Accepted for publication ASP-DAC'97

AND/OR Reasoning Graphs for Determining Prime Implicants in Multi-Level Combinational Networks

Dominik Stoffel *Wolfgang Kunz* *Stefan Gerber*

Max-Planck Fault-Tolerant Computing Group
at the University of Potsdam, Germany

Abstract

This paper presents a technique to determine prime implicants in multi-level combinational networks. The method is based on a graph representation of Boolean functions called AND/OR reasoning graphs. This representation follows from a search strategy to solve the satisfiability problem that is radically different from conventional search for this purpose (such as exhaustive simulation, backtracking, BDDs).
The paper shows how to build AND/OR reasoning graphs for arbitrary combinational circuits and proves basic theoretical properties of the graphs. It will be demonstrated that AND/OR reasoning graphs allow us to naturally extend basic notions of two-level switching circuit theory to *multi-level* circuits.
In particular, the notions of prime implicants and permissible prime implicants are defined for multi-level circuits and it is proved that AND/OR reasoning graphs represent all these implicants.
Experimental results are shown for PLA factorization.

1 Introduction

This paper presents a method to calculate prime implicants in *multi-level* circuits, thus generalizing basic notions from classical two-level minimization theory to multi-level circuits. Our reasoning techniques to analyze multi-level circuits are based on a radically different way of solving satisfiability compared to conventional concepts. This motivates the following basic discussion attempting to give a global view on the nature of the algorithm.

Algorithms to solve satisfiability rely on appropriate *searching* techniques. Every search process can be viewed as a *traversal of a directed graph*. Standard literature, e.g. [14], distinguishes between two basic types of search graphs. In the most simple case to be considered, the graph is a so called *OR graph*. A node in the OR graph represents a given problem to be solved and each arc emerging from this node represents a possible move or decision that can be made at the current state of the search process. A solution is found by traversing the graph following a certain strategy and being guided by some heuristics exploiting problem-specific knowledge.

As is well-known however, for some problems it is useful to allow graphs with two types of nodes, AND nodes and OR nodes in order to represent a different type of search process. If at a given search state a certain move is made this may lead to *several* new problems that *all* have to be solved. Such *AND/OR reasoning graphs* or simply *AND/OR graphs* are the basis of many searching methods employed in the field of automatic theorem proving with predicate logic and are used in proof-by-refutation strategies. For a description of general problem-solving techniques in computer science and for more information on the basic concepts of OR graphs and AND/OR graphs, the reader may refer to a standard text book, e.g. [14].

Conventionally, in the field of switching theory, when exploring the functionality of a given Boolean function or combinational circuit, techniques enumerate through the finite Boolean space being defined by the set of all combinations of value assignments at the input variables. A common search scheme to prove satisfiability or related problems like test generation is the *decision tree*. A decision tree can be considered as an example for an *OR graph*.

Notice that there can also be a different interpretation to such graphs. They do not only represent a possible search process to solve a specific problem, for certain problem formulations (like satisfiability) they can also represent the underlying Boolean function. If exhaustive simulation is represented as tree we obtain a Shannon tree. This tree can be reduced by sharing isomorphic subtrees so that we obtain a *binary decision diagram* [2],[6]. Graph representations of Boolean functions by *ordered binary decision diagrams* [6] have encountered wide-spread popularity.

The following observation is crucial for the motivation of this paper: the conventional concepts to solve satisfiability and related problems in computer-aided circuit design like decision tree based backtracking, exhaustive simulation, Shannon trees or binary decision diagrams can be interpreted as OR trees and *not* as AND/OR trees.

Note that any Boolean expression can be understood as an AND/OR tree. However, such a general AND/OR tree does not decide whether the implemented Boolean function is satisfiable or not. (As mentioned, this problem is usually solved by resorting to an OR tree based enumeration). Here we are interested in specific AND/OR trees for Boolean functions that decide satisfiability. This is important for many applications in design automation and leads to other useful properties of these graphs. A technique which solves Boolean satisfiability based on AND/OR search is the recursive learning approach of [11].

The differences between the two searching schemes are of great practical interest in the field of design automation. In particular OR search techniques are hard to use for *systematic reasoning*. The goal of any reasoning is to derive the

logic consequences of a given assumption. Specifically, for some Boolean statement A we would like to derive some statement B that is true if A is true, i.e. $A \Rightarrow B$. Previous representations of Boolean functions are not well suited for this kind of task. For example, given statement A, a BDD-based approach cannot *derive* or *imply* statement B, it can only *check* if $A \Rightarrow B$ is true if both A and B are given.

Boolean reasoning techniques have always played an important role for two-level circuits. Here, we propose a reasoning technique for *multi-level* circuits. In the sequel we will speak of AND/OR reasoning graphs or simply AND/OR graphs interchangeably.

2 AND/OR Enumeration in Combinational Networks

In the following, we consider combinational networks C with n primary inputs and m primary outputs where all gates in the circuit have a unique label and their output signals y_i realize Boolean functions $y_i(\underline{x})$: $B^n{}_2 \rightarrow B_2$ 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 [4], that a signal *f lies in the transitive fanout of y* if and only if there exists a directed path from y to f. Similarly, a signal *f lies in the transitive fanin of y* if and only if there exists a directed path from f to y. 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. Extensions to our methods using external don't cares are possible, but will not be further considered here.

2.1 Revisiting basic notions of [11]

AND/OR enumeration is performed by injecting and reversing signal values and by evaluating their logic consequences using the ordinary event-driven implication techniques for combinational circuits. *Direct* implications play an important role in this reasoning. By direct implications as in [11] we understand the evaluation of the set of value assignments at every gate that has an event and the propagation of value assignments according to the connectivity in the circuit. To perform AND/OR enumeration in a multi-level combinational network we need the two basic notions of *unjustified gates* and *justifications* from [11]. For the time being, assume that we operate in a ternary logic alphabet (0, 1, X) where X is the don't care value. A signal is called *specified* if it is assigned the logic value 0 or 1, it is unspecified if it has the value X.

