LOGIC SYNTHESIS FOR CONCURRENT ERROR DETECTION

Nur A. Touba
Edward J. McCluskey

Technical Report: CSL-TR-93-59 1

(CRC TR 93-6)

November, 1993

This work was supported in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and administered through the Office of Naval Research under Contract No. N00014-92-J-1782, and in part by the National Science Foundation under Grants No. MIP-9107760.

Imprimatur: LaNae Avra and Xingning Xu

Logic Synthesis for Concurrent Error Detection

Nur A. Touba and Edward J. McCluskey

CENTER FOR RELIABLE COMPUTING
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

ABSTRACT

The structure of a circuit determines how the effects of a fault can propagate and hence affects the cost of concurrent error detection. By considering circuit structure during logic optimization, the overall cost of a concurrently checked circuit can be minimized. This report presents a new technique called *structure-constrained logic optimization (SCLO)* that optimizes a circuit under the constraint that faults in the resulting circuit can produce only a prescribed set of errors. Using SCLO, circuits can be optimized for various concurrent error detection schemes allowing the overall cost for each scheme to be compared. A technique for quickly estimating the size of a circuit under different structural constraints is described. This technique enables rapid exploration of the design space for concurrently checked circuits. A new method for the automated synthesis of self-checking circuit implementations for arbitrary combinational circuits is also presented. It consists of an algorithm that determines the best parity-check code for encoding the output of a given circuit, and then uses SCLO to produce the functional circuit which is augmented with a checker to form a self-checking circuit. This synthesis method provides fully automated design, explores a larger design space than other methods, and uses simple checkers. It has been implemented by making modifications to SIS (an updated version of MIS [Brayton 87a]), and results for several MCNC combinational benchmark circuits are given. In most cases, a substantial reduction in overhead compared to a duplicate-and-compare implementation is achieved.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Concurrent error detection is an increasingly important requirement in the design of systems in which reliability and data integrity are important. As the complexity and density of VLSI continues to increase, transient faults are becoming a greater concern [Lala 88]. Concurrent error detection circuitry has the ability to detect both transient and permanent faults as well as enhance off-line testability and reduce BIST overhead [Sedmak 79], [Gupta 92].

A measure of the effectiveness of concurrent error detection in a circuit is the error *coverage* that it provides. Error coverage is defined as the percentage of the possible errors for a specified fault class that are detected. If concurrent error detection in a circuit provides 100% error coverage, then no fault in the specified fault class can cause an undetected error. Much research has been done on the problem of designing circuits with concurrent error detection that provides a certain error coverage. Many concurrent error detection schemes have been proposed that differ widely in the types of errors that can be detected and the area overhead required.

Notice that error coverage depends on two factors: 1) the set of errors that are detected, and 2) the set of errors that can occur. While most work has centered on increasing error coverage by improving the concurrent error detection circuitry such that more errors are detected, the focus of this paper will be on achieving a desired error coverage by decreasing the set of errors that faults in a circuit can cause. As will be shown in Sec. 2, the structure of a circuit limits the types of errors that faults can cause [Russell 71]. This property can be utilized to design circuits with high error coverage using low-cost concurrent error detection schemes.

When designing a concurrently checked circuit, the goal is to obtain the required error coverage while minimizing the overall cost of the circuit. In the case where cost is defined by area, the overall cost of a circuit is the sum of the area required by the functional circuit itself plus the area required by the concurrent error detection circuitry. The usual design method is to use normal logic optimization techniques to minimize the area of the functional circuit, and then choose the lowest-cost concurrent error detection scheme capable of detecting enough of the possible errors in the functional circuit to achieve the required error coverage. However, a concurrent error detection scheme with less error detection capability can still achieve the required error coverage if the functional circuit is redesigned such that the set of errors that can occur is reduced. This can often be cost effective because the area saved by using a concurrent error detection scheme with less error detection capability may be more than the increase in the area of the functional circuit needed to reduce the set of errors that occur. Therefore, the circuit with the best overall cost may not use the minimal area functional circuit, and hence may be in the portion of the design space that is not explored by the usual design method.

In Sec. 3, a new logic optimization technique called *structure-constrained logic optimization (SCLO)* is presented. SCLO does logic optimization under the constraint that faults in the resulting circuit can produce only a prescribed set of errors. By using SCLO, it is possible to reverse the order of design. The concurrent error detection scheme can be chosen first, and then based on the set of detectable errors, SCLO can be used to generate the functional circuit such that the required error coverage is obtained. The advantage of this method is that it can be used to determine the overall cost for various concurrent error detection schemes regardless of their error detection ability, thus allowing a larger design space to be explored.

In general, SCLO has to be done for each concurrent error detection scheme being considered for a circuit in order to determine the overall cost for each scheme. However, a method for estimating the overall cost without doing SCLO is described in Sec. 4. Using this method, the computation time required for comparing the overall cost of different schemes is significantly reduced thereby enabling faster exploration of the design space.

A special class of concurrently checked circuits that have been studied extensively are self-checking circuits; these circuits guarantee 100% error coverage for both the functional circuit and the concurrent error detection circuitry. In Sec. 5, self-checking circuits are defined, and a new synthesis method for generating self-checking circuit implementations for arbitrary combinational circuits is described. Designing efficient self-checking circuits is a difficult problem, and researchers have proposed several solutions. The synthesis method proposed here is completely automated and works for any combinational circuit. It consists of an algorithm that determines the best parity-check code for encoding the output of a given circuit, and then uses SCLO to optimize the self-checking circuit. This synthesis method has been implemented, and the results for several benchmark circuits are given. In most cases, a substantial reduction in overhead compared to a duplicate-and-compare implementation is achieved.

2. RELATIONSHIP BETWEEN CIRCUIT STRUCTURE AND ERRORS

2.1 Circuit Representation and Fault Model

A combinational logic circuit can be partitioned into *maximal single-output (MSO) subcircuits,* which are single-output subcircuits that are not part of larger single-output subcircuits. Fig. 1 shows an example of a combinational logic circuit partitioned into MSO subcircuits. The circuit structure information that is of interest in this section is the interconnection of MSO subcircuits. This information can be represented by a Boolean network in which each node corresponds to a MSO subcircuit.

A *Boolean network* [Brayton 90] is a directed acyclic graph where each node corresponds to a boolean function. Inward edges to the node indicate the inputs of the function, and outward edges from a node indicate the fan-outs of the function. Each primary input (PI) of the circuit is represented by a special node in the graph that does not have a corresponding boolean function. In this paper, the special PI nodes will not be shown on Boolean network diagrams in order to increase readability. Fig. 2a is an example of a Boolean network that represents the combinational logic circuit in Fig. 1. The PI's are *A*-*H* and the PO's are W-Z. Q and *P* are intermediate nodes. In a Boolean network, if a directed path exists from a node to a PO, then the PO is said to be *reachable* from that node. For example, in Fig. 2a, X is reachable from *P,* but *W* is not reachable from *P*.

The fault class that concurrent error detection techniques usually targets is *internal single stuck-atfaults,* which are all single stuck-at faults except those on the stems of PI's. The reason for this is that device operating failures are mostly caused by single point errors which are modeled well by the stuck-at fault model. Detection of stuck-at faults on the stems of PI's cannot be guaranteed unless encoded inputs are used, however, if the inputs to the circuit are checked elsewhere (e.g., if they are outputs of another concurrently checked logic block), then the only undetectable PI stem faults would be break faults after the checker [Khodadad-Mostashiri 80]. Henceforth, a "fault" will refer to an internal stuck-at fault.

