# Verifying Designs Containing Black Boxes

**Abstract**

We define a notion of equivalence for designs containing black boxes. This notion is applicable to both gate-level designs, as well as designs operating on integer variables. Using this notion, we describe a sound and complete methodology for optimizing designs containing black boxes, i.e. components whose functionality is not known; these arise naturally in the course of hierarchical design.

**Keywords:** Hierarchical Logic Synthesis, Black Boxes, Don't Cares

## 1 Introduction

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

Logic synthesis algorithms originally targeted the optimization of PLA implementations [4]; this was followed by research in synthesizing more general multilevel logic implementations. Currently, the central thrust in logic synthesis is sequential synthesis, i.e. the automatic optimization of the entire system. This is for designs specified at the structural level in the form of netlists, or at the behavioral level, i.e. in the form of finite state machines. DeMicheli [15] gives an excellent introduction to logic synthesis.

Typically, the synthesis process has two stages: first, the set of all possible implementations is characterized using some finite structure and then one is chosen according to some optimality criteria (e.g. minimum state [12]). For both combinational and sequential designs, the problem of determining and using the flexibility afforded by "don't care" conditions is well solved in both theory and practise [16, 7, 21].

In this paper we describe algorithms for synthesizing gate-level hardware designs which contain "black boxes", i.e. components whose functionality is not to be used. These can arise in many ways:

1. In hierarchical synthesis, components are recursively optimized from the "bottom-up" and then treated as fixed blocks.

2. The choice for the implementation of certain components may not have been made yet; this happens when different parts are designed separately.

3. A conscious decision may have been made not to synthesize certain components; these could be pre-defined blocks which have been already carefully designed and hand optimized. The emerging trend towards the use of reusable cores, such as MPEGs and data link controllers, is example of such a design philosophy [1].

In both scenarios 1 and 3, using knowledge of the exact functionality of the black box can only improve the quality of the synthesized logic. However, there is a tradeoff involved — the added quality comes at the expense of increased computation. Consider for example a black box which is in fact a multiplier. Non-local synthesis procedures (such as those exploiting controllability and observability don't cares) will need to represent the functionality of the core which can be prohibitively large; BDD based techniques for computing don't cares will fail completely, since the black box contains multiplication circuitry.

In the past, the approach taken for synthesis as well as verification of designs with black boxes has been to make the inputs to the black boxes primary outputs, and output of the black boxes primary inputs. In this way the synthesized product is guaranteed to be a "safe replacement" for the original design, since the input-output functionality remains unchanged. This is the approach taken in a number of commercial tools [8, 2].

We will show that, for logic optimization, this approach is pessimistic in theory and in practise; the flexibility afforded by observability and controllability don't cares in the portion of the design to be synthesized is not fully used. Additionally, the fact that certain components may be instantiations of the same "variety" of black box, and consequently when presented with the same input are constrained to produce the same output is not used in this approach.

To the best of our knowledge, there has been no past work on synthesizing designs with black boxes which has done more than treat the black box inputs and outputs as design outputs and inputs. In the verification community, Jones, Dill, and Burch have addressed the problem of verifying designs with "uninterpreted functions" (UIFs). These UIFs arise in the context of verifying complex operations in

microprocessors, and provide a useful mechanism for abstraction [13]. They can be viewed in some sense as being black boxes. However, they are applied to complex datatypes (such as integers), and the decision procedure for verification in the presence of UIFs is based on a rewrite system [18], which is completely distinct from our approach, which is BDD based.

The remainder of this paper is structured as follows: we begin in Section 2 by giving syntax and semantics to designs using finite state machines and netlists; this is extended to designs containing black boxes. In Section 3, we formulate the appropriate notion of equivalence for such designs. The inadequacy of existing approaches to synthesizing designs with black boxes is described in Section 4; we then formulate a sound and complete synthesis procedure using the concept of an "observability network", and present experimental results. We conclude with a summary of our contributions and suggestions for future work in Section 5.

## 2 Models for Hardware

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

We first review some useful terminology in the theory of sequences. Recall that in the usual set-theoretic development of the natural numbers $0, 1, 2, \ldots$, each natural number is the union of its predecessors [20]. Thus 0 is $\{\}$, 1 is $\{0\}$, 2 is $\{0, 1\}$, and so on. A *finite sequence over* a set $\Sigma$ is a function whose domain is a natural number, and whose range is a subset of $\Sigma$. Given a finite sequence $f : \{0, 1, \ldots, n-1\} \to \Sigma$, we will find it convenient to denote $f$ by $\langle f(0), f(1), \ldots, f(n-1)\rangle$.

### 2.1 Finite State Machines

Finite state machines provide a natural way of describing discrete time systems for which the current output depends not only on the current input, but also on past values of the input, while possessing only a bounded amount of memory. Theoretical and practical aspect of FSMs are described by Hopcroft [9]; below we develop enough theory to suffice for this paper.

**Definition 1** A Finite State Machine (FSM) is a quintuple $(Q, I, O, \lambda, \delta)$ where $Q$ is a finite set referred to as the *states*, $I$, and $O$ are finite sets referred to as the set of *inputs* and *outputs* respectively,

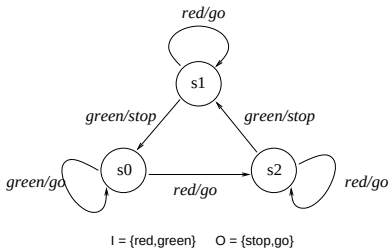


Figure 1: Graphically representing FSMs.

$\delta : Q \times I \to Q$ is the *next-state function*, and $\lambda : Q \times I \to O$ is the *output function.*

An FSM on inputs $X$ and outputs $Y$ can be represented graphically by a directed finite graph, referred to as a *state transition graph*, where the vertices are referred to as *states*, and the edges are referred to as *transitions*. The edges are labeled with input/output value pairs – the input value enables the transition, and the output value is produced. The destination node of the edge represents the next state for that input value. This is illustrated in Figure 1.

