[\mu]$ on application of u, which by hypothesis differ.

Now in order to show that for input $\tilde{u}$, the netlist $D^{\texttt{CHECKER}}_{\text{s-a-1}}$ produces a 1 on the output, it suffice to demonstrate that the output of each consistency gate in $D^{\texttt{CHECKER}}_{\text{s-a-1}}$ is 1.

Consider any consistency gate; let it correspond to a pair of black boxes of variety T. Either the nodes corresponding to inputs to the black boxes take different values for the input $\tilde{u}$, in which case the consistency gate outputs 1, or they take the same value. The critical observation here is that since the pair of black boxes are drawn from the same variety, they are given the same instantiation by μ. Thus on being presented with equal inputs, they produce equal outputs. Consequently, the values assigned in $\tilde{u}$ to the primary inputs for this black box pair must be equal, and so again the consistency node produces a 1.

Thus applying the input $\tilde{u}$ leads to $D^{\texttt{CHECKER}}_{\text{s-a-1}}$ producing a 1 on the output. Since $D^{\texttt{CHECKER}}$ always produces a 0, the gate corresponding to α in $D^{\texttt{CHECKER}}$ is not stuck-at-1 redundant.

(**ONLY IF:**) Now we show that if the gate corresponding to α (call it $\alpha^{\texttt{CHECKER}}$) is not stuck-at-1 redundant in $D^{\texttt{CHECKER}}$, then it is not stuck-at-1 redundant in D. In order to do so we need to exhibit an input to D and an instantiation μ for the black boxes so that the outputs of $D[\mu]$ and $D[\mu]$ with α set to 1 differ. Call the latter netlist $D_{\text{s-a-1}}[\mu]$.

Consider the netlist $D^{\texttt{CHECKER}}_{\text{s-a-1}}$ derived by setting $\alpha^{\texttt{CHECKER}}$ to 1. By hypothesis, $\alpha^{\texttt{CHECKER}}$ is not stuck-at-1 redundant in $D^{\texttt{CHECKER}}$; thus there is an input $\tilde{v}$ for which $D^{\texttt{CHECKER}}_{\text{s-a-1}}$ produces a 1. Let the projection of this input to the inputs which correspond to the primary inputs of D be v.

We claim that there is an instantiation to the black boxes so that the output of $D[\mu]$ and $D_{\text{s-a-1}}[\mu]$ differ on input v.

Let T be any variety. We create constraints on the instantiation $\mu[T]$ for T as follows: suppose u is a primary input in $D^{\texttt{CHECKER}}_{\text{s-a-1}}$ corresponding to the output of a black box of variety T; let $\vec{w}$ be the vector of nodes in $D^{\texttt{CHECKER}}_{\text{s-a-1}}$ corresponding to the inputs of this black box. Let $\mathbf{w}$ be the value of $\vec{w}$ in $D^{\texttt{CHECKER}}_{\text{s-a-1}}$ under input $\tilde{v}$; similarly, let $\mathbf{u}$ be the value assigned to u by $\tilde{v}$. Then constrain $\mu[T]$ to produce $\mathbf{u}$ on input $\mathbf{w}$.

Observe that the constraints on $\mu[T]$ are not inconsistent, i.e., we never require $\mu[T]$ to produce different values on the same inputs. This stems from the fact that the input $\tilde{v}$ causes $D^{\texttt{CHECKER}}_{\text{s-a-1}}$ to produce an output 1, which means that each consistency gate must produce a 1.

We now claim that for any instantiation μ of the varieties satisfying the above constraints, $D[\mu]$ and $D_{\text{s-a-1}}[\mu]$ produce distinct outputs on the input v. This immediately follows from the fact that $D^{\texttt{CHECKER}}_{\text{s-a-1}}$ produces a 1 on $\tilde{v}$, and that the values seen at the nodes corresponding to black box inputs and output in $D[\mu]$ and $D_{\text{s-a-1}}[\mu]$ are equal to the values of the corresponding nodes in $D^{\texttt{CHECKER}}_{\text{s-a-1}}$. Hence α is not stuck-at-1 redundant in D.

