Multi-Level Logic Optimization by Implication Analysis*

Wolfgang Kunz

Max-Planck-Society
Fault-Tolerant Computing Group
at the University of Potsdam
14415 Potsdam, Germany
email: wkunz@rz.uni-potsdam.de

Prem R. Menon

Dept. of Electrical & Comp. Eng.
University of Massachusetts
at Amherst
Amherst, MA 01003, U.S.A.

Abstract — This paper proposes a new approach to multi-level logic optimization based on *ATPG* (*Automatic Test Pattern Generation*). Previous *ATPG*-based methods for logic minimization suffered from the limitation that they were quite restricted in the set of possible circuit transformations. We show that the *ATPG*-based method presented here allows (in principle) the transformation of a given combinational network C into an arbitrary, structurally different but functionally equivalent combinational network C'. Furthermore, powerful heuristics are presented in order 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. In particular, it is shown that *Recursive Learning* can derive "good" Boolean divisors justifying the effort to attempt a Boolean division. For 9 out of 10 ISCAS-85 benchmark circuits our tool HANNIBAL obtains smaller circuits than the well-known synthesis system SIS.

1 Introduction

Multi-level logic optimization figures prominently in the synthesis of highly integrated circuits. The goal of multi-level 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 representing a good basis for subsequent steps targeting high speed, low power consumption and high testability.

The field of multi-level logic optimization is not as well delineated as the field of two-level optimization [6]. Even with much recent progress, e.g. [7, 9, 11, 18, 20, 23], the size and complexity of today's integrated circuits leave multi-level 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.

*) Research reported supported in part by NSF Grant MIP-9311185

The goal of this research is to work towards a general *ATPG-based* approach to logic synthesis. In particular, our work is motivated by recent advances in test generation. Over the years, a lot of progress has been achieved in combinational *ATPG*, e.g. [10, 13, 22, 24] and it seems wise to utilize the power of modern *ATPG* methods also in synthesis. Most test generation techniques operate on a structural gate-level description of the given circuit (netlist). Therefore, *ATPG* methods are very memory efficient and typically have memory requirements linear in the size of the gate-level description. The main limitation of previous *ATPG*-based methods in logic minimization is the lack of generality. *Redundancy removal* (e.g. [1]) alone is usually not sufficient because of the limited types of circuit transformations that can be performed. Very encouraging results using an *ATPG* approach to multi-level optimization have recently been obtained by Entrena and Cheng [9]. Their method is an extension to redundancy removal 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 [9] applied to combinational circuits.

A further advantage of operating on a gate-level description is that it is closer to the physical reality of the design than, say, a functional description based on BDDs. This has been recognized by Rohfleisch and Brglez [21] and it was shown that such structural approaches have the advantage of being applicable after technology mapping.

The methods of [9] and [21] have been shown useful for postprocessing networks that were preoptimized by traditional techniques. However, they only consider a very restricted set of possible network manipulations and hence do not provide the same power and flexibility as traditional synthesis methods.

Presently, the most flexible and powerful synthesis techniques for combinational circuits are based on Boolean and algebraic manipulations of Boolean networks, e.g. [7], only these enabling exploitation of the full range of possible transformations in a combinational network. Since they provide good optimization results and because they can handle circuits of realistic size, these methods, pioneered by Brayton, et al. [6, 7], have become widely accepted.

As mentioned before, a long-term (and quite ambitious) goal of this research is to replace such algebraic or Boolean

techniques for network manipulation by *ATPG-based* methods. Therefore, we present a new approach to multi-level logic minimization using a test generator as the basic Boolean reasoning engine. As will be proved, the method is general in the sense that, in principle, it can perform arbitrary manipulations in a combinational network. Throughout the paper, it is attempted to relate concepts of our *ATPG*-based method to common concepts in logic synthesis ("division", "permissible functions", "don't cares").

