Logic Optimization and Equivalence Checking by Implication Analysis

Wolfgang Kunz, *Member, IEEE*, Dominik Stoffel, *Member, IEEE*, and Prem R. Menon, *Fellow, IEEE*

***Abstract*—This paper proposes a new approach to multilevel logic optimization based on *automatic test pattern generation* (ATPG). It shows that an ordinary test generator for single stuck-at faults can be used to perform arbitrary transformations in a combinational circuit and discusses how this approach relates to conventional multilevel minimization techniques based on Boolean division. Furthermore, effective heuristics are presented to decide what network manipulations are promising for minimizing the circuit. By identifying indirect implications between signals in the circuit, transformations can be derived which are "good" candidates for the minimization of the circuit. A main advantage of the proposed approach is that it operates directly on the structural netlist description of the circuit so that the technical consequences of the performed transformations can be evaluated in an easy way, permitting better control of the optimization process with respect to the specific goals of the designer. Therefore, the presented technique can serve as a basis for optimization techniques targeting nonconventional design goals. This has already been shown for random pattern testability [11] and low-power consumption [28]. This paper only considers area minimization, and our experimental results show that the method presented is competitive with conventional technology-independent minimization techniques. For many benchmark circuits, our tool Hannover implication tool based on learning (HANNIBAL) achieves the best minimization results published to date. Furthermore, the optimization approach presented is shown to be useful in formal verification. Experimental results show that our optimization-based verification technique works robustly for practical verification problems on industrial designs.**

***Index Terms*—ATPG, implication analysis, logic synthesis, logic verification, miter, permissible function, recursive learning, redundancy elimination, transduction.**

I. Introduction

MULTILEVEL logic optimization figures prominently in the synthesis of highly integrated circuits. The goal of multilevel logic optimization is transforming an arbitrary combinational circuit C into a functionally equivalent circuit C', circuit C' being less expensive than C according to some cost function. The cost function typically incorporates area, speed, power consumption, and testability as the main objectives of the optimization procedure. This research focuses on optimizing a given circuit with respect to its area, a minimal area representation of the circuit being a good basis for subsequent steps targeting high speed, low power consumption, and high testability.

The field of multilevel logic optimization is not as well delineated as the field of two-level optimization [7], and there exist many different views on the multilevel optimization problem. An early systematic approach was proposed by Ashenhurst and Curtis [2], [13] and is known as *functional decomposition*. Functional decomposition, in general terms, is the process of expressing a switching function of n variables as a composition of a number of functions, each depending on less than n variables. Due to their complexity, early methods based on functional decomposition have been of limited use in practice. However, research in this area is still active. Recent contributions [19], [25], [26], [38] are encouraging, and have proven to be very useful in *field programmable gate array* (FPGA) synthesis.

Presently, the most flexible and powerful synthesis techniques for combinational circuits are based on Boolean and algebraic manipulations of Boolean networks, pioneered by Brayton *et al*. [8]. Since they provide good optimization results and can handle circuits of realistic size, these methods have become widely accepted.

Even with much recent progress, e.g., [3], [8], [10], [14], [16], [18], [19], [25]–[27], [31], [32], [34], and [38], the size and complexity of today's integrated circuits leave multilevel logic optimization a major challenge in the field of computer-aided circuit design. In particular, high-memory requirements represent the dominating limitation for many methods.

An important attribute of most common synthesis procedures is that they divide the synthesis process into an technology-independent minimization phase and a *cell-binding* procedure which maps the design to a specific target technology *(technology mapping)*. However, the strict separation of logic minimization from the specific technical design information can sometimes be of disadvantage since the powerful concepts for deriving circuit transformations cannot be oriented at the specific technical data.

Therefore, an important goal of this research is to work toward general logic minimization techniques which operate directly on the structural gate netlist description of the circuit so that the specific technological information of the given gate library is immediately available to guide the optimization process.

Our work is motivated by recent advances in test generation. Over the years, considerable progress has been achieved in combinational *ATPG*, and it seems wise to utilize the power of modern *ATPG* methods also in synthesis. *ATPG* methods are

Manuscript received October 4, 1994; revised March 25, 1996. This paper was recommended by Associate Editor, F. Somenzi. This work was supported in part by NSF Grant MIP-9311185.

W. Kunz and D. Stoffel are with the Max-Planck-Society, Fault-Tolerant Computing Group, University of Potsdam, Potsdam, Germany.

P. R. Menon is with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003 USA.

Publisher Item Identifier S 0278-0070(97)04326-1.

attractive for two reasons. First, in order to obtain effective test sets, *ATPG* techniques operate directly on a gate-level description of the circuit. Second, *ATPG* methods are very memory efficient and typically have memory requirements linear in the size of the gate-level description.

An important contribution exploiting testing techniques in logic minimization has recently been presented by Entrena and Cheng [16]. They propose an extension to *redundancy removal* (see, e.g., [1]) and describe an effective method which is based on adding and removing connections in the circuit. The approach to be presented in this paper can be seen as a generalization of the technique in [16] applied to combinational circuits. The advantage of operating directly on the gate netlist has also been recognized by Rohfleisch and Brglez [32] who presented a technique based on *permissible bridges* which can effectively optimize a circuit after technology mapping.

The methods of [16] and [32] have been shown to be very useful for postprocessing networks that were preoptimized by traditional techniques. On the other hand, they only consider a restricted set of possible network manipulations and, therefore, do not provide the same reasoning power and flexibility as traditional technology-independent synthesis methods.

Therefore, our goal is to develop a multilevel logic minimization method which is competitive with technology-independent minimization techniques such as [8] and which uses a test generator as the basic Boolean reasoning engine. In this paper, we present a method which is general in the sense that, in principle, it can derive arbitrary transformations in a combinational network. The second contribution of this paper is to introduce a new heuristic guidance for logic optimization. We show how logic minimization (for area) can be guided effectively by a *single* heuristic concept: the optimization process can be controlled by analyzing implications between circuit nodes. The complexity of reasoning required to derive logic implications is seen to be related to the optimality of the circuit structure and is used in optimization.

A major strength of our method is that it efficiently identifies or creates *permissible functions* [27]. Therefore, it relates to Muroga's *transduction* method. There are two main ingredients to our method: the *D-calculus* of Roth [33] and *Recursive Learning* [22]. The latter, which is discussed briefly in Section II, is used to derive logic implications in combinational circuits. Analyzing implications is crucial for deriving good circuit transformations. In this aspect our method also relates to [3] and [18].

Throughout the paper, we attempt to relate the concepts of our *ATPG*-based method to common concepts in logic synthesis ("division," "permissible functions," "don't cares," and "common kernel extraction").

As pointed out [8], "division" is central to Boolean/algebraic methods of logic optimization. For example, take the function $y = ac + bc + ad + bd$. A simpler representation of the same function is $y' = (a+b)(c+d)$. This representation can be obtained by defining a division operation '$/$' such that $(ac+bc+ad+bd)/(a+b) = c+d$. The expression $a+b$ is referred to as a "divisor" of y. When developing an *ATPG*-based method for logic minimization, the following two central issues have to be addressed.

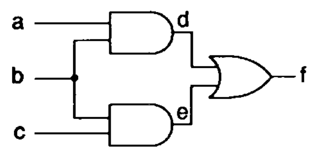

Fig. 1. Implications in combinational circuit [35].

1) How can an *ATPG*-based method perform Boolean division?
2) How can an *ATPG*-based method provide "good" divisors?

The paper is outlined as follows. The first of the above questions will be addressed in Section III, and the second question is discussed in Section IV. Section V describes and illustrates the general flow of our optimization procedure. Section VI is dedicated to an unconventional application of an optimization technique; we formulate the formal logic verification problem as an optimization problem and demonstrate how the described method can be tailored for logic verification. Section VII shows experimental results.

II. Indirect Implications

The optimization method to be presented heavily depends on analyzing implications derived by recursive learning [22]. A more general method to determine *implicants* in a multilevel network based on *AND/OR graphs* has been presented recently in [36]. The optimization procedure described in the following sections does not yet exploit the concepts of [36] but only uses recursive learning. Some previous results and terminology are briefly summarized. Recursive learning is a method to determine *all* value assignments which are necessary for the detection of a single stuck-at fault in a combinational circuit. This involves finding *all* value assignments necessary for the consistency of a given assignment of values to a set of nodes in the circuit. Determining value assignments necessary for the consistency of a given set of value assignments is often referred to as *performing implications*.

Consider the gate-level circuit of Fig. 1. Assume that the value assignments $a = 0, f = 1$ have been made in the circuit. By considering the truth table of an AND-gate, we imply $d = 0$. The variable d is an input variable of f, and by another implication, we obtain $e = 1$. Variables b and c are input variables of e, and we perform the implications $b = 1$ and $c = 1$. In [22], this type of implication has been referred to as *direct* implication.

As defined in [22], direct implications are identified by evaluating the value assignments at each gate and by propagating the signal values according to the connectivity in the circuit. An implication which cannot be determined in this simple way has been called *indirect*.

While the performance of direct implications is a straightforward procedure, it is more difficult to perform implications which are not direct. Reconsider the circuit in Fig. 1 and assume a value assignment of $f = 1$. A closer study reveals that $f = 1$ implies $b = 1$ [35]. The implication $f = 1 \Rightarrow b = 1$ is not direct, and more sophisticated techniques are required to derive such *indirect* implications. Recursive learning as presented in [22] represents a technique which allows us to derive *all* direct and indirect implications for a given situation of value assignments.

Indirect implications play an important role in our strategy for circuit optimization. As will be shown, *indirect* implications identify *promising* divisors for transforming the circuit. For a more detailed description on how the reasoning in recursive learning can be used to identify circuit transformations, see also [36].

III. Manipulating Combinational Networks by ATPG

Assume we are given a combinational circuit C with n primary inputs and m primary outputs and containing only the primitive gates AND $(\cdot)$, OR $(+)$, NOT $(^-)$. The AND- and OR-gates can only have two inputs. These restrictions are made in order to simplify the theoretical analysis of our method. In the following, such circuits will be referred to as *combinational networks*. (Of course, a reasonable implementation of our approach can also handle multi-input gates including NAND, NOR, and possibly XOR.) Furthermore, signals in the circuit can have constant values of '0' or '1.' All gates in the circuit have unique labels, and their output signals y_i realize Boolean functions $y_i(\underline{x}) : B_2^n \rightarrow B_2$ with $B_2 = \{0, 1\}$, where the variables $x_1, \cdots, x_n$ correspond to the primary input signals of the circuit C. Following the usual representation of a combinational circuit as a directed acyclic graph (DAG), we say, as in [8], that a signal f *lies in the transitive fanout of* y if and only if there exists a directed path from y to f in the image of C as DAG. Avoiding formalism, depending on the context, we will refer to the primary input signals and the output signals y_i of the gates in circuit C as "signals," "functions," or "nodes." Furthermore, we assume that there are no *external* don't cares; the function of the combinational network $C(\underline{x}) : B_2^n \rightarrow B_2^m$ with $B_2 = \{0, 1\}$ is completely specified. An extension to our method using external don't cares is possible but will not be further considered in this work.

Two combinational networks, C and C', are called *equivalent*, denoted $C = C'$, if they implement the same function $C(\underline{x}) : B_2^n \rightarrow B_2^m$ with $B_2 = \{0, 1\}$. They are called *structurally identical* or simply *identical* if there exists a one-to-one mapping between C and C', such that for every node y_i in C there is a y_i' in C' and vice versa, where y_i and y_i' implement the same function. We denote identical combinational networks by $C \equiv C'$.

