[16] Layout synthesis benchmark set, Microelectronics Center of North Carolina, Research Triangle Park, NC, May 1990.
[17] ACEO, Inc., *ACEO Migration and Partitioning Reference Manual*. ACEO, 1995.
[18] Aptix, Inc., *The Aptix FPID Data Book*. Aptix, Feb. 1993.
[19] Xilinx, Inc., *The Programmable Logic Data Book*. San Jose, CA: Xilinx, 1994.

## Logic Synthesis of Multilevel Circuits with Concurrent Error Detection

Nur A. Touba and Edward J. McCluskey

***Abstract*—This paper presents a procedure for synthesizing multilevel circuits with concurrent error detection. All errors caused by single stuck-at faults are detected using a parity-check code. The synthesis procedure (implemented in Stanford CRC's TOPS synthesis system) fully automates the design process, and reduces the cost of concurrent error detection compared with previous methods. An algorithm for selecting a good parity-check code for encoding the circuit outputs is described. Once the code has been selected, a new procedure called *structure-constrained logic optimization* is used to minimize the area of the circuit as much as possible while still using a circuit structure that ensures that single stuck-at faults cannot produce undetected errors. It is proven that the resulting implementation is path fault secure, and when augmented by a checker, forms a self-checking circuit. The actual layout areas required for self-checking implementations of benchmark circuits generated with the techniques described in this paper are compared with implementations using Berger codes, single-bit parity, and duplicate-and-compare. Results indicate that the self-checking multilevel circuits generated with the procedure described here are significantly more economical.**

### I. Introduction

Concurrent error detection is an important technique in the design of systems in which dependability and data integrity are important. Concurrent error detection circuitry has the ability to detect both transient and permanent faults, as well as to enhance off-line testability and reduce BIST overhead [1]–[3].

One general approach for concurrent error detection is to encode the outputs of a circuit with an error-detecting code, and to have a checker that monitors the outputs and gives an error indication if a noncodeword occurs. A *systematic code* is a code in which codewords are constructed by appending check bits to the normal output bits. Using a systematic code for concurrent error detection has the advantage that no decoding is needed to get the normal output bits. Fig. 1 shows the general structure of a circuit checked with a systematic code. There are three parts: function logic, check symbol generator, and checker. The function logic generates the normal outputs, the check symbol generator generates the check bits, and the checker determines if they form a codeword. Two types of systematic codes that are used for concurrent error detection are Berger codes and parity-check codes [4].

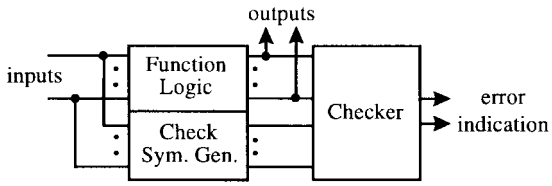


Fig. 1. Concurrent error detection using a systematic code.

While methods exist for designing PLA's and simple functional units (e.g., adders, multipliers, etc.) with concurrent error detection [4], the conventional approach for designing arbitrary multilevel circuits with concurrent error detection has been to use duplication. The circuit is simply duplicated, and the outputs are compared using a two-rail checker (equality checker). While this provides very high error-detection capability, it requires a large area overhead. Recently, research has been done on using automated logic synthesis techniques (such as those used in MIS [5]) to design multilevel circuits with concurrent error detection requiring less area overhead than duplication while still being able to detect all errors due to *internal single stuck-at faults* [6]–[8]. Internal single stuck-at faults are all single stuck-at faults, except those at the primary inputs (PI's). Note that for any concurrent error-detection scheme (including duplication), detection of stuck-at faults at the PI's cannot be guaranteed unless encoded inputs are used. However, if the inputs to the circuit are outputs of another concurrently checked logic block, then the only undetectable PI faults are break faults after the checker [9].

Jha and Wang [6] proposed a synthesis method 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 is *inverter free*, i.e., it has inverters only at the PI's. The primary outputs (PO's) are then encoded with a Berger code, which is a unidirectional error-detecting code. Since the inverters are only at the PI's, any error caused by an internal single stuck-at fault will produce a unidirectional error at the PO's, and therefore is guaranteed to be detected.

De *et al.* [7] have proposed two schemes for generating multilevel circuits with concurrent error detection. The first scheme uses a Berger code. It fully automates the synthesis method proposed in [6] by automatically adding the logic equations for the Berger check bits and checker, and then using a constrained technology mapping procedure that maintains the inverter-free property during technology mapping. The second scheme uses a parity-check code. 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 $\boldsymbol{H}$ [4]. Fig. 2 shows the parity check matrices $\boldsymbol{H}$ 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. The synthesis method proposed in [7] partitions the outputs to form

Manuscript received December 19, 1995; revised August 12, 1996 and November 22, 1997. This work was supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate, and administered through the Department of the Navy, Office of Naval Research under Grant N00014-92-J-1782, by the National Science Foundation under Grant MIP-9107760, and by the Advanced Research Projects Agency under Prime Contract DABT63-94-C-0045. This paper was recommended by Associate Editor A. Saldanha.

N. A. Touba was with the Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305 USA. He is now with the Computer Engineering Research Center, Department of Electrical and Computer Engineering, University of Texas, Austin, TX 78712-1084 USA.