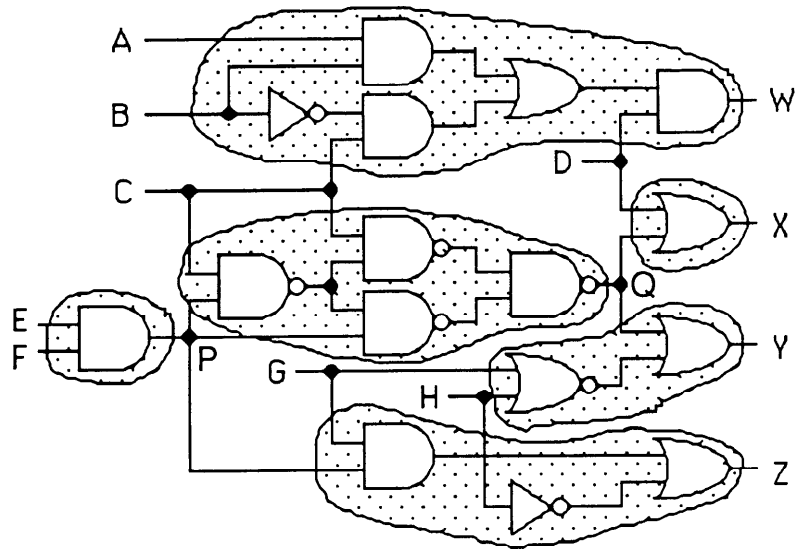


Figure 1. Example: Combinational logic circuit partitioned into MSO subcircuits

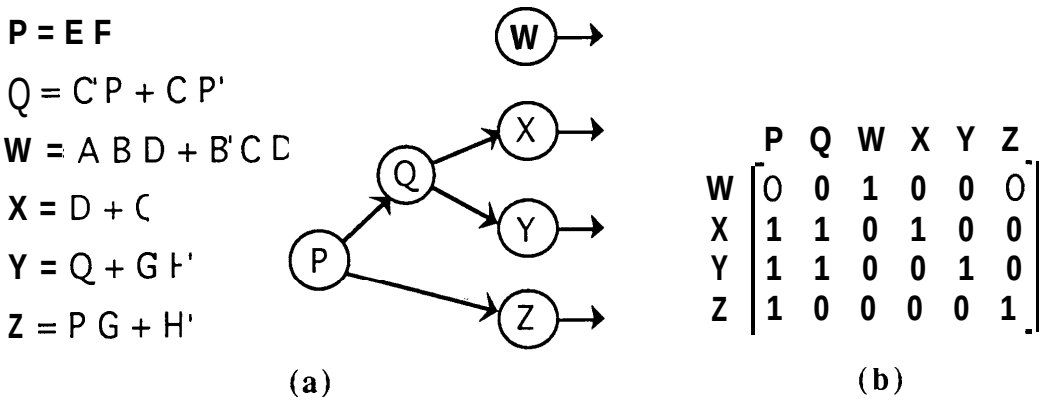


Figure 2. Example for circuit in Fig. 1: (a) Boolean Network; (b) Reachable Output Matrix

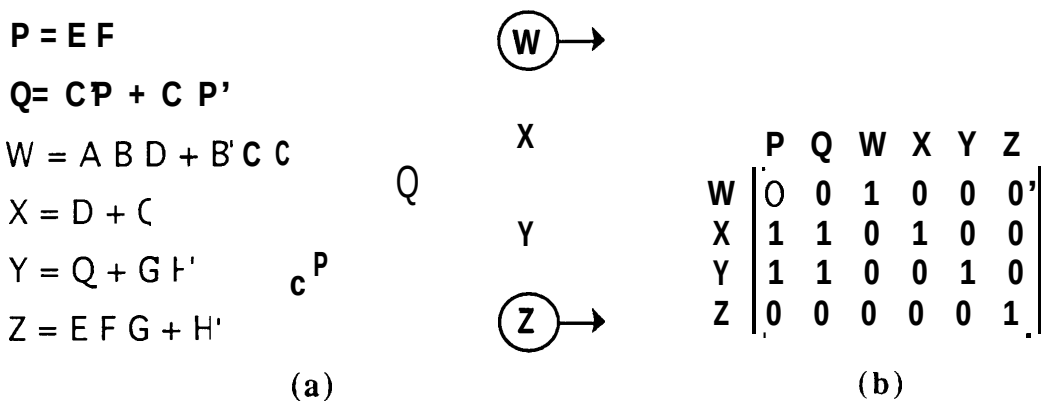


Figure 3. Node Z in Fig. 2 is collapsed: (a) Boolean Network; (b) Reachable Output Matrix

2.2 Possible Errors due to Faults

Any multiple fault that is confined to one MSO subcircuit (this includes all single faults) can be modeled at the Boolean network abstraction as changing the logic function of one node. A node whose logic function has been changed by a fault will be referred to as a *faulty node.* In order for a faulty node to cause an error at a PO, a necessary, but not sufficient, condition is that a structural path must exist from the faulty node to the PO, i.e., the PO must be reachable from the faulty node. If a PO is not reachable from the faulty node, then the logic value at the PO doesn't depend on the logic value at the output of the faulty node, so the faulty node cannot cause an error at the PO.

DEFINITION 1: An error *vector* has one element for each PO, where the element is a 1 if there exists an error at the PO, or a 0 if the PO is correct.

DEFINITION 2: The *reachable output vector* for node v, ro_v, has one element for each PO, where the element is a 1 if there exists a directed path from v to the PO, or a 0 if no path exists.

LEMMA 1: Let E_v be the set of possible error vectors due to a fault that changes the logic function of node v. Then E_v is a subset of the set of all subvectors of the reachable output vector for node v, where vector a is said to be a *subvector* of vector b if a can be obtained by changing zero or more elements in b from 1 to 0.

$$E_v \subseteq \{ \text{ all subvectors of } ro_v \}$$

PROOF: In order for a faulty node v to cause an error at a PO, a sensitized path must exist from v to the PO such that if the logic value at the output of v is incorrect, the logic value at the PO will be incorrect [Armstrong 66]. In order for a sensitized path to exist from v to a PO, a structural path must exist, thus the PO must be reachable from v. Therefore, the reachable output vector for v indicates the possible PO's that could be in error. For each input pattern, any combination of these PO's could be in error, so all subvectors of the reachable output vector for v form a superset of the possible error vectors.

As can be seen from Lemma 1, the reachable output vectors give a bound on the set of possible error vectors due to any multiple fault in one MSO subcircuit. Because the reachable output vectors depend only on the structure of the circuit, they can be determined quickly using an algorithm such as [Unger 69] that finds the reachability matrix of a graph.

DEFINITION 3: The *reachability matrix* of a graph, R, is a square matrix having one row and one column for each node. Each entry $r_{i,j}$ is 1 if a directed path exists from the row i node to the column j node, otherwise it is 0.

In this paper, a node will be defined to have a path to itself, so the diagonal entries in the reachability matrix will be all 1's. The reachability matrix actually contains more information than what is needed, so it can be reduced to the reachable output matrix which is defined as follows:

DEFINITION 4: The *reachable output matrix* of a graph, M, is a matrix having a row for each PO, and a column for each non-PI node. Each entry $m_{i,j}$ is 1 if the row i PO is reachable from column j node, otherwise it is 0.

The reachable output matrix can be derived from the reachability matrix by removing the PI rows and columns, and the non-PO columns, and then taking the transpose. Note that the reachable output vector for each node is given by its column vector in the reachable output matrix. The reachable output matrix for the Boolean network in Fig. 2a is shown in Fig. 2b. By looking at the column vectors, it can be seen that the worst case error would be for a fault in node P which can cause up to three PO's to be incorrect. The Boolean network in Fig. 3a is the same as in Fig. 2a except that node Z's dependency on node P has been eliminated by replacing P by EF in the logic equation for Z. The reachable output matrix for the Boolean network in Fig. 3a is shown in Fig. 3b. Now the worst case error due to any fault can only cause up to two PO's to be incorrect.

3. STRUCTURE-CONSTRAINED LOGIC OPTIMIZATION (SCLO)

The normal goal of logic optimization is to minimize area without regard to the structure of the circuit. However, as was shown in Sec. 2, the structure of a circuit determines the types of errors that can occur, thus if concurrent error detection is a requirement, circuit structure becomes an important consideration. The purpose of SCLO is to minimize area under structure constraints.

Multilevel logic optimization improves circuit area by restructuring and minimizing the logic represented by a Boolean network. After the Boolean network is optimized, technology mapping is done to translate the boolean functions into primitive cells belonging to a specified target library. Commonly used technology mapping procedures, such as tree-mapping [Keutzer 87], [Detjens 87], follow the structure of the Boolean network such that the structure of the mapped circuit corresponds to the structure of the Boolean network. Therefore, by constraining the structure of the Boolean network, the structure of the mapped circuit can be constrained. In SCLO, restrictions are placed on the normal restructuring operations to ensure that the resulting circuit will satisfy the structural constraints.

3.1 Constraint Specification

The constraints for SCLO are specified as a set of reachable output vectors that each non-PI node in the Boolean network is allowed to have; this set is called the *reachable output set (ROS).* If the circuit has n PO's, then there are 2^n possible reachable output vectors for each node, so the ROS can contain some subset of these vectors. If the ROS contains only the weight-l vectors, then each non-PI node can have a path to only one output, which corresponds to having a circuit structure in which no logic is shared between outputs. By adding more vectors to the ROS, the constraints on the structure of the circuit can be relaxed allowing more logic sharing. If all possible vectors are included in the ROS, then any circuit structure can be used, which corresponds to normal unconstrained logic optimization.

3.2 Constraints on Restructuring Operations

Given the ROS and the initial multilevel logic equations, SCLO optimizes the logic under the constraint that the reachable output vector for each non-PI node is contained in the ROS. Multilevel logic optimization involves using operations that attempt to restructure the logic in order to reduce its size. SCLO can be accomplished by requiring that the initial logic equations satisfy the constraints, and then constraining the restructuring operations so that they never cause nodes to violate the constraints. Two restructuring operations that can cause a node to violate the constraints are resubstitution and extraction [Brayton 90].

