FAULT ESCAPES IN DUPLEX SYSTEMS

Subhasish Mitra, Nirmal R. Saxena and Edward J. McCluskey
Center for Reliable Computing
Departments of Electrical Engineering and Computer Science
Stanford University, Stanford, California
http://crc.stanford.edu

Abstract

Hardware duplication techniques are widely used for concurrent error detection in dependable systems to ensure high availability and data integrity. These techniques are vulnerable to common-mode failures (CMFs). Use of duplex systems with diverse implementations of the two modules has been proposed in the past for protection against CMFs. In this paper, we define a category of faults, called non-self-testable faults that undermine the data integrity of dependable systems. These faults produce identical errors at the outputs of the two modules of a duplex system and can potentially be caused by CMFs. The main contributions of this paper are: (1) techniques that identify non-self-testable faults in duplex systems, and (2) design methods that reduce the number of non-self-testable faults by test point insertion. We show that our algorithm for identifying non-self-testable faults runs orders of magnitude faster than exact techniques with minimal loss of accuracy. Also, there is a significant reduction in the number of test points required for duplex systems with diverse implementations compared to duplex systems with identical implementations. Thus, we can detect common-mode failures in diverse duplex systems using very few test points. These results are especially useful for systems with user-programmable logic elements that enhance the practicality of using diverse designs in duplex systems.

1. Introduction

Concurrent Error Detection (CED) techniques are widely used for designing systems with high availability and data integrity. A duplex system is an example of a classical redundancy scheme that has been used in the past for concurrent error detection. There are many examples of commercial dependable systems from companies like Stratus and Sequoia using hardware duplication [Kraft 81, Pradhan 96]. Hardware duplication has also been used in the IBM G6 processor. Figure 1.1 shows the basic structure of a duplex system.

In a duplex system there are two modules (shown in Fig. 1.1 as Module 1 and Module 2) that implement the same logic function. The two implementations need not be identical; for example, one could be the complement of the other. A comparator is used to check whether the outputs from the two modules agree. If the outputs disagree, the system indicates the presence of an error. *Data integrity* is the property of a system which either produces correct outputs or generates an error signal when incorrect outputs are produced. For a duplex system, data integrity is maintained as long as both the modules do not produce identical erroneous outputs (assuming that the comparator is fault-free). Since the comparator is crucial to the correct operation of the duplex system, special designs are needed to ensure that the data integrity of the system is not compromised due to comparator failure. The comparator design in [Hughes 84] can be used for this purpose.

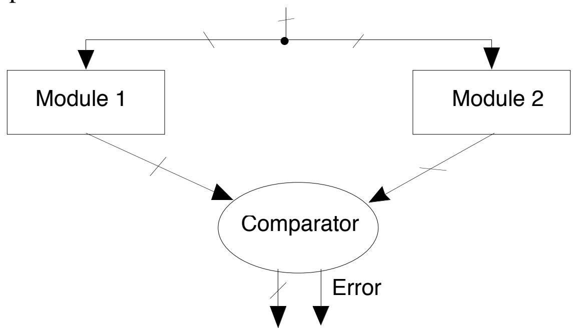


Figure 1.1. A Duplex System

Any duplex system is vulnerable to *common-mode failures* (CMFs) that affect both the modules of the system [Lala 94]. Design diversity through *independent* generation of *different* implementations of the two modules was identified as a possible solution to this problem [Avizienis 84]. In the presence of CMFs, the data integrity of a duplex system is not guaranteed to be preserved. Hence, CMFs must be detected using special techniques.

The main contributions in this paper are:

- An efficient algorithm for identifying non-self-testable faults (formally defined in Sec. 2) in duplex systems. These faults undermine the system data integrity.
- New techniques that use test points to detect all non-self-testable faults.

Our results indicate that the number of test points required for duplex systems with diverse implementations is significantly lower than those required for duplex systems with identical implementations.

We discuss the effects of diversity on the detectability of faults in duplex systems in Sec. 2. In Sec. 3, we present techniques to identify non-detectable faults in a duplex system that can potentially cause data integrity problems. Section 4 describes test point insertion techniques to detect these faults. We present simulation results in Sec. 5 and conclude in Sec. 6.