E. J. McCluskey is with the Center for Reliable Computing, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305 USA.

Publisher Item Identifier S 0278-0070(97)07563-5.

| *Group* | $Z_1$ | $Z_2$ | $Z_3$ | $c_1$ |
|---|---|---|---|---|
| *1* | 1 | 1 | 1 | 1 |

(a)

| *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 |

(b)

Fig. 2. $\boldsymbol{H}$ matrix for circuit with three outputs: (a) single-bit parity code and (b) duplication code.

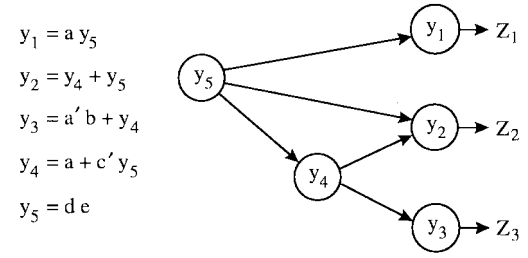


Fig. 3. Example of a Boolean network.

logic blocks in such a way that logic sharing within each block is maximized, but no logic sharing between blocks is allowed. If $k$ is the number of outputs in the logic block with the most outputs, then $k$ parity groups are used. Outputs are assigned to the $k$ parity groups in such a way that each parity group contains no more than one output from each logic block. The parity check functions for each of the $k$ parity groups are computed and form another logic block. Each of these logic blocks is then synthesized separately using MIS. Since no logic sharing exists between the logic blocks, an error caused by an internal stuck-at fault can only affect the outputs of a single logic block. Since each of the outputs in a single logic block is in a different parity group, and therefore checked by a different parity bit, the error cannot be masked, and therefore will be detected. In [7], the actual layout areas using the Berger code scheme, the parity-check code scheme, and duplication are given for some benchmark circuits. For almost all of the circuits, the parity-check code scheme requires the least area overhead.

This paper describes a new approach for generating multilevel circuits with concurrent error detection based on parity-check codes. A logic synthesis procedure is introduced which provides a significant improvement in the quality of the result. The two basic steps in generating a circuit that uses a parity-check code for concurrent error detection are: 1) determining which parity check code to use, and 2) performing logic optimization under the constraint that the structure of the circuit is such that errors are not masked. Major improvements are presented in this paper for both of these steps. In [7], the selection of the parity-check code is not fully automated. The number of output partitions is a user-supplied parameter. Results are shown in [7] that indicate that the final circuit area strongly depends on the number of output partitions used. Outputs are assigned to each partition using a cost function that is based only on logic sharing in an optimized multilevel implementation. In this paper, a fully automated parity-check code selection algorithm is presented. It is a greedy algorithm that tries to find the optimal code for minimizing the overall area of the circuit. It uses a cost function that considers the area of the parity check functions and area of the checker in addition to logic sharing.

Once the code is selected, the next step is to perform logic optimization under structural constraints. In [7], circuit structure is constrained by optimizing each logic block separately. This constraint is overly restrictive. For example, if output $X$ is in logic block 1 and output $Y$ is in logic block 2, then even though $X$ and $Y$ are in different logic blocks, they may be in different parity groups, and hence can still share logic without compromising the detectability of faults in the circuit. In this paper, a new logic optimization technique, called *structure-constrained logic optimization* (SCLO), is presented. By considering structural constraints when factoring, SCLO optimizes the area of the circuit as much as possible under the structural constraints. Using SCLO, the whole circuit can be synthesized together, allowing additional logic-sharing opportunities to be explored, thereby producing a better result. Some preliminary results have been published in [8].

The paper is organized as follows. In Section II, some definitions and terminology are explained. In Section III, the parity-check code selection algorithm is presented, and its cost function, heuristics, and time complexity are explained. In Section IV, the SCLO technique is described, and implementation of its factoring procedures is discussed. Section V shows how to use these techniques to generate a self-checking circuit. In Section VI, results are presented showing how much area overhead is required for self-checking implementations of some of the MCNC benchmark circuits. The results for the synthesis procedure described in this paper are compared with previous techniques. Section VII gives a summary and conclusions.

## II. Definitions and Terminology

A multilevel circuit can be represented by a *Boolean network* [10] which 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 PI and PO 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. 3 is an example of a Boolean network. The PI's are $a$–$e$, and the PO's are $Z_1$–$Z_3$. $y_1$–$y_5$ are intermediate nodes. In a Boolean network, if a directed path exists from node $i$ to node $j$, then node $i$ is a *transitive fan-in* of node $j$, and node $j$ is a *transitive fan-out* of node $i$. For example, in Fig. 3, $y_3$, $y_4$, and $y_5$ are transitive fan-in's of $Z_3$, and $Z_1$ is a transitive fan-out of $y_1$ and $y_5$, but not of $y_4$.

*Definition 1:* A circuit is *fault secure* if and only if, for every fault in a specified fault class, the circuit never produces an incorrect codeword output for any codeword input.

*Definition 2:* A circuit is *self-testing* if and only if, for every fault in a specified fault class, the circuit produces a noncodeword output for at least one codeword input.

*Definition 3:* A circuit is *path-fault secure (PFS)* if and only if, for every fault in a specified fault class, error propagation down any set of structural paths from the fault site to the outputs will never produce an incorrect codeword output [11].

The *totally self-checking (TSC) goal* is to detect the first error that occurs due to any fault in a specified fault class [11]. 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; such a circuit is said to be *self-checking* [12]. Smith and Metze [11] introduced the concept of PFS circuits, and showed it to be a subclass of fault-secure circuits. Whereas the fault-secure property depends on the functional paths in the circuit that are sensitized by codeword inputs, the PFS property depends on the structural paths, and therefore is independent of the input set that is applied. It was shown in [11] that PFS circuits that are checked by a TSC checker achieve the TSC goal regardless of which 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.