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

## 2.2 Netlists

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

### 2.2.1 Netlist Syntax

**Definition 2** A *simple netlist* is a directed 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 variable $w_i$. For simplicity, we will assume that the netlist is *Boolean*, i.e. all variables take values in $\{0, 1\}$. The three basic circuit elements are *primary inputs, latches* and *gates*. Primary input nodes have no fanins; latches have a single input, referred to as the *next state*. Every gate $G$ has an associated Boolean function $f_G : \{0,1\}^n \to \{0,1\}$, and an ordered list of $n$ nodes referred to as its *inputs*. Some nodes are designated as being *primary outputs*. Additionally, a total ordering on the

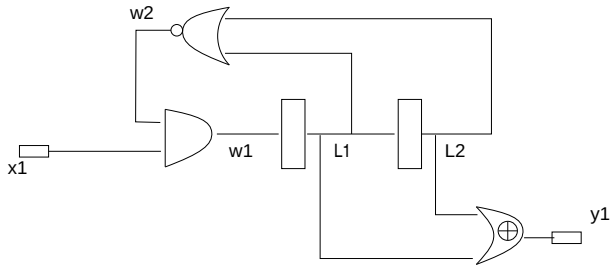


Figure 2: A simple netlist: the node $x_1$ is a primary input; $l_1, l_2$ are latches, and $w_1, w_2, y_1$ are gates. $y_1$ is designated a primary output.

primary inputs and primary outputs is specified. A simple netlist will be referred to as a *combinational* netlist if no latches occur in it; otherwise, it will be referred to as a *sequential* netlist.

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. Black boxes are akin to gates, in that they have an associated *variety*, and an ordered list of input nodes. Black boxes of the same variety are required to have the same number of inputs. Again, a complex netlist will be referred to as a *combinational* netlist if no latches occur in it; otherwise, it is *sequential*.

By definition, we require that there be no *combinational cycles*, where a combinational cycle is a cycle of gates. Issues related to combinational cycles have been dealt with elsewhere [5, 14].

Figure 2 depicts a simple netlist. Figure 3 illustrates a complex netlist, containing a single black box: the node $w_1$ is a black box of variety $F$. A combinational complex netlist is shown in Figure 4; this netlist has two black boxes of variety $G$ and one black box of variety $F$.

The definition given above of a simple netlist is essentially the data structure used by many logic synthesis tools to internally represent designs derived from HDL descriptions of gate/RTL level designs. Note that there is no loss of generality in restricting black boxes to be single output, since multi-output design can always be split into single output components; the single output assumption has been taken for pedagogical reasons.

### 2.2.2 Netlist Semantics

For a simple netlist, given an *input* (i.e., an assignment to each primary input node) and a *state* (i.e. an assignment to each latch), one can uniquely compute the values of each node in the netlist by evaluating the functions at gates in topological order, starting at the primary inputs and latches. In this way, a netlist $D$ on inputs $a_1, a_2, \ldots, a_n$, outputs $b_1, b_2, \ldots, b_m$ and latches $r_1, r_2, \ldots, r_k$ bears

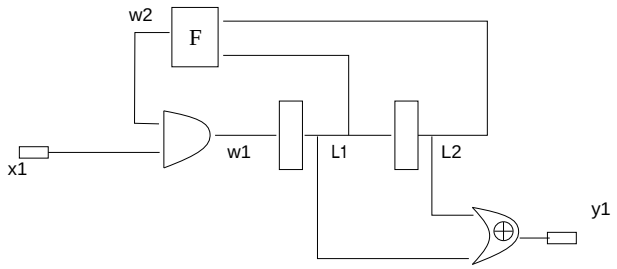


Figure 3: A complex netlist: the node $w_1$ is a black box of variety $F$

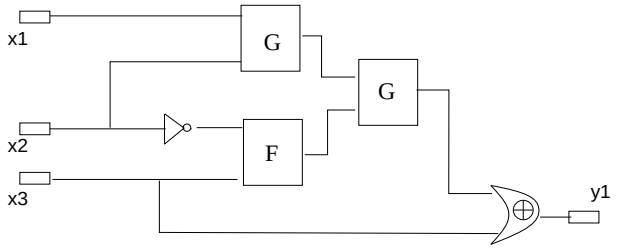


Figure 4: A combinational complex netlist containing two black boxes of variety $G$ and one of variety $F$.

a natural correspondence to an FSM $M_D$ on inputs $X_D = 2^n$, outputs $Y_D = 2^m$, and state space $S_D = 2^k$. The next-state function of $M_D$ is defined by the composed logic gates in the following manner: for each latch $r_i$ we can find a function $f_i : S_D \times X_D \to \{0, 1\}$ by composing the functions of the gates from the inputs and latch outputs to the input of the latch. We will refer to $f_i$ as the *next-state function* of the latch $i$. Thus $\delta_{M_D}(w_1, w_2, \ldots, w_n, r_1, r_2, \ldots, r_k) : S_D \times X_D \to S_D$ is simply the vector valued function $[f_1 f_2 \ldots f_k]$. Similarly, the output function is defined by composing the functions of gates from inputs and latches to output nodes.

An example of this correspondence is given in Figure 5. We will refer to the netlist as being an *implementation* of the finite state machine.

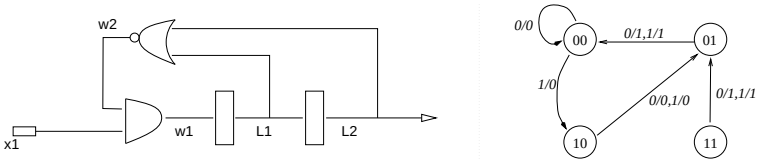


Figure 5: Extracting an FSM from a netlist.