Def. 2.1: Given a gate g in a combinational network that has at least one specified input or output signal and the values at g are logically consistent: *Gate g* is called *unjustified*, if there are one or several unspecified input or output signals of g for which there exists a combination of value assignments yielding a conflict at g. Otherwise, g is called justified.

Unjustified gates represent OR nodes in the AND/OR tree. The special case of an unjustified gate where the specified signal is at the gate output is commonly referred to as *unjustified line* in test generation literature [1].

Def. 2.2: Let $f_1, f_2, ... f_n$ be some unspecified input- or output signals of an unjustified gate g and let $V_1, V_2, ...,V_n$ be logic values which specify them. The set of signal assignments, $J = \{f_1 = V_1 , f_2 = V_2, ... f_n = V_n\}$, is called *justification* for g, if the combination of value assignments in J makes g justified.

Justifications represent AND-nodes in the AND/OR tree. To complete the picture we need the following definition:

Def. 2.3: Let ${}^gC$ be a set of m justifications $J_1, J_2, ... J_m$ for an unjustified gate g. If there is at least one justification $J_i \in {}^gC$, i=1,2...m for any possible justification J^* of g, such that $J_i \subseteq J^*$, then set ${}^gC$ is called *complete*.

As an example, the following represents a complete set of justifications for a logic OR gate with five unspecified inputs and a logical 1 at the output: $C = \{J_1, J_2, J_3, J_4, J_5\}$ with $J_1 = \{a = 1\}$, $J_2 = \{b = 1\}$, $J_3 = \{c = 1\}$, $J_4 = \{d = 1\}$, $J_5 = \{e = 1\}$. Note, that for example the justification $J^* = \{a = 1, b = 0\}$ does not have to be in C since all assignments in J_1 are contained in J^*. Finally the following definition is needed:

Def. 2.4: Let R be the set of value assignments $f_i = V_i$ for those variables f_i in a combinational network whose value has been changed by making implications for a given set of value assignments S. Further, U is the set of variable assignments at the outputs of those unjustified gates, which have an input with a variable assignment contained in R. The set $E(S) = R \cup U$ is called the *event list E for S*.

In other words, when performing (e.g. direct) implications for a given set of value assignments S, the event list E contains all variables whose value has been changed. This includes the output signals of new unjustified gates. Furthermore, also the output signals of old unjustified gates are included if their status has changed, i.e. one of their inputs has assumed a different value.

2.2 AND/OR Reasoning Trees

The procedure in Table 1 performs AND/OR enumeration for an initial set of value assignments S at arbitrary signals in a combinational circuit. The technique in Table 1 results from routine *make_all_implications()* in [11] after removing the statements to extract necessary assignments. If the

initial set of value assignments is inconsistent, the routine produces a conflict. If the value assignments are satisfiable this is proved by the absence of a conflict after exhausting the complete AND/OR tree. (This is similar as in proof-by-refutation strategies for theorem proving.) The completeness of this method follows from [11].

Note an important difference to conventional techniques: unlike other techniques in computer-aided circuit design this method proves the satisfiability of a set of value assignments in a combinational circuit without actually generating a satisfying input vector. (This may only happen in special cases). When conventional methods check satisfiability, as a side result, they produce *sufficient solutions*, i.e. inputs that satisfy the function. In contrast, the AND/OR enumeration based approach presented here, as a side result, can generate the *necessary conditions* for a solution in terms of *implications* or *implicants*. This will be further developed in Section 4.

```
initially: r:=0;
/* this procedure operates on a global data structure representing the circuit,
it takes as input r, r_max and a set of initial value assignments S in a circuit */

and_or_enumerate(S, r, r_max)
{
    /* determine OR nodes of AND/OR tree */
    make all direct implications for S in circuit and
    set up a list U^r of unjustified gates in event list E(S);

    if (value assignments are logically inconsistent)
        return INCONSISTENT;

    /* determine AND nodes of AND/OR tree */
    if (r<r_max)
    {
        for (each unjustified gate g in U^r)
        {
            set up list of justifications ^gC^r;

            /* try justifications */
            for (each justification J_i ∈ ^gC^r)
                consistency_i := and_or_enumerate(J_i, r+1, r_max);

            /* check logic consistency */
            if (consistency_i = INCONSISTENT for all i)
                return INCONSISTENT;
        }
    }
    return CONSISTENT;
}
```

Table 1: Pseudo-code for AND/OR enumeration

We now introduce AND/OR trees constructed by the routine of Table 1. An AND/OR tree is a bipartite tree, one type of node is referred to as AND node, the other type is the OR node. The justifications as performed by *and_or_enumerate()* form the AND nodes. The situation of value assignments being implied from the justifications are represented by the OR nodes. *Justified* gates are OR nodes without successors, i.e. they are the leaves of the tree. Unjustified gates require justifications and have AND nodes as children.

Consider the circuit in Figure 1. We apply *and_or_enumerate()* for an initial situation of value assignments $S = \{y = 1\}$. The initial event list is $E = \{y = 1\}$. Node y in the circuit of Figure 1 becomes an unjustified line and the complete set of justifications is ${}^{y}C = \{\{g = 1\}, \{e = 1\}\}$. This produces the two AND nodes in the AND/OR tree of Figure 2. To distinguish AND nodes from OR nodes, AND nodes are marked by an arc. (This convention is adopted from standard literature.) For each justification direct implications imply logic signal values and produce new unjustified gates. Every value assignment forms an OR node in the tree. For $g = 1$ we imply $c = 1$, $f = 1$ and $u = 1$, where node f becomes a new unjustified gate. This requires new justifications and the technique continues to enumerate the AND/OR tree as shown in Figure 2 in a depth-first way.