A basic technique to describe many manipulations of switching functions is the well-known Shannon expansion. Let y be a Boolean function of n variables $x_1 \cdots x_n$. The Shannon expansion for y with respect to x_i is given by

$$\begin{aligned} y(x_1, \cdots, x_n) = {} & x_i \cdot y(x_1, \cdots, x_i = 1, \cdots, x_n) \\ & + \bar{x}_i \cdot y(x_1, \cdots, x_i = 0, \cdots, x_n). \end{aligned} \tag{1}$$

Shannon's expansion can be understood as a special case of an orthonormal expansion [6]

$$y(\underline{x}) = \sum_{i=1}^{k} f_i(\underline{x}) \cdot y_i(\underline{x})|_f$$

where the functions $f_i(\underline{x}), i = 1, 2, \cdots, k$ represent an orthonormal basis, i.e.,

i) $f_i(\underline{x}) \cdot f_j(\underline{x}) = 0, \quad \forall_i \neq j \in \{1, 2, \cdots, k\}$;

ii) $\sum_{i=1}^{k} f_i(\underline{x}) = 1$.

The functions $y_i(\underline{x})|_f : B_2^n \rightarrow B_2, B_2 = \{0, 1\}$ are called the *(generalized) cofactors* with respect to the functions $f_i(\underline{x})$. Shannon's expansion is the special case of the above expansion where

$$k = 2$$
$$f_1(\underline{x}) = x$$

and

$$f_2(\underline{x}) = \bar{x}$$

and x is some variable of y. Next, we consider another special case of the above orthonormal expansion where, more generally than in Shannon's expansion, we choose

$$k = 2$$
$$f_1(\underline{x}) = f(\underline{x})$$

and

$$f_2(\underline{x}) = \bar{f}(\underline{x})$$

where $f(\underline{x})$ is some arbitrary Boolean function $f(\underline{x}) : B_2^n \rightarrow B_2, B_2 = \{0, 1\}$. This means we obtain an expansion given by the following equation:

$$\begin{aligned} & y(\underline{x}) = f(\underline{x}) \cdot y(\underline{x})|_{f(\underline{x})=1} + \bar{f}(\underline{x}) \cdot y(\underline{x})|_{f(\underline{x})=0} \\ & (\text{short notation: } y = f \cdot y|_1 + \bar{f} \cdot y|_0). \end{aligned} \tag{2}$$

The terms $y(\underline{x})|_{f(\underline{x})=1}$ and $y(\underline{x})|_{f(\underline{x})=0}$ denote the cofactors of this expansion. In the special case of Shannon's expansion, the cofactors are chosen by restricting the original function with respect to a particular variable, as in (1). We obtain the cofactor for y with respect to a variable x by setting $x = 1$ in the expression for y, similarly, the cofactor for $\bar{x}$ results when setting $x = 0$. Note that there is no such simple rule in the more general case of (2).

Let the cofactors be denoted $y(\underline{x})|_{f(\underline{x})=V}$, with $V \in \{0, 1\}$. Further, let $y^*(\underline{x})|_{f(\underline{x})=V}$ denote an incompletely specified function $B_2^n \rightarrow B_3$, $B_3 = \{0, 1, X\}$. The cofactors in (2) must be chosen such that the following equation holds:

$$y(\underline{x})|_{f(\underline{x})=V} \supseteq y^*(\underline{x})|_{f(\underline{x})=V} := \begin{cases} y(\underline{x}), & \text{if } f(\underline{x}) = V \\ \text{X (don't care)} & \text{otherwise.} \end{cases} \tag{3}$$

It is easy to see why (3) is true. Assume that the truth table of y is divided into two parts such that f is false for all rows in the first part and true for all rows in the second part. If we first consider the part of the truth table of y for which f is true, we can set y to the don't care value for all rows in which f is false. This means that the cofactor function must only have the same value as y in those rows where y is not don't care. Therefore, any valid cofactor for the expansion of (2) *covers* (denoted '$\supseteq$') the incompletely specified function $y^*(\underline{x})|_{f(\underline{x})=1}$ as given by (3). This first part of the function is described by the expression $f(\underline{x}) \cdot y(\underline{x})|_{f(\underline{x})=1}$. In the second part, we are looking at those rows of the truth table for which f is false and obtain $\bar{f}(\underline{x}) \cdot y(\underline{x})|_{f(\underline{x})=0}$.

Equation (2) is the basis of our approach to transforming a combinational network. In order to relate our approach to the Boolean/algebraic techniques of [8], we can refer to function $f(\underline{x})$ as *divisor* of $y(\underline{x})$. Similarly, $y(\underline{x})|_{f(\underline{x})=1}$ can be referred to as *quotient,* and $\bar{f}(\underline{x}) \cdot y(\underline{x})|_{f(\underline{x})=0}$ represents the *remainder* of the division. Further, note that the combined don't care sets of the two cofactors in (3) are identical to the don't care set passed to a minimization algorithm for Boolean division, described in [8]. The main issue in our approach, as well as in [8], is to find appropriate (divisor) functions $f(\underline{x})$ such that the internally created don't cares as given by (3) provide "degrees of freedom" in the combinational network which can be exploited to minimize its area.

Obviously, the result of such an orthonormal expansion (or Boolean division) depends on how the don't cares are used in order to minimize the circuit. (Boolean division is not unique.) In [8], the don't cares are explicitly passed to an optimization run by ESPRESSO. The approach to be described here, proceeds in a different way and uses a *test generator* to determine the cofactors in the above expansion. As already observed by Brand in [4], circuitry tends to have an increased number of untestable single stuck-at faults if it is not properly optimized with respect to a given don't care set. This suggests that the don't cares created by the expansion of (2) can also cause untestable stuck-at faults which can be removed by the standard procedure of *redundancy elimination.* In fact, redundancy elimination is a simple way to minimize the circuit with respect to don't care conditions. Note that redundancy elimination does not require any *explicit* knowledge about the don't care sets. Throughout this paper, transformations are examined that create *internal* don't cares. However, these don't care conditions are not explicitly calculated or represented. They are only considered in our theoretical analysis to illuminate where the redundant faults to be eliminated come from.

Example 3.1: To illustrate how don't cares as given by (3) lead to untestable stuck-at faults, consider Shannon's expansion as an example, i.e., take the special case where the divisor $f(\underline{x})$ is some variable x_i. Note that the original function y is a possible cover for both $y^*(\underline{x})|_{f(\underline{x})=1}$ and $y^*(\underline{x})|_{f(\underline{x})=0}$ so that according to (2) we can form the expression $y = x_i \cdot y + \bar{x}_i \cdot y$. In Fig. 2(b), this is implemented as combinational circuit for the example, $y = a \cdot b + c$ and $x_i = a$. Note that the choice of the original function y as a (trivial) cofactor ignores the don't care conditions as given by (3). The fact that the cofactors are not optimized with respect to these don't cares leads to untestable stuck-at faults as indicated. (The cofactors are shaded grey). It is determined by *ATPG* that a, stuck-at-one, and a, stuck-at-zero, in the respective cofactor are untestable and can be removed by setting a to a constant one or zero, respectively. Fig. 2(c) shows the circuit after redundancy removal. Redundancy removal in this case obviously corresponds to setting x_i to one or zero in the respective cofactors of (1).

By viewing redundancy elimination as a method to set signals in cofactors to constant values, we have just described

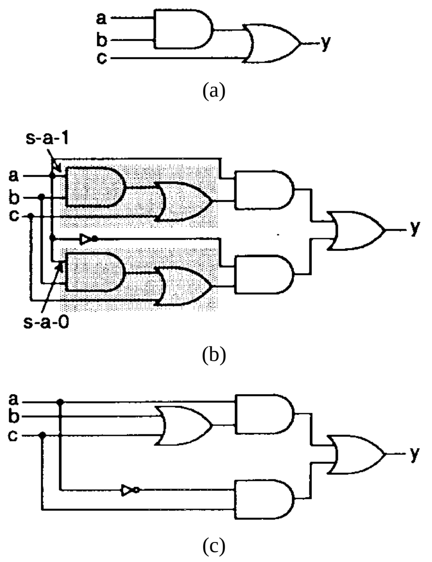

Fig. 2. Shannon's expansion by ATPG.

an *ATPG*-based method to perform a Shannon expansion. Clearly, it is not sensible to use a test generator in order to prove that x_i in the Shannon expansion of (1) can be set to constant values. However, this *ATPG* interpretation of Shannon's expansion is quite useful in the more general case of (2), i.e., when we expand in terms of some arbitrary function $f(\underline{x})$. In the general case, it is a priori *not* known if and what signals in the cofactors can be set to constant values. This, however, can be determined by means of a test generator.

Let y be an arbitrary node in a combinational network C and f be some Boolean function represented as a combinational network. The variables of f may or may not be nodes of the combinational network C. A new combinational network C' is constructed as follows. We duplicate all nodes in the transitive fanin of y so that there are two implementations of node y. This has been illustrated in Fig. 2(a) and 2(b). One version is ANDed with f, the other version is ANDed with $\bar{f}$, and the outputs of the AND gates are combined by an OR gate whose output replaces the node y in the original network. In the following, this construction will be represented by the equation $y = f \cdot y + \bar{f} \cdot y$.

Letting y and f be Boolean functions represented as a combinational network, we propose to expand function y in terms of function f by the following method:

1) transform network: $y = f \cdot y + \bar{f} \cdot y$;
2) redundancy elimination with an appropriate fault list. (4)

This expansion can also be understood as a special ATPG-based *transduction* [27] as it consists of a transformation and a reduction. In the following, we use the terms expansion and transduction synonymously. Since this ATPG-based transduction is one out of many possibilities to perform a Boolean division or orthonormal expansion in a combinational network, it is important to investigate what network transformations are theoretically possible using it. In the following theorem, we prove that the construction of (4) and redundancy elimination are sufficient for performing arbitrary Boolean transformations in a network.

Theorem 3.1: Let y^i be a node of a combinational network C^i. The gates in the combinational network can have no more than two inputs. Further, let f^i be a divisor which is represented as a combinational network and realizes a Boolean function of *no more than two* variables which may or may not be nodes in C^i such that

1) The transformation of node y^i into y^{i+1} given by

$$y^{i+1} = f^i \cdot y^i + \bar{f}^i \cdot y^i$$

followed by

2) Redundancy removal (with an appropriate fault list) generates a combinational network C^{i+1}.

For an arbitrary pair of equivalent combinational networks C and C' there exists a sequence of combinational networks $C^1, C^2, \cdots, C^k$ such that $C^1 \equiv C$ and $C^k \equiv C'$.

Proof: Switching algebra is isomorphic to two-valued Boolean algebra. A Boolean algebra can be defined by Huntington's axioms. First, we show that all operations (transformations) defined by the axioms can equally be performed by the above manipulations in a combinational network. For each axiom, it has to be shown that the corresponding transformation can be performed in both directions.

1) *Idempotent laws*: $a + a = a; a \cdot a = a$
for $a + a$:
$y = a + a, f = a$
Equation (4) $\rightarrow y = a \cdot (a + a) + \bar{a} \cdot (a + a) \rightarrow$ redundancy elimination: $y = a \cdot (a + a) \rightarrow$ redundancy elimination: $y = a$
$y = a, f = a + a$
Equation (4) $\rightarrow y = (a + a) \cdot a + (\overline{a + a}) \cdot a \rightarrow$ redundancy elimination: $y = (a + a) \cdot a \rightarrow$ redundancy elimination: $y = a + a$
for $a \cdot a$: analogous

2) *Commutative laws*: $a + b = b + a$; $a \cdot b = b \cdot a$
fulfilled by construction (definition) of primitive gates AND, OR.