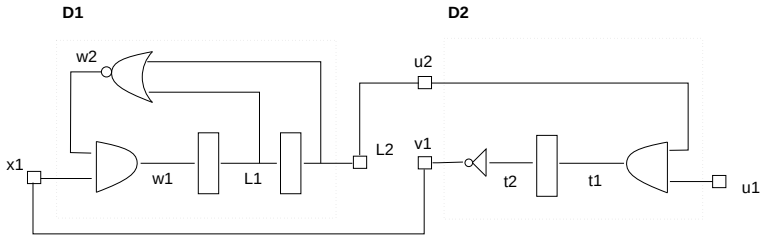


Figure 6: Composing netlists.

### 2.2.3 Composing Netlists

Large designs are invariably built up in a hierarchical fashion out of smaller components. Composition of two designs entails hiding some signals – each signal that is hidden is the output of one design, and an input of the other. The remaining input (or output) signals become the inputs (outputs) of the composed design.

Composition of two netlists simply consists of placing the two netlists next to each other and connecting the nets for input/output signals which are required by the composition. This is illustrated in Figure 6, where the inputs $x_1$ and $u_2$ are "tied" to $v_1$ and $l_2$ respectively; $v_1$ is designated an output in the composed design. For simplicity of presentation, we will assume that given two designs $C$ and $D$, it is understood which pairs of nets are tied together— henceforth, such a composition will be denoted by $C \otimes D$, without specifying the pairs of nets which are tied together. As stated in the introduction, our notion of composition is synchronous, i.e. all the latches are assumed to be driven by a common clock, and hence change state in lockstep.

Note that composing netlists does not necessarily result in a netlist, since the requirement that there be no combinational cycles may be violated. We will only consider netlist composition when it does not result in cyclic logic. We will revisit this issue when discussing equivalence in Section 3.1.

### 2.2.4 Instantiating Netlists

An *instantiation* $\mu$ 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 $\mu$ 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 $\mu$ of $D$, denoted by $D[\mu]$, is the simple netlist resulting from the composition of the the $\mu$-instantiation$\Phi$ 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; we will find

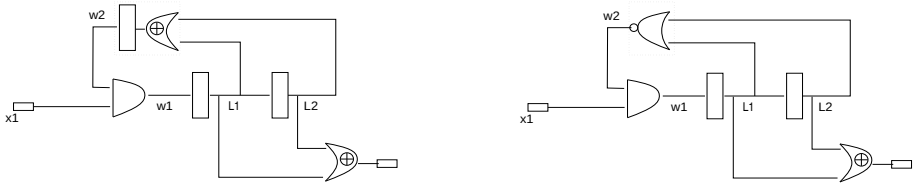


Figure 7: Two instantiations for the netlist of Figure 3.

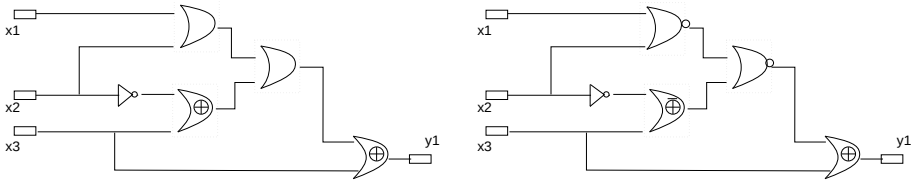


Figure 8: Combinational instantiations for the complex combinational netlist in Figure 4.

it useful to denote this family by $[\![D]\!]$. In Figure 7, we present two instantiations of the complex netlist in Figure 3

We will find occasion to consider what we will refer to as *combinational instantiations* of a complex netlist; these are instantiations where the instantiations of the individual black boxes are combinational netlists. In Figure 8, we give an example of combinational instantiations of a complex combinational netlist.

Once again, we note that an instantiation of a complex netlist may result in combinational cycles. We disallow such instantiations, and defer further discussion to Section 3.1

# 3 Design Equivalence

We now address the following fundamental question – when are designs equivalent? It is imperative that a notion of equivalence be compositional, i.e. if design $D_1$ is equivalent to design $D_0$ then the composition of $D_1$ with any environment should be equivalent to the composition of $D_0$ with the same environment; this is similar to the argument of plug-in plug-out replaceability [19] – see Figure 9.

We first need to redress the issue of combinational loops since it is intimately related to composition which is essential for any notion of design equivalence.

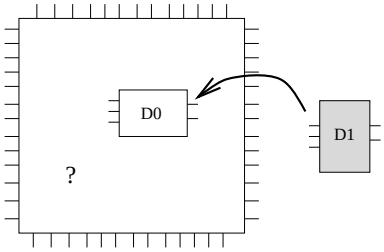


Figure 9: Plug-in plug-out replaceability.

### 3.1 Combinational loops

Since we require that there be no combinational loop in a design (a reasonable requirement), we also require that both composition and instantiation not create any loops as well. Thus, we will require that two designs $D_0$ and $D_1$ be equivalent only if for every environment $E$ and for every instantiation $\mu$, $D_1[\mu] \otimes E$ contains a combinational loop *if and only if* $D_0[\mu] \otimes E$ contains a combinational cycle. This requirement is necessitated to avoid the scenario where two designs may satisfy the intuitive notions of equivalence and still fail be replaceable in an environment (or instantiation) because one design may cause a combinational cycle and the other may not.

For the purposes of optimization, where $D_1$ is a replacement for $D_0$, we only need the *only if* half in the above condition, since a replacement may work for more environments (or instantiations) than the original design but not for less. Our synthesis algorithm in Section 4 ensures that there is a combinational path in $D_1$ from an input $i$ to output $j$ passing through black boxes $(B_1, B_2, \ldots, B_k)$ on the respective inputs $(p_1, p_2, \ldots, p_k)$ only if there is a combinational path in $D_0$ from $i$ to $j$ passing through black boxes $(B_1, B_2, \ldots, B_k)$ on the respective inputs $(p_1, p_2, \ldots, p_k)$; here $k$ may be 0 which corresponds to a simple combinational path. Thus, it is not possible for any instantiation of the synthesized design to create a combinational cycle with any environment unless the original design caused a combinational cycle under the same instantiation and the same composition.