It will be proven in Section V that the techniques described in this paper generate self-checking circuits that are PFS.

## III. Selecting a Parity-Check Code

Given an arbitrary combinational circuit for which concurrent error detection circuitry is to be added, the first step is to select a parity-check code for encoding the outputs of the circuit. If a requirement is that the circuit be PFS for all internal single stuck-at faults, then a tradeoff exists between the number of parity groups (i.e., check bits) and the constraints on logic sharing between outputs. Logic cannot be shared between two outputs in the same parity group because then, if a fault occurred in the shared logic, the effects of the fault could propagate to both outputs, causing a 2-bit error which would not be detected by the parity checker. So, the more parity groups there are, the more logic sharing is possible; however, more parity groups require more parity predict logic to generate the check bits. Consider the two extreme cases: duplication and single-bit parity prediction. In duplication, each output is in its own parity group, so there are no constraints on logic sharing; however, the parity predict logic is large. In single-bit parity prediction, there is only one parity group, so only one check bit needs to be generated; however, no logic sharing is possible.

The goal is to select the code that will require the least amount of area to implement. The area of the circuit is equal to the sum of the areas of the function logic, parity predict logic, and checker. The area required by the function logic depends on how much logic sharing is possible. The area required by the parity predict logic depends on the size of the parity functions that must be implemented for each check bit. The area required by the checker depends on how many parity groups there are. All of these factors should be considered when selecting the parity-check code. The cost function proposed here incorporates all three factors, whereas the cost function used in [7] considers only the area required by the function logic.

### A. Cost Function

The cost function is used to compare the area required by two different codes. The exact cost function would require actually synthesizing the circuit for each code and measuring the area; the computation time required for this is obviously not practical. Therefore, some estimation techniques are required. First, consider the area required by the function logic. The least area required can be found by performing normal logic optimization with no restrictions on logic sharing to produce an unconstrained implementation. Let *lits_shared*$(x, y)$ be the factored form literal count of the logic shared between outputs $x$ and $y$ in the unconstrained implementation. *lits_shared*$(x, y)$ can easily be computed by summing the literal counts for each node that is a transitive fan-in of both $x$ and $y$, i.e., that has a structural path to both $x$ and $y$. If a code has both $x$ and $y$ in the same parity group, then no logic can be shared between these outputs when implementing the function logic. The extra area required by the function logic because of this constraint on logic sharing between $x$ and $y$ can be approximated by *lits_shared*$(x, y)$ This value is only an approximation of the area complexity because it may be possible to restructure the circuit to exploit other opportunities for logic sharing that would compensate for some of the loss. Using this approximation, the total extra area required by the function logic for a code can be estimated by summing *lits_shared*$(x, y)$ for each pair of outputs $x$ and $y$ that are members of the same parity group in the code.

Next, consider the area required by the parity predict logic for a particular code. Each check bit is a parity function equal to the modulo-2 sum of the logic equations for the outputs in its parity group. Finding the area required by the parity predict logic for each code involves generating the check-bit equations and doing multilevel logic optimization; this is far too time consuming to do for each code. It is difficult to even estimate this area. A more tractable task is to compare the relative area requirements for two similar codes. If one code can be generated from another code by merging two parity groups, then the difference in the parity predict logic area 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 area of the parity predict logic can be approximated by comparing the factored form literal count of $c_1$ plus $c_2$ with that of $(c_1 \oplus c_2)$ after node minimization

$$\begin{aligned}&\text{parity predict logic area } (A) - \text{parity predict logic area } (B)\\ &\quad \approx \text{lits}(c_1) + \text{lits}(c_2) - \text{lits}(c_1 \oplus c_2).\end{aligned}$$

This approximation neglects the effect of global restructuring operations that would be performed in multilevel logic optimization.

Last, consider the area required by the checker for a particular code. The literal count for the checker can be easily calculated because it depends only on the number of inputs to the checker. Both parity checker trees and two-rail checker trees [4] require [4*(number of checker inputs* $-$ 2)] literals. For example, a four-input TSC parity checker consists of two two-input exclusive-or gates, and thus requires eight literals. A four-input two-rail checker also requires eight literals. Although the checker structure for each parity-check code will be composed of differing numbers of parity checker components and two-rail checker components, the total number of literals required is always equal to

$$\begin{aligned}&[4(number\ of\ function\ outputs\\ &\qquad + number\ of\ parity\ groups - 2)].\end{aligned}$$

The proposed algorithm uses these area estimation techniques to predict which parity-check code will require the least amount of area to implement. The area requirements for two codes are quickly compared by estimating the literals saved by using one code instead of another.

