Automatic test pattern generation for logic synthesis systems

Michiel M. Ligthart *
Philips Research Labs Palo Alto
Giovanni De Micheli
Center for Integrated Systems
Stanford University
Stanford, CA. 94305

Abstract

This paper describes a novel automatic test pattern generation algorithm for combinational circuits. The algorithm, based on the propagation of care sets in multi level combinational circuits, generates an expression for the entire set of input vectors that tests a particular node of a circuit stuck-at one and stuck-at zero. The salient feature of this approach is that it generates all test patterns for all nodes in the circuit or, if no test exists for a node, positively identifies redundancies. A covering algorithm is described that constructs a near-minimal set of test patterns from the original test set, detecting all single stuck-at faults in the circuit.

*Now with Exemplar Logic, 2550 Ninth street suite 102, Berkeley, Ca. 94710

1 Introduction

Combinational circuits can be tested for the presence of single stuck-at faults by applying test patterns at the primary inputs and measuring circuit responses at the primary outputs. These test patterns are constructed in such a way that for each stuck-at fault a circuit response is provoked different from the 'correct circuit' response if that fault is present. The process of finding test patterns is a difficult one and is hampered by the presence of redundancies, or untestable faults, in the circuit. A redundancy is a node in the circuit that can be removed without changing the functional behavior of the circuit. One of the objectives of Automatic Test Pattern Generation (ATPG) programs is to identify these redundancies, which is also beneficial for logic synthesis programs. Because combinational test algorithms are often used to test sequential circuits through incorporation of scan-design techniques [Fujiwara85] the number of test patterns required to fully test a circuit should be kept to a minimum. The goal of test pattern generation programs is therefore not only to generate test patterns for all (stuck-at) faults in the circuit but also to identify redundancies and to minimize the number of test patterns.

Many algorithms for test pattern generation have been proposed over the years. A multiple path sensitization approach, called the D-algorithm, was first formulated by Roth [Roth66]. The main disadvantage of the D-algorithm is the possibility of abundant backtracking, especially for circuits employing ex-or gates or circuits with high reconvergent fanout. The PODEM algorithm [Goel81] was shown to be more efficient, reducing the number of backtracks considerably. The FAN algorithm [Fujiwara83] introduced several strategies (implication, unique sensitization, and multiple backtracking) to improve the performance of PODEM. Recently, acceleration of test pattern generation and improvement of redundancy identification has been reported for the test pattern generator Socrates [Schulz88]. This test pattern generator utilizes an improved implication and sensitization procedure that yields a considerable speedup and is able of discovering all redundancies in benchmark circuits. with only a small number of backtracks. A conceptually different approach is taken in [Larabee89]. Here a test pattern generator is presented that constructs the Boolean difference formula for a certain fault in the network and then applies a Boolean satisfiability algorithm to solve this formula. As the Boolean difference method defines the complete set of test patterns for a given fault, satisfying this formula will yield a test pattern for that fault if one exists.

Although powerful, the algorithms described above have certain limitations.

Most test pattern generators operate only on primitive gates (and, or, inv, nand, nor) and replace all high level primitives like and-or-inverts, adders, muxes, etc., by their internal gate representation. The noticeable exception here is Socrates [Schulz88], which is capable of handling xors, xnors, and some high-level primitives. However, even Socrates has only a limited number of recognizable primitives. As a result, these programs only approximate the 'real' stuck-at faults, and do not allow for hierarchical testing. Another limitation is that nothing can be said about the size of the test set. Fault simulation techniques are used extensively to keep the number of test patterns low, but it is not known whether the test set is even close to a minimal size.

In this paper we present Cerberus, the test pattern generator of the Olympus Synthesis System [DeMicheli90] developed at Stanford University. This novel approach to automatic test pattern generation is based on the notion of care and don't care sets as developed in logic synthesis research. The method used in Cerberus resembles the Boolean difference method [Sellers68] in the way that it manipulates circuit equations in order to compute an expression for fault propagation. However, while the Boolean difference method computes these expressions from primary inputs towards primary outputs, requiring internal nodes in the circuit to be considered as additional inputs, Cerberus computes test sets incrementally from primary outputs towards primary inputs.