Henceforth, while discussing equivalence between two designs we will ignore the possibility of exactly one of the two designs causing a combinational loop under a certain instantiation for a given environment.

### 3.2 Equivalences for Combinational Netlists

First we consider the problem of equivalence for simple combinational netlists. Consider the netlists in Figure 10; intuitively, they can be safely interchanged in any larger design without affecting the

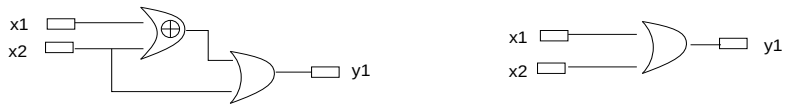


Figure 10: Simple combinational netlists which are equivalent: in both cases $y_1 = x_1 + x_2$.

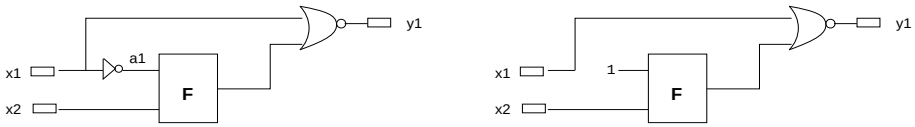


Figure 11: Combinationally equivalent complex combinational netlists: for both designs, when $x_1$ is 1, $y_1$ is zero and when $x_2$ is 0, $y_1 = F(1, x_2)$.

logical functionality of the overall design.

More formally, let $D_1$ and $D_2$ two simple combinational designs. 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 2.2.2, we can construct Boolean functions $f_{D_1}$ and $f_{D_2}$ from $\{0,1\}^n$ to $\{0,1\}^m$, for $D_1$ and $D_2$.

**Definition 3** *Two simple combinational designs $D_1$ and $D_2$ are equivalent if $f_{D_1} = f_{D_2}$.*

We can now define equivalence for complex combinational designs:

**Definition 4** *Two complex combinational designs $D_1$ and $D_2$ are equivalent if for any combinational instantiation $\mu$ 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 combinational designs which are combinationally equivalent is given in Figure 11. Note that the inputs to the black boxes are not the same in the two designs, and yet they are combinationally equivalent. Furthermore, it is not necessary for the number of black boxes to be equal for designs to be combinationally equivalent; consider the example of Figure 12.

Our definition of combinational equivalence differs from the traditional criterion of matching the black boxes for the two designs, 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 [3, 2]. The examples above demonstrate the incompleteness of this definition; we will see in Section 4 how the traditional notion of equivalence results in suboptimal synthesis.

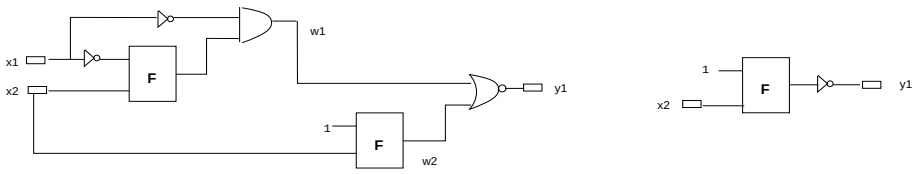


Figure 12: Combinationally equivalent complex combinational netlists with no correspondence between black boxes. When $x_1$ is 0, $y_1 = \bar{w_1} = \bar{w_2} = \bar{F}(1, x_2)$; when $x_1$ is 1, $w_1 = 0$, and so $y_1 = \bar{w_2}$; thus $w_1$ can be replaced by the constant 0, resulting in the design on the right.

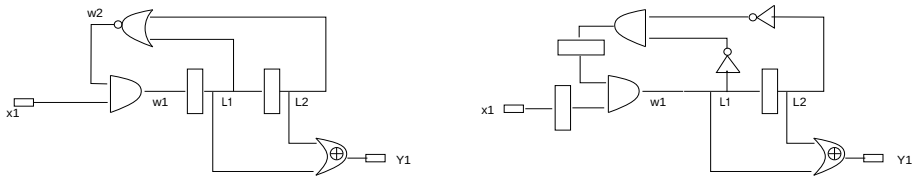


Figure 13: Simple sequential netlists which are equivalent for the reset state where each latch is assigned 0.

## 3.3 Equivalences for Sequential Netlists

### 3.3.1 Equivalences for simple sequential netlists

For sequential designs, we restrict our attention to designs where a hardware reset value is associated with each latch, and we assume that the circuit operation for any sequential design always starts in the designated initial state (which corresponds to the reset values for the respective latches). We assume that this designated initial state (or *reset* state) is part of the description of the design (and of any sequential instantiation of a black box).

As an example, consider the netlists in Figure 13; it should be intuitively clear that with respect to an initial state of 0's for all the latches, they can be safely interchanged without affecting the logical functionality of the overall design.

For simple sequential netlists with reset states, it is easy to develop an adequate notion of equivalence.