As pointed out [7], "division" can be considered a central point in 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:

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

A major strength of our method is that it efficiently identifies or creates *permissible functions* [18]. Therefore, it also relates to Muroga's *transduction* method.

There are two main ingredients to our method: the *D-calculus* of Roth [22] and *Recursive Learning* [13, 14]. The latter represents an efficient technique to derive implications in a combinational circuit. Analyzing implications is crucial for deriving good circuit transformations. In this aspect our method also relates to [2] and [11].

2 Indirect Implications

The method to be presented heavily depends on analyzing implications being derived by recursive learning [13, 14]. Some previous results will be 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 situation of value assignments. 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 [13, 14], this type of implication has been referred to as *direct* implication. Direct implications are performed by evaluating the value assignments at each gate and by propagating the signal values according to the connectivity in the circuit. 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 [24]. 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 [13, 14] represents a technique which, more generally than [24], allows to derive *all*, direct and indirect implications for a given situation of value assignments.

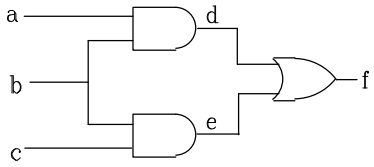

Fig. 1: Implications in combinational circuit [24]

Indirect implications play an important role in our strategy for circuit optimization. As will be shown, *indirect* implications indicate *promising* divisors for transforming the circuit.

3 Manipulating Combinational Networks by ATPG

Assume we are given a combinational circuit C with n primary inputs and m primary outputs containing only the primitive gates AND ($\cdot$), OR (+), NOT ($\bar{}$). The AND- and OR-gates can only have two inputs. These restrictions have been made in order to simplify the theoretical analysis of our method. 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'. More formally, such a combinational circuit can be understood as an interpretation of a Boolean network as defined e.g. in [12]. A combinational circuit consisting of only the above elements is in the following referred to as *combinational network*. All gates in the circuit have a unique label 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 $x_1,... x_n$ 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 [7], 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 ≡ C'.

Multi-level logic optimization relies heavily on the efficient use of factorization techniques. A fundamental factorization technique of switching algebra is the well-known Shannon expansion. Let y be a Boolean function of n variables $x_1...x_n$. The Shannon expansion for y with respect to x_i is given by:

$$y(x_1,... x_n) = x_i \cdot y(x_1,.. x_i = 1,.., x_n) + \bar{x}_i \cdot y(x_1,.. x_i = 0,.. x_n) \quad \text{(Eq. 1)}$$

The suggested approach to logic optimization is based on expanding Boolean functions as given in Eq. 2. Let f be a function of n variables $x_1...x_n$, $f(\underline{x})$: $B_2^n \rightarrow B_2$, $B_2 = \{0, 1\}$ and let $y(\underline{x})|_{f(\underline{x})=V}$, $V \in \{0, 1\}$ be an incompletely specified function: $B_2^n \rightarrow B_3$, $B_3 = \{0, 1, X\}$, defined as follows:

$$y(\underline{x})|_{f(\underline{x})=V} := \begin{cases} y(\underline{x}), \text{ if } f(\underline{x}) = V \\ \text{'X' (don't care) otherwise} \end{cases}$$

For an arbitrary Boolean function $f(\underline{x})$: $B_2^n \rightarrow B_2$ and $y(\underline{x})$: $B_2^n \rightarrow B_2$ with $B_2 = \{0, 1\}$ the following equation holds:

$$y(\underline{x}) = f(\underline{x}) \cdot y(\underline{x})|_{f(\underline{x})=1} + \bar{f}(\underline{x}) \cdot y(\underline{x})|_{f(\underline{x})=0} \quad \text{(Eq. 2)}$$

short notation: $y = f \cdot y|_1 + \bar{f} \cdot y|_0$

In the special case, that $f(\underline{x}) = x_i$, Eq. 2 is identical to Shannon's expansion. Note that Eq. 2 is a special case of a well-known generalization of Shannon's expansion using an orthonormal basis (see e.g. [5]).