The intimate relationship between logic synthesis and test pattern generation was first pointed out by Bartlett et.al. [Bartlett86], [Bartlett88] who showed how test patterns for stuck-at faults could in principle be supplied as a byproduct of 2-level logic minimization. Utilizing new developments and heuristics in the computation of fanout don't care sets, Cerberus generates an expression for the complete set of test patterns for each node in a combinational network, while simultaneously identifying all redundancies. Having the complete set of all test patterns, finding a minimal set of test patterns testing all stuck-at faults is then reduced to a covering problem. Where other test pattern generation programs operate solely on primitive gates Cerberus manipulates Boolean functions and is therefore capable of handling any complex gate. This allows for a precise enumeration of all stuck-at faults and their test patterns. Finally, Cerberus is capable of exploiting hierarchy, which makes it extremely useful for sophisticated techniques like Macro testing [Beenker86].

The remainder of this paper is organized as follows. Section 2 gives a brief explanation of care sets and presents new methods and heuristics to compute these sets. Section 3 describes the computation of local test sets from the care sets. Subsequently, section 4 describes how Cerberus constructs a near-minimal

test set for a circuit from the local test sets. As pointed out above, Cerberus handles hierarchy to a certain extent and can generate test patterns for use with fault models other than the single stuck-at model. This is discussed in section 5. Finally, in section 6 we conclude by summarizing the approach and showing some preliminary results.

2 Cares and don't cares

A multi-level combinational circuit can be represented as a directed acyclic graph (DAG) where the vertices represent the nodes of the circuit and the edges represent their interconnects. With each edge a network variable y_i is associated. A Boolean expression $f(y_1, \ldots, y_n)$ of its fanin edge variables is associated with each vertex. This representation enables us to distinguish between different stuck faults on multiple fanouts. In the DAG only Boolean functions are represented, i.e. no distinction is made between an actually mapped circuit or a Boolean description. Figure 1 shows an example of a combinational circuit and its representation in a DAG.

The cofactor of a sum of products representation f with respect to a variable i, notation $f|_i$, is the function obtained by assuming $i = 1$. Similarly, $f|_{\overline{i}}$, is obtained by assuming $i = 0$.

For each node i a $CARE_j(i)$ set can be defined as the condition, or set of conditions, for which the output of i is observable at node j. The CARE set [1] for node i with respect to a primary output z is given by

$$CARE_z(i) = (z|_i \neq z|_{\overline{i}}). \tag{1}$$

The full CARE set for node i is the summation over all primary outputs

$$CARE(i) = \sum_{j=1}^{n} CARE_{z_j}(i). \tag{2}$$

As a simple illustration of the CARE set consider the circuit in figure 2. The value of node a is only propagated through the 'and' gate g_2 if both b and c are '1', i.e.

$$CARE_i(a) = bc.$$

[1] The CARE set is the complement of the transitive fan-out don't care set as defined in [2].