3.2.1 Constraints on Resubstitution

Resubstitution is an operation where some node a, which is divisible (see Def. 5 below) by another node b, is rewritten as a function of node b, thus creating an arc from node b to node a. Since resubstitution adds an arc to the graph, it may create a path such that node b's reachable output vector is no longer in the ROS, thus violating the constraints. Therefore, in SCLO, resubstitution can be performed between two nodes only if the resulting arc does not violate the constraints.

DEFINITION 5: *Node a* is *algebraically divisible* by node b if an algebraic expression (minimal sum of products) for a is such that $a = kb + r$ where kb is an algebraic product in which k and b have no input variables in common. Note that $(w + x)(y + z) = wy + wz + xy + xz$ is an algebraic product, but $(x + y)(x + z) = x + xy + xz + yz$ and $(x + y)(y' + z) = xy' + xz + yz$ are not [Brayton 90].

Because of computational costs, algebraic division is preferred to boolean division in resubstitution operations. Fig. 4a shows a Boolean network in which node Q is algebraically divisible by node P. The dashed arrow indicates the new arc that would be added if resubstitution of P into Q is performed. In order to check if this resubstitution would violate the constraints, the new reachable output vector for P needs to be determined. From the reachable output matrix, which is shown in Fig. 4b, the current reachable output vectors for Q and P can be found by taking their column vectors. If the arc is added, P will be able to reach every PO that Q can reach in addition to all the PO's that P can already reach. Thus, the new reachable output vector for P will be equal to the logical OR of its current reachable output vector and Q's reachable output vector. This calculation is shown in Fig. 4c. If this new reachable output vector is in the ROS, then the resubstitution can be done, otherwise it cannot be done.

In the general case, in order for node b to be an algebraic divisor of node a, every variable that is used in the logic expression for b must also be used in the logic expression for a, thus any fan-in (inward arc) to node b must also be a fan-in to node a, hence any node that has a directed path to node b must also have a directed path to node a. Therefore, adding an arc from node b to node a would only change the reachable output vector of node b, so resubstitution of node b into node a can be performed in SCLO as long as the new reachable output vector for node b is in the ROS. This is formally stated as follows:

CONSTRAINT: Resubstitution of node b into node a can only be done if the following is true:

$$(ro_a \vee ro_b) \in \text{ROS}$$

Implementing constrained resubstitution is relatively easy. Various filters are generally used to reduce the number of node pairs for which resubstitution is attempted [Brayton 87a]. So, this constraint can simply be added as an additional filter.

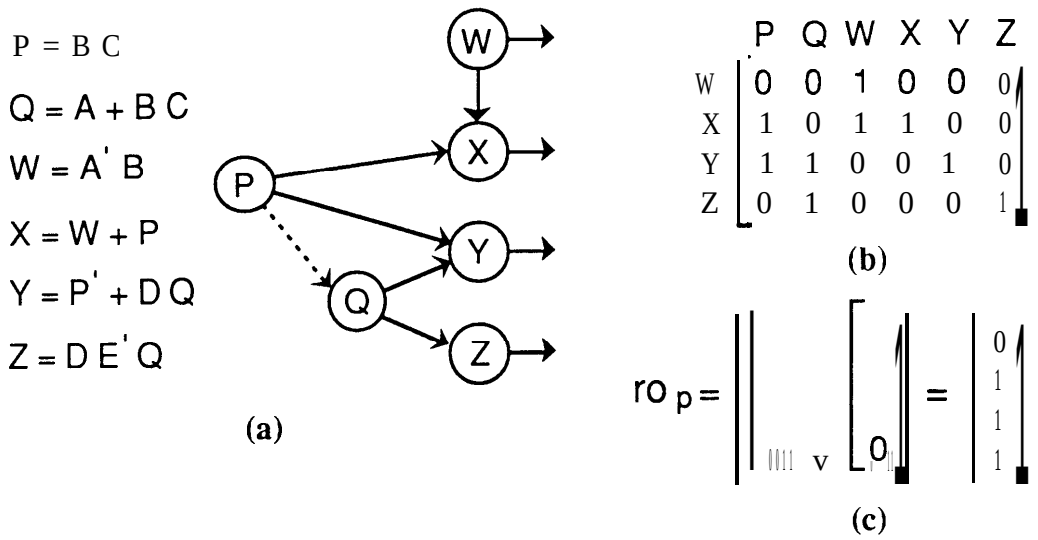


Figure 4. Resubstitution Example: (a) Boolean Network; (b) Reachable Output Matrix; (c) Calculation of New Reachable Output Vector for Node *P*

3.2.2 Constraints on Extraction

Extraction is an operation in which common subexpressions from a set of nodes are used to create intermediate nodes. These intermediate nodes are then used as inputs to the original nodes, thereby reducing the overall literal count. A newly created intermediate node may fan out in such a way that its reachable output vector is not in the ROS, thus violating the constraints. Therefore, in SCLO, extraction must be done in such a way that it doesn't create an intermediate node that violates the constraints.

An intermediate node is created by extracting a common subexpression from a set of nodes. The intermediate node will have an arc to each node in the set of nodes from which it was extracted, and therefore can reach any PO that any node in the set can reach. So, the reachable output vector for an intermediate node is equal to the logical OR of the reachable output vectors of the set of nodes from which it was extracted. Therefore, common subexpressions can only be extracted from a set of nodes if the logical OR of the reachable output vectors for each node in the set is in the ROS. This is formally stated as follows:

CONSTRAINT: A common subexpression from a set of nodes $(v_1, v_2, ..., v_n)$ can only be extracted if the following is true: $(ro_{v_1} \vee ro_{v_2} \vee \ldots \vee ro_{v_n}) \in$ ROS

In order to implement constrained extraction, the process of selecting common subexpressions to extract needs to be modified. In MIS [Brayton 87a], the two types of subexpressions that are considered for extraction are kernels and cubes. Rectangle covering can be used to find either the optimal common kernels or optimal common cubes to extract. In cube extraction, the cube-literal matrix is formed with one row for each term in the disjunctive form of each function and one

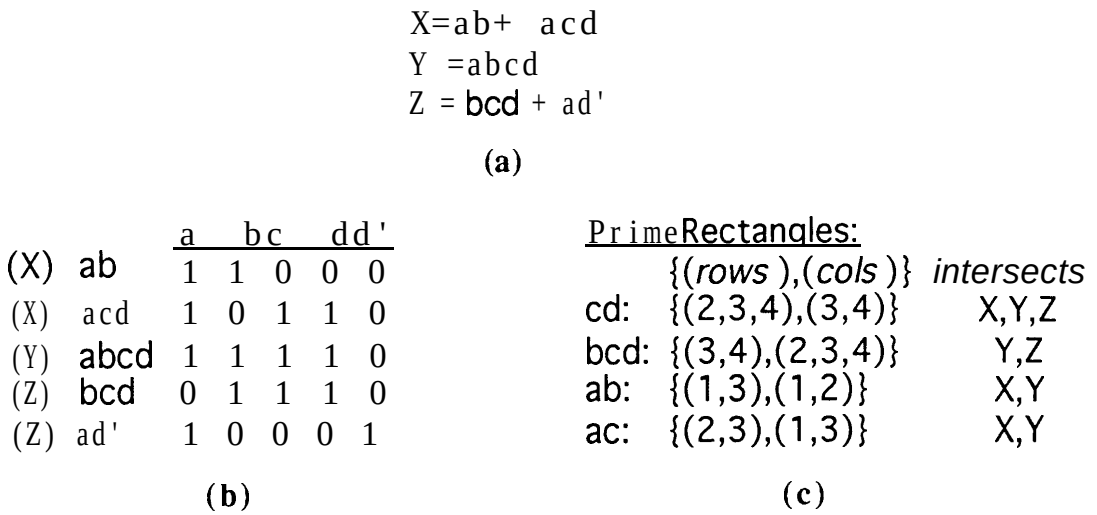


Figure 5. Cube-Literal Matrix Example: (a) Functions; (b) Cube-Literal Matrix; (c) Prime Rectangles

column for each different literal, see example in Fig. 5. Rectangles in the cube-literal matrix correspond to common cubes. The "value" of a rectangle is defined as the number of literals that would be saved if it is extracted. A heuristic procedure for finding the optimal common cubes is to extract the "best" rectangle, substitute it into the expressions, update the cube-literal matrix, and then reapply the procedure [Brayton 90]. Kernel extraction is very similar, except that the co-kernel cube matrix is used instead of the cube-literal matrix (see [Brayton 87b] for more details).