It is easy to see why Eq. 2 is true: 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 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 $\bar{f}(\underline{x}) \cdot y(\underline{x})|_{f(\underline{x})=0}$.

In the case of Shannon expansion, $f(\underline{x}) = x_i$, and the expressions $y(\underline{x})|_{f(\underline{x})=1}$ and $y(\underline{x})|_{f(\underline{x})=0}$ are often called the *cofactors* of y with respect to the *variable* x_i [6]. More generally, the partially specified functions $y(\underline{x})|_{f(\underline{x})=1}$ and $y(\underline{x})|_{f(\underline{x})=0}$ can be termed as *cofactors* of y with respect to the *function f*.

Eq. 2 is the basis of our approach to transforming a combinational network. In the well-known notation of [7], we 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. The main issue of this approach is to divide function $y(\underline{x})$ by appropriate divisors $f(\underline{x})$ such that the exploitation of the internally created don't cares results in a reduction of the circuit.

Obviously, the result of such a Boolean division depends on how the don't cares are used in order to minimize the circuit. (Boolean division is not unique.) As already observed in [3], circuitry may contain untestable single stuck-at faults if it is not properly optimized with respect to a given don't care set. The approach to be presented will exploit don't cares for minimization, exclusively, by simply performing redundancy elimination. Throughout this paper we examine transformations that create *internal* don't cares. However, these don't care conditions are only considered in our theoretical analysis in order to keep track of where the redundant faults to be eliminated come from. They are not used in our constructions.

As an example, consider the special case where the divisor $f(\underline{x})$ is some variable x_i. Take function y as a possible cover for both $y(\underline{x})|_{f(\underline{x})=1}$ and $y(\underline{x})|_{f(\underline{x})=0}$, i.e. we form $y = x_i \cdot y + \bar{x}_i \cdot y$. Suppose this is implemented by some combinational logic. The fact that the cofactors y are not carefully optimized with respect to the don't cares existing according to Eq. 2 explains why we obtain untestable stuck-at faults in the cofactors. By *ATPG* we can find that x_i , stuck-at-1 and x_i , stuck-at-0 in the respective cofactor are untestable and can be removed by setting x_i to a constant '1' or '0', respectively.

By viewing redundancy elimination as a method to set signals in cofactors to constant values, we have just described an *ATPG*-based method to perform a Shannon expansion. This *ATPG* interpretation of Shannon's expansion may seem awkward but it is quite useful in the general case where we expand in terms of some arbitrary function $f(\underline{x})$. In the general case, unlike in Shannon's expansion, 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. We therefore suggest the following two-step methodology:

1) Transformation: $y = f \cdot y + \bar{f} \cdot y$ (Eq. 3)
2) Reduction: Redundancy elimination in each cofactor

By combining the transformation of Eq. 3 with *ATPG*-based redundancy elimination we obtain an expansion that can also be seen as a special *transduction* (transformation and reduction) according to Muroga et al. [18]. We therefore refer to the above two-step methodology as "expansion" or "transduction" interchangeably. Since Eq. 3 is only one out of many possible transformations in a network and since redundancy elimination is only one out of many possibilities to simplify the representation of each cofactor, it is important to investigate what network manipulations are theoretically possible using the above transduction. In the following theorem we prove that by using Eq. 3 and redundancy elimination as the only means of manipulating a network, we do not lose any generality. It turns out that the above *ATPG*-based expansion can be used to perform arbitrary manipulations in a combinational network.

Theorem 3.1: Let y^i be a node of a combinational network C^i as defined above. Further, let f^i be a divisor which is real-

ized as a function of *no more than two* signals 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 $y^{i+1} = f^i \cdot y^i + \bar{f}^i \cdot y^i$ or its dual $y^{i+1} = (f^i + y^i)(\bar{f}^i + y^i)$ 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, ...C^k such that $C^1 \equiv C$ and $C^k \equiv C'$.