3) *Associative laws*: $a + (b + c) = (a + b) + c$; $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
for $a \cdot (b \cdot c) = (a \cdot b) \cdot c$:
$y = a \cdot (b \cdot c),\ f = c$
Equation (4) $\rightarrow y = c \cdot (a \cdot (b \cdot c)) + \bar{c} \cdot (a \cdot (b \cdot c)) \rightarrow$ redundancy elimination: $y = c \cdot (a \cdot (b \cdot c)) \rightarrow$ redundancy elimination: (because of commutative laws) $y = (a \cdot b) \cdot c$; in opposite direction analogous
for $a + (b + c) = (a + b) + c$: analogous.

4) *Absorption*: $a \cdot (a + b) = a$; $a + (a \cdot b) = a$
for: $a \cdot (a + b) = a$
$y = a \cdot (a + b), f = a$
Equation (4) $\rightarrow y = a \cdot (a \cdot (a + b)) + \bar{a}(a \cdot (a + b)) \rightarrow$ redundancy elimination: $y = a \cdot (a \cdot (a + b)) \rightarrow$ redundancy elimination: $y = a$
$y = a, f = a + b$
Equation (4) $\rightarrow y = (a + b) \cdot a + (\overline{a + b}) \cdot a \rightarrow$ redundancy elimination $\rightarrow y = (a + b) \cdot a$
for: $a + (a \cdot b) = a$ analogous

5) *Distributive laws*: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$; $a + (b \cdot c) = (a + b) \cdot (a + c)$
for: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
$y = a \cdot (b + c),\ f = a \cdot b$
Equation (4) $\rightarrow y = (a \cdot b) \cdot (a \cdot (b + c)) + \overline{ab} \cdot (a \cdot (b + c)) \rightarrow$ redundancy elimination: $y = a \cdot (b + c) + a \cdot b \rightarrow$ redundancy elimination: $y = (a \cdot b) + (a \cdot c)$
$y = (a \cdot b) + (a \cdot c),\ f = a$
Equation (4) $\rightarrow y = a \cdot (a \cdot b + a \cdot c) + \bar{a} \cdot (a \cdot b + a \cdot c) \rightarrow$ redundancy elimination: $y = a \cdot (a \cdot b + a \cdot c) \rightarrow$ redundancy elimination: $y = a \cdot (b + c)$
for: $a + (b \cdot c) = (a + b) \cdot (a + c)$ analogous

6) *Universal bounds*: $0 + a = a; 0 \cdot a = 0; 1 + a = 1; 1 \cdot a = a$
for $0 + a = a$:
$y = 0 + a, f = a$
Equation (4) $\rightarrow y = a \cdot (0 + a) + \bar{a} \cdot (0 + a) \rightarrow$ redundancy elimination: $y = a \cdot (0 + a) \rightarrow y = a$
$y = a, f = 1$
Equation (4) $\rightarrow y = 1 \cdot a + 0 \cdot a \rightarrow$ redundancy elimination for stuck-at-one fault at a in second summand: $y = 1 \cdot a + 0 \rightarrow$ redundancy elimination for stuck-at-one at signal with constant one: $y = a + 0$

7) *Unary operation*: $a \cdot \bar{a} = 0; a + \bar{a} = 1$
for: $a + \bar{a} = 1$
$y = a + \bar{a}, f = 1$
Equation (4) $\rightarrow y = 1 \cdot (a + \bar{a}) + 0 \cdot (a + \bar{a}) \rightarrow$ redundancy elimination: $y = a + \bar{a} \rightarrow$ redundancy elimination: 1
$y = 1, f = a + \bar{a}$
Equation (4) $\rightarrow (a + \bar{a}) \cdot 1 + \bar{a} \cdot a \cdot (1) \rightarrow$ redundancy elimination: $y = a + \bar{a}$

In order to complete the proof, it must be shown that the above expansion also allows arbitrary sharing of logic. This follows easily from the following construction. Let C be the original network and C' be the target network. Further, let C_{tree} denote a network that has tree structure and results from C if all sharing of logic is removed by duplication. Similarly, let C'_{tree} denote the tree version of the target network. Consider the following construction. First we remove all sharing of logic between the different output cones of the original network C so that we obtain C_{tree} . It is easy to derive C_{tree} by the above expansion. Let y be some internal fanout branch and assume its stem is the output of an AND gate with input signals a and b. By choosing a divisor $f = a \cdot b$ and by performing the above expansion with an appropriate fault list, the AND gate is duplicated, and the fanout point is moved to the inputs of the AND gate. For other gate types, the procedure is analogous. This process is repeated until no more internal fanout points exist and C_{tree} has been obtained. After all sharing of logic has been removed, each output cone is isomorphic to a Boolean expression that can be manipulated arbitrarily as shown using the above axioms. Therefore, it is also possible to obtain the network C'_{tree} by the above expansion. The target network C' results if the duplicated logic is removed. This can be accomplished if equivalent nodes are substituted. If node y is to be substituted by node y', this can be accomplished by selecting $f = y'$ and performing the above expansion. This process can be repeated for well-selected nodes in C'_{tree} until network C' is reached. □

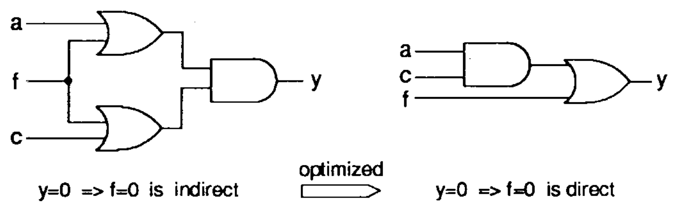

Fig. 3. Indirect implication and optimization.

Suppose C is the given combinational network and C' is the combinational network which is optimal with respect to the given cost function. Theorem 3.1 states that there always exists a sequence of the specified expansion operations such that the optimal combinational network is obtained. However, it does not say which divisors shall be used when applying (4). As stated in the theorem, if the network has gates with no more than two inputs, it is sufficient to only consider divisors created as function of two nodes in the network. This reduces the number of divisors that (theoretically) have to be examined. Of course, this restriction does not imply that more complex divisors are of no use in the presented expansion scheme. If more complex divisors are used, the network is transformed in bigger steps. Theorem 3.1 does not put any restriction on the choice of divisors to transform the network. Further degrees of freedom for the expansions lie within redundancy elimination. The result of redundancy elimination depends on what faults are targeted and in which order they are processed.

Theorem 3.1 represents the theoretical basis of a *general ATPG-based* framework to logic optimization. As mentioned, redundancy elimination and the transformation of (4) *per se* do not represent an optimization technique. However, they provide the basic tool kit to modify a combinational network. In order to obtain good optimization results, efficient heuristics have to be developed to decide what divisors to choose and how to set up the fault list for redundancy elimination. This will be described in the following.

IV. Identifying Divisors by Implications

Our method of identifying divisors has been motivated by an observation first mentioned in [30]. *Indirect* implications indicate suboptimality in the circuit. This is illustrated in Fig. 3.

In the left circuit of Fig. 3, we consider $y = 0$ as the initial situation of value assignments for which we can *indirectly* imply $f = 0$. This is can be accomplished by means of recursive learning. Note that the existence of the indirect implication $y = 0 \Rightarrow f = 0$ is due to the fact that the circuit is not properly optimized. In the optimized right circuit which is functionally equivalent to the left circuit, we note that the implication $y = 0 \Rightarrow f = 0$ is direct. One may verify that all examples of indirect implications shown in [22] or [35] are also due to poorly optimized circuitry. Apparently, indirect implications are a key to identifying and optimizing suboptimal circuitry.

Before developing an optimization strategy based on distinguishing between direct and indirect implications, we first study the role of implications in general for multilevel minimization.

Consider again the example of Fig. 3. For the above expansion, the circuit transformation of (4) requires that all combinational circuitry in the transitive fanin of y is duplicated before redundancy elimination is applied. This seems impractical and in the following, we, therefore, consider special cases of the expansion where only one cofactor has to be considered. These special cases are obtained if only such divisor functions f are considered which follow from y by *implication*. For the following lemmas, let f and y be nodes of the combinational network C such that f is not in the transitive fanout of y. (This restriction ensures that the circuit remains combinational after the transformation).

Lemma 4.1: Consider the transformation $T_1 : y' = y|_1 + \bar{f}$. Then $y' = y$ if and only if the implication $y = 0 \Rightarrow f = 1$ is true.

Proof:

"$\Rightarrow$":

$(y = 0 \Rightarrow f = 1) \Leftrightarrow (f = 0 \Rightarrow y = 1) \Rightarrow$ in Eq. (2) y_0 can be set to '1' and we obtain: $fy_1 + \bar{f}y_0 = fy_1 + \bar{f} = y_1 + \bar{f}$

"$\Leftarrow$":

$y_1 + \bar{f} = fy_1 + \bar{f}y_0$ (Eq. *)

$\Rightarrow$ $\bar{f}(y_1 + \bar{f}) = \bar{f}(fy_1 + \bar{f}y_0) \Leftrightarrow \bar{f} = \bar{f}y_0 \Rightarrow f + \bar{f} = f + \bar{f}y_0$

$\Leftrightarrow$ $1 = f + y_0$

$\Rightarrow$ for $f = 0$ it follows that $y_0 = 1$, for $f = 0$ the function y_0 assumes the same values as y (by definition), hence the implication $f = 0 \Rightarrow y = 1$ must be true and by contraposition the implication $y = 0 \Rightarrow f = 1$ must also be true if Eq. * can be fulfilled. □

Lemma 4.2: Consider the transformation $T_2 : y' = f + y|_0$. Then $y' = y$ if and only if the implication $y = 0 \Rightarrow f = 0$ is true.

Lemma 4.3: Consider the transformation $T_3 : y' = f \cdot y|_1$. Then $y' = y$ if and only if the implication $y = 1 \Rightarrow f = 1$ is true.

Lemma 4.4: Consider the transformation $T_4 : y' = \bar{f} \cdot y|_0$. Then $y' = y$ if and only if the implication $y = 1 \Rightarrow f = 0$ is true.

The lemmas state that implications determine exactly those functions f with respect to which function y has only *one* cofactor. In other words, in a combinational network, the expansion of Theorem 3.1 can be simplified without any circuit duplication if the specified implications are present. It is interesting to note that this does not sacrifice the generality of the approach.

Theorem 4.1: Let y^i be a node of a combinational network C^i. The gates in the combinational network can have no more than two inputs. Further, let f^i be a divisor which is represented as combinational network and realizes a Boolean function of no more than two variables which may or may not be nodes in C^i such that

1) The transformation of node y^i into y^{i+1} given by

$$\begin{array}{ll} y^{i+1} = y^i + \bar{f}^i & \text{for } y = 0 \Rightarrow f = 1 \\ y^{i+1} = f^i + y^i & \text{for } y = 0 \Rightarrow f = 0 \\ y^{i+1} = f^i \cdot y^i & \text{for } y = 1 \Rightarrow f = 1 \\ y^{i+1} = \bar{f}^i \cdot y^i & \text{for } y = 1 \Rightarrow f = 0 \end{array}$$

followed by

2) Redundancy removal (with appropriate fault list) generates a combinational network C^{i+1}. For an arbitrary pair of equivalent combinational networks C and C', there exists a sequence of combinational networks $C^1, C^2, \cdots, C^k$ such that $C^1 \equiv C$ and $C^k \equiv C'$.