A procedure for finding the best rectangle for extraction is to find all the prime rectangles, compares their value, and then chooses the best one [Rudell 89]. A *prime rectangle* is a rectangle that is not contained in another rectangle. Since any non-prime rectangle is a *subrectangle* of some prime rectangle, it will always have a smaller value than the prime rectangle that it is contained in. Thus, only prime rectangles need to be considered.

In SCLO, some rectangles cannot be extracted because they would result in intermediate nodes which violate the constraints. In order to check if a rectangle can be extracted, the set of nodes it intersects must be determined. The set of nodes that a rectangle intersects is equal to the union of the nodes corresponding to the rows covered by the rectangle, see example in Fig. 5. A rectangle can only be extracted if the logical OR of the reachable output vectors for each node it intersects is contained in the ROS, otherwise it violates the constraints.

In SCLO, the previously described procedure for finding the best rectangle to extract is modified so that it finds the best rectangle that satisfies the constraints. An algorithm for doing this is shown in Table 1. All the prime rectangles are found and compared as before. However, if the best prime rectangle doesn't satisfy the constraints, then it must be replaced by the best subrectangle contained in it which does satisfy the constraints. Then this subrectangle is compared

Table 1. Algorithm for Selecting Best Rectangle that Satisfies Constraints

```
SELECT-BEST-RECTANGLE (matrix):
    rect-set = FIND_PRIME_RECTS(matrix)
    best-rect = MAX_VALUE_RECT(rect_set)
    while ( best-rect doesn't satisfy constraints ) {
        rect-set = rect_set - best-rect
        rect-set = rect-set u FIND-BEST-SUBRECT(best-rect)
        best-rect = MAX_VALUE_RECT(rect_set)
    }
    return (best-ret t)


FIND-BEST-SUBRECT (rect):
    node set = ∅
    for each row r in rect {
        node-set = node-set u NODE(r)
        node_weight[NODE(r)] += VALUE(r)
    }
    comp_graph = create compatibility graph for node-set
    mm_clique_set = find maximum weight clique in comp_graph
    subrect = 0
    for each row r in rect {
        if ( NODE(r) ∈ max_clique_set )
            subrect = subrect u r
    }
    return(subrect)
```

with the other prime rectangles, and the best one is chosen. This process continues until either a prime rectangle which satisfies the constraints is chosen, or one of the replacement subrectangles is chosen. Finding the best subrectangle can be done using the compatibility graph for the nodes that are intersected by the prime rectangle. Two nodes are compatible if the logical OR of their reachable output vectors is contained in the ROS. Each node in the compatibility graph is assigned a weight based on the relative number of literals that would be saved if the node is used, so the maximum weight clique in the compatibility graph indicates the best subrectangle. A *clique* is a maximal complete subgraph [Bron 71].

		a	a'	b	b'	c	d	e
(W)	b'cd	0	0	0	1	1	1	0
(X)	acde	1	0	0	0	1	1	1
(Y)	abcd	1	0	1	0	1	1	0
(Z)	bcde	0	0	1	0	1	1	1
(Z)	a'cd	0	1	0	0	1	1	0

Prime Rectangles:	{(*rows*),(*cols*)}	Value
cd:	{(1,2,3,4,5),(5,6)}	3
acd:	{(2,3),(1,5,6)}	1
bcd:	{(3,4),(3,5,6)}	1
cde:	{(2,4),(5,6,7)}	1

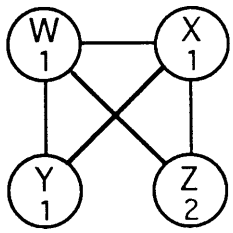
Constraints:

$(ro_W \vee ro_X) \in ROS$ $(ro_X \vee ro_Y) \in ROS$

$(ro_W \vee ro_Y) \in ROS$ $(ro_X \vee ro_Z) \in ROS$

$(ro_W \vee ro_Z) \in ROS$ $(ro_Y \vee ro_Z) \notin ROS$

Compatibility Graph for cd:



Maximum Weight Clique is W,X,Z $\Rightarrow$ Best Subrectangle: {(1,2,4,5),(5,6)}

Figure 6. Example of Finding Best Rectangle that Satisfies ROS Constraints

Fig. 6 shows an example of finding the best rectangle to extract from a cube-literal matrix. The best prime rectangle is *cd*, but this rectangle violates the constraints because it intersects both Y and Z. In the compatibility graph for *cd*, node Z is weighted with twice the value of the other nodes because *cd* is used twice in Z but only once in the other nodes The maximum weight clique in the graph is *W,X,Z*. So, the best subrectangle contains all rows in the original prime rectangle for *cd* that intersect *W,X,Z*. This subrectangle has a value of 2 which is more than all the other prime rectangles in the matrix. Therefore, it is the best rectangle which satisfies the constraints.

3.3 Application to Concurrent Error Detection

As was shown in Sec. 2, the set of possible error vectors in a circuit is bound by the reachable output vectors for each node in the corresponding Boolean network. Therefore, since the ROS constrains the reachable output vectors for each node, the ROS can be used to constrain the set of possible error vectors in a circuit. So, if a concurrent error detection scheme is to be used for a circuit, then the ROS can be chosen such that SCLO will generate a circuit in which faults produce only detectable errors. For example, if a concurrent error detection scheme that is capable of detecting 1 or 2 bit errors is used, then by putting only weight-l and weight-2 vectors in the ROS, the circuit structure is constrained such that non-PI nodes have paths to only one or two outputs. Thus, any fault in a non-PI node can only produce a 1 or 2 bit error at the PO's, which would be detected.

4. ESTIMATING STRUCTURE-CONSTRAINED CIRCUIT SIZE

Choosing the best concurrent error detection scheme requires knowing the size of the functional circuit for each scheme. This involves doing SCLO with the appropriate ROS for each scheme. In order to reduce the computation time required to evaluate various concurrent error detection schemes, a method for estimating the size of the functional circuit without doing SCLO is presented in this section.

The circuit that results from doing unconstrained logic optimization will be referred to as the *unconstrained circuit,* and the circuit that results from doing SCLO will be referred to as the *constrained circuit.* The size of the constrained circuit can be estimated by determining which nodes in the unconstrained circuit have reachable output vectors that are not in the ROS. If a node in the unconstrained circuit violates the constraints, it can be replicated in such a way that each copy will satisfy the constraints. Consider the unconstrained circuit shown in Fig. 7. If the all l's vector is not contained in the ROS, then nodes P and Q violate the constraints. In Fig. 8, these two nodes have been replicated so that each copy satisfies the constraints, i.e., none of the nodes can reach all the PO's.

By determining the number of extra copies of each node in the unconstrained circuit that are needed to satisfy the constraints, an upper bound on the size of the constrained circuit can be computed. In the worst case, the constrained circuit could be generated by replicating nodes in the unconstrained circuit, and its size would then be equal to the size of the unconstrained circuit plus the size of the extra nodes, where size is defined as the factored form literal count. However, since SCLO takes the constraints into consideration when deciding how to restructure the circuit, it will usually generate a smaller circuit. The size of the constrained circuit is bounded as follows:

$$\text{unconstrained size} \leq \text{constrained size} \leq \text{unconstrained size} + \text{size of extra nodes}$$

Extra copies of a node in the unconstrained circuit are needed if the reachable output vector for the node is not contained in the ROS. When a node is replicated, the PO's that it reaches can be distributed among the copies. For example, if a node that can reach three PO's is triplicated, each copy need only reach one PO. If a node's reachable output vector is not in the ROS, it must be replicated enough times so that each copy has a reachable output vector that is in the ROS, and the logical OR of the reachable output vectors for all the copies equals the node's original reachable output vector. Based on this requirement, an algorithm can be written to scan through each node in the unconstrained circuit and determine how many extra copies are needed to satisfy the constraints imposed by the ROS. By summing the sizes of these extra nodes, the worst case size of the constrained circuit can be computed without doing SCLO. The worst case size gives a relative measure of the area complexity for different structural constraints. This value, or some fraction of it, can be used as an estimate of the constrained circuit size.

Using this estimation technique, the design space for a concurrently checked circuit, in which each point corresponds to using a different concurrent error detection scheme, can be rapidly explored to find the minimal area implementation. The overall circuit area for each scheme can be computed by determining the ROS that would provide the required error coverage and then estimating the size of the constrained functional circuit. The sum of the size of the constrained functional circuit and the size of the concurrent error detection circuitry gives the overall area for each scheme. For the scheme with the smallest overall area, the functional circuit can then be generated using SCLO and augmented with the concurrent error detection circuitry to produce a concurrently checked circuit with the required error coverage.