2. Self-testing Properties of Fault Pairs in Duplex Systems

Consider a duplex system consisting of two implementations N_1 and N_2 of the same logic function and a comparator comparing their outputs. The duplex system is *self-testing* with respect to a fault pair (f_1, f_2) (f_1 affecting N_1 and f_2 affecting N_2) if there exists an input combination for which the two implementations produce different outputs in the presence of the faults. The corresponding fault pair is said to be *self-testable*. If the two implementations produce different outputs in the presence of the fault pair, then the comparator will produce a *Mismatch* signal that can be used to initiate repair action.

Common-mode failures can have permanent effects on the behavior of redundant systems. This has been observed in the past [Lala 94]. For example, in dependable adaptive computing systems [Saxena 00] using SRAM-based FPGAs (e.g., Xilinx 4000 and Vertex series), single and multiple event upsets from radiation sources [Reed 97] can have permanent fault effects on the configuration bits. These faults can potentially be non-self-testable. The objective of our technique is to ensure that these faults are detected. In this paper, we consider all single stuck-at fault pairs pair $(f_1, f_2), f_1$ affecting N_1 and f_2 affecting N_2. This model includes common-mode failures that manifest themselves as single stuck-at faults in the individual implementations.

Table 2.1. Self-testing properties of duplex systems

Circuit Name	Modules	# Single-stuck-at fault pairs (millions)	% non-self-testable
Z5xp1	Identical	0.30	0.73
	Diverse	0.36	0.02
clip	Identical	0.49	0.58
	Diverse	0.46	0.02
inc	Identical	0.26	0.84
	Diverse	0.25	0.03
rd84	Identical	0.16	1.1
	Diverse	0.23	0.04

In Table 2.1, we show simulation results comparing the percentage of non-self-testable fault pairs in duplex systems with identical and diverse implementations. Diverse implementations were obtained by synthesizing the logic functions in different ways. More detailed information about the synthesis of different implementations can be obtained from Sec. 5.

For duplex systems with identical implementations, a common-mode failure (CMF) can be considered as one for which the corresponding leads in the two implementations are stuck at the same value. It is obvious that the self-testability of common-mode failures is 0 % in a duplex system with identical implementations. However, for a duplex system with different implementations, we have very few non-self-testable fault pairs. Thus, many of the potential CMFs can be detected by using diverse implementations. Self-testability enables on-line detection of faults (and CMFs included in the fault model) that affect the two modules of a duplex system. In the next section, we describe efficient techniques to identify non-self-testable fault pairs in a duplex system.

3. Identifying Non-self-testable Fault Pairs

In this section, we describe a technique to identify fault pairs that are not self-testable in a duplex system. The technique is approximate and is based on compaction of output responses for each fault.

We calculate a *signature* corresponding to each fault in each implementation. We call a fault pair non-self-testable if and only if the two faults forming the pair have the same signature. The reason behind this will be discussed later in this section. The algorithm is shown below (Algorithm 1).

```
ALGORITHM 1: Find Non-Self-Testable Fault Pairs
For each input combination p
        Simulate $N_1$ and store response $R_1$
        For each fault $f_1$ in $N_1$
          Inject $f_1$ in $N_1$, simulate and store response $R_1(f_1)$
                If $R_1 \neq R_1(f_1)$, update signature($f_1$)
        Endfor
        For each fault $f_2$ in $N_2$
          Inject $f_2$ in $N_2$, simulate and store response $R_2(f_2)$
                If $R_1 \neq R_2(f_2)$, update signature($f_2$)
        Endfor
Endfor
For each fault $f_1$ in $N_1$ and $f_2$ in $N_2$
        If signature($f_1$) $\neq$ signature($f_2$), ($f_1$, $f_2$) self-testable
        Else ($f_1$, $f_2$) not self-testable
Endfor
```

To calculate the signature associated with each fault, we use a Multiple-Input Signature Register (MISR) and a counter. The length of the MISR is at least 20 and greater than the number of outputs. For example, suppose that we want to calculate the signature associated with a particular fault f in N_1. For each input combination, if the response of N_1 in the presence of f is different from the fault-free response, then the counter is incremented and the faulty response is compacted into the MISR. The counter counts the number of test patterns that detect a particular fault. The signature of a fault is given by the pair <*content of the MISR, value of the counter*>. While *updating* a signature in Algorithm 1, the counter value is incremented by 1 and the circuit outputs are compacted in the MISR.