Proof: Follows from the proof of Theorem 3.1, by noting that all functions y and divisors f used satisfy one of the implications specified above. □

Note that Lemmas 4.1–4.4 only cover those cases where a node y in a combinational network can be replaced by some equivalent function y'. A function at node y can also be replaced by some nonequivalent function y' if this does not change the function $C(\underline{x}): B_2^n \rightarrow B_2^m$ of the combinational network as a whole. Such functions are called *permissible functions* [27]. By considering permissible functions rather than only equivalent functions as candidates for substitution at each node, we exploit additional degrees of freedom as given by *observability don't cares* [8]. Permissible functions can also be obtained by recursive learning:

Definition 4.1: For an arbitrary node y in a combinational network C, assume the single fault y stuck-at-V, $V \in \{0, 1\}$. If $f = U, U \in \{0, 1\}$ is a value assignment at a node f which is *necessary* to detect the fault at at least one primary output of C, then $f = U$ follows from $y = \bar{V}$ by "D-implication" and is denoted $y = \bar{V} \xrightarrow{D} f = U$.

The conventional implications are a special case of such D-implications. Replacing the implications in Lemmas 4.1–4.4 by D-implications, we obtain the following generalization.

Theorem 4.2: Let f and y be arbitrary nodes in a combinational network C where f is not in the transitive fanout of y and both stuck-at faults at node y are testable.
The function $y': B_2^n \rightarrow B_2, B_2 = \{0, 1\}$ with $y' = y|_1 + \bar{f}$ is a *permissible function* at node y if and only if the D-implication $y = 0 \xrightarrow{D} f = 1$ is true.

Proof: "$\Rightarrow$":
If $y = 0 \Rightarrow f = 1$ is true, then $y = 0 \xrightarrow{D} f = 1$ is true. We partition the set of possible combinations of input assignments (rows in the truth table) into two disjoint subsets, where each fulfills one of the following conditions:

Case 1: ($y = 0$ and $f = 1$) or $y = 1$:
For these inputs, $y = 0 \Rightarrow f = 1$ is true and Lemma 4.1 applies, for these inputs y and y' always assume the same value.
Case 2: $y = 0$ and $f = 0$:
For these inputs, y' can have a different value than y. With $y = 0$ the function y' can only assume the faulty value '1.' However, this cannot lead to a wrong value at the primary outputs of C because the fault stuck-at-one at node y cannot be tested since $f = 0$ and $y = 0 \xrightarrow{D} f = 1$.

"$\Leftarrow$": The transformation is permissible if one of the following cases is fulfilled:

Case 1: y and y' are equivalent
Lemma 4.1 applies, if $y = 0 \Rightarrow f = 1$ is true then $y = 0 \xrightarrow{D} f = 1$ is true.
Case 2: $y = 1$ and $y' = 0$ and y' stuck-at-zero is untestable under this condition
$y = 1 \Rightarrow y_1 + \bar{f} = y' = 1$, i.e., this case cannot occur.
Case 3: $y = 0$ and $y' = 1$ and y' stuck-at-one is untestable under this condition.
With $y' = y_1 + \bar{f}, y = 0$ and $y' = 1$ can only occur if $f = 0$ is true. The term y_1 in the above transformation means that this node can be implemented as an arbitrary function assuming the same values as y for $f = 1$. As a special case, assume that $y_1 = y$. In this special case, if $y = 0$ then $f = 0$ is sufficient to produce a "faulty" signal '1' at node y'. Now consider the set of all test vectors for y stuck-at-one in the original circuit that produce $f = 0$. Every such test will result in a faulty response of the transformed circuit. Therefore, the transformation is only allowed if such a test does not exist. However, if a test for y stuck-at-one exists in general, it is required that there is none which produces $f = 0$. This means that $f = 1$ is necessary for fault detection and $y = 0 \xrightarrow{D} f = 1$ must be true. If this condition is necessary for the special case that $y_1 = y$, it is also necessary for the general statement since y is one of the possible choices to implement y_1. □

Theorem 4.3: Analogous to Lemma 4.2.
Theorem 4.4: Analogous to Lemma 4.3.
Theorem 4.5: Analogous to Lemma 4.4.

Theorems 4.2–4.5 represent the basis for the circuit transformations in our optimization method. As the transformations given in Theorems 4.2–4.5 represent special simplified cases of (2), they also provide simplified cases of (4). As will be illustrated in Section V, the constructions based on (4) and the above theorems provide good candidates for the expansion of Theorem 3.1.

Recursive learning can be used to determine all value assignments necessary to detect a single stuck-at fault, i.e., it is a technique to perform all D-implications. This is accomplished by two routines *make_all_implications()*, and *fault_propagation_learning()* as given in [22] if they are performed for the five-valued logic alphabet $B_5 = \{0, 1, X, D, \bar{D}\}$ of Roth [33]. Therefore, by recursive learning it is possible to derive all cases where Theorems 4.2–4.5 apply.

The number of implications and D-implications can be very large so that it is impossible to examine all transformations. At this point, however, we come back to the observation discussed earlier. Implications which can only be derived by "great effort" represent the promising candidates for the transformations as given in Theorems 4.2–4.5. These *indirect* implications are only a small fraction of all possible implications. In the following, we refer to a D-implication $y = V \xrightarrow{D} f = U, U, V \in \{0, 1\}$ as *indirect* if it can neither be derived by direct implication nor by *unique sensitization* [17] at the dominators [21] of y. In other words, all those necessary assignments obtained by the learning case of routines *fault_propagation-_learning()* and *make_all_implications()* are implied *indirectly* and provide the set of promising candidates for the circuit transformations.

As it turns out, the concept of relating the complexity of the implication problem to minimality of the combinational network permits a new and promising approach to guiding logic minimization techniques.

V. Optimization Procedure

The described concepts have been implemented as part of the HANNover Implication Tool Based on Learning (HANNIBAL) tool system. Table I summarizes the general program flow for circuit optimization. HANNIBAL performs logic optimization by applying the described concepts stepwise to all nodes in the combinational network. The optimization procedure moves from node to node in the combinational network. Experiments showed that the optimization results are only moderately sensitive to the order in which the different circuit nodes are processed. However, best results were generally obtained by processing the nodes according to their topological level moving from the primary inputs toward the primary outputs. For a selected node, recursive learning is used to derive promising divisor functions. The candidates found promising are stored in lists and tried in sequence. When identifying implications, it is important that we run recursive learning only for one node at a time and then transform the given node by the implications obtained. Therefore, after modifying the circuit, we have to update the data only for the current node.

For each candidate implication, the circuit is transformed according to the rules given in Section IV. After each transformation, redundancy elimination is employed. To make this process as fast as possible, the deterministic test set is always maintained for the most recent version of the circuit. After each circuit transformation, this test set is simulated to quickly discard many faults from further consideration so that a only few faults have to be targeted explicitly by deterministic ATPG. After redundancy elimination has been completed, it is checked whether the circuit became smaller or not. If it became smaller, the current circuit is maintained, otherwise the previous version is recovered. This is continued for all nodes in the network until no more improvements can be found. In HANNIBAL, several runs are made through the circuit varying the recursion depth and the number of candidate implicants being tried at each node in different runs.

For each step of redundancy removal, we determine the fault list as follows:

1) include in the fault list both stuck-at faults at all signals that were "touched" by recursive learning when deriving the current divisor;
2) exclude from the fault list, all faults in the circuitry added for the current transformation.

Limiting the fault list to signals being processed by the event-driven recursive learning routine proved to be a very good heuristic to speed up fault simulation and ATPG (up to a factor of 4) without significantly sacrificing optimization quality.

TABLE I
Program Flow of Hannibal in Optimization Mode

Names	# I/O	# Conn.	equivalent ?	CPU-time h : min : sec
c6288 with c6288.rug	32/32 32/32	4768 4695	yes	00 :01 : 32
M1.1.32 with M1.2.32	32/16 32/16	1508 1508	yes	00 : 00 : 11
M1.1.32 with M1.3.32	32/16 32/16	1508 1616	yes	00 : 01 : 27
M1.1.64 with M1.2.64	64/32 64/32	5988 5988	yes	00 : 01 : 39
M1.1.64 with M1.3.64	64/32 64/32	5988 6065	yes	00 : 03 : 26
M1.1.64 with M1.4.64	64/32 64/32	5988 5981	no	00 : 01 : 12
M2.1 with M2.2	74/32 74/32	4318 4318	no	00 : 03 : 27
M2.1 with M2.3	74/32 74/32	4318 4509	yes	00 : 03 : 50
M2.1 with M2.4	74/32 74/32	4318 4508	no	00 : 02 : 20

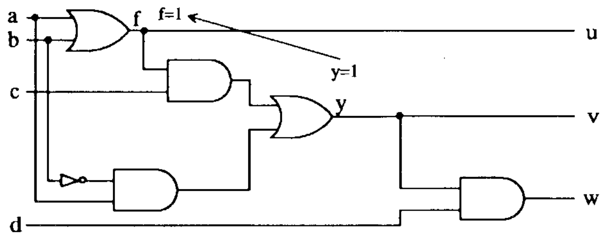

Fig. 4. Combinational network C with *indirect* implication.

Example 5.1—"Good" Boolean Division: Consider Fig. 4. By recursive learning, it is possible to identify the indirect implication $y = 1 \Rightarrow f = 1$. (Please refer to [22] for details of recursive learning.) The fact that the implication $y = 1 \Rightarrow f = 1$ is *indirect* means that it is promising to attempt a Boolean division at node y using the divisor f. This could be performed by any traditional method of Boolean division. Instead, we use the *ATPG*-based expansion introduced in Section III.

Applying Theorem 4.4, we obtain the combinational network as shown in Fig. 5. Actually, in this case we could also apply Lemma 4.3 since $y = 1 \Rightarrow f = 1$ is obtained without using any requirements for fault propagation. Note that Theorem 4.4 states that $y' = f \cdot y|_1$ is a permissible function for y. (In this case, y and y' are equivalent.) By transformation as shown in Fig. 5, we introduce the node $y' = f \cdot y$. Since y is used as a cover for $y|_1$, it is likely that the internal don't cares result in untestable single stuck-at faults. This is used in the next step (reduction).

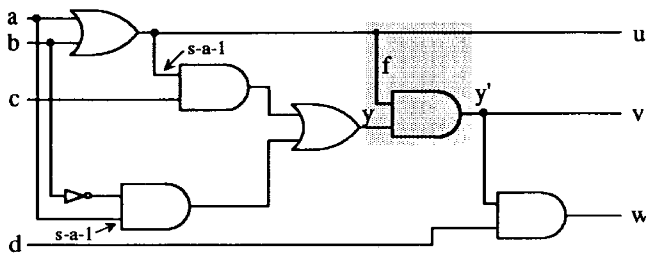

Fig. 5. Combinational network after transformation $y' = f \cdot y$ using internal node f as divisor.