Let *area_reduce*$(A, B)$ be the estimated number of literals saved by using code $B$ instead of code $A$ where code $B$ is formed by merging two parity groups, corresponding to check bits $c_1$ and $c_2$, in code $A$. *area_reduce*$(A, B)$ is computed as follows:

$$\begin{aligned}&area\_reduce(A, B)\\ &\quad = parity\_reduce(A, B) + checker\_reduce(A, B)\\ &\qquad - shared\_logic\_reduce(A, B)\\ &parity\text{-}reduce(A, B) = \text{lits}(c_1) + \text{lits}(c_2) - \text{lits}(c_1 \oplus c_2)\\ &checker\_reduce(A, B) = 4\\ &shared\_logic\_reduce(A, B)\\ &\quad = \text{literals shared between all pairs of outputs } x \text{ and } y\\ &\qquad \text{where } x \text{ is checked by } c_1 \text{ and } y \text{ is checked by } c_2.\end{aligned}$$

All three components of the circuit implementation are considered in the value of *cost_reduce*$(A, B)$. *parity_reduce*$(A, B)$ estimates how many literals will be saved by combining check bits $c_1$ and $c_2$ in the parity predict logic. *checker_reduce*$(A, B)$ is the number of literals that will be saved since the checker required for code $B$ is smaller than the checker for code $A$ because code $B$ has one less parity group.

TABLE I
ALGORITHM FOR SELECTING PARITY-CHECK CODE

```
SELECT_PARITY_CHECK_CODE (function_logic):
    opt_funct_logic = unconstrained logic optimization of function_logic
    best_code = duplication code
    /* one parity group for each output */
    for i = 1 to NUM_PO(function_logic)
        parity_group_i = { i }
    repeat {
        for each code_{i,j} formed by combining parity groups i and j in best_code
            Compute AREA_REDUCE(best_code, code_{i,j})
        code_{p,q} = code that maximizes AREA_REDUCE
        if ( AREA_REDUCE(best_code, code_{p,q}) > 0 ) {
            best_code = code_{p,q}
            /* combine parity groups p and q */
            parity_group_p = parity_group_p ∪ parity_group_q
            delete parity_group_q
            improvement = true
        }
        else
            improvement = false
    } until ( ! improvement || best_code == single-bit parity code )
    return ( best_code )

AREA_REDUCE (best_code, code_{i,j}):
    shared_{i,j} = LITS_SHARED(opt_funct_logic, parity_group_i, parity_group_j)
    c_{i,j} = SIMPLIFY( XOR(c_i, c_j) )
    parity_reduce_{i,j} = LITS(c_i) + LITS(c_j) - LITS(c_{i,j})
    check_reduce_{i,j} = 4
    return ( parity_reduce_{i,j} + check_reduce_{i,j} - shared_{i,j} )
```

*shared_logic_reduce*$(A, B)$ estimates how many additional literals will be required in the function logic due to the added constraints on logic sharing imposed by using code $B$. By considering all of these factors, *area_reduce*$(A, B)$ predicts how much better (or worse if it is a negative value) the overall circuit area will be using code $B$ instead of code $A$ for concurrent error detection.

### B. Algorithm

The number of possible parity-check codes for a circuit with $n$ outputs is equal to the number of partitions of a set of $n$ objects. This number is exponential in $n$, so heuristics are needed in searching for the minimal area code. A greedy algorithm is given in Table I which uses the heuristic of pairwise combining of parity groups in searching for a minimal area code. It begins with the duplication code in which each output is in its own parity group. It then computes the reduction in the cost function for all codes that can be formed by combining two of the parity groups, and chooses the code that offers the largest cost reduction. This process continues until no further reduction in the cost function is possible through combining parity groups.

If $n$ is the number of outputs in the function logic, then the initial duplication code has $n$ parity groups. The algorithm will consider all possible codes with $n-1$ parity groups, so if the optimal code has $n$ or $n-1$ parity groups, it is guaranteed to be found. The algorithm only considers a subset of the codes with fewer than $n-1$ parity groups. This subset is determined by the heuristic of greedily combining parity groups in a pairwise manner. In the worst case, the algorithm will execute the COST_REDUCE function $\Sigma_{i=2}^{n} C_2^i$ times. Thus, a solution is obtained using only $O(n2\log_2 n)$ operations, where each

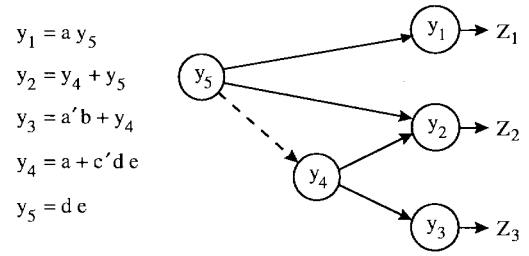


Fig. 4. Resubstitution example.

operation requires computing the exclusive OR of two nodes and simplifying it.

## IV. STRUCTURE-CONSTRAINED LOGIC OPTIMIZATION (SCLO)

Once the parity-check code has been selected, the next step is to optimize the circuit under the constraints on logic sharing that are needed to ensure that no internal single stuck-at fault will cause an undetectable error. A new synthesis procedure that does this, called SCLO, is described here. Multilevel logic optimization improves circuit area by using operations that restructure and minimize the logic represented by a Boolean network. In SCLO, restrictions are placed on the restructuring operations to ensure that the resulting circuit will satisfy the structural constraints.

Given the initial multilevel logic equations and the parity-check code, SCLO optimizes the logic under the constraint that a non-PI node cannot be a transitive fan-in of more than one PO in a parity group. SCLO is accomplished by starting with an initial Boolean network that satisfies the constraints, and then constraining the restructuring operations so that they never cause nodes to violate the constraints. The two restructuring operations that can cause a node to violate the constraints are resubstitution and extraction [10].

### A. Constraints on Resubstitution

*Resubstitution* is an operation where some node $a$, which is divisible 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 $b$ (or some node that is a transitive fan-in of node $b$ if Boolean resubstitution [10] is considered) becomes a transitive fan-in of more than one PO in a parity group, thus violating the constraints. Therefore, in SCLO, resubstitution can be performed between two nodes only if the resulting arc does not violate the constraints.