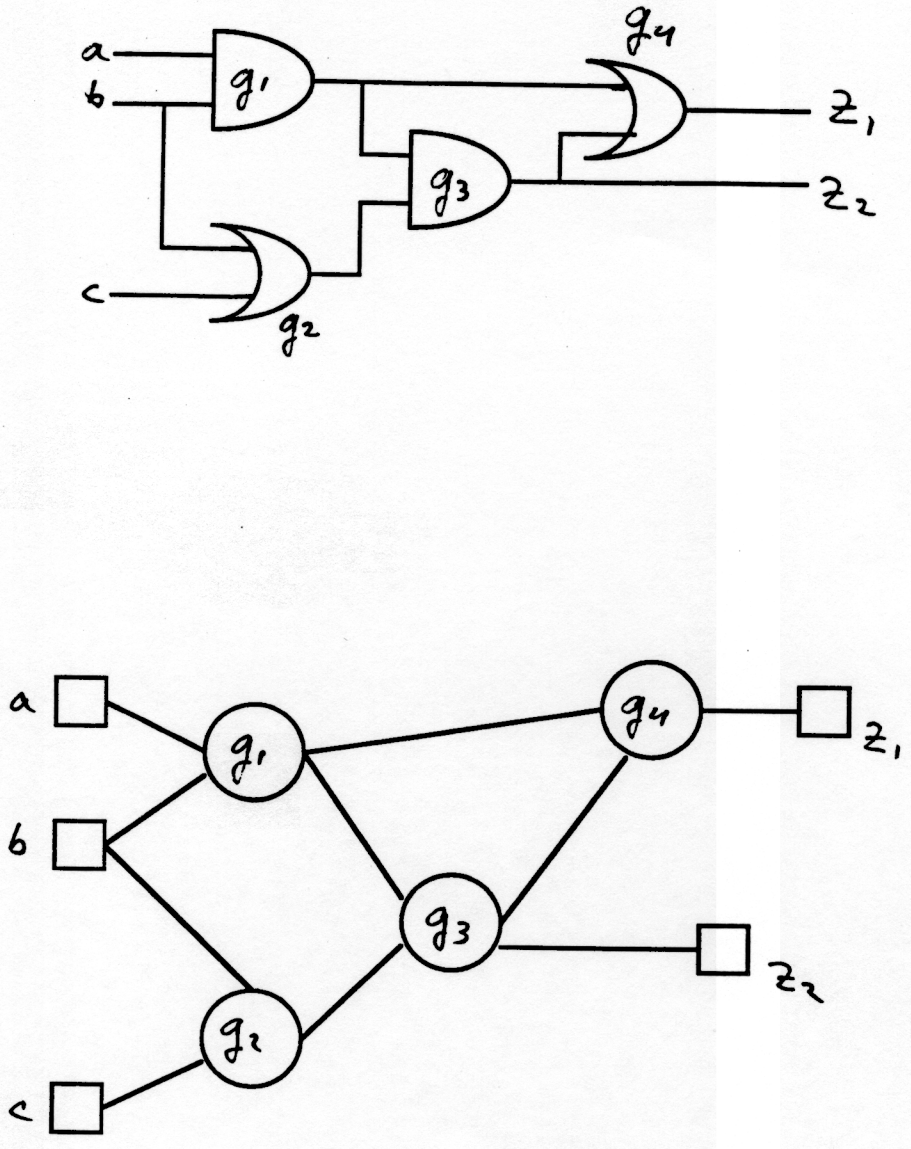

Figure 1: Directed Acyclic Graph representation of a combinational circuit.

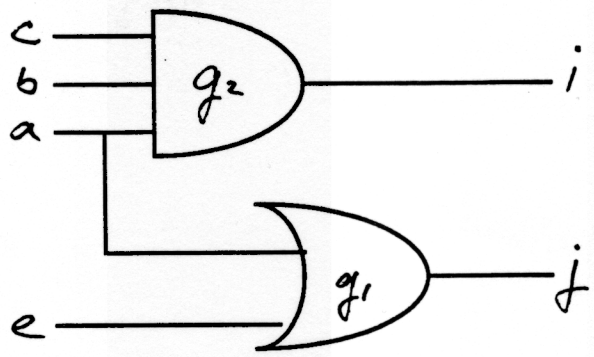

Figure 2: Example of the CARE set.

As a also fans out to node j

$$CARE_j(a) = \overline{e},$$

yielding

$$CARE(a) = CARE_i(a) + CARE_j(a) = bc + \overline{e}.$$

Computation of the CARE set is a cumbersome and CPU intensive process as each internal node has to be propagated to the primary outputs before its CARE set can be derived. This has always hampered proper use of care and don't care sets and made, for instance, the Boolean difference method impractical.

Recent advances in logic synthesis research, however, have yielded ways to compute the CARE set incrementally. These advances are best explained first with an illustrative example. Assume a single-output network that contains a node x with two fanouts y_1 and y_2 as in fig. 3. According to eqs. 1 and 2,

$$CARE(x) = (z|_x \neq z|_{\overline{x}}) = (z|_x \oplus z|_{\overline{x}}).$$

Because $x = y_1 y_2$ and $\overline{x} = \overline{y_1 y_2}$,

$$CARE(x) = (z|_{y_1 y_2} \oplus z|_{\overline{y_1 y_2}}).$$

Using $a \oplus a = 0$ and $a + 0 = a$

$$CARE(x) = (z|_{y_1 y_2} \oplus z|_{\overline{y_1 y_2}}) \oplus (z|_{\overline{y_1} y_2} \oplus z|_{\overline{y_1} y_2}).$$

Figure 3: Example of a node x with multiple fanouts in a single output network.