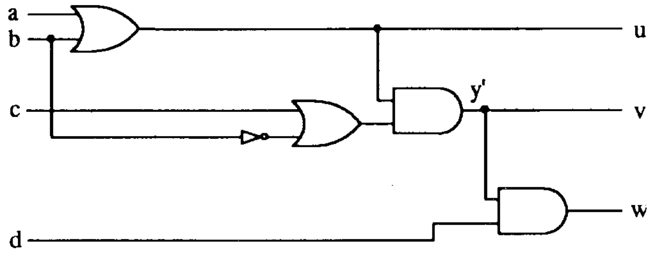

Fig. 6. Combinational network after reduction by redundancy elimination.

By *ATPG*, the untestable faults indicated in Fig. 5 can be identified. Performing redundancy removal (e.g., [1]) results in the minimized combinational network as shown in Fig. 6. Note that we have to exclude the stuck-at faults in the added circuitry in the shaded area of Fig. 5. If we performed redundancy elimination on line f in Fig. 5, we would return to the original network.

In the example, node y in Fig. 4 is implemented by $y = c(a + b) + a\bar{b}$. By *indirect* implication, we identified the Boolean divisor $f = a + b$ as "promising" and performed the (nonunique) division $c(a + b) + a\bar{b}/(a + b)$, resulting in $y' = (a+b) \cdot (c+\bar{b})$ in Fig. 6. Note that this is a *Boolean*—as opposed to *algebraic*—division [8]. As the example shows, *indirect* implications help to identify good divisors that justify the effort to attempt a Boolean division.

Example 5.2–"Common Kernel Extraction": Consider the circuitry of Fig. 7. The circuit implements two Boolean functions: $u = b(a+c)+cd$ and $v = b(f+e)+ed$, each of which cannot be optimized any further. Note, however, that the two functions have a common kernel, $b+d$, which can be extracted and shared so that a smaller circuit is obtained

$$\begin{aligned} u &= b(a + c) + cd = ab + c(b + d) = ab + cg \\ &\quad \text{with } g = b + d \\ v &= b(f + e) + ed = bf + e(b + d) = bf + eg \\ &\quad \text{with } g = b + d. \end{aligned}$$

It is interesting to examine how the suboptimality of the original circuit is reflected by the indirectness of implications.

Consider Fig. 7. By recursive learning, it is possible to identify the D-implication $d = 0 \xrightarrow{D} b = 0$. Remember that this means that $b = 0$ is necessary for detection of d, stuck-at-one. As can be noted, the necessary assignment $b = 0$ is not "obvious." It can neither be derived by direct implications nor by sensitization at the dominators of d. The reader may verify

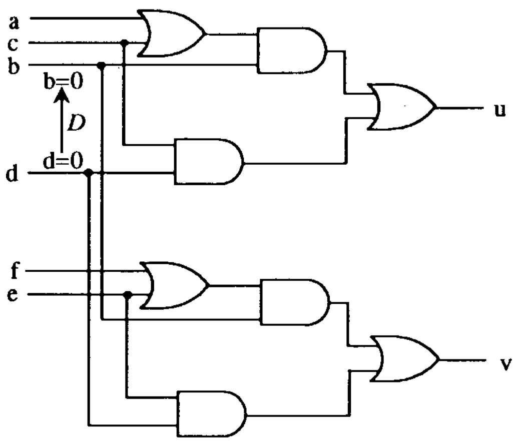

Fig. 7. Extraction of common kernel, $b + d$, by D-implication.

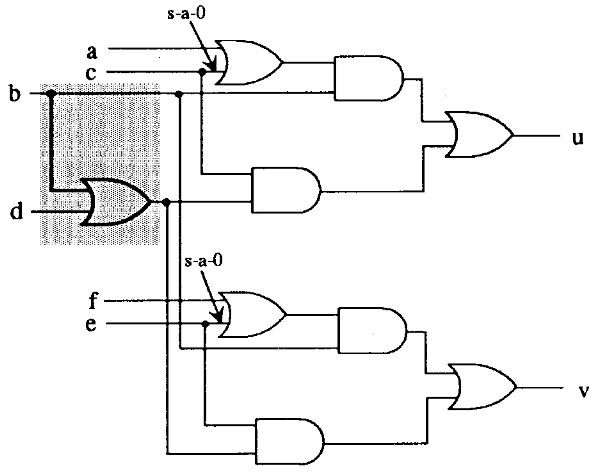

Fig. 8. Replacing d by permissible function $d' = b + d$.

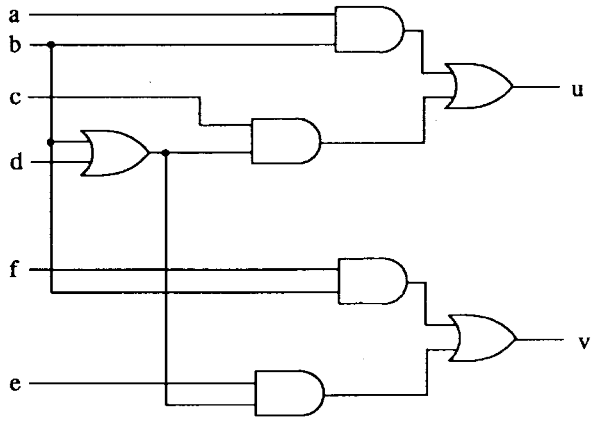

Fig. 9. Optimized circuit with logic sharing.

that $b = 0$ can be obtained by the learning case of recursive learning using *fault_propagation_learning()* [22].

Now the transduction is performed in the usual way. According to Theorem 4.3, the circuit can be modified as shown in Fig. 8, and redundancy elimination yields the optimized circuit in Fig. 9.

Note that our method can perform transformations which cannot be performed by the method of Entrena and Cheng

[16] and the method of [10]. To the best of our understanding, in the above example, the minimization cannot be obtained by only adding and removing connections as in [10] and [16]. This is because the methods of [10], [16] require the existence of gates of a certain type at the location where the added connection (gate) is anchored. Based on the expansion described in Section III, our approach uses a wider spectrum of circuit transformation. This could possibly impose higher computational costs, however, our results show that the heuristic strategy of only using indirect implications for circuit transformation can effectively limit the search space.

As presented in [22], recursive learning consists of two techniques, *make_all_implications()* and *fault_propagation_learning()*. It is interesting to note that implications obtained by *make_all_implications()* result in transductions that, in conventional terms, often lead to transformations that are most adequately described by Boolean division or Boolean resubstitution. This was illustrated in Example 5.1. If the implication is obtained by *fault_propagation_learning()* as in Example 5.2, the expansion often performs what is commonly referred to as common kernel extraction.

Limitations: 1) The examples also show the limitation of our method. By implication analysis we only consider divisors that are already present as nodes in the network. Therefore, we do not completely utilize the generality of our basic approach as given by Theorem 4.1. Extensions are under way to derive *implicants* and *D-implicants* [36] for a given node in the network which are not explicitly present as nodes in the network. AND/OR graphs, using which such implications can be derived, have been introduced in [36].

2) Our techniques operate on a gate-level netlist description. As mentioned, this is of advantage if specific technical information shall be considered in the optimization process. However, it has not yet been considered how the presented techniques can handle circuits with *complex gates* in an efficient way. Our tool, HANNIBAL, at this point, is limited to handling only the basic gate types, AND, OR, NAND, NOR, INV, XOR. Future work will therefore extend our techniques to handling complex gates so that arbitrary libraries can be processed.

VI. Application to Logic Verification

The described minimization approach can also be applied to logic verification. Formal logic verification of integrated circuits has become of great interest for many industrial designers and manufacturers of highly integrated circuits. Especially, in safety-critical applications, it is of great importance to verify that the implemented logic circuit is equivalent to its specification. When verifying digital circuits, an important subproblem is to check whether two combinational circuits are functionally equivalent. Traditionally, this problem is approached by generating a canonical (= unique) form of the circuits to be verified. The circuits are equivalent if their canonical forms are isomorphic. Unfortunately, canonical forms of Boolean functions may grow extremely large even for relatively small designs. The most compact canonical forms known to date are *Reduced Ordered Binary Decision Diagrams* (ROBDD's) [9] and related graph representations of Boolean functions. Therefore, binary decision diagrams (BDD's) have become very popular for solving logic verification problems. Some classes of circuits, however, are not amenable to a BDD analysis, since the size of the BDD's grows exponentially with the size of the circuit.

More recently, to overcome the limitations of BDD-based approaches, a different approach to logic verification has been proposed in [5], [23] which exploits the structural "similarity" between the designs. Instead of producing canonical forms these techniques extract the similarity between designs by ATPG and implications between signals in the two circuits. These techniques have only little memory requirements and proved successful in verifying circuits that cannot be verified by BDD-based approaches. Further developments based on these techniques have been proposed in [15], [20], [29], and [37]. Note, however, that such techniques may require excessive amounts of central processing unit (CPU)-time if the circuits have little structural similarity. Therefore, it is an important problem to study how to exploit the "similarity" between designs as efficiently as possible. In this section, we propose to use the presented optimization procedure for this purpose. There is a wealth of powerful synthesis methods, and it should be noted that many of these methods can also be useful in logic verification.

A. Logic Verification by Optimization

Logic verification as proposed by [5], [23] relies on combining the circuits to be verified as shown in Fig. 10. This construction has been called *miter* in [5] and represents a circuit, which maps the verification problem to solving the satisfiability problem for the output line e. In [5] and [23], a test generator is used for this purpose. Proving whether the output of the miter is satisfiable or not is generally a very complex problem. To overcome this difficulty, the approaches in [5], [23] make use of the fact that structural similarity between the two designs can help to break the problem down. In [23], implications are identified between different signals of the subcircuits, and these implications are stored at the respective nodes. Similarly, the complexity of the verification problem can be reduced by identifying signals in one circuit which can be used to substitute signals in the other circuit [5].

Making physical connections between the circuits or storing of implications have a similar effect. They simplify the reasoning for the satisfiability solver by introducing "short cuts" between the circuits so that the satisfiability solver does not necessarily need to fully exhaust both circuits. This has been shown in [5] and [23] if the satisfiability solver is a test generator and in [20] and [29] if the satisfiability solver is based on BDD's.

Note that this type of approach works well if the circuits for comparison have a certain degree of similarity but it may fail otherwise. Therefore, it is important to investigate what techniques can capture a wide spectrum of similarity in an efficient way. The techniques of [5] and [23] rely on relatively strict requirements. The approach of [5] requires that lines

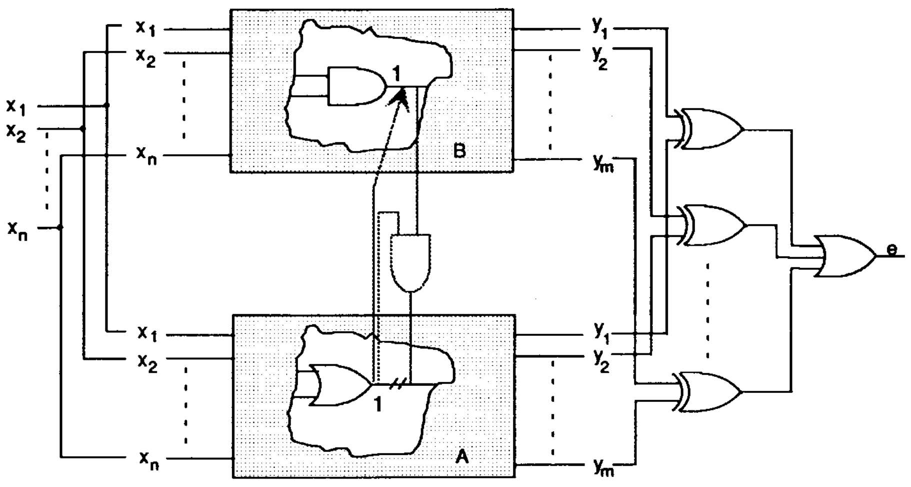

Fig. 10. Exploiting structural similarity in a miter.

in one circuit can be replaced by lines in the other circuit exploiting observability don't cares. In [20] or [23], it is required that there exist logic implications, e.g., $f = 1$ in circuit A implies $g = 1$ in circuit B. This can be a looser requirement than demanding a substitution, but on the other hand, [20] and [23] do not exploit observability don't cares.