Our results show that using the counter or the MISR alone results in highly sub-optimal results. The sub-optimality arises from the fact that faults f_1 and f_2 may have the same signature although the fault pair (f_1, f_2) may be self-testable. In this situation, our algorithm will declare the fault pair (f_1, f_2) to be non-self-testable. This situation is referred to as *signature aliasing*. However, as the results in Sec. 5 indicate, using both the counter and the MISR, we obtain very close to optimum and often fully optimum results with negligible aliasing.

If the signatures for faults f_1 and f_2 are different, then the fault pair (f_1, f_2) is self-testable. If a fault pair (f_1, f_2) is non-self-testable, then the corresponding signatures are equal. However, the converse may not be true. For example, signatures for faults f_1 and f_2 may be equal due to aliasing while the fault pair (f_1, f_2) is self-testable. In this case, we classify a self-testable fault pair as non-self-testable. Thus, aliasing results in one-sided error and makes our algorithm pessimistic (unlike fault detection where aliasing may cause a defective part to be treated as a fault-free part).

A similar argument holds for the number of input patterns that must be applied to identify the self-testable fault pairs. In the worst case, for a system with n inputs, we have to apply all the 2^n input combinations for identifying self-testable fault pairs. If we use a reduced number of input combinations, a self-testable fault pair may be declared as being non-self-testable. However, the reverse situation cannot happen. Thus, using a reduced number of input combinations produces one-sided errors and pessimistic results. Thus, depending on the number of inputs of the circuits, the required execution time, and the desired level of accuracy, we can appropriately select the number of input combinations.

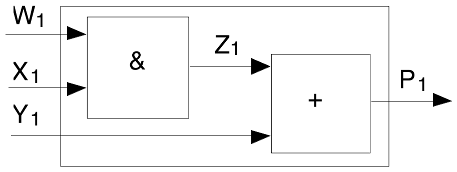
The running time of Algorithm 1 can be further reduced by using deductive fault simulation techniques [Abramovici 90, Armstrong 72]. The simulation results presented in Sec. 5 clearly show a distinct advantage in execution time by using Algorithm 1 over exact techniques. In the next section, we describe test point insertion techniques so that all fault pairs that are identified as being non-self-testable become testable.

4. Enhancing Self-testing Properties Using Test Points

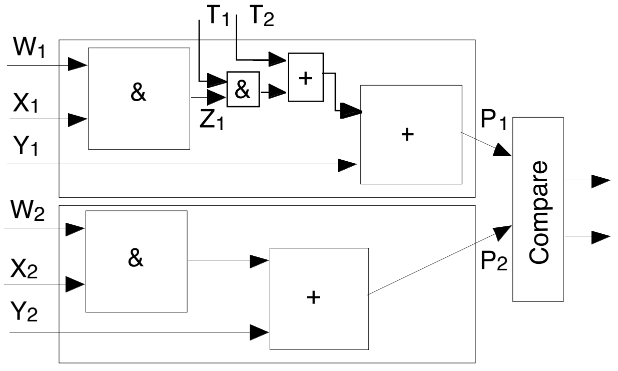
Test points have been used to enhance fault-coverage of logic circuits [Eichelberger 83][Abramovici 90][Touba 96]. In this section, we discuss test point insertion techniques to enhance the self-testability of duplex systems. There are two types of test points: control test points and observation test points. In Secs. 4.1 and 4.2, we describe self-testability enhancement using control and observation test points, respectively.

4.1. Control Test Points

Consider the duplex system consisting of two identical modules each implementing the logic circuit shown in Fig. 4.1a. Consider the fault pair in the presence of which, the signal line corresponding to Z_1 is stuck-at-0 in both the modules. It is obvious that the duplex system will never produce any mismatch signal in the presence of these two faults. Thus, the fault pair is not self-testable. Next, suppose that for one of the two modules, we add test points T_1 and T_2 as shown in Fig. 4.1b. We make $T_1 = 0$ and $T_2 = 0$ and *apply a test pattern* for Z_1 stuck-at-0. If the fault pair is not present, a *mismatch* signal will be produced. If the fault pair or other fault pairs are present, no mismatch signal will be produced. This observation can be used to detect the presence of the fault pair. A similar case with $T_1 = 0$ or 1 and $T_2 = 1$ arises when the fault pair Z_1 is stuck-at-1 in both the modules. Thus, control test points can enhance the self-testability of fault pairs in a duplex system. Note that, in a duplex system with two identical implementations, the fault pairs affecting the same leads in the two implementations are not self-testable. Thus, we have to add test points at each lead of the circuit in Fig. 4.1a.