Because the ex-or operation is commutative

$$\begin{aligned} CARE(x) &= (z|_{y_1 y_2} \oplus z|_{\overline{y_1} y_2}) \oplus (z|_{\overline{y_1} y_2} \oplus z|_{\overline{y_1}\,\overline{y_2}}) \\ &= (z|_{y_1} \oplus z|_{\overline{y_1}})|_{y_2} \oplus (z|_{y_2} \oplus z|_{\overline{y_2}})|_{\overline{y_1}}, \end{aligned}$$

which, together with eq. 1, gives the surprising result

$$CARE(x) = CARE(y_1)|_{y_2} \oplus CARE(y_2)|_{\overline{y_1}}.$$

Generalized, the CARE set of a node i in a single output network can be computed as a function of its immediate fanouts using the following equation [Damiani90]:

$$CARE(i) = \bigoplus_{j=1}^{n} (F_j|_{y_j} \neq F_j|_{\overline{y_j}}) CARE(F_j)|_{y_1,\ldots,y_{j-1},\overline{y_{j+1}},\ldots,\overline{y_n}}. \qquad (3)$$

where $y_1, \ldots, y_n$ are the immediate fanouts of node i and F_j is the function associated with the vertex that y_j fans out to. Equation 3 computes the contribution to the CARE set of one fanout at the time, while simultaneously asserting a value on the other fanouts. The full CARE set is obtained by taking the exclusive-or over the contributions of all fanouts. The CARE set for a multi-output network is simply the summation over all outputs as in eq. 2.

If a node in the network does not reconverge in any of its path to the primary outputs, eq. 3 can be drastically simplified. For those nodes, the complete CARE set is a function of its immediate fanouts:

$$CARE(i) = \sum_{j=1}^{n} [(F_j|_{y_j} \neq F_j|_{\overline{y_j}}) CARE(F_j)] \qquad (4)$$

where $y_1, ..., y_n$ are again the immediate fanouts of node i.

Equations 3 and 4 provide a means of computing CARE sets as a function of a node's immediate fanouts. Hence, if a network is sorted in topological order, i.e. such that each node in the network is preceded by all its fanins, it is possible to compute CARE sets incrementally from the primary outputs towards the primary inputs.

3 Test pattern generation

The CARE set for a node as discussed in the previous section gives all the conditions under which the value of that node propagates to at least one of the primary outputs. In other words, the CARE set gives the complete set of necessary and sufficient conditions under which that node is ***observable*** at the primary outputs of the network. The ON set of node i, notation $ON(i)$, is defined as the set of primary input vectors that generate a logical '1' on that node. Similarly, the OFF set of node i, notation $OFF(i)$, is defined as the set of primary input vectors that generate a logical '0' on that node. The set of all vectors testing node i stuck-at zero, denoted $sa0(i)$, is then equal to

$$sa0(i) = CARE(i) \cap ON(i). \tag{5}$$

Similarly, the set of all vectors testing node i stuck-at one, denoted $sa1(i)$, is equal to

$$sa1(i) = CARE(i) \cap OFF(i). \tag{6}$$

The salient feature of expressions (5) and (6) is that they provide us with the ***complete*** set of all stuck-at patterns per node in the network. Rather then computing one pattern per fault, as the path sensitization algorithms do, this algorithm generates an expression for all patterns that will detect node i stuck-at 0 or stuck-at 1. If either $sa1(i)$ or $sa0(i)$ is empty this indicates that the stuck fault associated with node i is not testable. In logic synthesis terminology this means that the node is redundant and a constant value can be propagated. Figures 4 and 5 summarize the algorithms for computation of the test patterns and the CARE set respectively.

```
procedure test_pattern_generation()
{
    topo_list = order_nodes_topologically ;
    tag_all_reconvergent_nodes;
    for i = 1 to num_of_nodes do
    {
        node = topo_list[i];
        node→ON = compute_ON_set(node);
        node→OFF = compute_OFF_set(node);
    }
    for i = num_of_nodes downto 1 do
    {
        node = topo_list[i];
        node→CARE = comp_CARE_set(node);
        node→sa0 = node→CARE ∩ node→ON;
        node→sa1 = node→CARE ∩ node→OFF;
    }
}
```

Figure 4: Pseudo code for the computation of test patterns.

```
procedure compute_CARE_set(node)
{
    CARE = NULL;
    if (node_does_not_reconverge(node) == 1)
    {
        for i = 1 to num_of_fanouts do
        {
            fanout = node→fanout[i];
            F = cofactor(node, fanout);
            G = cofactor(node, fanout');
            CARE = CARE + [(F ⊕ G)fanout→CARE];
        }
    }
    else
    {
        for i = 1 to num_of_fanouts do
        {
            fanout = node→fanout[i];
            F = cofactor(node, fanout);
            G = cofactor(node, fanout');
            for j = 1 to num_of_outputs do
            {
                CARE_j = CARE_j ⊕ [(F ⊕ G)fanout→CARE_j];
                CARE = CARE + CARE_j
            }
        }
    }
    return CARE;
}
```

Figure 5: Pseudo code for the computation of the CARE set.