Fig. 4 shows a Boolean network in which node $y_4$ is divisible by node $y_5$. The dashed arrow indicates the new arc that will be added if resubstitution of $y_5$ into $y_4$ is performed. To check if this resubstitution will violate the constraints, the new set of PO's that will become transitive fan-outs of $y_5$ needs to be determined. If the arc is added, any PO that is a transitive fan-out of $y_4$ will become a transitive fan-out of $y_5$, so the new set of PO's that will be transitive fan-outs of $y_5$ is $\{Z_1, Z_2, Z_3\}$. If each PO in this new set is in a different parity group, then the resubstitution can be done; otherwise, it cannot be done.

Implementing constrained resubstitution is easy. Various filters are generally used to reduce the number of node pairs for which resubstitution is attempted [5]. So, this constraint can simply be added as an additional filter. Note that some procedures for node

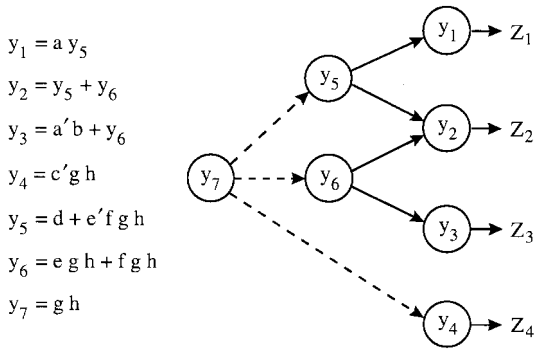


Fig. 5. Extraction example.

minimization may implicitly perform resubstitution, so the additional filter must be used in those node minimization procedures as well.

### B. Constraints on Extraction

*Extraction* is an operation in which an intermediate node is created by factoring out a common subexpression from a set of nodes $S$. The intermediate node is then used as an input to each node in $S$, thereby reducing the overall literal count. The intermediate node will have an arc to each node in $S$, and hence will be a transitive fan-in of every PO that is a transitive fan-out of any node $S$. Therefore, common subexpressions can only be extracted from a set of nodes if all of the PO's that are transitive fan-outs of the set of nodes are in different parity groups.

Fig. 5 shows a Boolean network in which node $y_7$ can be extracted from the set of nodes $\{y_4, y_5, y_6\}$. The dashed arrows indicate the new arcs that will be added. To check if this extraction will violate the constraints, the new set of PO's that will become transitive fan-outs of $y_7$ needs to be determined. If $y_7$ is extracted from the full set of nodes, then its set of transitive fan-outs will be $\{Z_1, Z_2, Z_3, Z_4\}$. If $Z_2$ and $Z_4$ are in the same parity group, then this extraction will violate the constraints because $y_7$ will fan out to both $Z_2$ and $Z_4$. However, note that if $y_7$ is extracted only from the set of nodes $\{y_5, y_6\}$, then its set of transitive fan-outs will be $\{Z_1, Z_2, Z_3\}$, and thus it will not fan out to both $Z_2$ and $Z_4$, and therefore will not violate the constraints.

In order to implement constrained extraction, the process of selecting common subexpressions to extract needs to be modified. Two methods that are used for selecting common subexpressions to extract are rectangle covering [13] and the concurrent decomposition procedure in [14]. In both cases, subexpressions are identified between nodes, and are assigned a value based on the number of literals that will be reduced if it is extracted. In SCLO, subexpressions cannot always be extracted from the full set of possible nodes due to the structure constraints. Thus, the value assigned to each subexpression must be adjusted according to the maximum number of literals that can be reduced without violating the constraints. For example, in Fig. 5, if $Z_2$ and $Z_4$ are in the same parity group, then the subexpression $gh$ cannot be extracted from the full set of possible nodes $\{y_4, y_5, y_6\}$, but it can be extracted from the set of nodes $\{y_5, y_6\}$. So the value assigned to the subexpression $gh$ must be adjusted to reflect the true number of literals that will be reduced if it is extracted under the structural constraints. In SCLO, after the subexpressions are identified, an additional step is required in which the value assigned to each subexpression is adjusted based on the constraints. The adjusted value is computed by identifying the subset of nodes that the subexpression cannot be extracted from due to the constraints, and subtracting out the literal reduction for those nodes. Subexpressions are then selected based on the adjusted values, and are extracted from the maximal set of nodes that the constraints allow. Details of one particular procedure for extraction in SCLO using rectangle covering can be found in [15].

| *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 |

Fig. 6. Example: $\boldsymbol{H}$ matrix for parity-check code.

### C. Technology Mapping

After the Boolean network is optimized, it must be mapped to a set of library elements; this process is called *technology mapping*. In order to ensure that the resulting mapped circuit satisfies the structural constraints, a technology mapping procedure that follows the structure of the Boolean network, such as tree mapping [16], [17], must be used to map the Boolean network to single-output library cells.

## V. Generating a Self-Checking Circuit

This section proves that the SCLO procedure guarantees the PFS property, and describes how to augment the circuit with a TSC checker to make it self-checking.

*Theorem 1:* After SCLO has been performed, the resulting circuit will be PFS for all internal single stuck-at faults.