Symmetric arguments shows that the gate corresponding to α is not stuck-at-0 redundant in $D^{\texttt{CHECKER}}$ iff α is not stuck-at-0 redundant in D. □

Theorem 4.1 states that the consistency netlist can be used to determine gates which can be replaced by constants. We can similarly prove a theorem analogous to Theorem 4.1 which states that for a gate α on n inputs, the minterm $s \in \{0,1\}^n$ is in the don't care set for α if and only if it is in the don't care of the gate corresponding to α in the simple netlist $D^{\texttt{CHECKER}}$. Thus we can compute the don't care set for α by computing the don't care set for the gate corresponding to α in the simple netlist $D^{\texttt{CHECKER}}$.

4.5 Experiments

We now report experimental results on the synthesis of complex netlists; these experiments were performed in the SIS environment [Sentovich, Singh, Moon, Savoj, Brayton, and Sangiovanni-Vincentelli 1992].

Specifically, we selected several combinational circuits from the ISCAS suite, and manually identified a region of the logic internal to the netlist to be treated as a black box. We then formed the consistency netlist, computed the don't cares for the gates as described in Section 4.4, and simplified the functions for the gates using these don't cares. Specifically, we used the existing code for the `full_simplify` command in SIS which computes don't cares and minimizes the local functions with respect to these don't cares; we simply restricted the gates in the consistency netlist that `full_simplify` optimized to be from D and not from $D_{\texttt{dup}}$ or the consistency logic. The `full_simplify` routine uses BDDs [Bryant 1986] in the course of computing don't cares; BDD size explosion is the limiting factor in running this command. We report the size of the largest BDD built over the course of optimization.

In Table I we provide a comparison of our procedure with the conventional approach of making the black box inputs and outputs primary outputs and primary inputs. The experiments show that there is more reduction to be achieved by using the new procedure; in many cases, this difference is substantial, e.g., `i4`. The runtime and memory usage of our procedure is of the same order of magnitude as conventional don't care optimization.

References

Alliance, V. 1996. Virtual Socket Interface Proposal 1.0. `http://www.vsi.org/`.

Aziz, A. 2000. `http://www.ece.utexas.edu/~adnan/publications/www-core-seq.ps`.

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

Bryant, R. 1986. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers C-35*, (Aug.), 677–691.

Benchmarks		*Conventional Optimization*			*New Optimization*		
Name	Size	Reduction	Time	BDD Size	Reduction	Time	BDD Size
`pm1`	144	62	1	12	84	1	12
`b9`	338	124	1	52	138	4	63
`i4`	496	156	1	44438	318	6	44438
`9symml`	720	446	4	58	512	10	58
`cordic`	266	86	1	139	95	3	175
`apex6`	1869	988	7	454	1054	47	256
`comp`	404	238	1	21128	291	3	166125

Table I. Comparing conventional optimization with complete optimization based on the *consistency netlist*. For each netlist, we report the initial size of the circuit as measured by the number of literals in factored form representation of the node functions, the literal savings after optimization, the time taken for optimization (in seconds), and the number of BDD nodes for the largest BDD built by *full_simplify* in the course of computing the don't cares.

Liu, T., Sajid, K., Aziz, A., and Singhal, V. 1997. Optimizing Designs Containing Black Boxes. In *Proceedings of the Design Automation Conference* (June 1997). 113–116.

Savoj, H. and Brayton, R. K. 1991. Observability Relations and Observability Don't Cares. In *Proceedings International Conference on Computer-Aided Design* (Nov. 1991). 518–521.

Savoj, H., Brayton, R. K., and Touati, H. 1991. Extracting Local Don't Cares for Network Optimization. In *Proceedings International Conference on Computer-Aided Design* (Nov. 1991). 514–517.

Sentovich, E. M., Singh, K. J., Moon, C., Savoj, H., Brayton, R. K., and Sangiovanni-Vincentelli, A. L. 1992. Sequential Circuit Design Using Synthesis and Optimization. In *Proceedings International Conference on Computer Design* (Oct. 1992). 328–333.

Stok, L. 1996. BooleDozer: Logic Synthesis for ASICs. *IBM J. Res. and Devep.*, (July), 407–430.

Watanabe, Y. and Brayton, R. K. 1993. The Maximum Set of Permissible Behaviors for FSM Networks. In *Proceedings International Conference on Computer-Aided Design* (1993).

Watanabe, Y., Guerra, L., and Brayton, R. K. 1993. Logic Optimization with Multi-Output Gates. In *Proceedings International Conference on Computer Design* (Boston, MA, Oct. 1993). 416–420.