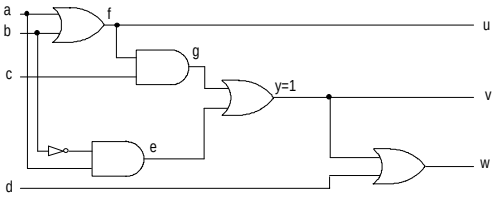

Figure 1: Circuit with value assignment $y = 1$

The proposed AND/OR trees are described more precisely by the following definitions:

Def. 2.5: An *AND/OR tree* is a bipartite rooted directed tree with two disjoint vertex sets V_{AND} and V_{OR}. The root node v_r is an element of V_{AND} . The terminal nodes (*leaves*) of the tree are elements of V_{OR}. Adjacent nodes belong to different vertex sets. Each node $v_{OR} \in V_{OR}$ has as attribute a variable assignment $f = V$ where f is element of a set of variables $\{f_1, f_2, ..., f_n\}$ and V is element of a set of values B. Each node $v_{And} \in V_{AND}$ has as attribute a set of variable assignments $S = \{f_1 = V_1, f_2 = V_2, ..., f_k = V_k\}$. Furthermore, each vertex v has as attribute an integer $level(v)$, such that

i) The root (AND) node v_r has
$$level(v_r) = 0.$$
ii) OR nodes v_{OR} have the same $level(v_{OR})$ as their immediate (AND) predecessors v_{pred}:
$$level(v_{OR}) = level(v_{pred}).$$
iii) AND nodes v_{AND} with their immediate (OR) predecessors v_{pred} have
$$level(v_{AND}) = level(v_{pred}) + 1.$$

Def. 2.6: An AND/OR tree with root node v_r can be associated with the AND/OR enumeration of Table 1 as follows:

i) each AND node v_{AND} belongs to a set $S = \{f_1 = V_1, f_2 = V_2, ..., f_k = V_k\}$ of variable assignments at nodes in the combinational network, where this set is given either by the initial set of variable assignments if $v_{AND} = v_r$ (root node), or by justifications for unjusti-

fied gates if $v_{AND} \neq v_r$ (intermediate nodes). If a set S turns out to be logically inconsistent, the corresponding AND node and all its successors are removed from the tree.

ii) each OR node v_{OR} belongs to a variable assignment $f = V$ at a node in the combinational network which is required for the logic consistency of the set S associated with the parent AND node of v_{OR}, i.e. an OR node belongs to a variable assignment in the event list $E(S)$. If $f = V$ is at the output of an unjustified gate g then v_{OR} has m AND children, each belonging to a justification $J \in {}^{g}C$, with $m = |{}^{g}C|$. If $f = V$ is at the output of a justified gate then v_{OR} is a leaf of the tree.

Such a tree is called the *AND/OR reasoning tree* for the initial set of value assignments S and the given combinational network.

Implications can be extracted in an easy way by examining the *topology* of the AND/OR tree. If all AND nodes that succeed a given OR node, say y, have succeeding OR nodes that all correspond to the same value assignment, then, these OR nodes with identical value assignments can be attached as OR nodes to the immediate predecessor of y. As an example, in Figure 2, the two AND nodes corresponding to justifications $\{g = 1\}$ and $\{e = 1\}$ have a succeeding OR node corresponding to $f = 1$. Therefore this OR node can be attached to the immediate predecessor of y. This process has been referred to as "learning" in [11] and is schematically shown in Figure 2. It can occur in any recursion level and the value assignments resulting in the previous level can change the course of subsequent enumeration so that more logical consequences can be examined faster. Recursive learning is one possible application of AND/OR trees and allows to determine all (not only direct) implications for the given set of value assignments in the circuit.

An important practical property of AND/OR graphs is that important information about the given problem can be derived without visiting the complete graph. In the above example the implication $y = 1 \Rightarrow f = 1$ can be derived already in the first recursion of *and_or_enumerate()*. Partial graphs can be visited by restricting the maximum recursion depth for *and_or_enumerate()* to some value r_{max}.

It is illuminating to apply *and_or_enumerate()* of Table 1 to two-level circuits. Consider the two-level circuit in Figure 3. Figure 4 shows the AND/OR graph for the assignment $y = 0$. AND/OR enumeration for the initial set of value assignments $S = \{y = 0\}$ at the output of a two-level SOP type circuit performs a tautology test. The SOP is a tautology if and only if a conflict is produced by *and_or_enumerate()*. As can be noted, the AND/OR tree for a *unate* SOP is very simple and has the same structure as the two-level circuit. The root AND node in the AND/OR tree corresponds to the OR gate in the circuit and the succeeding OR nodes correspond to the AND gates in the circuit. Obviously, this is because the AND gates in the circuit represent implicants for function y and therefore the value assignment $y = 0$ implies OR nodes which correspond to these implicants. Since the circuit implements a unate function the AND/OR tree terminates in the next level, all AND nodes have only one succeeding OR node representing a leaf of the tree. This reflects the well-known fact that tautology checking for unate functions is of polynomial complexity.