**Definition 5** *If $D_1$ and $D_2$ are simple sequential netlists with the respective reset states $s_1$ and $s_2$, $D_1$ and $D_2$ are equivalent if for every input sequence* $\mathbf{i}$ *applied to $D_1$ in state $s_1$, the resulting output sequence* $\mathbf{o_1}$ *is equal to the output sequence* $\mathbf{o_2}$ *resulting on application of* $\mathbf{i}$ *to $D_2$ starting at state $s_2$.*

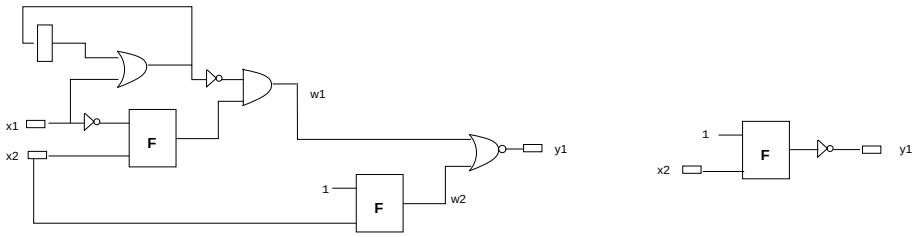


Figure 14: An example of equivalent complex sequential netlists. While $x_1$ is 0, $w_1 = w_2$ and $y_1 = \bar{w_2}$. As soon as $x_1$ goes to 1, $w_1$ goes to 0 and stays there forever; consequently, $y_1 = \bar{w_2}$. Hence the node $w_1$ can be replaced by the constant 0, resulting in the design on the right.

### 3.3.2 Equivalences for complex sequential netlists

Now we consider the case of complex sequential netlists.

**Definition 6** *Two complex sequential netlists $D_1$ and $D_2$ are equivalent with respect to the reset states $s_1$ and $s_2$ if for every instantiation $\mu$ (which includes the reset latch values for the latches in the black boxes), the simple sequential netlists $D_1[\mu]$ and $D_2[\mu]$ are equivalent at the corresponding reset states.*

Notice that all instantiations of the black boxes of a given variety must have the same designated reset values for the latches in the black boxes.

As an example of equivalent complex sequential netlists, consider the netlists in Figure 14. Also note that the combinationally equivalent complex combinational netlists of Figure 11 are not equivalent when the instantiations are allowed to be noncombinational. There is a two state implementation for $F$ which can "remember" the previous input, and produce 0 forever if $x_1$ ever goes to 1.

#### Equivalence in the absence of a fixed powerup state

When simple sequential designs $\eta_1$ and $\eta_2$ do not have designated initial states, the notion of equivalence becomes less obvious. It is straightforward to extend our work if we assume the behavior of such designs is defined over a 3-valued algebra on the set $0, 1, X$ [11] and latches which do not have a hardware reset value begin operation in the value $X$. $X$ denotes the *unknown* value. We assume the standard functionality of all logic gates in this algebra; notice that this can sometimes lead to non-intuitive reasoning: for example, the AND of $y$ and $\overline{y}$ is not 0 because if $y = X$, then $y = \overline{y} = X$ and $AND(X, X)$ is defined to be $X$, not 0. For equivalence we can say that two designs are equivalent[1] if

[1]Yet another notion of design equivalence is that of safe replacement [19]. This notion is analogous to language containment over the set of I/O traces from the non-deterministic initial state. The verification problem for this equivalence

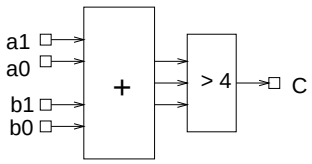


Figure 15: A modular design.

for any 3-valued input sequence, both designs produce the same 3-valued output sequence from their respective initial states (which may have some latches as $X$'s). For the purposes of optimization, we may only want to require that whenever the original design produces a 0 (or 1) the replacement design produces 0 (or 1); if the original design produces $X$ the replacement design can produce either 0 or 1. This definition of replacement (discussed earlier in [10]) is also compositional. On the other hand, the notion of 3-valued equivalence defined in [6] is not compositional as shown in [19].

### 3.3.3 Compositionality for equivalent designs

We state, without the easy but tedious formal proofs, that our four notions of equivalence (Definitions 3, 4, 5 and 6) for simple and complex designs are all compositional as long as the two conditions in Section 3.1 are met, i.e. if two designs are equivalent then for any instantiation of these designs any composition with a given environment yields equivalent designs.

## 4 Synthesis

We now describe algorithms for optimizing designs with black boxes. The most powerful synthesis procedures are those which use global knowledge of the design. As an example, consider the design of Figure 15. Since the output of the 2-bit adder is being compared with 5, the adder circuitry can be substantially simplified. This example of the power of nonlocal optimization partly shows how logic synthesis has freed designers from worrying about low level optimizations.

is PSPACE-hard in the size of state space of the design (which itself is exponential in the number of latches). Thus, we prefer to do analysis over the 3-valued notion which may be less desirable but is more tractable– checking equivalence is polynomial in the size of the state space.

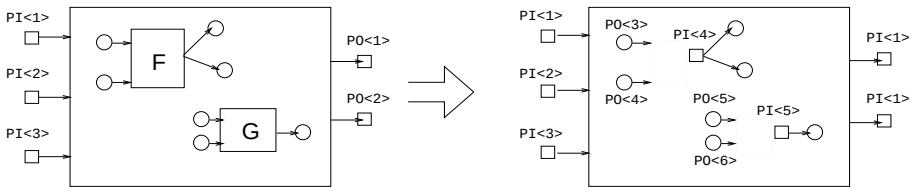


Figure 16: Traditional approach to synthesis; nodes marked PI⟨∗⟩ are primary inputs, nodes marked PO⟨∗⟩ are primary outputs.

## 4.1 Traditional Approaches

Traditionally, designs with black boxes have 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; this is illustrated in Figure 16. In this way the synthesized product is guaranteed to be a "safe replacement" for the original design, since the input-output functionality remains unchanged. This is the approach taken in a number of commercial tools [8].

However, this approach is pessimistic; the flexibility afforded by observability and controllability don't cares in the portion of the design to be synthesized is not fully used. This is illustrated in Figure 11, in which gate $a_1$ can be replaced by the constant one, 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 consequently when presented with the same input are constrained to produce the same output is not used in this approach. This is illustrated in Figure 12: since in all instantiations, when $w_1$ is not 0, it has the same value as $w_2$, $w_1$ can be safely set to 0.