Taking all of this into account suggests that the verification problem should be simplified effectively by performing logical transformations in the miter so that logic common to the two designs can be extracted and shared. If the circuits are equivalent, then one circuit must eventually be merged into the other circuit. As a special case, the substitutions of [5] perform such an operation. More generally, any known synthesis technique can be used to accomplish this task. The general goal is to *optimize* the miter. If the miter is reduced to a constant zero, the two circuits are proved equivalent. If this is not (or only partially) possible, then it must be attempted to generate a distinguishing vector using ATPG.

If the circuits have a fair amount of structural *similarity,* this means that the miter can be optimized by a sequence of fairly *local* circuit transformations. If the circuits become less similar, then deriving these transformations becomes more and more complex, and it becomes important to fully exploit the range and power of modern synthesis techniques. The advantage of formulating verification as a *miter optimization* problem is that the power of modern synthesis techniques becomes available to the difficult problem of logic verification.

As experimentally confirmed in Section VII, circuit transformations derived by indirect implications cover a large spectrum of the circuit manipulations performed in standard synthesis procedures like [8]. Further, since implications permit an easy and effective guidance of the optimization process, we base our verification procedure on the optimization procedure of Section V.

B. Heuristic Guidance in a Miter

Optimization in a miter has special characteristics which are discussed in this section with respect to the optimization procedure of Section V.

Selecting Implications: Using our approach, it must be attempted to identify implications that are valid between two nodes that belong to *different* subcircuits of the miter. If the corresponding transformations are performed, this introduces a sharing of logic between the circuits. Enforcing a sharing of logic between the circuits has two beneficial effects. It generally reduces the size of the miter, and it tends to increase the degree of similarity in the remaining, unshared parts of the circuits if the original circuits are equivalent. If the two networks are forced to share the same subfunctions, this leaves less "freedom" for the implementation of the remaining parts. This is illustrated in the following example.

Example 6.1: Fig. 11 shows two circuit examples that shall be verified to be equivalent. The circuits are combined to form a miter. For reasons of clarity, we depict the circuits without the extra logic to form the miter. Consider signal a in the upper circuit and signal z in the lower circuit. By recursive learning, it is possible to identify the D-implication $a = 0 \xrightarrow{D} z = 0$. The reader may verify that any test for a, stuck-at-one produces the value assignment $z = 0$. According to Section VI, we can perform the circuit transformation as shown in Fig. 12.

In the transformed circuit, untestable faults can be identified as indicated in Fig. 12. Removing these redundancies leads to the circuit in Fig. 13. Note that this transformation has not only introduced a sharing of logic between the two circuits and reduced the size of the miter, it has also increased the degree of structural similarity in the remaining unshared portions of the circuit. As a matter of fact, in this example, the remaining circuit portions are now structurally identical and can be shared by a sequence of very simple circuit transformations.

Substitution: Often, a lot of CPU-time can be saved by restricting the circuit transformations to node substitutions. Notice in Example 6.1 that node a in the upper circuit is substituted by node z in the lower circuit after removing the redundancy a, stuck-at-zero. Often it may be sufficient to restrict all transformations to only finding such substitutions [5]. In this case, if a transformation has been performed for an implication between two nodes y and f as given in Theorems 4.2–4.5, redundancy elimination needs to be performed only

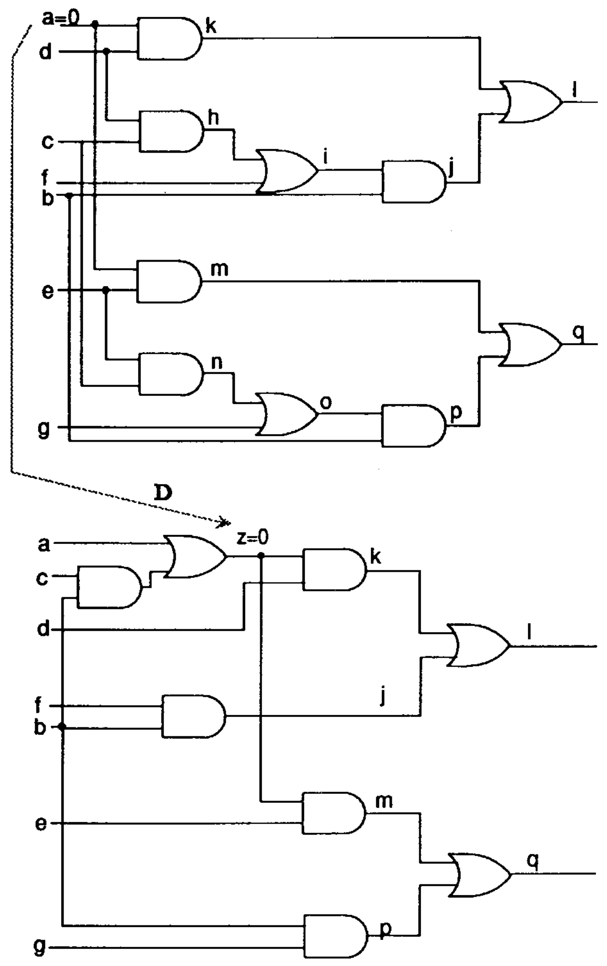

Fig. 11. Circuits to be verified.

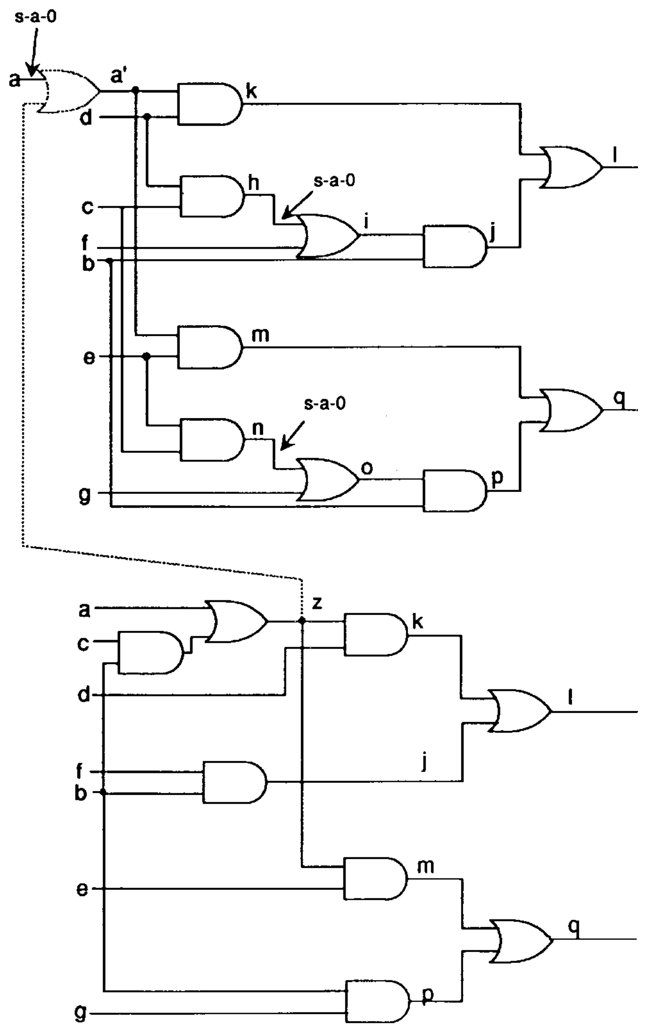

Fig. 12. Circuit transformation for D-implication $a = 0 \xrightarrow{D} z = 0$.

for the appropriate fault at signal y. This is faster than considering all faults in the circuit, but on the other hand, it overlooks miter transformations which cannot be obtained by a simple substitution. Therefore, we pass through the circuit several times. In the early passes of our verification procedure, we restrict the redundancy check to the node to be substituted. In the later passes we perform redundancy elimination in the whole circuit.

ATPG in a Miter: Finally, another important aspect should be considered when running the optimization approach of Section V in a miter. The described method heavily relies on evaluating circuit transformations by ATPG. However, for many target faults in the circuits for comparison, the ATPG problem becomes severely more difficult if the circuits are connected to form a miter. In fact, a large number of faults become redundant, but proving these redundancies practically has the same complexity as the verification problem itself. The reason for this is the global reconvergence created by the miter. Therefore, the ATPG tool may waste a lot of time on numerous target faults which eventually have to be aborted.

The effect of the global miter reconvergence on the ATPG process can be eliminated by the following trick. When performing ATPG or fault simulation, faults are declared "detected" as soon as the fault signal has reached the outputs of the subcircuits, i.e., if it has reached the inputs of the XOR-tree that forms signal e. In Fig. 10, these signals are labeled y_1 to y_m. Alternatively, the XOR-portion of the miter could be removed during the ATPG-procedure. Note that this is extremely important for an efficient ATPG-process.

VII. Experimental Results

The described methods have been implemented by making extensions to the HANNIBAL tool system. For efficient fault simulation, we integrated the public domain fault simulator FSIM [24] into HANNIBAL. HANNIBAL contains the recursive learning technique of [22] and has options to apply this technique to test generation [22], logic verification [23], and logic optimization. Section VII-A shows the results for logic optimization. The results for logic verification using implications and BDD's have been shown in [29]. In Section VII-B, we show results for the verification part of HANNIBAL enhanced by the optimization approach presented in this paper.

A. Results for Logic Minimization

We compare HANNIBAL with other state-of the art optimization tools. For a fair comparison, it is very important to take into account that several different ways of measuring

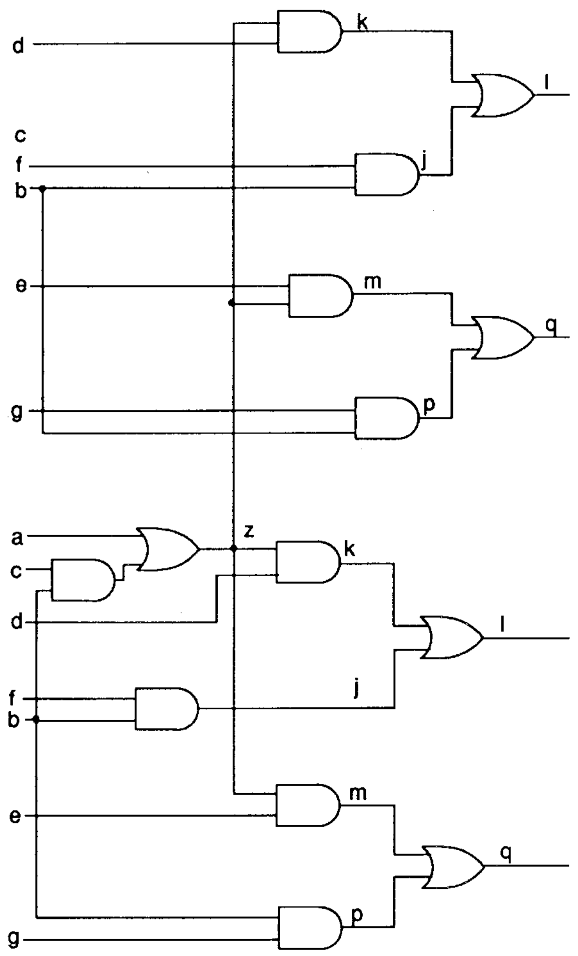

Fig. 13. Miter subcircuits after redundancy elimination.