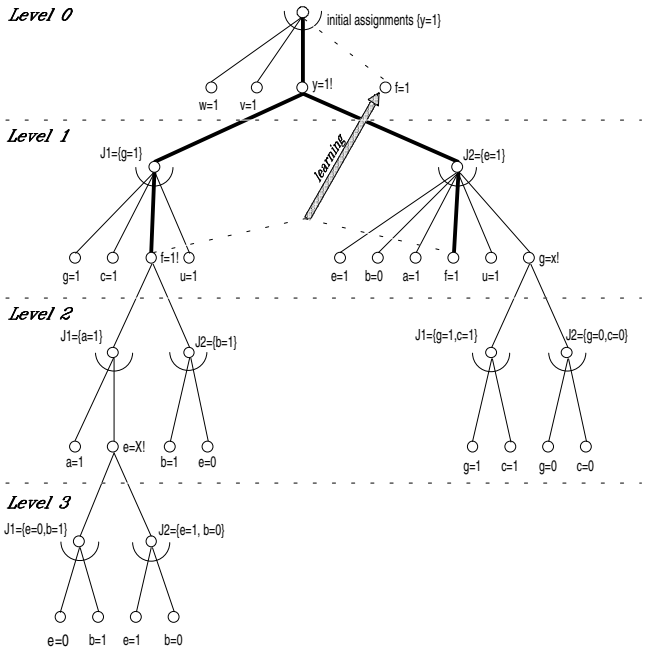

Figure 2: AND/OR tree for assignment $y = 1$ in the circuit of Figure 1

Let the AND/OR tree be levelized according to Def. 2.5, then each level consists of a set of AND nodes with their OR children. The following theorem holds:

> **Theorem 2.1:** Let y be the output signal of a two-level combinational circuit in SOP form. The AND/OR tree for the initial set of value assignments $S = \{y = 0\}$ (tautology test) has only two levels if the SOP is unate.

The fact that the AND/OR tree for a unate SOP has only two levels is related to the well-known result that all prime implicants in a unate SOP are essential, i.e. the unate SOP is a syllogistic formula [5]. If the circuit is not unate the AND/OR tree has to be continued after level 1 in order to explore the logic consequences which are not covered by the implicants included in the SOP.

The situation for the non-unate case is illustrated in Figure 5 and 6 where the circuit of Figure 3 is modified such that it becomes non-unate in variable c. Also in the case of a non-unate circuit, level 0 of the AND/OR tree reflects the implicants in the SOP. If the circuit is not unate however, the

AND/OR tree continues after level 1. (Now there are non-essential prime implicants). This is because the justifications at some unjustified line, e.g. $h = 0$ in Figure 5, produce events at other unjustified lines without justifying them. In Figure 5, the justification $c = 0$ at gate h produces a logical 1 at the input of gate j. This changes the status at gate j and represents an event so that the unjustified line $j = 0$ is added to the list of unjustified gates for the next recursion level.

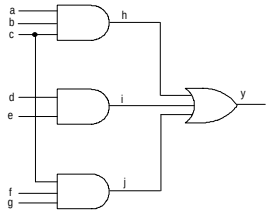

Figure 3: Unate two-level circuit

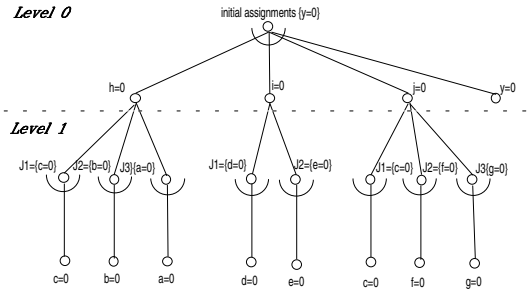

Figure 4: AND/OR tree for circuit in Figure 3

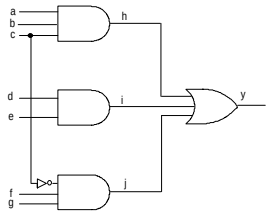

Figure 5: Non-unate circuit

Note that conventional OR search exploits the properties of unate functions only by additional heuristic guidance. For example, a decision tree based test generator usually employs a backtrace procedure, see [1], to direct the search such that non-unate signals are enumerated first. Similarly, in tautology checking by the *unate recursive paradigm* [3] heuristics are used to direct the Shannon expansions to the non-unate signals. AND/OR enumeration on the other hand does not require any such guidance. As the above examples illustrate it is an inherent property of AND/OR enumeration that the enumeration simplifies at the presence of unate signals or unate circuit partitions. This is not only of theoretical but also of practical importance as will be demonstrated in Section 5.

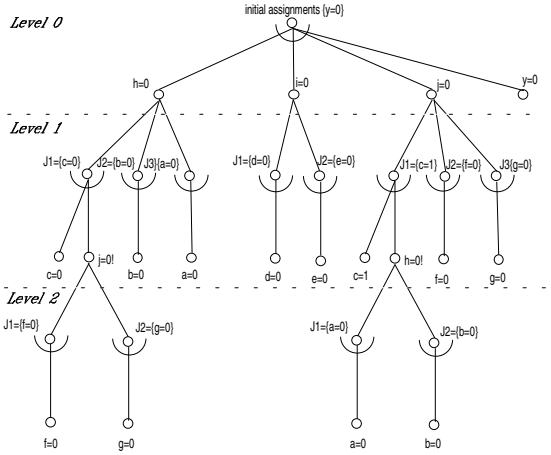

Figure 6: AND/OR tree for circuit in Figure 5

3 Implicants in Multi-Level Circuits

Section 3.1 provides the theoretical links between basic notions of two-level minimization theory, namely *implicants* and *prime implicants*, and certain subtrees of the AND/OR reasoning tree. Section 3.2 shows that AND/OR enumeration allows to naturally extend these notions to important concepts of multi-level optimization theory like *observability don't cares* and *permissible functions* [13]. Section 3.3 illustrates these concepts by means of an example.

3.1 Prime Implicants

The notions of *implicants* and *prime implicants* of Boolean functions are central elements in *two-level* minimization theory. However, when dealing with general, multi-level combinational circuits the basic concepts of implicants and prime implicants have been used only rarely. One reason why the notion of implicants has only played a limited role in multi-level circuit design, we believe, is that in the classical approach implicants are always expressed in terms of the variables of the considered Boolean function. This is sufficient for two-level circuits, however, in multi-level circuits this does permit to describe all transformations which are possible in a multi-level network. Therefore, in this section, we extend the meaning of the notion of prime implicants for broader use in multi-level circuits. A second reason why the notion of prime implicants has not become popoular for multi-level circuits is that previously no techniques have been available to actually calculate them. The classical Boolean reasoning techniques like *consensus-methods* are based on a two-level description of the circuit. Therefore we now present how prime implicants can be determined in multi-level circuits based on the concepts of Section 2.