4 Finding a minimal set of test patterns

Minimization of the length of test sets is important in those situations where combinatorial test pattern generation is used in conjunction with scan path testability. Here one tries to keep the number of clock cycles required to shift in the patterns to a minimum in order to save time at the tester. Besides that, many testers have a limited memory size, and can only store a certain amount of data. Once the complete set of all stuck-at patterns has been computed the problem of finding a minimal test set is reduced to a covering problem. The set of test patterns per fault (node stuck-at 0 or node stuck-at 1) are called local test sets. The global test set is the set of patterns testing all single faults in the network. Obviously, the global test set can be obtained by choosing one pattern from each local test set, but normally a much smaller set will do. This section addresses the problem of constructing a near-minimal global test set from the abundance of available test patterns.

In order to state the problem clearly we first introduce some terminology from the theory of two-level minimization [Brayton84]. A variable is a symbol representing a single coordinate of the Boolean space e.g. a. The dimension of a Boolean space is its number of variables, e.g. the variables a, b, c, d span a 4-dimensional Boolean space. A literal is a variable or its negation, e.g. a or $\overline{a}$. A cube is a set of literals such that a variable cannot occur together with its negation, e.g. $a\overline{b}c$ is a cube but $a\overline{a}c$ is not. A minterm is a cube that does not contain any other cube than itself, and can therefore be considered as the atomic constituent of a cube. For instance, in the 4-dimensional Boolean space the cube $a\overline{b}c$ can be decomposed into the minterms $a\overline{b}cd$ and $a\overline{b}c\overline{d}$. A cube c is said to contain another cube d, $c \supseteq d$, if each of the minterms of d is also a minterm of c, e.g. if $c = pq$ and $d = pq\overline{r}$ then $c \supseteq d$. An expression is a set of cubes, e.g. $f = ab + \overline{b}c$. An expression f covers a cube c if each of the minterms of c is contained by at least one of the cubes of f.

Each local test set is a Boolean expression and comprises 0, 1, or more cubes. The global test set is a set of minterms that are offered as test patterns at the primary inputs of the circuit. Hence, the problem is to construct a global test set such that for each local test set there exists at least one minterm in the global test set that is covered by that local test set. Moreover, the cardinality of the global test set should be minimal.

A heuristic has been developed to solve this covering problem. All stuck-at faults with their associated local test sets are first placed in a fault list. If a local test

set is empty, the fault is untestable and the associated node is redundant. Subsequently, the fault is removed from the fault list and reported untestable. The remaining fault list is passed to a procedure find_global_test_set(). (See figures 6, 7 and 8 for an outline of the algorithms in pseudo code.) Find_global_test_set() first searches the fault list for local test sets that contain only one cube. These are called essential cubes, because they contain the only minterms that can test the fault associated with this local test set. As soon as an essential cube is found, procedure reduce() is used to recursively extract a smaller cube that is covered by as many other local test sets as possible. Reduction is the operation where a cube is split into two smaller cubes contained in it. E.g. in a 4-dimensional space over $\{p, q, r, s\}$ cube $c = pq$ can be split into cubes $c_1 = pqr$ and $c_2 = pq\overline{r}$. After a cube is split into 2 smaller cubes the number of local tests sets covering each cube is determined in the procedure cover(). Cover() returns a list of faults that are covering the current cube. The cube with the lowest count is discarded and the reduction is continued recursively with the other cube. This is repeated until a cube is found that cannot be further reduced, i.e. a minterm. The minterm is added to the global test set, and all faults with local test sets covering the minterm are removed from the fault list. Once all essential cubes are processed this way, the algorithm continues with those local test sets that contain two cubes. For both cubes the potential coverage is computed and the one with the lowest number is discarded. The same reduction technique is used to extract the minterm from the remaining cube. This process is repeated until the fault list is empty.