## 4.2 Sound and Complete Synthesis

### 4.2.1 Combinational netlists

We first examine the problem of optimizing combinational netlists, subject to the requirement that the resultant design be combinationally equivalent to the starting design.

Our approach to synthesizing such netlists 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 networks [15]; the same is true of "compatible sets of permissible functions" [7, 16] is useful. The

latter correspond to subsets of the complete set of don't cares for individual gates with the property that gates can be independently simplified with respect to these subsets, without requiring that don't cares to be recomputed.

**Definition 7** *Let $D$ be a complex combinational design. For a gate $G$ in $D$ on $n$ inputs with function $f_G$, the input set $S \subset 2^n$ is a* don't care *set, if the gate function $f_G$ can be replaced by any function $f_G^*$ so that $f_G^*$ takes the same values as $f_G$ on any input $c \in \{0,1\}^n - S$ (on inputs from $S$, $f_G^*$ is allowed to take arbitrary values), while preserving the equivalence of the resulting complex combinational design and $D$.*

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 design. However, as discusses in Section 4.1, this approach is suboptimal. In order to compute the full set of don't cares at a gate we use the concept of a *consistency network*.

### The consistency network

Let $D$ be a complex combinational network. We construct the consistency network as follows:

**Step 1:** Form a new netlist $D^{\texttt{PM}}$ from $D$: duplicate $D$ to obtain $D_{\texttt{dup}}$ and combine $D$ and $D_{\texttt{dup}}$ by merging corresponding primary inputs, and creating a single primary output which is 1 precisely when there is a primary output of $D$ which is not equal to the corresponding primary output of $D_{\texttt{dup}}$.

**Step 2:** Replace all black boxes in $D^{\texttt{PM}}$ by primary inputs to form the simple combinational netlist $D^{\texttt{SIMP}}$.

**Step 3:** For each pair of primary inputs in $D^{\texttt{SIMP}}$ which correspond to black boxes of the same variety in $D^{\texttt{PM}}$, add to $D^{\texttt{SIMP}}$ a "consistency" gate, which outputs 1 exactly when the pair of primary inputs have the same value or the inputs to the corresponding black boxes from $D^{\texttt{PM}}$ take distinct values. Call this netlist $D^{\texttt{CONSISTENT}}$.

**Step 4:** Form the gate $G_{\texttt{CONSISTENT}}$ by taking the conjunction of all consistency logic nodes and the output of $D^{\texttt{CONSISTENT}}$. Add this gate to $D^{\texttt{CONSISTENT}}$; designate $G_{\texttt{CONSISTENT}}$ to be the only primary output. Call the resulting netlist $D^{\texttt{CHECKER}}$; we will refer to $D^{\texttt{CHECKER}}$ as the *consistency network*.

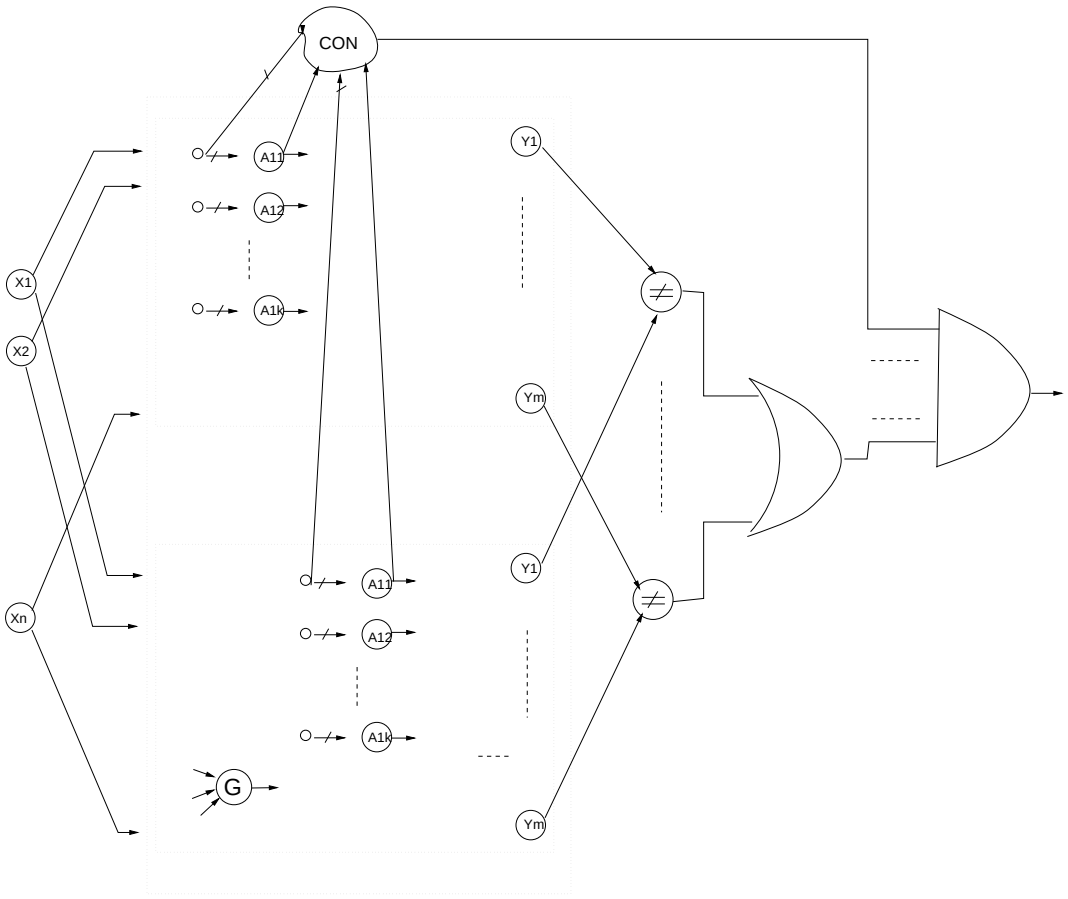


Figure 17: Redundancy removal for complex combinational netlists. The netlist depicted is $D^{\texttt{CHECKER}}$. The nodes marked $A_{1j}$ are primary inputs derived from black boxes from single variety $A$; consistency logic has been shown for a single pair. (Consistency logic exists for pairs within a netlist too.)