TABLE II
MINIMIZATION RESULTS FOR SIS_1.2, RAMBO, AND HANNIBAL (SUN SPARC 5)

	Original	SIS_1.2 script.rugged	RAMBO_C	HANNIBAL	
Name	#conn. (#lit.)	#conn. (#lit.)	#conn. (#lit.)	#conn. (#lit.)	CPU time h : min : s
c1355	992 (562)	778 (554)	837 (546)	764 (540)	00:01: 30
c1908	1059 (769)	708 (535)	784 (551)	696 (511)	00:04:38
c2670	1559 (1023)	1082 (752)	1520 (816)	1064 (701)	00:03:37
c3540	2226 (1658)	1649 (1288)	1810 (1331)	1628 (1221)	01:13:57
c432	296 (270)	247 (205)	271 (207)	207 (181)	00:00:55
c499	368 (562)	776 (554)	837 (546)	348 (540)	00:00:16
c5315	3492 (2425)	2548 (1731)	3201 (1851)	2661 (1779)	00:15:39
c6288	4768 (3313)	4695 (3337)	3834 (3294)	3723 (3252)	00:29:18
c7552	4734 (3087)	3457 (2312)	3385 (2188)	2542 (1826)	00:36:00
c880	640 (433)	594 (413)	643 (410)	578 (400)	00:00:44
9symml	387 (237)	384 (186)	324 (217)	286 (217)	00:03:19
alu2	669 (453)	507 (361)	509 (359)	347 (274)	00:05:29
alu4	1299 (855)	975 (694)	1006 (722)	826 (646)	00:36:40
apex6	1214 (835)	1074 (743)	1327 (759)	1000 (697)	00:03:54
apex7	410 (289)	331 (245)	412 (251)	309 (229)	00:00:24
dalu	3533 (2610)	1364 (979)	2007 (1344)	1710 (1102)	02:10:18
frg2	2244 (2005)	1182 (887)	1734 (1157)	1315 (982)	00:27:35
pair	2795 (1803)	2356 (1602)	2594 (1636)	2155 (1636)	00:15:29
rot	1085 (764)	928 (672)	1093 (662)	834 (633)	00:02:20
term1	773 (456)	235 (170)	363 (248)	208 (149)	00:00:44
ttt2	434 (324)	303 (219)	300 (191)	204 (148)	00:00:26
x1	627 (445)	409 (298)	503 (333)	392 (298)	00:00:58
x3	1589 (1133)	1101 (787)	1547 (985)	1110 (1035)	00:03:59
x4	843 (1607)	512 (380)	777 (449)	583 (400)	00:03:34

the area costs are currently common practice. HANNIBAL and RAMBO [16] operate on a gate netlist description and measure the area in terms of the number of *connections*. A connection is an input to a gate with at least two inputs, i.e., single-input gates (inverters, buffers) are not counted. Technology-independent optimization tools like SIS measure the area in terms of numbers of literals. A literal is a variable or its complement used to describe the Boolean function at a node in the Boolean network. In a gate netlist, the number of literals can be obtained by counting the number of inputs of the *fanout-free zones* (FFZ's) in the network. For a fair evaluation of our tool, we present our results in terms of both, number of connections and number of literals. For RAMBO and HANNIBAL, the number of literals (factored form) has been obtained by reading the optimized circuits into SIS and postprocessing them such that a technology-independent factored form is obtained. For this purpose, we used a SIS script obtained from [12] which performs some standard network manipulations. To count connections for SIS, we map the optimized circuit to a generic library which contains the basic gates that are allowed in our netlist description.

Note that comparing connections or literals may slightly bias the results. Since RAMBO and HANNIBAL optimize in terms of connections whereas SIS uses literals, comparing connections can bias the results in favor of HANNIBAL and RAMBO. Comparing literals gives a certain advantage to SIS. Therefore, for all circuits we always present both area measures.

In all experiments, HANNIBAL passes through the circuit four times performing expansions at every node where recursive learning can identify indirect implications. The recursion depth is 'one' for the first two passes and 'two' for the final two passes. We also experimented with higher depth of recursion. It turned out that recursion depth higher than 'two' did not lead to improved optimization results because a transformation that can be derived by high recursion depth can usually also be obtained by a sequence of local transformations derived by small recursion depth. (Anyway, for the larger designs a recursion depth of 'four' and more is usually not affordable in terms of CPU-time.)

Table II shows results for SIS_1.2, RAMBO_C and HANNIBAL. SIS_1.2 is run using *script.rugged* which includes the powerful techniques of [31] and [34]. No preoptimization is used to process the circuits in RAMBO and HANNIBAL. As can be noted, for most benchmark circuits, HANNIBAL produces the smallest circuits. This is quite remarkable because it shows that most circuit manipulations performed by conventional technology-independent minimization techniques are covered by the netlist transformations presented in this paper. In particular, heuristic guidance by indirect implications proved surprisingly powerful.

In the next experiment, it is examined how much optimization is possible by HANNIBAL if the circuits are preprocessed

TABLE III
RESULTS FOR HANNIBAL AFTER PREPROCESSING WITH SIS (SUN SPARC 5)

Name	SIS_1.2 (script.rugged)	SIS_1.2 (script.rugged) + HANNIBAL	
	#conn. (#lit.)	#conn. (#lit.)	CPU time h : min : s
c1355	778 (554)	759 (543)	00:01: 20
c1908	708 (535)	690 (516)	00:02:38
c2670	1082 (752)	1021 (773)	00:02:37
c3540	1649 (1288)	1571 (1144)	00:34:18
c432	247 (205)	212 (165)	00:00:28
c499	776 (554)	763 (540)	00:01:26
c5315	2548 (1731)	2425 (1679)	00:07:16
c6288	4695 (3337)	3720 (3210)	00:31:28
c7552	3457 (2312)	2516 (1778)	00:22:24
c880	594 (413)	589 (416)	00:00:48
9symml	384 (186)	227 (178)	00:01:49
alu2	507 (361)	355 (279)	00:04:44
alu4	975 (694)	776 (596)	00:26:38
apex6	1074 (743)	999 (687)	00:02:36
apex7	331 (245)	294 (224)	00:00:14
dalu	1364 (979)	1171 (735)	00:44:02
frg2	1182 (887)	1052 (834)	00:05:46
pair	2356 (1602)	2149 (1509)	00:15:35
rot	928 (672)	822 (641)	00:01:45
term1	235 (170)	183 (131)	00:00:10
ttt2	303 (219)	225 (165)	00:00:24
vda	688 (615)	630 (566)	00:09:36
x1	409 (298)	377 (287)	00:00:31
x3	1101 (787)	995 (758)	00:03:48
x4	512 (380)	482 (357)	00:00:47

TABLE IV
EXPERIMENTAL COMPARISON WITH [10]

Name	SIS +		
	Rambo	Perturb/SIM	HANNIBAL
	# 2-input gates	# 2-input gates	# 2-input gates
c3540	988	938	922
c5315	1458	1321	1288
c6288	2334	1883	1872
c7552	1761	1426	1292
alu2	366	281	157
alu4	700	555	426
apex6	647	543	525
rot	569	452	452
term1	203	113	103
ttt2	174	118	117
x3	617	552	542

TABLE V
INDIRECT D-IMPLICATIONS BEFORE AND AFTER OPTIMIZATION

Circuit	Original	SIS (script.rugged)	HANNIBAL
c432	568	308	134
c880	580	269	138
c1355	942	320	72
c1908	6,239	755	835
c2670	4,041	617	419
c3540	16,770	18,082	10,767
c5315	9,383	3,149	3,157
c6288	3,516	3,275	618
c7552	27,644	3,189	1,118

by SIS. As shown in Table III, substantial area gains are possible in many cases. For seven out of 25 circuits, the gain is more than 20%. Also note that the CPU-times for HANNIBAL are significantly shorter in many cases if the circuits are run through a technology-independent minimization first, like in the experiment of Table III.

Finally, we also compare our results with [10]. In [10] the circuit is mapped to a library with only two-input gates, and results are only shown after preprocessing with SIS. Table IV shows the results for RAMBO (taken from [10]), PERTURB/SIMPLIFY [10], and HANNIBAL if the area is measured in terms of two-input gates. We take the subset of the above benchmarks for which results are shown in [10], and all circuits are preoptimized by SIS. As can be noted, HANNIBAL obtains smaller or equal circuits than RAMBO or PERTURB/SIMPLIFY for all circuits. Note that the results of HANNIBAL and RAMBO could be somewhat improved when compared with [10] if their cost function was changed to optimize the number of two-input gates.

Further experiments confirmed the heuristic that *indirect* implications indicate promising divisors. We examined how many indirect D-implications existed in the circuits before and after optimization. For the ISCAS85 circuits, Table V shows the number of indirect D-implications that have been identified by recursive learning with depth '2' for the original circuits as well as for the optimized circuits. We note that HANNIBAL reduces the number of indirect D-implications drastically for all circuits. It is interesting that this is also true for SIS in most cases, which confirms that optimization in general is related to reducing the number of *indirect* implications in the circuit. The results of Table V reflect that many (but not all) "good" divisors for optimization can be obtained by *indirect* implication. Also, note that Table V explains why the CPU-times for HANNIBAL are generally shorter if HANNIBAL is run after SIS. If SIS is used first, there are less indirect implications and, hence, less expansions need to be performed.

B. Results for Logic Verification

We demonstrate the performance of our verification technique based on optimization by means of the public domain multiplier c6288 which we verified against its optimized version. The optimized version has been obtained by SIS1.2 [8] using script.rugged. The other circuits listed in Table VI have been obtained from Mentor Graphics Autologic II Logic Synthesis Team. The designs are highly datapath oriented, contain multipliers and rotators and were created in Verilog, synthesized by Autologic II to a commercial ASIC vendor library. The designs were synthesized with different design goals in mind (such as area or performance). The test cases also contain (intentionally) nonequivalent designs.

TABLE VI
LOGIC VERIFICATION WITH HANNIBAL (SPARC 5)

Names	# I/O	# Conn.	equivalent ?	CPU-time h : min : sec
c6288 with	32/32	4768	yes	00 :06 : 20
c6288.rug	32/32	4695		
M1.1.32 with	32/16	1508	yes	00 : 00 : 35
M1.2.32	32/16	1508		
M1.1.32 with	32/16	1508	yes	00 : 06 : 58
M1.3.32	32/16	1616		
M1.1.64 with	64/32	5988	yes	00 : 09 : 45
M1.2.64	64/32	5988		
M1.1.64 with	64/32	5988	yes	00 : 18 : 29
M1.3.64	64/32	6065		
M1.1.64 with	64/32	5988	no	00 : 08 : 32
M1.4.64	64/32	5981		
M2.1 with	74/32	4318	no	00 : 06 : 27
M2.2	74/32	4318		
M2.1 with	74/32	4318	yes	00 : 21 : 00
M2.3	74/32	4509		
M2.1 with	74/32	4318	no	00 : 03 : 20
M2.4	74/32	4508		

The results show that logic verification based on the optimization procedure performs efficiently and robustly for these practical verification problems. The proposed technique may be outperformed by techniques such as [5] and [23] if the circuits have a high degree of similarity. On the other hand, for circuits with less structural similarity, logic verification by optimization provides a general framework for a more robust verification approach. Our results show that the optimization procedure of Section V can be tailored for efficient miter optimization. In all examined cases the miter could be minimized to a constant signal '0' within short CPU-times.