(a)



(b)

Figure 4.1. Control Test Points

The primary advantage of using control points is that, we can utilize the available resources (comparators) of the duplex system for observing the response of the system in response to an input combination. Thus, we do not have to store simulated fault-free responses and compare the system response with these pre-stored responses to detect the presence of faults. However, we have to ensure that when the test points are activated, we apply a test vector for the untestable stuck-at fault pair. This can be achieved by using deterministic test patterns or pseudo-random patterns using an *LFSR* (Linear Feedback Shift Register). If LFSRs are used, some technique similar to the mapping logic technique [Touba 95] can be used for test point activation. For detecting the presence of faults when the test points are activated, we can XOR the *mismatch* signal output of the comparator with a *Test* signal that is 1 when one of the test points is activated. The *Test* signal may be generated externally or by the test controller.

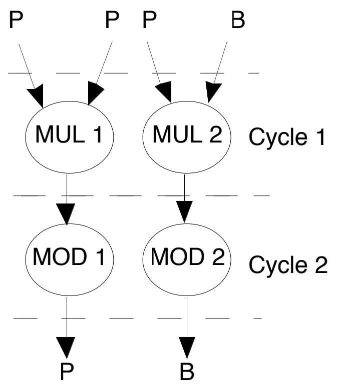


Figure 4.2. L-Modular exponentiation algorithm

During the idle cycles of the system, we can apply input combinations and activate appropriate test points (if necessary) to detect different fault pairs. Consider the example of a computation process shown in Fig. 4.2. Multipliers MUL1 and MUL2 are used in every alternate cycle (Cycle 1, Cycle 3, Cycle 5, etc.). Thus, during every even cycle, we can apply test patterns to the multiplier inputs. This can be done by adding extra instructions if the algorithm is implemented in a processor, or by using an LFSR. The basic advantage here is that we do not need any extra mechanism to process the response of the multipliers during the idle cycles. Use of idle cycles for concurrent error detection is also described in [Sohi 89].

Control points require extra area, may affect the circuit performance of the circuit and require more design effort.

4.2. Observation Test Points

Instead of adding control points, we can increase the self-testability of a duplex system by using observation test points. For example, in the duplex system of Fig. 4.1b, we can observe the logical values on the node Z_1 instead of adding any control test point. As a result, we can detect all fault pairs involving a stuck-at fault on Z_1. For observation purposes, we can perform signature analysis or directly observe the node Z_1.

This approach has a distinct advantage over control points because we do not have to add extra gates. However, we have to route the observation points to signature analyzers and perform comparison of the computed signature of the logic values on the observation test points with golden signature. For self-testable fault pairs, we can steal idle cycles of the system to apply test patterns. For non-self-testable faults, we must observe the logical values on the added test points (possibly through signature analysis). Thus, fault simulation and storage of fault-free signatures are necessary.

With observation test points, each application can be preceded and followed by testing phases. A high-level block diagram of an application with testing phases is shown in Fig. 4.3. Input patterns are applied using LFSRs or compiling deterministic finite state machines.

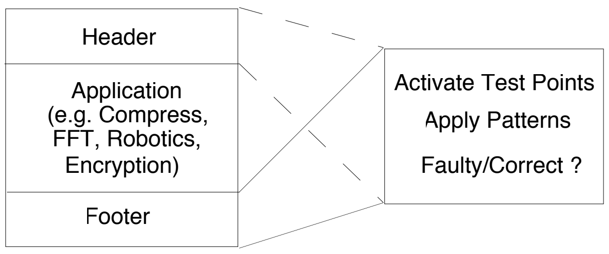


Figure 4.3. Applications with testing phases

Note that, with observation test points, the faults can be detected more easily — we just have to excite the fault and not worry about sensitizing the fault effect. Hence, we have to toggle logic values at the test point sites rather than propagating the fault effects to the outputs. Thus, the fault-free signature on the observation points is a 0 or a 1 for a stuck-at-1 or a stuck-at-0 fault, respectively. Finally, test points can help us perform quick fault-location and self-repair. Table 4.1 summarizes the advantages and disadvantages of the control and observation test points.