A *literal* is a variable in the *combinational network* or its complement. A *product term* is a conjunction of literals.

Def. 3.1: A *1-implicant* (*0-implicant*) for a given function y in a combinational network C is a product term t such that y assumes the value 1 (0) for every set of value assignments at the primary inputs of C for which t assumes the value 1. An *implicant* for a node y is product term which is either a 0-implicant or a 1-implicant for y.

Note that 1-implicants correspond to the classical notion of implicants as used in the theory of two-level minimization.

Def. 3.2: An *implicant* is called *prime* if the removal of any literal makes the implicant a product term that is not an implicant.

By definition, the literals of an implicant for some node y in a *multi-level* combinational network can belong to arbitrary nodes in the network. Unlike in the case of a two-level circuit, they are not necessarily primary inputs and do not even have to be in the transitive fanin of y. Prime implicants at a node in the network may be composed out of arbitrary network variables.

Consider the node y in the circuit of Figure 7. It is immediately obvious that c and bd are prime (1-)implicants of function y. If we allow that the literals of the implicants do not have to belong exclusively to primary input signals but can belong to arbitrary nodes of the network, additional prime implicants can be determined. Note that $x = 1$ and $z = 0$ can simultaneously only occur for input assignments that produce $y = 1$. Therefore, $x\bar{z}$ is a 1-implicant for y. This implicant is prime because neither x nor $\bar{z}$ represent a 1-implicant for y.

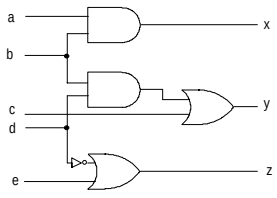

Figure 7: Circuit example for multi-level implicants

We will now examine how implicants in combinational networks can be derived by AND/OR reasoning trees. It has been shown in [11] how all *single-literal* implicants can be extracted from the AND/OR enumeration process (= recursive learning). This is now extended to extract arbitrary, *multi-literal* implicants.

The following definition is useful to relate implications and implicants in a combinational network to certain subtrees of the AND/OR reasoning tree defined in Section 2.

Def. 3.3: An *implication subtree* (*IST*) is an AND/OR tree with the following properties:

i) it is a subtree of an AND/OR reasoning tree according to Def. 2.6,
ii) the AND/OR reasoning tree and its subtree have the same root node,
iii) for each AND node included in the subtree, all its siblings in the AND/OR reasoning tree are also included in the subtree.

Theorem 3.1: Let y be an arbitrary node in a combinational network and T be the AND/OR reasoning tree for an initial set of value assignments $S = \{y=0\}$. Consider a product term $t = x_1 \cdot x_2 \cdot \ldots x_m$ where x_i is a literal corresponding to a variable f_i or its complement in the combinational network. Further, consider an IST of T with a set of leaves L.

If there is a one-to-one mapping between the literals x_i of t and the elements ($f_i = V_i$) of L such that $V_i = 0$ if x_i represents the uncomplemented variable, and $V_i = 1$ otherwise, then t is a 1-implicant for y. Analogously, t is a 0-implicant for y if the IST is a subtree of the AND/OR reasoning tree with the initial assignment $S = \{y=1\}$.

Theorem 3.1 states the rule for deriving implicants from an AND/OR tree. An implicant is formed by the conjunction of variables belonging to the leaves of an IST. If a variable at a leaf of the IST is assigned to 0 then we have to take the uncomplemented variable, if it is assigned to 1 we have to take the complemented variable as a literal in the implicant.

As mentioned above, implications as performed in test generation can be related to single-literal implicants. Viewed in AND/OR trees, *direct* implications correspond to an IST completely contained in level 0, i.e. it only consists of the root node with its immediate OR successors. For an *indirect* implication, say $y = 1 \Rightarrow f = 1$, there must exist an IST with root node y and an initial set of value assignments $\{y = 1\}$ which also includes OR nodes further down in the AND/OR tree, and which fulfills the condition that *all* leaves belong to the same value assignment $f = 1$. As an example, the bold lines in Figure 2 indicate an IST which corresponds to an indirect implication.

How are *prime* implicants represented in the AND/OR tree? Note in Def. 3.3 there is no requirement to include in the IST more than *one* child of each AND node of the original tree. In fact, including more than one OR child of any AND node makes the IST non-minimal. Since this non-minimal IST can contain leaves with new variable assignments which are not needed to make the product term an implicant, the corresponding implicant is non-prime.

Def. 3.4: An IST is called *minimal implication subtree* (*MIST*) if each AND node has exactly one OR child.

Theorem 3.2: Let y be an arbitrary node in a combinational network and T be the AND/OR reasoning tree for an initial set of value assignments $S = \{y = V\}$, $V \in \{0, 1\}$. For every *prime* implicant of y there exists a minimal implication subtree (MIST) of T such that the leaves of the MIST correspond to the literals of the prime implicant as given in Theorem 3.1.

Note that not every MIST corresponds to a prime implicant. For a given MIST with a set of leaves L there may be some other MIST with a set of leaves L' such that $L' \subset L$. Obviously, then, the implicant belonging to the first MIST cannot be prime. Fortunately, by tracing from the leaves towards the root of the AND/OR tree, it is very easy to check for a given MIST with its set of leaves, whether a subset of these leaves can belong to another MIST.

3.2 Observability Don't Cares

Often, the value of an internal logic function cannot be *observed* at any of the circuit outputs and therefore, in such situations the value of the function need not be specified. This leads to so called *observability don't cares*. Any function that covers such an incompletely specified function is called a *permissible function* according to Muroga [13].