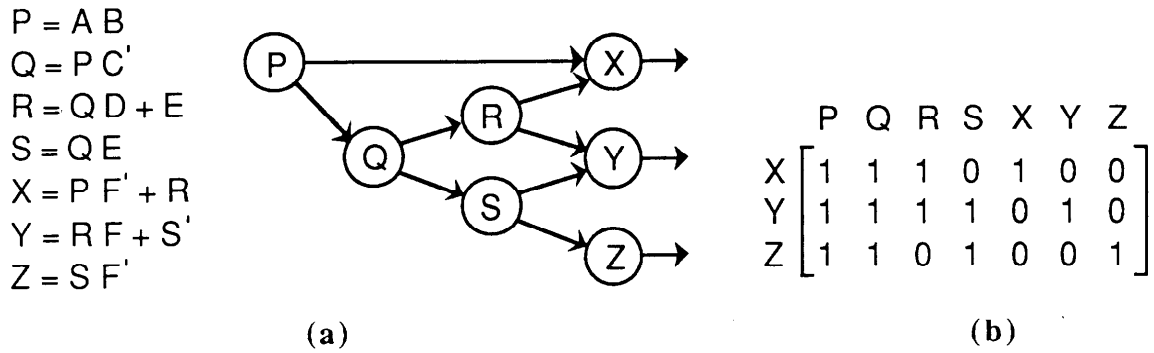


Figure 7. Example: Unconstrained Circuit (Lower Bound)
(a) Boolean Network; (b) Reachable Output Matrix

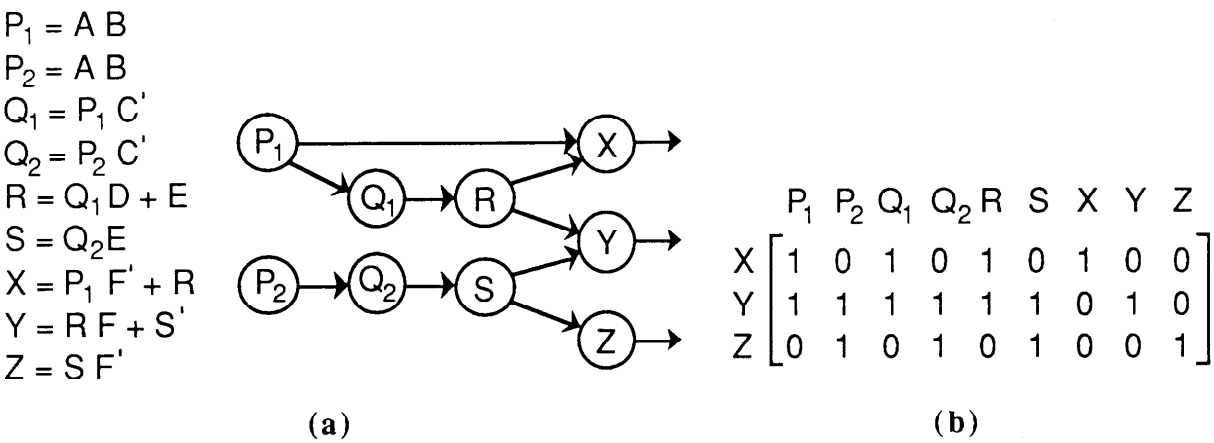


Figure 8. Circuit Obtained by Replicating Nodes to Satisfy Constraints (Upper Bound)
(a) Boolean Network; (b) Reachable Output Matrix

5. SYNTHESIS OF SELF-CHECKING CIRCUITS

The synthesis techniques described in this paper can also be used in the automated synthesis of self-checking circuit implementations for arbitrary combinational circuits. In this section, a new synthesis method is presented. It consists of an algorithm that determines the best parity-check code for encoding the outputs of a given circuit, and then uses SCLO to produce the functional circuit which is augmented with a checker to form a self-checking circuit implementation.

5.1 Definitions

The goal of a self-checking circuit is called *the totally self-checking (TSC) goal,* which is to detect the first error that occurs due to any fault in a specified fault class [Smith 78]. The general structure of a self-checking circuit is shown in Fig. 9. The output of the functional circuit is encoded such that if an error occurs, it will produce a non-codeword. A *totally self-checking (TSC) checker* is used to monitor the output of the functional circuit and indicate an error if a non-codeword appears or if a fault occurs in the checker itself. One way to achieve the TSC goal is if the functional circuit is both fault secure and self-testing and its output is checked by a TSC checker [Anderson 7 1]. However, as was shown in [Smith 78], the self-testing requirement can be relaxed if the functional circuit is strongly fault secure (SFS). [Smith 78] also introduced the concept of path fault secure (PFS) circuits, and showed that it is a subclass of SFS circuits. PFS circuits can be defined using the terms described in Sec. 2; this is done in Def. 9.

DEFINITION 6: A circuit is *fault-secure* if for every fault in a specified fault class, the circuit never produces an incorrect codeword output for any codeword input.

DEFINITION 7: A circuit is *self-testing* if for every fault in a specified fault class, the circuit produces a non-codeword output for at least one codeword input.

DEFINITION 8: A circuit is *strongly fault-secure (SFS)* if for each fault in a specified fault class, one of the following is true:

(i) the circuit is fault secure and self-testing, or

(ii) the circuit is fault secure and remains SFS in the presence of the fault

A SFS circuit remains fault secure until it becomes self-testing, thus no sequence of faults can cause an undetected error.

DEFINITION 9: A circuit is *path fault secure (PFS)* if for each fault in a specified fault class, each of the possible errors determined from the structural paths, i.e., reachable output vectors (by way of Lemma 1), must produce non-codewords.

Assuming no fault can add a structural path, a PFS circuit remains fault secure in the presence of any faults and therefore is SFS.

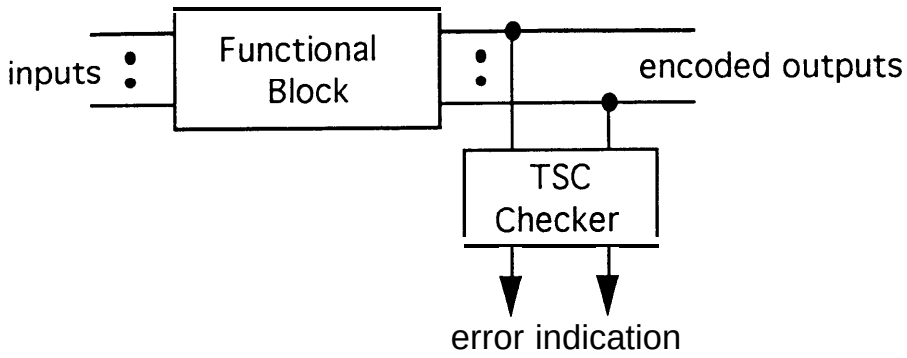


Figure 9. General Structure of a Self-Checking Circuit

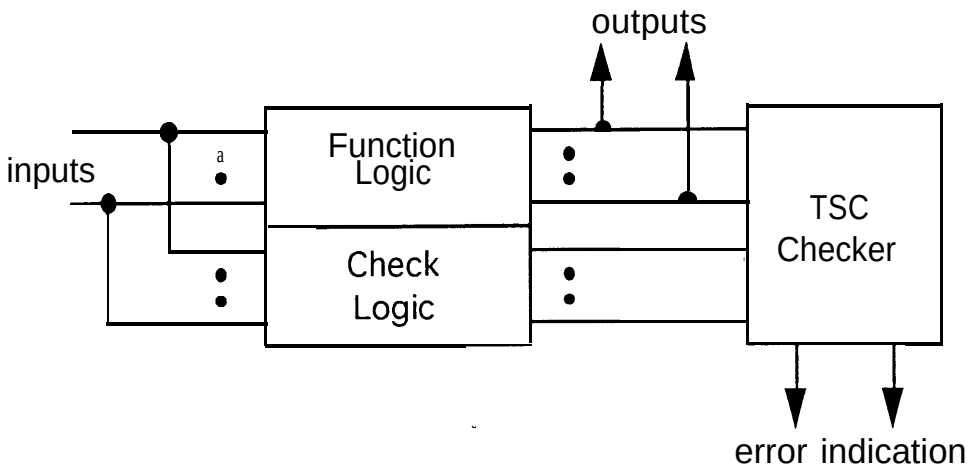


Figure 10. General Structure of a Self-Checking Circuit using a Systematic Code

In Sec. 2, it was shown how the set of errors due to internal single stuck-at faults depends on the reachable output vectors of the nodes in the circuit. Based on this dependence, a circuit whose output is encoded such that each possible error causes only non-codewords is PFS for internal single stuck-at faults. As was shown in [Smith 78], PFS circuits that are checked by a TSC checker achieve the TSC goal regardless of what input patterns are applied to the circuit during normal operation. This property is important because it eliminates the need to consider the operating input patterns in the design process.