As an example, consider a circuit with primary inputs a, b, c and d that yielded the following local test sets:

```
sa0(a)  = d'ba + bac;          sa1(a)  = d'ba' + ba'c;
sa0(b)  = d'ab + da'b;         sa1(b)  = d'ab' + da'b';
sa0(c)  = a'c;                 sa1(c)  = a'c';
sa0(d)  = a'c'd + db';         sa1(d)  = a'c'd' + d'b';
sa0(g1) = d'a' + d'b' + a'bc;  sa1(g1) = ad' + abc;
sa0(g2) = a'd c' + db';        sa1(g2) = ba'd;
sa0(g3) = a'd' + a'bc + d'b';  sa1(g3) = a'dc' + db';
sa0(z)  = ab + b'd + dc';      sa1(z)  = a'd' + a'bc + d'b';
```

The first essential cube is $\overline{a}c$. Splitting in variable b yields $\overline{a}cb$ and $\overline{a}c\overline{b}$. Cube $\overline{a}cb$ is potentially covered by faults sa0(b), sa0(c), sa0(g1), sa0(g3), sa1(a), sa1(g2) and sa1(z), which is a total of 7. Cube $\overline{a}c\overline{b}$ is potentially covered by

faults sa0(c), sa0(d), sa0(g1), sa0(g2), sa0(g3), sa0(z), sa1(b), sa1(d), sa1(g3) and sa1(z), which is a total of 10..

Continuing with $\overline{a}c\overline{b}$ and splitting in d yields $\overline{a}c\overline{b}d$ detecting sa0(c), sa0(d), sa0(g2), sa0(z), sa1(b) and sa1(g3), and $\overline{a}c\overline{b}\overline{d}$ detecting sa0(c), sa0(g1), sa0(g2), sa1(d) and sa1(z). Hence, $\overline{a}\overline{b}cd$ is the best choice as test pattern. After removal of the covered faults the fault list becomes:

```
sa0(a)  = d'ba + bac;          sa1(a)  = d'ba' + ba'c;
sa0(b)  = d'ab + da'b;         sa1(c)  = a'c';
sa0(g1) = d'a' + d'b' + a'bc;  sa1(d)  = a'c'd' + d'b';
sa0(g3) = a'd' + a'bc + d'b';  sa1(g1) = ad' + abc;
                               sa1(g2) = ba'd;
                       sa1(z)  = a'd' + a'bc + d'b';
```

Repeating the same thing for essential cube $\overline{a}\overline{c}$ gives pattern $\overline{a}b\overline{c}d$ detecting sa0(g1), sa0(g3), sa1(a), sa1(c), sa1(d) and sa1(z). The fault list after removal of these faults is as follows:

```
sa0(a)  = d'ba + bac;          sa1(g1) = ad' + abc;
sa0(b)  = d'ab + da'b;         sa1(g2) = ba'd;
```

The next essential cube is $b\overline{a}d$ detecting faults sa0(b) and sa1(g2). Two faults, sa0(a) and sa1(g2), remain on the fault list which are both tested by $abc-$. So, the total number of patterns to test this circuit for single stuck-at is 4.

5 Application to hierarchical testing

Cerberus operates solely on Boolean equations and is therefore capable of generating stuck-at patterns for any type of gate, if its functional behavior is known. This also allows for the introduction of hierarchy and hierarchical tests in the system. A cell or module in a library can be equipped with a custom set of test patterns. These patterns might be handcrafted especially for that particular cell or module, based on detailed knowledge about the physical cell. Such a cell test can be regarded as a cell property equivalent to rise and fall delays, area, etc.. When generating local test sets, Cerberus can now take the predefined test patterns into account while intersecting the ON and OFF sets with the CARE set.

```
procedure find_global_test_set(fault_list)
{
    foreach fault ∈ fault_list do
    {
        if (| local_test_set | == 1)
        {
            cube = local_test_set→cube;
            test_pattern = reduce(cube);
            covered_faults = cover(test_pattern, fault_list);
            global_test_set = global_test_set + test_pattern;
            fault_list = fault_list - covered_faults;
        }
    }
    for num_of_cubes = 2 to n do
    {
        while (fault_list ≠ ∅)
        {
            if (| local_test_set | == num_of_cubes)
            {
                best_cube = local_test_set→cube[1];
                tmax = |cover(best_cube, fault_list)|;
                for i = 2 to num_of_cubes do
                {
                    cube = local_test_set→cube[i];
                    t = |cover(cube, fault_list)|;
                    if (t < tmax)
                    {
                        good_cube = cube;
                        tmax = t;
                    }
                }
                test_pattern = reduce(best_cube);
                covered_faults = cover(test_pattern, fault_list);
                global_test_set = global_test_set + test_pattern;
                fault_list = fault_list - covered_faults;
            }
        }
    }
}
```