Proof: see [16]

Note that Theorem 3.1 only states the *existence* of a sequence of the specified transduction operations to transform a combinational network C into some other equivalent network C'. It does not say which divisors shall be used when applying Eq. 3. As stated in the theorem, it is sufficient to only consider two-input divisors created as a function of two nodes in the network. This drastically reduces the number of divisors that (theoretically) have to be examined. However, this restriction does not imply that more complex divisors are of no use in the presented transduction 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 transductions lie within redundancy elimination. The result of redundancy elimination depends on what faults are targeted and in which order they are processed.

In some sense, the transduction steps of Theorem 3.1 can be understood as an *ATPG*-based generalization of Shannon's expansion. In both cases, the circuit (function) is transformed by a) using Eq. 3 and then b) setting certain signals to a constant value. Both are methods to change the representation of a Boolean function or combinational network. Shannon's expansion is the classical approach to bring a Boolean function into its canonical representation. More generally, the transduction in Theorem 3.1 allows us to transform a combinational network into *any* equivalent one.

Theorem 3.1 represents the theoretical basis of a *general ATPG-based* framework to logic optimization. However, redundancy elimination and the transformation given by Eq. 3 *per se* do not represent an optimization technique. They only 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 section.

4 Identifying Divisors by Implications

Our method of identifying divisors has been motivated by an observation first mentioned in [19] and further discussed in [14]. *Indirect* implications indicate suboptimality in the circuit. This is illustrated in Fig. 2:

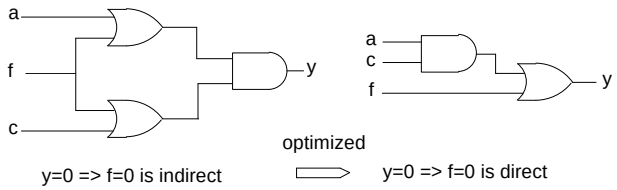

Fig. 2: Indirect implication and optimization

In the left circuit of Fig. 2 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 [13] or [24] are also due to poorly optimized circuitry. Apparently, indirect implications are the 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 multi-level minimization. Implications can be used in an easy way to identify divisors for Eq. 3. If a value assignment at a node y allows to imply a unique value assignment at node f, then Eq. 2 can be simplified as stated in the below lemmas. Let f and y be nodes of the combinational network C and f is not in the transitive fanout of y. The node f must not be in the transitive fanout of y, in order to ensure 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.

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 proofs can be found in [16]. The lemmas state that implications determine exactly those functions *f present in the network* with respect to which function y has only *one* cofactor. Such cases are of interest in logic optimization because they

often permit a simplification of the circuit. With recursive learning it is possible to derive all implications in a combinational network so that - if given enough time - all cases can be determined where the above lemmas apply.

Note however 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 non-equivalent function *y'* if this does not change the function C(x): $B_2^n \rightarrow B_2^m$ of the combinational network as a whole. Such functions are called *permissible function* [18]. 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* [7]. Permissible functions can also be obtained by recursive learning:

Definition: 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 $y=\overline{V}$ by "D-implication" and is denoted: $y=\overline{V} \xrightarrow{D} f=U$

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 4.1: 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: see [16]

Theorems 4.2 - 4.4: analogous to Lemmas 4.2 - 4.4

Theorems 4.1 - 4.4 represent the basis for the circuit transformations in our optimization method. As the transformations given in Theorems 4.1 - 4.4 represent special, simplified cases of Eq. 2, they also provide simplified cases of Eq. 3. As will be illustrated in Section 5, the constructions based on Eq. 3 and the above theorems provide good candidates for the setting up of the transduction of Theorem 3.1.

Recursive learning permits 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 [13] if they are performed for the five-valued logic alphabet $B_5 = \{0, 1, X, D, \overline{D}\}$ of Roth [22]. Therefore, by recursive learning it is possible to derive all cases where Theorems 4.1 - 4.4 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 4.1 - 4.4. 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* [10] at the dominators 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.