*Proof:* In order for an incorrect codeword output to occur in a parity-check code, some parity group must have an even number of errors on its outputs. SCLO constrains restructuring operations so that non-PI nodes in the resulting Boolean network have a path to no more than one output in any parity group. Any internal single stuck-at fault can change the logic function of only one cell. Since the technology mapping procedure follows the structure of the Boolean network and each library cell has a single output, the effects of a fault in a cell can propagate to no more than one output in any parity group using any set of possible structural paths. Therefore, no internal single stuck-at fault can cause the circuit to produce an incorrect codeword output.

For a self-checking circuit, a TSC checker needs to be added to the PFS circuit generated by SCLO. TSC parity checkers [18] can be used to check each parity group, and then a TSC two-rail checker [19] 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 [20] can be used. An example of a parity-check code is given in Fig. 6, and the block diagram for a self-checking circuit using this code is shown in Fig. 7. The check bits are $c_1, c_2, c_3$, and the normal circuit outputs are the $Z_i$'s.

## VI. Results

The synthesis method proposed in this paper 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. Using this implementation, self-checking circuits were generated for some of the MCNC combinational benchmark circuits.

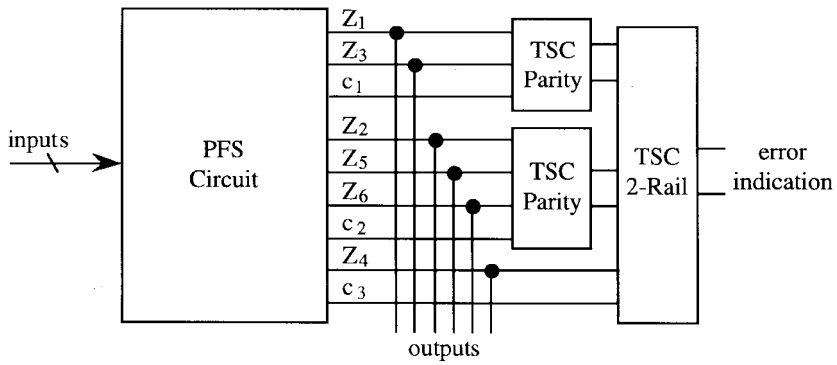


Fig. 7. Block diagram of self-checking circuit using code in Fig. 6.

TABLE II
LITERAL COUNT COMPARISON FOR CODES WITH DIFFERENT NUMBERS OF PARITY BITS

| b12 | |
|---|---|
| Parity Bits | Literal Count |
| 9 (dup) | 238 |
| 6 | 205 |
| **4 (sel)** | **194** |
| 2 | 219 |
| 1 | 206 |

| misex1 | |
|---|---|
| Parity Bits | Literal Count |
| 7 (dup) | 156 |
| 5 | 149 |
| **3 (sel)** | **128** |
| 2 | 123 |
| 1 | 134 |

| misex2 | |
|---|---|
| Parity Bits | Literal Count |
| 18 (dup) | 344 |
| 11 | 325 |
| 4 | 325 |
| **2 (sel)** | **278** |
| 1 | 291 |

| pcle | |
|---|---|
| Parity Bits | Literal Count |
| 9 (dup) | 202 |
| 6 | 199 |
| **3 (sel)** | **196** |
| 2 | 209 |
| 1 | 219 |

Table II shows results comparing the factored form literal counts for self-checking circuits based on codes with different numbers of parity bits. The literal counts include the function logic, parity predict logic, and the checker. The first row for each circuit corresponds to the duplication code (the number of parity bits is equal to the number of outputs in the circuit). The last row for each circuit corresponds to the single-bit parity code (there is only one parity bit). The code that was selected by the code selection algorithm described in this paper is shown in bold. As can be seen, the code selection algorithm selected the best code for all of the circuits except *misex1*.

Table III shows factored form literal count data for three implementations: the circuit with no concurrent error detection, the circuit with duplication, and the circuit generated by the synthesis procedure described in this paper. Under the first major heading, information about each circuit is given: number of primary inputs, number of primary outputs, and literal count after normal unconstrained logic optimization. Under the second and third major headings, results for duplication and for the synthesis procedure described in this paper are given: number of parity bits, literal count for the circuit, literal count for the checker, and total literal count for the self-checking circuit. For duplication, the number of parity bits is equal to the number of outputs and the literal count for the circuit is twice the normal literal count since there are two copies of the circuit. For the synthesis procedure described in this paper, the number of parity bits in the selected code ranged from single-bit parity (e.g., $cu$) up to duplication (e.g., $alu4$). In most cases, something between single-bit parity and duplication turned out to be best.