Figure 6: Pseudo code for find_global_test_set.

```
procedure reduce(cube)
{
    if (cube != minterm)
    {
        cube_split(cube, cube1, cube2);
        t1 = |cover(cube1, fault_list)|;
        t2 = |cover(cube2, fault_list)|;
        if (t1 > t2)
            red_cube = reduce(cube1);
        else
            red_cube = reduce(cube2);
        return red_cube;
    }
    else
        return cube;
}
```

Figure 7: Pseudo code for reduce().

```
procedure cover(cube, fault_list)
{
    covered_faults_list = NULL;
    foreach fault ∈ fault_list
    {
        foreach fcube ∈ fault→local_test_set
        {
            if (fcube ⊇ cube)
            {
                covered_faults_list = covered_faults_list + fault;
                break;
            }
        }
    }
    return covered_faults_list;
}
```

Figure 8: Pseudo code for cover().

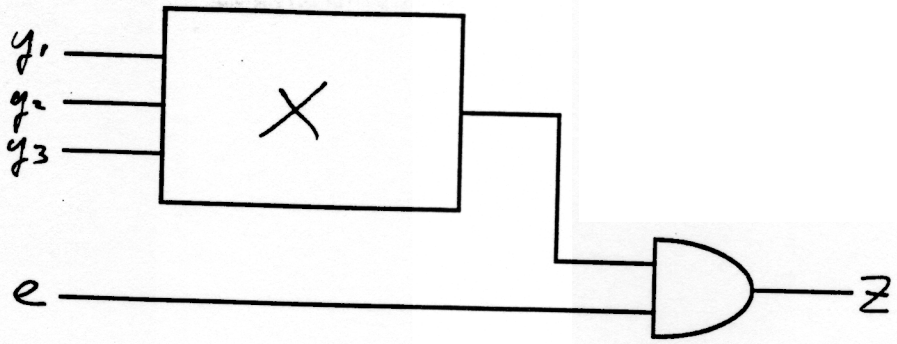

Figure 9: Example of cell test propagation.

As an example of this methodology, consider the circuit in fig. 9. Cell X, 3 inputs, 1 output, has a predefined test consisting of two patterns. This cell test is given in the library, expressed in generic inputs of the cell:

$$TST(X) = \{\overline{in_0 in_1} in_2, in_0 in_1\}$$

Using eq. 4 we find

$$CARE(X) = e.$$

The intersection $CARE(X) \cap TST(X)$ yields $\{\overline{y_1 y_2} y_3 e, y_1 y_2 e\}$. After substitution of the ON sets of y_1, y_2, y_3 and e the patterns testing cell X are found expressed in primary inputs.

6 Implementation and results

Cerberus is currently being implemented as a part of the Olympus Synthesis System developed at Stanford University. The Olympus Synthesis System is a vertical system for synthesis of digital ASICs from a behavioral description to layout and consists of several tools. All synthesis tools operating on the logic level interface with the intermediate language SLIF (Structured Logic Intermediate Format) and share the same data structures. The SLIF language and accompanying parser enable Cerberus to operate on Boolean circuit descriptions, netlist descriptions, hierarchical circuit descriptions, or any combination of these. Although using the Olympus front end and SLIF data structure, Cerberus actually runs as a standalone tool.

circuit	num pi	num po	num nodes	num lits	non recon	untest. faults	num of test patterns	cpu in seconds
c17	5	2	11	12	9	0	4	1
b1	3	3	19	30	16	0	4	2
x2	10	7	44	96	38	0	13	53
con1	7	2	18	32	13	0	6	5
tcon	17	8	49	72	32	8	4	12
cc	21	15	49	105	37	7	6	50
misex1	8	7	50	157	46	0	13	80
cu	14	11	58	119	49	7	21	55
b9	41	21	157	251	134	29	16	131
f2	4	4	8	36	4	0	2	1
pm1	16	13	47	98	39	0	8	17
pcle	19	9	35	78	19	0	6	239
cm85	11	3	35	68	20	11	9	42
cm151	12	2	42	66	25	3	5	60
unreg	36	16	68	144	66	0	7	230

Table 1: Results of Cerberus for several benchmark circuits.