With the transduction procedure described above, we pursue a simple optimization strategy: At every node in the circuit we check by means of recursive learning whether indirect D-implications can be derived. If this is the case, the circuit is transformed according to Theorems 4.1 - 4.4. Then we perform redundancy elimination. If the resulting circuit is smaller than the previous one the transformation is maintained, otherwise it is reversed. This process is repeated until no more improvements can be found. In order to make the redundancy elimination process as efficient as possible the deterministic test set for the original circuit is stored and simulated after each transformation in order to quickly reduce the fault set for which redundancy has to be checked.

When identifying implications there are several advantages of recursive learning over other implication techniques like e.g. [2], [19], or [24]. First, recursive learning can identify more implications than previous techniques. If given enough time to perform a sufficient number of recursions, *all* implications can be identified. However, the CPU-time grows exponentially with the number of recursions. Fortunately, our experiments show that usually a small number of recursions is sufficient to identify most of the implications that exist in a realistic circuit. Secondly, recursive learning can perform D-implications. This is essential for identifying permissible functions and would not be possible, e.g., with the learning method of [24]. Thirdly, recursive learning identifies implications for a given node by local analysis. This avoids problems of incremental updating after modifying the circuit as they would occur for methods like in [2] or [19].

In the current implementation, after each transformation the fault list for redundancy identification is set up in the following simple way:

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.

5 Example

In this section it is illustrated how Theorems 4.1 - 4.4 are used to construct the transformations for the transduction of Theo-

rem 3.1 and how redundancy removal is used to reduce the circuit.

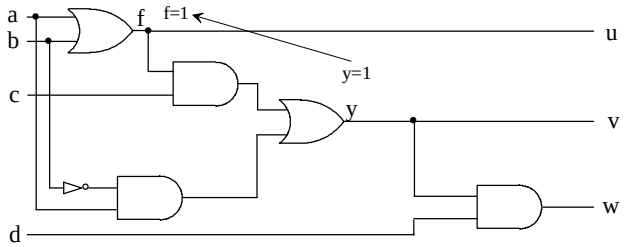

Fig. 3: Combinational network C with *indirect* implication

Consider Fig. 3. By recursive learning it is possible to identify the indirect implication $y=1 \Rightarrow f=1$. (Please refer to [14] 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 3.

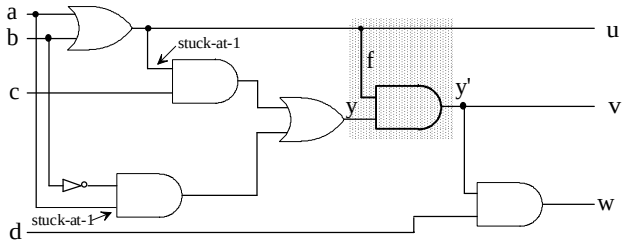

Fig. 4: Combinational network after transformation $y' = f \cdot y$ using internal node f as divisor

Applying Theorem 4.3 we obtain the combinational network as shown in Fig. 4. 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 4.3. 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. 4 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).

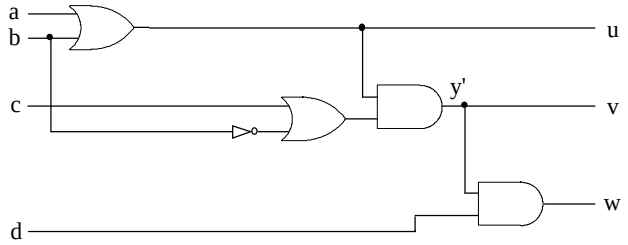

Fig. 5: Combinational network after reduction by redundancy elimination

By *ATPG* the untestable faults indicated in Fig. 4 can be identified. Performing redundancy removal (e.g. [1]) results in the minimized combinational network as shown in Fig. 5. Note that we have to exclude the stuck-at faults at the added circuitry in the shaded area of Fig. 4. If we performed redundancy elimination on line f in Fig. 4 we would return to the original network.