TABLE III
LITERAL COUNT COMPARISON FOR DUPLICATION VERSUS CODE SELECTED BY PROPOSED PROCEDURE

| Circuit | | | | Duplication | | | | Proposed Procedure | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | PI | PO | Opt. Lits | Par. Bits | Circuit Lits | Chkr Lits | Total Lits | Par. Bits | Circuit Lits | Chkr Lits | Total Lits |
| apla | 10 | 12 | 312 | 12 | 624 | 88 | 712 | 3 | 316 | 52 | 368 |
| br1 | 12 | 8 | 196 | 8 | 392 | 56 | 448 | 2 | 243 | 32 | 275 |
| bw | 5 | 28 | 178 | 28 | 356 | 216 | 572 | 8 | 302 | 136 | 438 |
| chkn | 29 | 7 | 398 | 7 | 796 | 48 | 844 | 3 | 706 | 32 | 738 |
| dc1 | 4 | 7 | 45 | 7 | 90 | 48 | 138 | 3 | 62 | 32 | 94 |
| dc2 | 8 | 7 | 162 | 7 | 324 | 48 | 372 | 2 | 188 | 28 | 216 |
| exp | 8 | 18 | 435 | 18 | 870 | 136 | 1006 | 5 | 526 | 84 | 610 |
| luc | 8 | 27 | 211 | 27 | 422 | 208 | 630 | 6 | 335 | 24 | 359 |
| p82 | 5 | 14 | 127 | 14 | 254 | 104 | 258 | 3 | 162 | 60 | 222 |
| signet | 39 | 8 | 335 | 8 | 670 | 56 | 726 | 5 | 310 | 44 | 354 |
| wim | 4 | 7 | 60 | 7 | 120 | 48 | 168 | 2 | 61 | 28 | 89 |
| 5xpl | 7 | 10 | 134 | 10 | 268 | 72 | 340 | 3 | 217 | 44 | 261 |
| alu4 | 14 | 8 | 800 | 8 | 1600 | 56 | 1656 | 8 | 1600 | 56 | 1656 |
| b12 | 15 | 9 | 87 | 9 | 174 | 64 | 238 | 4 | 150 | 44 | 194 |
| cmb | 16 | 4 | 52 | 4 | 104 | 24 | 128 | 2 | 64 | 16 | 80 |
| cu | 14 | 11 | 53 | 11 | 106 | 80 | 186 | 1 | 83 | 40 | 123 |
| f51ml | 8 | 8 | 130 | 8 | 260 | 56 | 314 | 2 | 218 | 32 | 250 |
| misex1 | 8 | 7 | 54 | 7 | 108 | 48 | 156 | 3 | 96 | 32 | 128 |
| misex2 | 25 | 18 | 104 | 18 | 208 | 136 | 344 | 2 | 202 | 72 | 278 |
| pcle | 19 | 9 | 69 | 9 | 138 | 64 | 202 | 3 | 156 | 40 | 196 |
| term1 | 34 | 10 | 179 | 10 | 358 | 72 | 430 | 7 | 326 | 60 | 386 |
| ttt2 | 24 | 21 | 191 | 21 | 382 | 160 | 542 | 9 | 374 | 112 | 486 |
| x2 | 10 | 7 | 51 | 7 | 102 | 48 | 150 | 2 | 79 | 28 | 107 |

The circuits were placed and routed using the TimberwolfSC 4.2c standard cell package [21], [22]. Results are shown in Table IV, comparing the resulting layout areas with those reported in [7]. The layout areas are given in units of 1000 $\lambda^2$, where $\lambda$ is the minimum size in a technology. The percentage of area overhead required is computed as shown below:

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

In [7], results are given for using a Berger code (as proposed in [6]) and for using a single-bit parity code. As can be seen, in most cases, the procedure described in this paper provides a significant

TABLE IV
LAYOUT AREA COMPARISON FOR DIFFERENT METHODS

| Circuit | | Duplication | | Berger Code [7] | | Sgl-Bit Parity [7] | | Proposed Procedure | |
|---|---|---|---|---|---|---|---|---|---|
| Name | Layout Area | Layout Area | Ovh % | Layout Area | Ovh % | Layout Area | Ovh % | Layout Area | Ovh % |
| apla | 1039 | 2391 | 173 | 1422 | 37 | 1485 | 43 | 1247 | 20 |
| br1 | 648 | 1481 | 128 | 1076 | 66 | 1117 | 72 | 875 | 35 |
| bw | 742 | 2790 | 276 | 2520 | 240 | 1344 | 81 | 1066 | 44 |
| chkn | 1553 | 3650 | 135 | 2888 | 86 | 2485 | 60 | 3280 | 111 |
| dc1 | 130 | 423 | 224 | 346 | 165 | 257 | 97 | 252 | 94 |
| dc2 | 562 | 1316 | 134 | 1035 | 84 | 668 | 19 | 624 | 11 |
| exp | 1730 | 3862 | 123 | 2303 | 34 | 2137 | 24 | 2123 | 24 |
| luc | 756 | 2761 | 265 | 2260 | 199 | 1946 | 157 | 1523 | 101 |
| p82 | 437 | 1286 | 194 | 1145 | 162 | 718 | 64 | 672 | 54 |
| signet | 1378 | 3342 | 143 | 2736 | 99 | 2730 | 98 | 1755 | 27 |
| wim | 184 | 513 | 179 | 480 | 161 | 274 | 49 | 224 | 22 |
| 5xpl | 459 | 1060 | 131 | 1443 | 215 | 688 | 50 | 797 | 72 |
| alu4 | 3298 | 6796 | 106 | NA | NA | NA | NA | 6796 | 106 |
| b12 | 284 | 727 | 156 | NA | NA | NA | NA | 613 | 116 |
| cmb | 153 | 414 | 171 | NA | NA | NA | NA | 206 | 35 |
| cu | 181 | 537 | 197 | NA | NA | NA | NA | 356 | 97 |
| f51ml | 385 | 923 | 140 | NA | NA | NA | NA | 677 | 76 |
| misex1 | 154 | 403 | 162 | NA | NA | NA | NA | 355 | 131 |
| misex2 | 372 | 1166 | 213 | NA | NA | NA | NA | 935 | 151 |
| pcle | 229 | 659 | 188 | NA | NA | NA | NA | 523 | 128 |
| term1 | 617 | 1542 | 150 | NA | NA | NA | NA | 1211 | 96 |
| ttt2 | 630 | 1977 | 220 | NA | NA | NA | NA | 1707 | 171 |
| x2 | 144 | 433 | 201 | NA | NA | NA | NA | 279 | 94 |

reduction in area overhead compared with the previous techniques. There are only a few cases ($chkn$ and $5xpl$) where the heuristics in the proposed code selection algorithm did not find the best code. Note that for the circuits where results are not available in [7], an "NA" is placed in the corresponding entry.