The result of this construction is illustrated in Figure 17; note that the resulting netlist is a simple combinational netlist. It should be clear from the construction that the output of $D^{\texttt{CHECKER}}$ is 0 for any input.

### 4.2.2 Logic Optimization

We claim that the consistency network embodies all the flexibility available for synthesis. In order to illustrate this claim, we consider a simple yet surprisingly powerful global optimization technique known as *redundancy removal*. This consists of identifying gates which can be replaced by a constant valued gate, while ensuring that the resultant design is equivalent to the original design. These

constants are subsequently use to simplify the logic.

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

**Definition 8** *A gate is stuck-at-1(0) redundant in a combinational 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 combinational netlists, as given in Definition 4.*

The following theorem demonstrates that we can perform redundancy removal on the simple logic associated with $D$ by performing redundancy removal on the nodes corresponding to $D$ in the consistency network.

**Theorem 4.1** *Let $\alpha$ be any gate in $D$. Then $\alpha$ is s-a-1(0) redundant if and only if the node corresponding to $\alpha$ in $D^{\texttt{CHECKER}}$ is s-a-1(0) redundant.*

**Proof**:

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

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

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

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

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

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

Consider any consistency node; 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 node 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 $\mu$ 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 $D_{s-a-1}^{\texttt{CHECKER}}$ producing a 1 on the output. Since $D^{\texttt{CHECKER}}$ always produces a 0, the node corresponding to $\alpha$ in $D^{\texttt{CHECKER}}$ is not stuck-at-1 redundant.

(**ONLY IF:**) Now we show that if the node corresponding to $\alpha$ (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 this we need to show an input to $D$ and an instantiation $\mu$ for the black boxes so that the outputs of $D$ and $D$ with $\alpha$ set to 1 differ. Call the latter netlist $D_{s-a-1}$.

Consider the netlist $D_{s-a-1}^{\texttt{CHECKER}}$ 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 $D_{s-a-1}^{\texttt{CHECKER}}$ 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 outputs of $\mu$ and $\mu_{s-a-1}$ (the instantiations of $D$ and $D_{s-a-1}$ for this instantiation) differ on input $v$.

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

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 critical observation stems from the fact that the input $\tilde{v}$ causes $D_{s-a-1}^{\texttt{CHECKER}}$ to produce an output 1, which means that each consistency node must produce a 1.

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

Symmetric arguments shows that the node corresponding to $\alpha$ is not stuck-at-0 redundant in

$D^{\texttt{CHECKER}}$ iff $\alpha$ is not stuck-at-0 redundant in $D$. ■

More generally, one can compute don't cares which can be used to simplify the individual gates in $D$. Once again, it is tedious but straightforward to prove an analog of Theorem 4.1 to the effect that the don't cares computed for the gate corresponding to $\alpha$ in $D^{\texttt{CHECKER}}$ are exactly the don't cares for $\alpha$ in $D$. This gives us a mechanism for computing the don't cares for $D$.

After simplifying gate $G$ in $D$, the don't cares for other gates may have changed. Thus it is necessary to recompute the don't cares for the remaining gates, which is potentially expensive computationally. Almost the same degree of optimization can be achieved using the concept of *compatible don't cares* [7, 16]. These don't cares can be used independently to optimize the gates; they too can be directly computed from the network $D^{\texttt{CHECKER}}$.

### 4.2.3 Sequential netlists

We now consider the most general case, namely that of synthesizing complex sequential netlists, wherein black boxes can be instantiated by arbitrary netlists. Again, our approach is to find "sequential don't cares" which are the analog of don't cares for combinational netlists, for sequential netlists.

We construct the netlist $D^{\texttt{CHECKER}}$ from $D$ exactly as in Section 4.2.1 except that in Step 3, in place of the combinational consistency checker, we add a "sequential consistency" checker for each pair of black boxes of a particular variety. This is the netlist corresponding to a three state finite state machine; in the initial state $s_0$, as long as the corresponding inputs to the black boxes are equal and the corresponding primary inputs equal, the machine stays in $s_0$ and produces a 1; if the corresponding inputs to the black boxes are ever not equal, the machine goes to $s_1$ and produces a 1; it remains in $s_1$ and produces an output 1. When the corresponding inputs to the black boxes are equal and the corresponding primary inputs not equal, the machine goes to state $s_2$ and produces a 0. It remains in $s_0$ on all inputs with the output remaining 0.

The consistency network embodies all flexibility available for synthesizing complex sequential netlists. In particular, it can be used for "sequential redundancy removal". A gate $\alpha$ is stuck-at-1(0) redundant in a complex sequential 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 sequential netlists, given in Definition 6.

The following theorem is the analog of Theorem 4.1 for complex sequential netlists:

**Theorem 4.2** *Let $D$ be a complex sequential netlist and $\alpha$ any gate in $D$. Then $\alpha$ is stuck-at-1(0) redundant if and only if the node corresponding to $\alpha$ in $D^{\texttt{CHECKER}}$ is stuck-at-1(0) redundant.*

| **Benchmark** | Initial size | *Conventional Optimization* | | |
|---|---|---|---|---|
| | (literals) | Reduction | Time (sec) | BDD Size |
| pm1 | 144 | 62 | 0 | 12 |
| b9 | 338 | 124 | 1 | 52 |
| i4 | 496 | 156 | 1 | 44438 |
| s208 | 206 | 34 | 1 | 35 |
| 9symml | 720 | 446 | 4 | 58 |
| cordic | 266 | 86 | 1 | 139 |
| apex6 | 1869 | 988 | 7 | 454 |
| comp | 404 | 238 | 0 | 21128 |

Table 1: Conventional logic optimization.