In a *systematic code,* codewords are constructed by appending check bits to the normal output bits. Using a systematic code in a self-checking circuit has the advantage that no decoding is needed to get the normal outputs. Fig. 10 shows the general structure of a self-checking circuit

that uses a systematic code. The circuit has three parts: function logic, check logic, and checker. The function logic generates the normal outputs, the check logic generates the check bits, and the checker determines if they form a codeword.

A *parity-check code* is a code in which each check bit is a parity check for a group of output bits. Each group of outputs that is checked by a check bit is called a *parity group,* and corresponds to a row in the parity check matrix $\boldsymbol{H}$ [Pradhan 86]. Both single-bit parity and duplication are parity-check codes. Table 2 shows the parity check matrix $\boldsymbol{H}$ for a circuit with three outputs Z_1, Z_2, Z_3, encoded with single-bit parity and with duplication. In single-bit parity, there is one parity group which contains all the outputs. In duplication of a circuit with *n* outputs, there are *n* parity groups each containing one of the outputs.

Table 2. $\boldsymbol{H}$ Matrix for circuit with three outputs: (a) Single-Bit Parity Code; (b) Duplication Code.

(a)

Group	Z_1	Z_2	Z_3	c_1
1	1	1	1	1

(b)

Group	Z_1	Z_2	Z_3	c_1	c_2	c_3
1	1	0	0	1	0	0
2	0	1	0	0	1	0
3	0	0	1	0	0	1

5.2 Previous Work

Several methods for the design of self-checking circuits for arbitrary combinational circuits have been proposed. The simplest method is to duplicate the functional logic and use a two-rail checker (equality checker). This is commonly used, but requires more than 100% overhead. Another simple and commonly used method is conventional parity prediction in which a single parity check bit is generated [Khodadad-Mostashiry 79]. The check logic and checker are small, but severe restrictions are placed on the structure of the functional circuit since only errors that affect an odd number of PO's can be detected. An extended parity checking method capable of detecting errors affecting an even number of PO's was proposed in [Ko 78], and a generalized parity prediction checker for any parity-check code was described in [Fujiwara 87].

In [Jha 91], a synthesis method is proposed in which the functional circuit is optimized using a MIS script with only algebraic operations such that the resulting circuit can be transformed so that it has inverters only at the PI's. The outputs can then be encoded with a Berger code, which is an optimal systematic all-unidirectional error-detecting code, so that the circuit is self-checking for all internal single stuck-at faults. One drawback is that the checker required for a Berger code tends to

be large. However, in the case where the functional circuit produces only a small subset of all possible output patterns, a separable code with fewer check bits than a Berger code may be used.

In [De 92], two synthesis methods are described. The first uses a Berger code and is essentially the same as [Jha 91]. The second uses a parity-check code and is similar to the method proposed in this paper. The functional circuit outputs are partitioned to form logic blocks in such a way that logic sharing within each block is maximized. The parity functions for the necessary check bits are generated and form another logic block. MIS is then used to optimize each logic block separately thereby restricting the circuit structure such that no logic is shared between each logic block. A TSC checker for the appropriate parity-check code is added to form a self-checking circuit.

The synthesis method proposed here offers several advances over the previously proposed methods. It automatically selects the best parity-check code for encoding the outputs of a given functional circuit such that the overall area is minimized, and then performs SCLO to generate an optimized PFS circuit that is augmented with a TSC checker to produce a self-checking circuit. Whereas [De 92] only considers logic sharing when choosing the parity-check code, the algorithm given here considers the size of the parity functions and the size of checker as well as logic sharing when choosing the parity-check code, thereby minimizing the size of the overall circuit instead of just the size of the function logic. By using SCLO to generate a PFS circuit, the whole circuit is synthesized together allowing additional logic sharing opportunities to be explored, thus producing a better result. Also, since a parity-check code is used, the checker is usually smaller than the checker required for a Berger code.

5.3 Selecting Parity-Check Code

The procedure for the proposed synthesis method is to first select a parity-check code for encoding the outputs of the functional circuit. Then the logic equations for the check bits used in the selected code are computed. These logic equations are then combined with the functional logic equations and synthesized under the constraint that the resulting circuit is PFS with respect to the selected code. The PFS circuit is then augmented with a TSC checker for the selected code to produce a self-checking circuit. Since each step of the synthesis process is affected by the selected code, selecting the best code is very important.

Given a functional circuit, selecting a parity-check code involves choosing the number of parity groups (i.e. check bits), and then assigning the outputs to the parity groups. It is very rare that placing an output in more than one parity group will reduce the overall size of the circuit, thus only codes in which each output is in exactly one parity group are considered. Therefore, selecting the code requires partitioning the outputs into parity groups.

A parity-check code can detect any error in which an odd number of output bits in any parity group are incorrect. If the structure of the circuit is such that each node in the circuit can only reach

at most one output in each parity group, then the circuit is PFS because no fault can cause an undetectable error. So, there exists a tradeoff between the number of parity groups and the constraints on the structure of the function logic when generating a PFS circuit. Single bit parity and duplication are the two extreme cases. In single bit parity, there is one parity group, so the check logic and checker are small, but no logic can be shared between outputs, so the function logic may be large. In duplication, there is one parity group for each output, so the check logic and checker may be large, but there are no constraints on the structure of the function logic. By reducing the number of parity groups, the size of the check logic and checker decreases, but the additional structural constraints may increase the size of the function logic. The goal is to select the code that minimizes the overall size of the circuit.

5.3.1 Estimating Circuit Size

The exact solution to the problem of selecting the optimal code for a circuit requires adding the appropriate check bit equations to the function equations and doing SCLO for each possible code to compare the resulting circuit sizes. Since the number of possible codes is exponential in the number of outputs, heuristics are needed. The size of the circuit is equal to the sum of the sizes of the function logic, the check logic, and the checker. The size of the checker depends only on the number of parity groups, so it can easily be determined. The size of the function logic can be estimated using the method described in Sec. 4 with the appropriate ROS for each code. The size of the check logic, however, is difficult to estimate. An exact computation requires generating the check bit equations by taking the sum modulo-2 of the logic equations for the outputs in each parity group, and then doing multilevel logic optimization. The computation time required for this is obviously not practical.

While it is difficult to estimate the absolute size of the check logic for each code, the relative sizes for two codes can be approximated. If one code can be generated from another code by merging two parity groups, then the difference in the sizes of the check logic for the two codes can be approximated using only node minimization (two-level minimization of a node). Consider the case where code B is generated by merging two parity groups, corresponding to check bits c_1 and c_2, in code A. The change in the size of the check logic can be approximated by comparing the factored form literal count of c_1 and c_2 with $(c_1 \oplus c_2)$ after node minimization:

$$\text{check logic size } (A) - \text{check logic size } (B) \approx \text{lits}(c_1) + \text{lits}(c_2) - \text{lits}(c_1 \oplus c_2)$$

This approximation neglects the effect of global restructuring operations that would be performed in multilevel logic optimization. Since the only difference between the check logic for code A and code B is that two nodes were combined, this approximation gives a reasonable prediction of the change in size.

5.3.2 Algorithm

A greedy algorithm for selecting an optimal parity-check code for a circuit is given in Table 3. It begins with the duplication code in which each output is in its own parity group. Using the estimation techniques that have been described, it computes the change in circuit size (literal savings) if any two parity groups were merged. The pair that would give the largest reduction in size is merged, and the process is repeated for the resulting code. Parity groups continue to be merged until the size of the circuit can no longer be reduced by merging groups.

When computing the literal savings from merging some pair of parity groups, three factors are considered: 1) reduction in the size of check logic due to combining two parity functions into one *(parity-reduce),* 2) reduction in the size of checker due to eliminating one input *(chk_reduce),* and 3) increase in the size of function logic due to added structural constraints because of the decrease in error detection ability *(weight * shared).* The sum of the first two factors minus the third gives the net reduction in size *(savings).*

In estimating the effect of the structural constraints on the size of the function logic, a fraction *(weight)* of the literals shared between the two merged groups *(shared)* is used. When two parity groups are merged, logic can no longer be shared between the outputs in each group for the reasons that were explained in Sec. 4. However, this is an upper bound on the loss because SCLO will likely restructure the circuit in such a way that other logic sharing is possible, thereby compensating for some of the loss. *So weight* is a parameter that is used to estimate the actual loss in logic sharing as a fraction of the worst case loss. For most circuits, $weight=0.5$ has been found to give good results. In general, the synthesis process can be repeated for a few different values of *weight,* and the best result can be used.