Since we extend the notion of implicants from two-level circuits to multi-level circuits we do not only calculate prime implicants for functions at the outputs of the circuit but also for functions that belong to arbitrary internal nodes in the network. Therefore, we also want to take into account the concepts of observability don't cares and permissible functions. This leads to the definition of *permissible prime implicants*:

Def. 3.5: For some node y in the combinational network C, a product term t of some node variables of C is called a *permissible 1-implicant* for y, if and only if the following condition holds: If t is 1 then y is 1 or *not observable* at any primary output of C. Equally, t is called a *permissible 0-implicant* for y, if and only if the following condition holds: If t is 1 then y is 0 or *not observable* at any primary output of C. A permissible implicant is called *prime* if the removal of any literal makes the permissible implicant a product term that is not a permissible implicant.

AND/OR enumeration for a given node in the combinational network can easily take care of observability if the enumeration is based on Roth's D-alphabet. In [11] a method has been presented to calculate all value assignments *necessary* to observe a stuck-at fault at a given fault line in a combinational network. This routine is called *fault_propagation_learning()* in [11]. Analogously, like the AND/OR enumeration of Table 1 results from *make_all_implications()* [11], *D-AND/OR enumeration* results from *fault_propagation_learning()* by removing the statements to extract necessary assignments. The routine will be illustrated by an example in Section 3.3.

The AND/OR tree associated with these extensions starts with a single stuck-at fault assumption and, in the sequel, is referred to as *D-AND/OR reasoning tree*. The results of Section 3.1 can now be generalized for permissible implicants.

Theorem 3.3: Let y be an arbitrary node in a combinational network and T be the D-AND/OR reasoning tree for a fault assumption, y stuck-at-1. Consider a product term $t = x_1 \cdot x_2 \cdot \ldots x_m$ where x_i is a literal corresponding to a variable f_i or its complement in the combinational network. Further, consider an IST of T with a set of leaves L such that in the combinational network the nodes f_i *cannot be reached by the fault effect*.

If there is a one-to-one mapping between the literals x_i of t and the elements $(f_i = V_i)$ of L such that $V_i = 0$ if x_i represents the uncomplemented variable, and $V_i = 1$ otherwise, then t is a permissible 1-implicant for y. Analogously, t is a permissible 0-implicant for y if the IST is a subtree of the enumeration tree with y stuck-at-0.

Theorem 3.4 states an important property of AND/OR trees which makes them very attractive in multi-level logic synthesis:

Theorem 3.4: Let y be an arbitrary node in a combinational network and T be the D-AND/OR reasoning tree for a fault assumption, y stuck-at-V, $V \in \{0, 1\}$. For *every permissible prime implicant* at node y there exists a minimal implication subtree (MIST) of T such that the leaves of the MIST correspond to the literals of the permissible prime implicant as given in Theorem 3.1.

3.3 Example

For illustration of D-AND/OR enumeration consider the following example. Figure 9 shows a circuit for which the AND/OR tree is built in Figure 11. Consider the fault a, stuck-at 1.

We proceed as given by *fault_propagation_learning()* in [11]. There are two paths along which this fault can propagate to a primary output. At least one of them has to be *sensitized* for fault detection. One path traverses gates k and l, sensitization yields the value assignments $d = 1$ and $j = 0$. For the AND/OR tree in Figure 11, this produces the left AND-node in level 1 with its children. The second possibility is to sensitize the path through m and q resulting in the right portion of the AND/OR tree. The sensitizations yield value assignments and unjustified lines. These value as-

signments are enumerated in the usual way as given by Table 1, so that the AND/OR tree for the stuck-at-1 fault assumption at signal *a* results as shown in Figure 10.

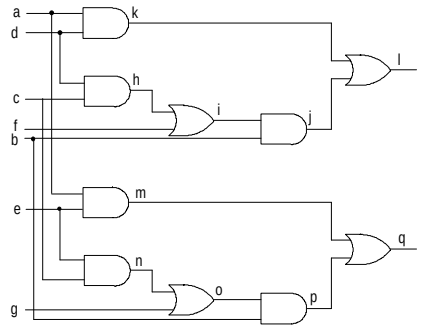

Figure 9: Example circuit for D-AND/OR enumeration

Note that for reasons of simplicity, in Figure 10, we only consider for inclusion in the AND/OR tree those unjustified gates that have a specified output signal, i.e. they represent what is referred to as unjustified lines in test generation literature. Although unjustified gates with unspecified outputs like in the AND/OR tree of Figure 2 are necessary for the theoretical completeness of the enumeration, it is possible to neglect them for most practical purposes [11].

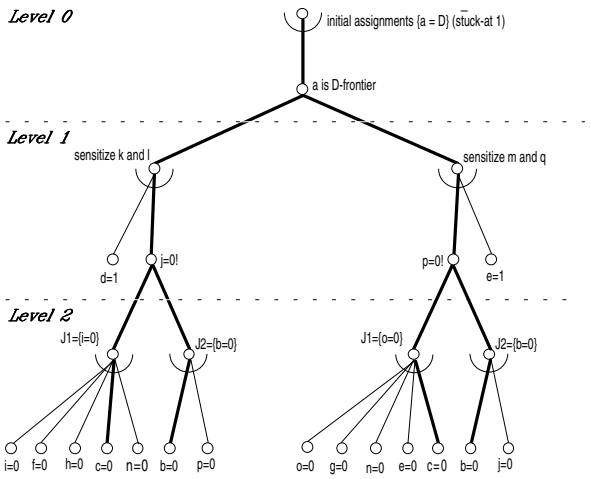

Figure 10: AND/OR tree for circuit in Figure 9, bold lines indicate the MIST for implicant *b·c*

The bold lines in Figure 10 indicate a MIST that represents a permissible implicant *b·c* for node *a* in the circuit.