The proof of this result follows on the lines of the proof for Theorem 4.1.

### 4.3 Experiments

In this section we report experimental results on the synthesis of complex combinational netlists; these experiments were performed in the SIS environment [17].

Specifically, we took several ISCAS benchmark circuits, and "cut out" a region of the logic internal to the design; this was treated as a black box. We then formed the consistency network, computed the compatible observability don't cares for the gates as described in Section 4.2.2, and simplified the functions for the gates using these don't cares [4]. All this was relatively simple to achieve, since we were able to use the existing code for the `full_simplify` command in SIS which computes compatible observability don't cares; we simply restricted the nodes chosen to be simplified in the consistency network to be from $D$ and not from $D_{\mathtt{dup}}$ or the consistency logic.

In Tables 1 and 2 we provide a comparison of our procedure with the conventional approach of making the black box inputs and outputs primary outputs and primary inputs. For each design, we report the initial size of the circuit, the literal savings after optimization, the time taken for optimization, and the size of the largest BDD built in the course of computing the don't cares [16].

Not surprisingly, 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`.

However, the true significance of these results is not the amount of reduction offered; these examples do not reflect the kind of hierarchical designs our procedure is suited to. The main point to note is that

| **Benchmark** | Initial size | *New Optimization* | | |
|---|---|---|---|---|
| | (literals) | Reduction | Time (sec) | BDD Size |
| pm1 | 144 | 84 | 1 | 12 |
| b9 | 338 | 138 | 4 | 63 |
| i4 | 496 | 318 | 6 | 44438 |
| s208 | 206 | 37 | 2 | 1901 |
| 9symml | 720 | 512 | 10 | 58 |
| cordic | 266 | 95 | 3 | 175 |
| apex6 | 1869 | 1054 | 47 | 256 |
| comp | 404 | 291 | 3 | 166125 |

Table 2: Complete optimization based in the *consistency network*.

the penalty in running time and memory usage is not significant. Thus we feel our procedure should take the same order of magnitude of running time as existing synthesis routines on more realistic designs, while providing substantial improvements to the quality of the resulting design.

## 5 Conclusion

To summarize, we have met our goal of establishing a sound and complete methodology for optimizing designs containing black boxes. We formalized the notion of a design containing black boxes, developed criterion for equivalence, and characterized all the flexibility available for synthesizing such designs; we pointed out the limitations in existing approaches. Preliminary experiments performed indicate that the additional flexibility can be useful for optimization, and the increased time taken for synthesis is acceptable. We have been offered a large hierarchical industrial design which contains many uninstantiated components; we are working on applying our procedure to it.

In practise, designers may have a partial characterization of the functionality of the black boxes in the design; we plan to extend our results to incorporate this. We also intend to address the verification problem for designs with black boxes. In the long term future, we would like to relate our results, which stem from work logic synthesis, to high-level synthesis. We feel existing high-level synthesis procedures lack a well defined characterization of the set of permissible implementations; we believe we can contribute to an enhanced understanding of these issues.

## References

[1] VSI Alliance. Virtual Socket Interface Proposal 1.0. `http://www.vsi.org/`, September 1996.

[2] D. P. Appenzeller and A. Kuehlmann. Formal Verification of a PowerPC Microprocessor. In *Proc. Intl. Conf. on Computer Design*, pages 79–84, Austin, TX, October 1995.

[3] D. Brand, R. Damiano, L. van Ginneken, and A. Drumm. In the driver's seat of BooleDozer. In *Proc. Intl. Conf. on Computer Design*, Boston MA, October 1994.

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

[5] J.R. Burch, D. Dill, E. Wolf, and G. DeMicheli. Modeling Hierarchical Combinational Circuits. In *Proc. Intl. Conf. on Computer-Aided Design*, 1993.

[6] K.-T. Cheng. Redundancy Removal for Sequential Circuits Without Reset States. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 12(1):13–24, January 1993.

[7] M. Damiani and G. De Micheli. Don't care Set Specifications in Combinational and Synchronous Logic Circuits. Technical Report CSL-TR-92-531, Stanford University, Computer Systems Laboratory, Stanford, CA 94305-4055, July 1992.

[8] L. Stok et al. BooleDozer: Logic Synthesis for ASICs. *IBM J. Res. and Devep.*, pages 407–430, July 1996.

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

[10] S.-Y. Huang, K.-T. Cheng, K.-C. Chen, and U. Glaeser. An ATPG-Based Framework for Verifying Sequential Equivalence. In *Proc. Intl. Test Conf.*, 1996.

[11] J. S. Jephson, R. P. McQuarrie, and R. E. Vogelsberg. A Three-Value Computer Design Verification System. *IBM J. Res. and Devep.*, pages 178–188, 1969.

[12] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. A Fully Implicit Algorithm for Exact State Minimization. In *Proc. of the Design Automation Conf.*, pages 684–690, San Diego, CA, June 1994.

[13] David E. Long. *Model Checking, Abstraction and Compositional Verification* . PhD thesis, Carnegie Mellon University, Pittsburgh, PA, July 1993.

[14] S. Malik. Analysis of Cyclic Combinational Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(7):950–956, July 1994.

[15] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw Hill, 1994.

[16] Hamid Savoj. *Don't Cares in Multi-Level Network Optimization.* PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, May 1992.

[17] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *Proc. Intl. Conf. on Computer Design*, pages 328–333, October 1992.

[18] R. E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, 1979.

[19] Vigyan Singhal. *Design Replacements for Sequential Circuits.* PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, 1996.

[20] R. Vaught. *Set Theory.* Birkhauser, 1995.

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