The algorithm starts with the duplication code which has n parity groups where n is the number of outputs in the circuit. It then considers all codes with **n-l** parity groups and greedily chooses the optimal one. Then it considers all codes with *n-2* parity groups that can be generated by merging two parity groups in the optimal code for *n-l* parity groups and greedily chooses the best one. This process continues until the selected code for *n-i* parity groups is better than any of the codes with *n-i-* 1 parity groups that are considered. All codes with n or n- 1 parity groups are considered, so if the optimal code has n or **n-l** parity groups, it is guaranteed to be found. Only a **subset** of the codes with fewer than *n-l* parity groups are considered. This subset is generated using the heuristic of greedily merging groups to minimize parity functions. Thus, a solution is obtained using only $O(n^2 log_2 n)$ operations, where each operation requires computing the exclusive OR of two nodes and simplifying it. These operations can be done fairly quickly in MIS thereby making the algorithm practical for circuits with a reasonable number of outputs.

Table 3. Algorithm for Selecting Parity-Check Code

SELECT-PARITY-CHECK-CODE (*funct_logic*, *weight*):

n = NUM_PO(*funct_logic*)
for i = 1 to n {
  $group_i = \{ i \}$
  c_i = NETWORK_GET_PO_NODE(*funct_logic*, i)
}
groups = { 1,2,...,n }
optf unct-logic = unconstrained logic optimization of *funct-logic*
for each unordered pair (i,j) where $i, j \in groups$ and $i \neq j$ {
  $shared_{i,j}$ = LITS_SHARED(*opt_f unct-logic*, $group_i$, $group_j$)
  $c_{i,j}$ = NODE_XOR(c_i, c_j)
  $c_{i,j}$ = NODE_SIMPLIFY($c_{i,j}$)
  $parity\_reduce_{i,j}$ = LITS(c_i) + LITS(c_j) - LITS($c_{i,j}$)
  $savings_{i,j} = parity\_reduce_{i,j} + chk\_reduce - weight * shared_{i,j}$
}
while (MAX($savings_{i,j}$) > 0) {
  $group_i = group_i \cup group_j$
  $c_i = c_{i,j}$
  groups = *groups* - { j }
  for each $k \in groups$ where $k \neq j$
    $savings_{j,k} = -00$
  for each $k \in groups$ where $k \neq i$ {
    $shared_{i,k}$ = LITS_SHARED(*opt_f unct-logic*, $group_i$, $group_k$)
    $c_{i,k}$ = NODE_XOR(c_i, c_k)
    $c_{i,k}$ = NODE-SIMPLIFY ($c_{i,k}$)
    $parity\_reduce_{i,k}$ = LITS(c_i) + LITS(c_k) - LITS($c_{i,k}$)
    $savings_{i,k} = parity\_reduce_{i,k} + chk\_reduce - weight * shared_{i,k}$
  }
}
$row = 0$
for each $i \in groups$ {
  $row = row + 1$
  for col = 1 to n
    if ($col \in group_i$)
      $H_{row,col} = 1$
    else
      $H_{row,col} = 0$
}
return (H)

5.4 Generating Self-Checking Circuit

Once the parity-check code has been selected, the next step of the synthesis method is to generate the actual circuit. The check bit equations are added to the function logic equations, and based on the parity-check code, the appropriate ROS for a PFS structure is determined. SCLO is then performed to produce a PFS circuit. Logic sharing between the function logic and check logic is considered as long as the PFS property is maintained. Therefore, even for a duplication code, SCLO may produce better results than doing logic optimization on the function logic and check logic separately.

For a complete self-checking circuit, a TSC checker needs to be added to the optimized PFS circuit. TSC parity checkers [Khakbaz 82] can be used to check each parity group, and then a TSC two-rail checker [Hughes 84] can be used to combine the error indication signals. If too few output codewords occur during normal operation to satisfy the self-testing requirement of the checkers, modifications such as those suggested in [Fujiwara 87] can be used. An example of a parity-check code is given in Table 4, and the block diagram for a complete self-checking circuit using this code is shown in Fig. 11. The check bits are c_1,c_2,c_3, and the normal circuit outputs are the Z_i's.

Table 4. Example: ***H*** Matrix for Parity-Check Code

Group	Z_1	Z_2	Z_3	Z_4	Z_5	Z_6	c_1	c_2	c_3
1	1	0	1	0	0	0	1	0	0
2	0	1	0	0	1	1	0	1	0
3	0	0	0	1	0	0	0	0	1

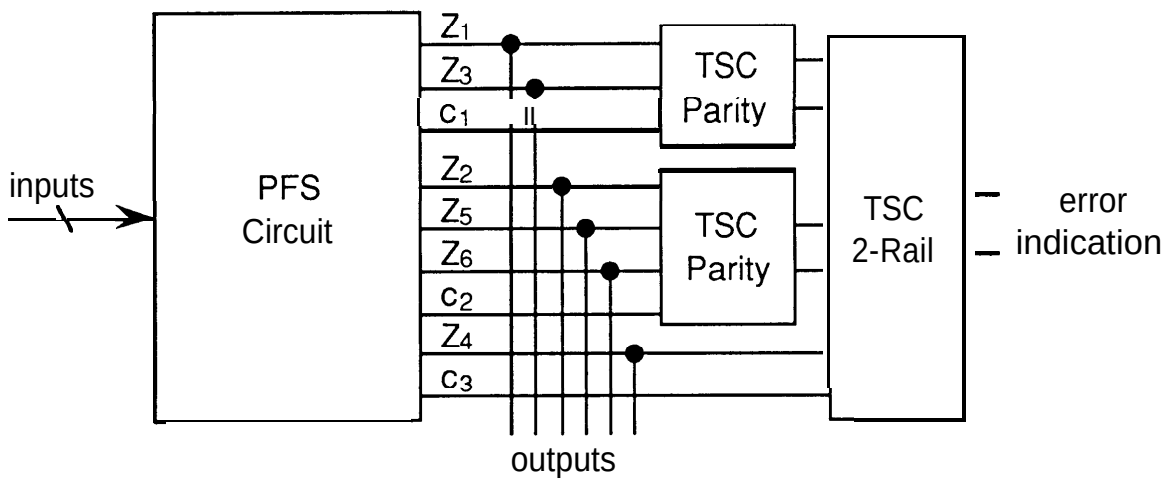


Figure 11. Block Diagram of Self-Checking Circuit Using Code in Table 4

5.5 Results for MCNC Benchmark Circuits

The synthesis method has been implemented by making modifications to SIS 1.1 (an updated version of MIS). The code selection algorithm was added, and the restructuring algorithms were extended to handle structural constraints so that SCLO could be performed. Self-checking circuits were generated for some of the MCNC combinational benchmark circuits [Yang 9 1] and then placed and routed using the TimberwolfSC 4.2c standard cell package [Sechen 86,871. The circuits were optimized using the script file script.boolean included with the SIS source code. Results are shown in Table 5. Literal counts are given in terms of factored form literals and layout areas are given in units of 1000 λ^2, where λ is the minimum size in a technology. Under the first major heading, information about each circuit is given: number of PI's, number of PO's, literal count after normal unconstrained logic optimization, and layout area for the optimized circuit (with no concurrent error detection). Under the second and third major headings, results for the duplication method and the synthesis method are given: number of check bits, literal count for the circuit, literal count for the checker, total literal count for the self-checking circuit, layout area for the self-checking circuit, and the percentage of area overhead required which is computed as shown below.

$$\% \text{ Area Overhead} = \frac{(\text{self - checking layout area}) - (\text{normal layout area})}{(\text{normal layout area})} \times 100$$

For the circuits where the number of check bits that are used in the parity-check code chosen by the synthesis method is equal to the number of PO's, the duplication code was selected. Where the number of check bits is equal to one, single-bit parity prediction was selected. For most circuits, something between duplication and single-bit parity prediction turned out to be best. In most cases, the synthesis method provides a significant reduction in the amount of overhead required compared to duplication.