In the example, node y in Fig. 3 is implemented by $y = c(a+b) + a\overline{b}$. By *indirect* implication we identified the Boolean divisor $f = a+b$ as "promising" and performed the (non-unique) division $c(a+b)+a\overline{b} / (a+b)$, resulting in $y' = (a+b) \cdot (c+\overline{b})$ in Fig. 5. Note that this is a *Boolean* - as opposed to *algebraic* - division [7]. As the example shows, *indirect* implications help to identify good divisors that justify the effort to attempt a Boolean division.

On the other hand, the example also shows 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 presented in Section 3. Future work will investigate how the information processed by the recursive learning routines can be exploited more generally in order to *create* divisors that are not present in the network.

Note that our method is more general than the method of Cheng and Entrena [9]. In the above example, the minimization cannot be obtained by only adding and removing connections. Cheng's method can be viewed as a special case of the transduction of Section 3, where both, the divisor *and* the gates needed to form the expansion are already present in the network. Note that also Entrena and Cheng use necessary assignments in order to derive transformations. However, their reasoning to derive transformations is different. Being more restricted, the advantage of the method in [9] is that it can be predicted where the redundant faults will be created. Our method has the advantage that more general transformations are obtained and that the notion of "indirect" implications can be used to guide the search for good divisors. This leads to significantly better optimization results.

6 Experimental Results

The described methods have been implemented by making extensions to the tool HANNIBAL (HANNover Implication tool Based on Learning) [15]. Table 1 compares the optimization results of SIS 1.1 (based on [7]), RAMBO_C [9] and HANNIBAL for the ISCAS-85 circuits. Since HANNIBAL operates directly on a gate-level netlist description we post-processed the circuits optimized by HANNIBAL in the same way [8] as in [9] in order to compare literal counts between SIS, RAMBO_C and HANNIBAL. Since the literal counts may not always accurately reflect the number of gates and connections, we also list the number of connections when the circuit is mapped to a library that corresponds to the gate types in the ISCAS circuits.

Columns 3 to 6 show the results after optimizing the circuits with different scripts of SIS, *script.algebraic*, *script.boolean*, *script.rugged*, *script*. The script *script.rugged* is the recommended script and includes the methods of [20] and [23]. All scripts were run once; in the case of *script.rugged* we first applied the SIS command *full_simplify*. Columns 7 and 8 in Table 1 show the number of literals and connections for the circuits optimized by HANNIBAL as well as the required CPU-time. Column 9 shows the results for RAMBO_C as published in [9]. The last column shows the results if HANNIBAL is run after optimizing with SIS (*script.rugged*) first.

Name	Initial	SIS 1.1				HANNIBAL		RAMBO	SIS + H.
		alg	bol	rug	scr		time		
	#L #C	#L #C	#L #C	#L #C	#L #C	#L #C	h:min:s	#L	#L #C
c1355	992 994	670 911	554 771	**550** 768	554 771	**547** 759	0:09:11	546	544 759
c1908	1058 1046	564 723	552 734	**536** 713	552 734	**515** 711	0:15:35	551	517 701
c2670	1570 1465	840 1251	759 1130	**746** 1166	759 1130	**697** 1067	0:29:42	861	718 1008
c3540	2221 2141	1486 1821	1299 1870	**1288** 1818	1299 1870	**1188** 1707	1:53:05	1331	1154 1609
c432	335 344	252 315	240 310	**190** 246	240 310	**179** 224	0:01:35	207	161 217
c499	992 994	670 911	554 771	**550** 768	554 771	**547** 759	0:09:03	546	544 759
c5315	3531 3443	2008 2769	1815 2664	**1758** 2564	1815 2664	**1756** 2683	4:20:11	1851	1697 2456
c6288	4705 4734	3787 4294	3550 5222	**3337** 4373	3550 5222	**3252** 3768	3:48:23	3294	3240 3758
c7552	4750 4655	2584 3593	2297 3423	**2157** 3388	2584 3423	**1894** 2703	6:54:20	2188	1855 2607
c880	648 591	473 640	427 625	**415** 599	427 625	**398** 542	0:04:29	410	417 596