Note the special characteristics of this prime implicant: the network function *a* does not depend on the variables *b* and *c*. Still, according to our definitions, it is possible to determine a prime implicant for *a* using these variables. This is not possible with the conventional two-level notion of prime implicants. Further, this example illustrates another important property of our approach. The implicant actually exploits observability don't cares at node *a*. The fact that *b·c* is a permissible prime implicant means that we can modify the combinational network of Figure 9 as shown in Figure 11. The node *a'* assumes a different function than *a* but the full circuits in both figures are still functionally equivalent.

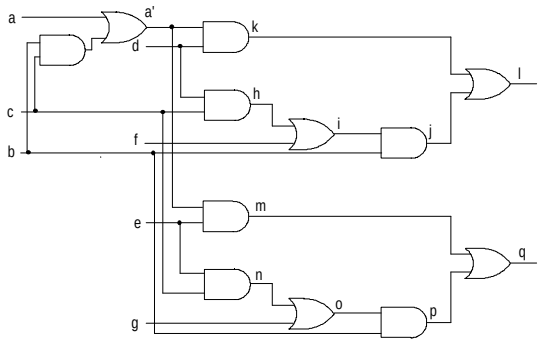

Figure 11: Adding permissible implicant *b·c* at signal *a*

3.4 Heuristics

A) Heuristics to select an implicant

AND/OR reasoning trees, in principle, can be used to generate all prime implicants for nodes in a *multi-level* combinational network, however there may be a huge number of prime implicants for a given node in the network, especially if we take into account that the implicants can be expressed in terms of arbitrary (internal) nodes of the network. Therefore, this section is dedicated to demonstrate how the *topology* of the AND/OR trees can be used to generate certain implicants being particularly promising for the given application. Here we consider circuit minimization. In [12] it has been observed and experimentally confirmed that, for a given node *y*, certain single-literal implicants, being obtained by *indirect* implications, represent *good* divisors for *y*. As explained above, an indirect implication corresponds to a MIST with *several* leaves all belonging to the *same* value assignment. Intuitively, a subtree of the AND/OR tree which has *several* leaves that all correspond to the *same* value assignment indicates suboptimal circuitry. If such a subtree is a MIST, then we have an implication and transformations like in [12] can be performed. Hence, we intend to identify MISTs with many identical leaves. In Figure 10 of Section 3.1, the bold lines indicate a promising MIST with four leaves that only belong to two different value assignments. Therefore, the permissible implicant *b·c* is promising for inclusion in a permissible function at node *a*. In Section 4 it will be shown how this optimizes the circuit.

Importantly, in this work, we completely avoid building the trees to save memory. In [17], a method has been developed to extract MISTs with a maximum number of identical leaves from the D-AND/OR reasoning tree, solely by "monitoring" the enumeration process. This method is based on repeated enumeration, thus we are maintaining linear memory requirements at the cost of CPU-time. This technique must be omitted here for reasons of space.

B) Heuristics to select sets of implicants

Often it is not sufficient to add a single implicant to a network in order to obtain reductions in other parts of the network. A different cover of the node function can often only be obtained if several new implicants are added to the network. If there are several implicants that are all particularly "good" according to our heuristics then they all are added to the network. This is a pretty rough heuristic and current research examines whether topological properties of the AND/OR tree can give insight what implicants should be grouped together in the attempt to find a better cover.

4 Optimization Procedure

The optimization procedure is along the lines of [12] and, for reasons of space, is only illustrated by an example. Reconsider the circuit of Figure 9. The function of the circuit is given by the following Boolean expressions:

$$l = ad + b(cd + f) = (a + bc)d + bf$$
$$q = ae + b(ce + g) = (a + bc)e + bg$$

By manipulating the equations, it can be noted that there exists a common kernel, $a + bc$. Minimization can be achieved by sharing this kernel. It was demonstrated for this example in Section 3.3 how to generate the permissible implicant bc for signal a. Note that the suboptimality of the original circuit is reflected by the existence of a MIST with several leaves belonging to the same value assignments, $b = 0$ and $c = 0$. In this case it points out a common kernel, $a + bc$, that can be shared to save area.

If bc is a permissible implicant for a then the circuit is modified as shown in Figure 11. After adding the implicant to the circuit, redundancy elimination is used to reduce the circuit. This results in the circuit of Figure 12.

The optimization in this example can also be obtained by an algebraic kernel extraction technique [4]. Note however, that the procedures based on AND/OR trees and redundancy elimination are capable of performing *general Boolean manipulations* and are not restricted to algebraic transformations. Also note that this kind of transformation cannot be obtained by the netlist optimization methods of [8], [9], [12].

In [18] it has been proved that manipulating circuits based on prime implicants can perform arbitrary transformations in a combinational network. This means that the classical notion of prime implicants is not only sufficient to fully describe circuit transformations for two-level circuits but also for multi-level circuits. Indeed, all transformations in combinational networks commonly referred to as "division", "decomposition", "kernel extraction", "transduction" etc. can also be described by operations based on our notion of prime implicants in networks, thus extending the classical notions of two-level theory to multi-level circuits.