Table 5. Results for MCNC Benchmark Circuits

Circuit					Duplication Method						Synthesis Method					
Name	PI	PO	opt lits	layout area	chk bits	ckt lits	chkr lits	total lits	layout area	ovrhd %	chk bits	c k t lits	chkr lits	total lits	layout area	ovrhd %
5xpl	7	10	121	334	10	242	72	314	989	196	3	217	44	261	797	139
alu2	10	6	441	1717	6	882	40	922	3602	110	5	796	36	832	3146	83
alu4	14	8	800	3298	8	1600	56	1656	6796	106	8	1600	56	1656	6796	106
b12	15	9	87	284	9	174	64	238	727	156	4	150	44	194	613	116
bw	5	28	217	752	28	434	216	650	2566	241	8	302	136	438	1066	42
clip	9	5	157	462	5	314	32	346	1131	145	3	310	24	334	1084	135
cmb	16	4	52	153	8	104	24	128	414	171	2	64	16	80	206	35
cu	14	11	53	181	11	106	80	186	537	197	1	83	40	123	356	97
f51ml	8	8	130	385	8	260	56	314	923	140	2	218	32	250	677	76
misexl	8	7	54	154	7	108	48	156	403	162	3	96	32	128	355	131
misex2	25	18	104	372	18	208	136	344	1166	213	2	202	72	278	935	151
pcle	19	9	69	229	9	138	64	202	659	188	3	156	40	196	523	128
rd73	7	3	81	236	3	162	16	178	511	116	3	162	16	178	511	116
sao2	10	4	149	431	4	298	24	322	1076	150	1	268	12	280	785	82
term1	34	10	179	617	10	358	72	430	1542	150	7	326	60	386	1211	96
ttt2	24	21	191	630	21	382	160	542	1977	220	9	374	112	486	1707	171
x2	10	7	51	144	7	102	48	150	433	201	2	79	28	107	279	94

6. SUMMARY AND CONCLUSIONS

A new synthesis technique called structure-constrained logic optimization (SCLO) was proposed for minimizing area under the constraint that the set of PO's that each node in the circuit can reach is contained in a specified reachable output set (ROS). It was shown that the ROS places a limit on the types of errors that a fault can cause. By using the appropriate ROS, SCLO can be used to optimize circuits for various concurrent error detection schemes such that a particular error coverage is achieved. A method for estimating the size of structure-constrained circuits without actually doing SCLO was described. Using this method, the design space for a concurrently checked circuit can be rapidly explored. This technique forms the basis of a new method for automated synthesis of self-checking circuit implementations for arbitrary combinational circuits. This new synthesis method consists of an algorithm that selects the best parity-check code for encoding the outputs of a given circuit, and then uses SCLO to produce a path fault secure circuit that is augmented with TSC parity and two-rail checkers to produce a self-checking circuit. The advantages of this new synthesis method are: 1) provides fully automated design, 2) explores more of the design-space than other methods, and 3) uses simple checkers. Results were shown that suggest that this method can significantly reduce the area overhead required for self-checking circuits.

ACKNOWLEDGMENTS

The authors would like to thank LaNae Avra, Prof. Xingning Xu, and Dr. Nirmal Saxena for their useful comments and suggestions. This work was supported in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and administered through the Office of Naval Research under Contract No. N00014-92-J-1782, and by the National Science Foundation under Grant No. MIP-9 107760.

REFERENCES

[Anderson 71] Anderson, D.A., "Design of Self-Checking Digital Networks Using Coding Techniques, " *Tech. Report R-527,* Coordinated Science Laboratory, University of Illinois, Urbana, IL, 1971.

[Armstrong 66] Armstrong, D.B., "On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Networks," *IEEE Transactions on Electronic Computers ,* Vol. EC-15, Feb. 1966, pp. 66-73.

[Brayton 87a] Brayton, R.K., R. Rudell, A. Sangiovanni-Vincentelli, A.R. Wang, "MIS: A Multiple-Level Logic Optimization System," *IEEE Transactions on Computer-Aided Design,* Vol. 6, Nov. 1987, pp. 1062-1081.

[Brayton 87b] Brayton, R.K., R. Rudell, A. Sangiovanni-Vincentelli, A.R. Wang, "Multi-Level Logic Optimization and The Rectangular Covering Problem," *Proc. of Int. Conference on Computer-Aided Design (ICCAD), 1987,* pp. 66-69.

[Brayton 90] Brayton, R.K., G.D. Hachtel, and A.L. Sangiovanni-Vincentelli, "Multilevel Logic Synthesis," *Proceedings of the IEEE,* Vol. 78, No. 2, Feb. 90, pp. 264-300.

[Bron 71] Bron, C., and J. Kerbosch, "Finding all cliques of an undirected graph," *Collected Algorithms from CACM* (Algorithm 457), 1971.

[De 92] De, K.D., C. Wu, and P. Banerjee, "Reliability Driven Logic Synthesis of Multilevel Circuits", *Proc. of International Symposium on Circuits and Systems, 1992,* pp. 1105-1108.

[Detjens 87] Detjens, E., G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Technology Mapping in *MIS," Proc. of Int. Conference on Computer-Aided Design (ICCAD),* 1987, pp. 116-119.

[Fujiwara 87] Fujiwara, E., and K. Matsuoka, "A Self-Checking Generalized Prediction Checker and Its Use for Built-In Testing," *IEEE Transactions on Computers ,* Vol. C-36, No. 1, Jan. 1987, pp. 86-93.

[Gupta 92] Gupta, S.K., and D.K. Pradhan, "Can Concurrent Checkers Help BIST?," *Proc. of International Test Conference, 1992,* pp. 140-150.

[Hughes 84] Hughes, J.L.A., E.J. McCluskey, and D.J. Lu, "Design of Totally Self-Checking Comparitors with an Arbitrary Number of Inputs," *IEEE Transactions on Computers ,* Vol. C-33, No. 6, Jun. 1984, pp. 546-550.

[Jha 91] Jha, N.K., and S. Wang, "Design and Synthesis of Self-Checking VLSI Circuits and Systems," *Proc. of International Conference on Computer Design (ICCD), 1991,* pp. 578-581.

[Keutzer 87] Keutzer, K., "Dagon: Technology Binding and Local Optimization by DAG Matching," *Proc. of the 24th Design Automation Conference, 1987,* pp. 341-347.

[Khakbaz 82] Khakbaz, J., "Self-Testing Embedded Parity Trees", *Proc. of FTCS-12,* 1982, pp. 109-116.

[Khodadad-Mostashiry 79] Khodadad-Mostashiry B., "Parity Prediction in Combinational Circuits," *Proc. of FTCS-9,* 1979, pp. 185- 188.

[Khodadad-Mostashiry 80] Khodadad-Mostashiry B., "Break Faults in Circuits with Parity Prediction," *Tech. Note* No. *183*, CRC, Stanford University, Stanford, CA, Dec. 1980.

[Ko 78] Ko, D.C., and M.A. Breuer, "Self-Checking of Multi-Output Combinational Circuits Using Extended-Parity Technique," *Journal of Design Automation & Fault-Tolerant Computing,* Vol. 2, Jan. 1978, pp. 29-62.

[Lala 88] Lala, J.H., and L. Alger, "Hardware and Software Fault Tolerance: A Unified Architectural Approach," *Proc. of FTCS-18,* 1988, pp. 240-245.

[Nanya 89] Nanya, T., and M. Uchida, "A Strongly Fault-Secure and Stronly Code-Disjoint Realization of Combinational Circuits," *Proc. FTCS,* 1989, pp. 390-397.

[Nicolaidis 89] Nicolaidis, M., "Self-Exercising Checkers for Unified Built-in Self-Test (UBIST)," *IEEE Transactions on Computer-Aided Design,* Vol. 9, No. 3, Mar. 1989, pp. 203-218.

[Rudell 89] Rudell, R., "Logic Synthesis for VLSI Design," Ph. D. Thesis, University of California, Berkeley, 1989.

. [Russell 71] Russell J.D., and C.R. Kime, "Structural Factors in the Fault Diagnosis of Combinational Networks," *IEEE Transactions on Computers ,* Vol. C-20, No. 11, Nov. 197 1, pp. 1276-1285.

[Pradhan 86] Pradham, D.K., *Fault Tolerant Computing: Theory and Techniques,* Vol. 1, Englewood Cliffs, NJ: Prentice-Hall, 1986.

[Sechen 86] Sechen C., and A. Sangiovanni-Vincentelli, "TimberWolf3.2: A New Standard Cell Placement and Global Routing Package", *Proc. of the 30th Design Automation Conference,* 1986, pp. 432-439.

[Sechen 87] Sechen C., K. Lee, B. Swartz, D. Chen, and M. Lee, "The TimberWolfSC Standard Cell Placement and Global Routing Package, User's Guide for Version 4.2c", Yale University, Oct. 1987.

[Sedmak 79] Sedmak, R.M., "Design for Self-Verification: An Approach for Dealing with Testability Problems in VLSI-Based Designs", *Proc. International Test Conference, 1979,* pp. 112-120.

[Smith 77] Smith, J.E., and G. Metze, "The Design of Totally Self-Checking Combinational Circuits," *Proc. FTCS,* 1977, pp. 130-134.

[Smith 78] Smith, J.E., and G. Metze, "Strongly Fault Secure Logic Networks," *IEEE Transactions on Computers ,* Vol. C-27, No. 6, Jun. 1978, pp. 491-499.

[Smith 83] Smith, J.E., and P. Lam, "A Theory of Totally Self-Checking System Design," *IEEE Transactions on Computers* , Vol. C-32, No. 9, Sep. 1983, pp. 831-843.

[Unger 69] Unger, S.H., "An Algorithm for Finding the Reachability Matrix of a Directed Linear Graph," *IEEE Transactions on Circuit Theory,* Vol. CT- 16, Feb. 1969, pp. 130- 132.

[Yang 91] Yang, S., "Logic Synthesis and Optimization Benchmarks User Guide, Version 3.0", *1991 MCNC International Workshop on Logic Synthesis.*