Table 1: Results for ISCAS-85 circuits, Sun SPARC 10

As can be noted, the optimization results of HANNIBAL are superior to the ones of SIS 1.1 in all cases except for c5315 (# connections). If HANNIBAL is used in addition to SIS further improvements are obtained in most cases. Furthermore, our literal counts are significantly smaller than the ones obtained by RAMBO_C [9]. This is due to the fact that HANNIBAL can perform a larger set of network manipulations than RAMBO_C and is heuristically guided by the indirectness of implications.

These results are very encouraging and clearly show the great potential of the presented *ATPG*-based method in logic optimization. Given our general *ATPG*-based framework for optimization many other heuristics seem possible to further improve the method. The CPU-times of our preliminary implementation are still unsatisfactory but can be improved drastically. For fault simulation we use the fault simulator FSIM [17]. In our preliminary implementation, FSIM is used externally so that a lot of CPU-time is wasted for the communication between the two tools. In our prototype implementation the memory requirements range from 1.9 Mbytes for c432 to 16 Mbytes for c7552. In SIS 1.1 using *script.rugged* memory requirements ranged from 20 Mbytes for c432 to about 60 Mbytes for c7552.

All experiments are run with a recursion depth of '2' for recursive learning. At recursion depth '2' the CPU-time for recursive learning is still small compared to the fault simulation times that dominate in our current implementation. We also experimented with higher depths of recursion. In the circuits examined, only few cases occured where this provided additional necessary assignments. Although these additional necessary assignments are "particularly good" candidates according to our heursitic, optimization results in these cases did not improve substantially. In the circuits examined, the improvement obtained by a single good transformation derived with recursion depth greater than '2' could also be obtained from several smaller optimization steps with recursion depths of '1' or '2'.

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

Table 2: Indirect D-implications before and after optimization

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. Table 2 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. This experiment confirms that optimization in general is related to reducing the number of *indirect* implications in the circuit. The results of Table 2 reflect that many (but not all) "good" divisors for optimization can be obtained by *indirect* implication. Another experiment was conducted including some *direct* implications into the optimization procedure: The result was increased CPU-time without any improvement of the final optimization result.

7 Conclusion

This work was originally motivated by the observation that indirect implications indicate suboptimal circuitry. We have

presented a general *ATPG-based* approach to logic optimization deriving circuit transformations from implications. The essence of the presented method is the D-calculus of Roth in combination with the precise implication procedure of [13, 14]. 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. Our preliminary results clearly prove the great potential of our approach. They also show that our notion of "indirect" implications is indeed most helpful to identify good Boolean divisors. The theoretical relationship between the complexity of performing implications and minimality of the circuit is subject to current research.

Note that the presented approach to multi-level optimization is also useful for logic *verification.* As shown in [15] storing implications can be very useful to simplify the verification process. Another possibility is to use the implications to transform the circuit by Theorems 4.1 - 4.4. It is interesting to note that the transformations conducted in Brand's method for logic verification [4] can be identified by D-implications as a special case. With heuristics adjusted appropriately, Brand's *ATPG*-based verification algorithm is a special case of the optimization method presented here.

Acknowledgments

We acknowledge the contributions of Hitesh Ahuja in the initial phase of this research. We are also grateful to Daniel Brand for valuable comments and to Martin Cobernuss for his help in various ways.

References

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

[2] Berman L., Trevillyan L.: "Global Flow Optimization in Automatic Logic Design", IEEE Transactions on Computer-Aided Design, vol. 10, No. 5, May 1991.