Cerberus has been applied to several of the benchmark circuits from the MCNC examples. Table 1 shows some results. Each circuit is identified by its number of primary inputs, number of primary outputs, number of literals (sum of product representation), number of nodes, and number of reconvergent nodes (indicating in how many situations eq. 4 rather than eq. 3 could be used). Results are given for the number of redundancies found, the number of test patterns after test set minimnization, and CPU time. CPU times are measured in seconds on an Apollo DN4500.

Due to the underlying algorithm, Cerberus is not applicable for very large circuits. Circuit have to be flattened before the intersection between the CARE set and ON/OFF set can be taken. This is known to be a difficult and time consuming operation for certain type of circuits, especially those with a large number of reconvergent fanouts over longer paths. In its current implementation, Cerberus handles circuits with up to 500 gates.

7 Conclusions

We have presented Cerberus, a novel approach to automatic test pattern generation for combinational circuits. Cerberus computes an expression for the complete set of test patterns testing single stuck-at faults for each node in the network. From this pool of test patterns Cerberus extracts a near-minimal global test set using powerful heuristics. During test pattern generation Cerberus identifies all untestable faults (redundancies) in the circuit. Cerberus is different from the Boolean difference method in the way that it computes the expressions for the test sets per node incrementally. Finally, the algorithms underlying Cerberus allow for hierarchical test pattern generation.

References

[Bartlett86] K.A. Bartlett et.al., 'Multilevel Logic Minimization using Implicit Don't Cares' ICCD'86, pp.552-557.

[Bartlett88] K.A. Bartlett et.al., 'Multilevel Logic Minimization using Implicit Don't Cares', IEEE transactions on CAD, vol.7, no. 6, pp. 723-740, June 1988.

[Beenker86] F. Beenker et.al., 'Macro Testing: Unifying IC and Board Test', IEEE Design & Test, December 1986, pp.26-32.

[Brayton84] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, 'Logic Minimization Algorithms for VLSI Synthesis', Kluwer Academic Publishers, Dordrecht, The Netherlands, 1984.

[Brglez85] F. Brglez and H. Fujiwara, 'A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran', proc. of the IEEE International Symposium on Circuits and Systems, 1985.

[Fujiwara83] H. Fujiwara and T. Shimono, 'On the acceleration of test generation algorithms', IEEE transactions on computers, vol. c-32, pp.1137-1144, Dec. 1983.

[Damiani90] M. Damiani and G. De Micheli, 'Efficient Computation of Exact and Simplified Observability Don't Care Sets for Multiple Level

Combinational Networks', internal report CSL-TR90, Stanford University.

[Fujiwara85] H. Fujiwara, 'Logic testing and design for testability', The MIT Press, Cambridge, Massachusetts, 1985.

[Goel81] P. Goel, 'An implicit enumeration algorithm to generate tests for combinational logic circuits', IEEE transactions on computers, vol. c-30, pp.215-222, March 1981.

[Larabee89] T. Larabee, 'Efficient generation of test patterns using boolean difference', Proc. of the 1989 International Test Conference, pp.

[DeMicheli90] G. De Micheli et.al., 'The Olympus synthesis system', IEEE Design and Test, October 1990, pp.37-53.

[Roth66] J.P. Roth, 'Diagnosis of automata failures: a calculus and a method', IBM Journal of Research and Development, vol. 10, pp. 278-281.

[Sellers68] E.F. Sellers, M.Y.Hsiao and L.W.Bearnson, 'Analyzing errors with the Boolean difference', IEEE transactions on computers, vol.C-17, no.7, pp. 676-683, 1968.

[Schulz88] M. Schulz, E. Trischler and T. Sarfert, 'SOCRATES: a highly efficient automatic test pattern generation system', IEEE tranactions on CAD, vol.7, nr. 1, pp.126-137, 1988.