As described in Section VI, our verification approach uses two phases. The first phase performs substitution only, and the second phase considers more general transformations by running redundancy elimination in the whole circuit. In all cases, the first phase helped to significantly reduce the size of the miter before starting the more CPU-time expensive phase two. All circuit transformations have been derived by recursive learning with recursion depth '2'.

VIII. CONCLUSION

This work has introduced an *ATPG*-based generalization of Shannon's expansion that provides an adequate theoretical description for *ATPG*-based logic synthesis. Our research was originally motivated by the observation that indirect implications indicate suboptimal circuitry. We have presented an *ATPG*-based approach to logic optimization deriving circuit transformations from implications. It has been shown that implications can be used to determine for each node those functions in the network with respect to which this node has only one cofactor. Furthermore, it has been shown that the complexity of performing implications can be related to potential area reduction by Boolean division. This introduces new heuristic guidance and a different view on logic optimization problems. Our results clearly prove the great potential of our method. They also show that our notion of "indirect" implications is indeed most helpful to identify good Boolean divisors.

As has been shown, netlist optimization by HANNIBAL is competitive with technology independent minimization techniques. Future work will, therefore, exploit the main advantage of this approach. Optimization on the gate netlist provides much better insight in the technical properties of the design and, therefore, permits a better guidance when trying to achieve specific optimization goals. It has already been shown that the presented approach is very useful when optimizing for random pattern testability [11] and for low-power consumption [28].

Further, we formulated logic verification as an optimization problem and demonstrated the usefulness of our optimization approach for logic verification. Our verification method successfully verified a number of industrial designs. Current research examines extending the set of circuit transformations using the concept of AND/OR graphs [36]. This is expected to improve the capabilities of HANNIBAL for both logic synthesis and formal verification.

ACKNOWLEDGMENT

The authors would like to thank Mentor Graphics Inc. and in particular, A. Srinivasan for providing us with industrial designs for our experimental evaluation. The authors are also grateful to M. Cobenuss for his help in various ways.

REFERENCES

[1] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*. Rockville, MD: Computer Science, 1990.

[2] R. L. Ashenhurst, "The decomposition of switching functions," in *Proc. Int. Symp. Theory Switching*, Cambridge, MA, 1959, pp. 74–116.

[3] L. Berman and L. Trevillyan, "Global flow optimization in automatic logic design," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 557–564, May 1991.

[4] D. Brand, "Redundancy and don't cares in logic synthesis," *IEEE Trans. Comput.*, vol. C-32, pp. 947–952, Oct. 1983.

[5] ——, "Verification of large synthesized designs," in *Proc. Int. Conf. Computer-Aided Design*, Santa Clara, CA, Nov. 1993, pp. 534–537.

[6] F. Brown, *Boolean Reasoning*. Norwell, MA: Kluwer, 1990.

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

[8] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: Multi-level interactive logic optimization system," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 1062–1081, Nov. 1987.

[9] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, pp. 677–691, Aug. 1986.

[10] S. C. Chang and M. Marek-Sadowska, "Perturb and simplify: Multi-level Boolean network optimizer," in *Proc. Int. Conf. Computer-Aided Design*, San Jose, CA, Nov. 1994, pp. 2–5.

[11] M. Chatterjee, D. Pradhan, and W. Kunz, "LOT: Optimization with testability—New transformations using recursive learning," in *Proc. Int. Conf. Computer-Aided Design*, San Jose, Nov. 1995, pp. 318–325.

[12] K. T. Cheng, private communication, Apr. 1994.

[13] H. A. Curtis, "A generalized tree circuit," *J. Assoc. Comput. Mach.*, pp. 484–496, Aug. 1961.

[14] M. Daminani, J. C. Y. Yang, and G. De Micheli, "Optimization of combinational logic circuits based on compatible gates," in *Proc. Des. Automat. Conf.*, June 1993, pp. 631–636.

[15] P. Debjyoti and D. K. Pradhan, "New verification framework using generalized logic relationships and structural transformations," Dept. Comput. Sci., Texas A&M Univ., College Station, TX, Tech. Rep. 95-045, 1995.

[16] L. A. Entrena and K. T. Cheng, “Sequential logic optimization by redundancy addition and removal,” in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1993, pp. 310–315.
[17] H. Fujiwara and T. Shimono, “On the acceleration of test generation algorithms,” in *Proc. 13th Int. Symp. Fault Tolerant Comput.*, 1983, pp. 98–105.
[18] G. Hachtel, R. Jacoby, P. Moceyunas, and C. Morrison, “Performance enhancements in BOLD using implications,” in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1988, pp. 94–97.
[19] T. Hwang, R. M. Owens, and M. J. Irwin, “Exploiting communication complexity for multi-level logic synthesis,” *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 1017–1027, Oct. 1990.
[20] J. Jain, R. Mukherjee, and M. Fujita, “Advanced verification techniques based on learning,” in *Proc. Des. Automat. Conf. (DAC)*, June 1995, pp. 420–426.
[21] T. Kirkland and M. R. Mercer, “A topological search algorithm for ATPG,” in *Proc. 24th Des. Automat. Conf.*, June 1987, pp. 502–508.
[22] W. Kunz and D. K. Pradhan, “Recursive learning: A new implication technique for efficient solutions to CAD problems: Test, verification, and optimization,” *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 1143–1158, Sept. 1994.
[23] W. Kunz, “HANNIBAL: An efficient tool for logic verification based on recursive learning,” in *Proc. Int. Conf. Computer-Aided Design*, Santa Clara, CA, Nov. 1993, pp. 538–543.
[24] H. K. Lee and D. S. Ha, “An efficient forward fault simulation algorithm based on the parallel pattern single fault propagation,” in *Proc. Int. Test Conf.*, Sept. 1991, pp. 946–953.
[25] P. Molitor and C. Scholl, “BDD-based computation of common decomposition functions of multi-output boolean functions,” Universität des Saarlandes, Germany, Tech. Rep. TR-02/1993, 1993.
[26] R. Murgai, R. K. Brayton, and S. Vincentelli, “Optimum functional decomposition using encoding,” *Proc. ACM/IEEE Des. Automat. Conf. (DAC)*, 1994, pp. 408–414.
[27] S. Muroga, Y. Kambayashi, H. C. Lai, and J. L. Culliney, “The transduction method—Design of logic networks based on permissible functions,” *IEEE Trans. Comput.*, pp. 1404–1424, Oct. 1989.
[28] D. K. Pradhan, M. Chatterjee, and M. Swarna, “Implication-based gate level synthesis for low power,” Dept. Comput. Sci., Texas A&M Univ., College Station, TX, Tech. Rep. TR-95-042, 1995.
[29] S. Reddy, W. Kunz, and D. Pradhan, “A novel verification framework combining structural and OBDD methods in a synthesis environment,” in *Proc. Des. Automat. Conf. (DAC)*, June 1995, pp. 414–419.
[30] J. Rajski and H. Cox, “A method to calculate necessary assignments in algorithmic test pattern generation,” in *Proc. Int. Test Conf.*, 1990, pp. 25–34.
[31] J. Rajski and J. Vasudevamurthy, “Testability preserving transformations in multi-level logic synthesis,” in *Proc. Int. Test Conf.*, 1990, pp. 265–273.
[32] B. Rohfleisch and F. Brglez, “Introduction of permissible bridges with application to logic optimization after technology mapping,” in *Proc. EDAC/ETC/EUROASIC 1994*, pp. 87–93.
[33] J. P. Roth, “Diagnosis of automata failures: A calculus and a method,” *IBM J. Res. Develop.*, vol. 10, no. 4, pp. 278–291, July 1966.
[34] H. Savoj, R. K. Brayton, and H. Touati, “Extracting local don’t cares for network optimization,” in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1991, pp. 514–517.
[35] M. Schulz, E. Trischler, and T. Sarfert, “SOCRATES: A highly efficient automatic test pattern generation system,” *IEEE Trans. Computer-Aided Design*, vol. 10, Apr. 1991.
[36] D. Stoffel, W. Kunz, and S. Gerber, “AND/OR graphs,” Max-Planck Institut für Informatik, Germany, Tech. Rep. MPI-I-95-602, 1995.
[37] H. Walker and Shao, “Logic and transistor circuit verification using regression testing and hierarchical recursive learning,” submitted for publication.
[38] B. Wurth, K. Eckl, and K. Antreich, “Functional multiple-output decomposition: Theory and an implicit algorithm,” in *Proc. Des. Automat. Conf. (DAC)*, June 1995, pp. 54–59.

Wolfgang Kunz (S’90–M’91) was born in Saarbrücken, Germany, in 1964. He received the Dipl.-Ing. degree in electrical engineering from the University of Karlsruhe, Germany, in 1989. He received the Ph.D. degree from the University of Hannover, Germany, in 1992.

During 1989, he was Visiting Scientist at the Norwegian Institute of Technology, Trondheim, Norway. In 1989, he joined the Department of Electrical and Computer Engineering at the University of Massachusetts, Amherst, where he worked as a Research Assistant until August 1991. From October 1991 to March 1993, he worked with Institut für Theoretische Elektrotechnik, University of Hannover. Since April 1993, he has been with Max-Planck-Society, Fault-Tolerant Computing Group, University of Potsdam, Germany, and since 1995, he has a joint appointment as Research Assistant Professor with Texas A&M University, College Station. His research interests are in VLSI testing, formal verification, logic synthesis, and fault-tolerant computing.

Dr. Kunz is recipient of the 1996 IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS Best Paper Award.

Dominik Stoffel (M’95) was born in Xenia, OH, in 1966. He received the Dipl.-Ing. degree in electrical engineering from the University of Karlsruhe, Germany, in 1992.

From 1993 to 1994, he worked for Mercedes-Benz, Stuttgart, Germany, in the development of automated test systems for electronic car components. Since 1994, he has been with the Max-Planck Society, Fault Tolerant Computing Group, University of Potsdam, Germany. His research interests are in the field of VLSI design automation and fault-tolerant computing.

Prem R. Menon (M’70–SM’83–F’88) received the B.Sc. degree from Banaras Hindu University, India, in 1954 and the Ph.D. degree from the University of Washington, Seattle, WA, in 1962, both in electrical engineering.

From 1963 to 1986, he was with AT&T Bell Laboratories, where he was engaged in research in testing and simulation of digital circuits, and switching theory, and the development of test generation and simulation systems. He is currently a Professor in the Department of Electrical & Computer Engineering at the University of Massachusetts, Amherst. He is a coauthor of *Fault Detection in Digital Circuits* (Englewood, Cliffs, NJ: Prentice-Hall, 1971), *Theory and Design of Switching Circuits* (Rockville, MD: Computer Science, 1976), and a chapter in *Fault Tolerant Computing: Theory & Techniques*, (Englewood, Cliffs, NJ: Prentice-Hall, 1986). His current research interests include VLSI testing, testable design and logic synthesis.

Dr. Menon is a recipient of the Bell Laboratories Distinguished Technical Staff Award. He has served on the editorial boards of the IEEE TRANSACTIONS ON COMPUTERS and the *Journal of Design Automation and Fault Tolerant Computing* and on the program committees of several international conferences.