## VII. SUMMARY AND CONCLUSIONS

New techniques for the automated synthesis of multilevel circuits with concurrent error detection using a parity-check code were presented. Given the circuit description, the procedure is as follows.

1) Determine a parity-check code for encoding outputs of the circuit using a greedy algorithm.
2) Compute the parity functions for each check bit, and add them to the circuit description.
3) Use structure-constrained logic optimization to generate a PFS implementation.
4) Add a TSC checker to form a self-checking circuit implementation.

This procedure reduces area overhead compared with previously proposed synthesis methods because the parity-code selection algorithm uses a cost function that considers the area requirements for the function logic, parity predict logic, and checker, and the new SCLO technique considers the structural constraints during each step of the logic optimization. Results were presented that show that this method can significantly reduce the area overhead required for concurrent error detection in multilevel circuits while still detecting all internal single stuck-at faults. A possibility for future research is to apply these techniques to the synthesis of fault-tolerant finite-state machines where the states are encoded with a parity-check code [23]. Also, the SCLO procedure described here can easily be generalized for any types of structural constraints during logic synthesis, and may have other applications.

## REFERENCES


[1] R. M. Sedmak, "Design for self-verification: An approach for dealing with testability problems in VLSI-based designs," in *Proc. IEEE Int. Test Conf.*, 1979, pp. 112–120.

[2] S. K. Gupta and D. K. Pradhan, "Can concurrent checkers help BIST?," in *Proc. IEEE Int. Test Conf.*, 1992, pp. 140–150.

[3] ——, "Utilization of on-line (cocurrent) checkers during built-in self-test and vice-versa," *IEEE Trans. Comput.*, vol. 45, pp. 63–73, Jan. 1996.

[4] D. K. Pradham, *Fault Tolerant Computing: Theory and Techniques, Vol. 1*. Englewood Cliffs, NJ: Prentice-Hall, 1986, ch. 5.

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

[6] N. K. Jha and S. Wang, "Design and synthesis of self-checking VLSI circuits," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 878–887, June 1993.

[7] K. De, C. Natarajan, D. Nair, and P. Banerjee, "RSYN: A system for automated synthesis of reliable multilevel circuits," *IEEE Trans. VLSI Syst.*, vol. 2, pp. 186–195, June 1994.

[8] N. A. Touba and E. J. McCluskey, "Logic synthesis techniques for reduced area implementation of multilevel circuits with concurrent error detection," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, 1994, pp. 651–654.

[9] B. Khodadad-Mostashiry, "Break faults in circuits with parity prediction," Tech. Note 183, Center for Reliable Computing, Stanford Univ., Stanford, CA, Dec. 1980.

[10] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proc. IEEE*, vol. 78, pp. 264–300, Feb. 1990.

[11] J. E. Smith and G. Metze, "Strongly fault secure logic networks," *IEEE Trans. Comput.*, vol. C-27, pp. 491–499, June 1978.

[12] D. A. Anderson, "Design of self-checking digital networks using coding techniques," Tech. Rep. R-527, Coordinated Sci. Lab., Univ. Illinois, Urbana, 1971.

[13] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "Multi-level logic optimization and the rectangular covering problem," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, 1987, pp. 66–69.

[14] J. Rajski and J. Vasudevamurthy, "The testability-preserving concurrent decomposition and factorization of Boolean expressions," *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 778–793, June 1992.

[15] N. A. Touba and E. J. McCluskey, "Logic synthesis for concurrent error detection," Tech. Rep. 93-6, Center for Reliable Computing, Stanford Univ., Stanford, CA, Nov. 1993.

[16] K. Keutzer, "Dagon: Technology binding and local optimization by DAG matching," in *Proc. IEEE/ACM 24th Design Automation Conf.*, 1987, pp. 341–347.

[17] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Technology mapping in MIS," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, 1987, pp. 116–119.

[18] J. Khakbaz and E. J. McCluskey, "Self-testing embedded parity trees," *IEEE Trans. Comput.*, vol. C-33, pp. 753–756, Aug. 1984.

[19] J. L. A. Hughes, E. J. McCluskey, and D. J. Lu, "Design of totally self-checking comparators with an arbitrary number of inputs," *IEEE Trans. Comput.*, vol. C-33, pp. 546–550, June 1984.

[20] E. Fujiwara and K. Matsuoka, "A self-checking generalized prediction checker and its use for built-in testing," *IEEE Trans. Comput.*, vol. C-36, pp. 86–93, Jan. 1987.

[21] C. Sechen and A. Sangiovanni-Vincentelli, "TimberWolf3.2: A new standard cell placement and global routing package," in *Proc. IEEE/ACM 23rd Design Automation Conf.*, 1986, pp. 432–439.

[22] C. Sechen, 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 Univ., New Haven, CT, Oct. 1987.

[23] R. Leveugle, "Optimized state assignment of single fault tolerant FSM's based on SEC codes," in *Proc. IEEE/ACM 30th Design Automation Conf.*, 1993, pp. 14–18.