[3] Brand D.: "Redundancy and Don't Cares in Logic Synthesis", IEEE Trans. on Computers, vol. C-32, pp. 947-952, Oct. 1983.

[4] Brand D.: "Verification of Large Synthesized Designs", Proc. Int. Conf. on Computer-Aided Circuit Design, Santa Clara, Nov. 1993, pp. 534-537.

[5] Brown F.: "Boolean Reasoning", Kluwer Academic Publishers, Boston, MA 1990.

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

[7] Brayton R. K., Rudell R., Sangiovanni-Vincentelli A., Wang A. R.: "MIS: Multi-level Interactive Logic Optimization System", IEEE Trans. on CAD, CAD-6(6), pp. 1062-1081, Nov. 1987.

[8] Cheng K.T., April 1994, private communication

[9] Entrena L. A., Cheng K.T: "Sequential Logic Optimization by Redundancy Addition and Removal", Proc. Intl. Conf. on Computer-Aided Design, Nov. 1993, pp. 310-315.

[10] Fujiwara H., Shimono T.: "On the Acceleration of Test Generation Algorithms", in Proc. 13th Int. Symp. on Fault Tolerant Computing, 1983, pp. 98-105.

[11] Hachtel G. et al.: "Performance Enhancements in BOLD using Implications," Proc. Intl. Conf. on Computer-Aided Design, pp. 94-97, Nov. 1988.

[12] Hotz G.: "Einführung in die Informatik", Teubner Verlag, Stuttgart 1990.

[13] Kunz W., Pradhan D.K.: "Recursive Learning: An Attractive Alternative to the Decision Tree for Test Generation in Digital Circuits", Proceedings Intl. Test Conference, 1992, pp.816-825.

[14] Kunz W., Pradhan D.K.: "Recursive Learning: A New Implication Technique for Efficient Solutions to CAD Problems: Test, Verification and Optimization", accepted for publication in IEEE Transactions of Computer-Aided Design (probably Sept. 1994).

[15] Kunz W.: "HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning", Proc. Intl. Conference on Computer-Aided Design, Santa Clara, Nov. 1993, pp. 538-543.

[16] Kunz W., Menon P.: "Multi-Level Logic Optimization by Implication Analysis" Technical Report, Max-Planck Institut für Informatik, MPI -I-94-602, April 1994.

[17] Lee H.K., Ha D.S.: "An Efficient Forward Fault Simulation Algorithm Based on the Parallel Pattern Single Fault Propagation", Proc. Intl. Test Conference, pp. 946-953, Sept, 1991.

[18] Muroga S. et al.: "The Transduction Method - Design of Logic Networks Based on Permissible Functions", IEEE Trans. on Computers, Oct. 1989, pp. 1404-1424.

[19] Rajski J., Cox H.: "A Method to Calculate NecessaryAssignments in Algorithmic Test Pattern Generation", Proc., Int. Test Conf., 1990, pp. 25-34.

[20] Rajski J., Vasudevamurthy J.: "Testability Preserving Transformations in Multi-Level Logic Synthesis", Proc. Intl. Test Conference, 1990, pp. 265-273.

[21] Rohfleisch B., Brglez F.: "Introduction of Permissible Bridges with Application to Logic Optimization after Technology Mapping" , Proc. EDAC/ETC/EUROASIC 1994, pp. 87 - 93.

[22] Roth J.P.: "Diagnosis of Automata Failures: A Calculus and a Method", IBM Journal of Research and Development, Vol. 10, No. 4, July 1966, pp. 278-291.

[23] Savoj H., Brayton, R.K., Touati H.: "Extracting Local Don't Cares for Network Optimization", Proc. Intl. Conf. on Computer-Aided Design, Nov. 1991

[24] Schulz M., Trischler E., Sarfert T.: "SOCRATES: A highly efficient automatic test pattern generation system", IEEE Transactions on Computer-Aided Design, vol. 10. no.4, April 1991.