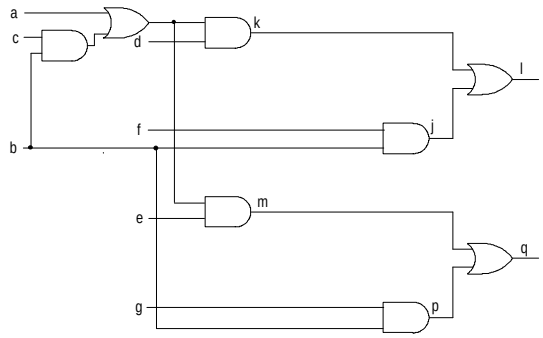

Figure 12: Circuit after redundancy removal

5 Experimental Results

The described methods have been implemented in the program system HANNIBAL and are applied to the problem of PLA factorization. This application has been chosen because this task cannot be accomplished in a satisfactory way by previous implication based minimization techniques [12] or ATPG-based methods [8], [9], [16]. Table 2 shows the results for some two-level MCNC benchmark circuits which are factorized into a multi-level description. Since our implementation does not accept external don't cares, at this point we selected only such examples which are completely specified. The results of HANNIBAL are compared with SIS1.2 (using *resub -a*, *simplify -d* followed by *script.rugged*). The area is measured in terms of number of connections based on a generic library of the basic gate types. For both tools we show results for fixed settings (single run of *script.rugged* for SIS) and interactive use (column "multi run" for SIS and column "best" for HANNIBAL). Column "RL" shows the results if only single literal implicants (recursive learning) are used.

The optimization results of Table 2 show the great promise of this approach although CPU-times are not yet satisfactory. As mentioned in Section 3, implicants are determined by repeated enumeration [17] without building the graphs. If the graphs are actually constructed (at the cost of memory) implicants can be determined much faster by simple operati-

ons on the graph. Current research investigates appropriate trade-offs between memory and time and how to include appropriate hashing and caching techniques similarly like for BDDs. This will be needed to reduce CPU-times of our approach. An important attribute of the presented AND/OR trees is that they need not be constructed to their full size in order to be useful. In these experiments, AND/OR trees have been examined only up to a recursion depth of '3'. This however proved sufficient to obtain the shown optimization results.

The experimental results clearly confirm our conjecture that topological properties of AND/OR reasoning graphs can be used to guide an optimization process.

PLA factorization results		SIS1.2 (script rugged)		HANNIBAL			
		single run	multi run	fixed settings		best	RL
name	# c.	# c.	# c.	# c.	CPU-time	# c.	# c.
5xp1	369	164	159	79	0:01:58	78	237
9sym	609	320	206	152	0:17:00	83	609
clip	1055	195	187	110	0:10:24	90	520
con1	32	30	30	27	0:00:01	27	30
duke2	995	540	510	416	1:12:33	355	612
e64	2144	253	253	253	0:15:19	253	253
misex1	154	77	77	59	0:00:51	55	81
misex2	206	121	121	121	0:02:47	111	134
o64	195	-	-	195	0:00:08	195	195
rd53	176	52	52	36	0:00:42	34	99
sao2	501	192	190	116	0:05:38	108	195
vg2	914	124	124	115	0:03:22	112	141

Table 2: Experimental results for MCNC PLAs (without external don't care conditions), Sparc 5

Our experiments also demonstrate the practical relevance of the theoretical result given in Theorem 3.1. This can be nicely observed at the example of the MCNC benchmark circuit *o64*. This circuit is small, nevertheless it is impossible to build an OBDD for this circuit. No literal in the circuit description appears in more than one product term (hence the SOP is unate) and all prime implicants are essential. Therefore no optimization is possible, neither with two-level nor with multi-level minimization techniques. SIS1.2 runs out of memory in both *script.rugged* and *script.algebraic* after about ten minutes of CPU-time in each script. However, since the circuit is unate (Theorem 3.1 applies) HANNIBAL has no problem with this example. The fact that this circuit cannot be further optimized is detected very fast and only little CPU-time is wasted. This benchmark circuit illustrates that the basic theoretical differences between variable enumeration and AND/OR enumeration that have been elaborated in this paper have important consequences in practical applications.

Conclusion

This paper has introduced specific AND/OR reasoning graphs as a new basis for solving design automation problems by implicant-based techniques. Conventional variable enumeration and its derivates are not the only possibility to fully explore the functionality of a given circuit. We have proved basic theoretical properties of the proposed AND/OR graphs permitting to extend basic concepts of two-level circuit theory to multi-level circuits.

References

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

[2] Akers S.: "Binary Decision Diagrams", IEEE Transactions on Computers, vol. 27, pp. 509-516, June 1978.

[3] Brayton R. K., Hachtel G. D., McMullen C. T., Sangiovanni-Vincentelli A. L.: "Logic Minimization Algorithms for VLSISynthesis", *Kluwer Academic Publishers*, Boston, MA, 1984.

[4] 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.

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

[6] Bryant R.: "Graph-based algorithms for Boolean function manipulation", IEEE Trans. on Computers, vol. 35, pp. 677-691, August 1986.

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

[8] Chang S.C., Marek-Sadowska M.: "Perturb and Simplify: Multi-Level Boolean Network Optimizer", Proc. International Conf. on Computer-Aided Design, San Jose, Nov. 1994.

[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] Jain J., Mukherjee R., Fujita M.: "Advanced Verification Techniques Based on Learning", Design Automation Conference (DAC), pp. 420 - 426, June 1995.

[11] Kunz W., Pradhan D.K.: "Recursive Learning: A New Implication Technique for Efficient Solutions to CAD Problems: Test, Verification and Optimization", IEEE Trans. on CAD, vol. 13, pp. 1143-1158, Sept. 1994.

[12] Kunz W., Menon P.: "Multi-Level Logic Optimization by Implication Analysis" Proc. Intl. Conference on Computer-Aided Design, San Jose, pp. 6-13, Nov. 1994.

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

[14] Rich E.: "Artificial Intelligence", McGraw-Hill, 1983.

[15] Roth J. P.: "Diagnosis of automata failures: A calculus & a method", IBM J. Res. Develop., vol. 10, July 1966, pp. 278-291.

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

[17] Stoffel D., Kunz W., Gerber S.: "AND/OR Graphs", *Technical Report*, Max-Planck-Society, MPI-I-95-602, 1995.

[18] Kunz W.: "Testing Techniques in Logic Synthesis", *Habilitation Thesis*, Department of Computer Science, University of Potsdam, 1996.