Table 4.1. Comparison of control and observation points

	Control Points	Observation Points
Area Overhead	Extra gates, Routing Area	Routing
Performance Impact	Possible	Small
Test Strategy	On-line (Idle Cycles)	Off-line (Start and End)
Effort	Fault Simulation	Fault Simulation, Response Analysis
Extra Pins	May be required	Required

4.3. Choice of Test Points

For duplex systems with identical implementations, the test points to be inserted can be determined very easily. In such a system with m leads in each implementation, there must be at least $2m$ single stuck-at fault pairs that are not self-testable. These are the same faults on the corresponding lines of the two implementations. Then we need m test points to detect all single stuck-at fault pairs [Mitra 00].

For a duplex system with different implementations, we find the non-self-testable fault pairs using Algorithm 1. Next, we choose the minimum number of test points that make all these fault pairs self-testable. This problem can be formulated as a *Covering* problem. A non-self-testable fault pair (f_1, f_2) can be detected if we insert a test point on the signal line corresponding to f_1 or f_2. Thus, for each non-self-testable fault pair, we have two candidate signal lines and at least one of these signal lines has to be selected in order to detect that fault pair. We explain the idea with the help of the following example.

Table 4.2. Test point insertion example

	(A_1, B_1)	(A_1, B_2)	(A_1, B_3)	(A_2, B_4)	(A_3, B_4)
A_1	X	X	X		
A_2				X	
A_3					X
B_1	X				
B_2		X			
B_3			X		
B_4				X	X

Let us suppose that we have 5 non-self-testable fault pairs, (A_1, B_1), (A_1, B_2), …, (A_3, B_4), as shown in Table 4.2. To detect this fault pair (A_1, B_1), we have to add a test point at the signal line corresponding to A_1 in the first implementation or at the signal line corresponding to B_1 in the second implementation. The rows of Table 4.2 correspond to candidate test points. We put an X in an entry if the fault pair in that column can be detected by inserting a test point at the signal line corresponding to the row of the entry. Selecting the minimum number rows in order to have X's under every column is the same as the classical covering problem encountered while finding the minimum number of prime implicants to represent a Boolean function [McCluskey 56].

```
Heuristic Test Point Selection
While there are columns in the table
    Sort rows in decreasing order of the number of X's
    Choose the row at the top of the sorted list
    Remove columns having X's in the selected row
Endwhile
```

The covering problem is NP-complete and has exponential complexity in the worst case. We implemented a simple heuristic algorithm as shown above. For the example in Table 4.2, we need two test points at the sites A_1 and B_4.

5. Simulation Results

In Tables 5.1(a) and 5.1(b), we show results on the number of test points needed to make all the fault pairs self-testable in a duplex system. For generating *different* implementations, we minimized the truth tables corresponding to some MCNC benchmark circuits using *espresso*. Then, we synthesized logic circuits after applying multi-level optimizations using the *rugged* script available in *sis* [Sentovich 92]. We subsequently mapped the multi-level logic circuits to the LSI Logic G-10p technology library [LSI 96]. These implementations are referred to as "T" in Table 5.1(a).

Next, we complemented the outputs in the truth tables of the benchmark circuits to generate new truth tables. We used the same synthesis procedure for these new truth tables. Finally, we added inverters at the outputs of the new designs obtained. These implementations are referred to as "C" in Table 5.1(a). In the third column of Table 5.1(a), we show the total number of single stuck-at fault pairs (in millions) in a duplex system containing the two implementations. This gives an idea of the complexity of the problem. In the next three columns we show the number of observation test points needed to detect all the single stuck-at fault pairs using different techniques. The exact algorithm that can find the minimum number of test points is based on ATPG (Automatic Test Pattern Generation). The ATPG tool available in *Sis* is used for this purpose.

Table 5.1(a). Test points for 100% self-testability

Circuit Name	Copy	# pairs (Million)	# Observation Test Points		
			Exact	Algo. 1 MISR	Algo. 1 MISR + counter
Z5xp1	T, T	0.30	275	275	275
	C, C	0.37	305	305	305
	T, C	0.36	**9**	**12**	**9**
clip	T, T	0.49	349	349	349
	T, C	0.46	**13**	**33**	**13**
inc	T, T	0.24	243	243	243
	C, C	0.26	253	253	253
	T, C	0.25	**12**	**13**	**12**
apex4	T, T	93	—	4818	4818
	C, C	74	—	4289	4289
	T, C	83	**—**	**46**	**46**
rd84	T, T	0.32	284	284	284
	C, C	0.16	199	199	199
	T, C	0.23	**10**	**22**	**11**
ex1010	T, T	84	—	4592	4592
	C, C	142	—	5961	5961
	T, C	109	**—**	**19**	**15**

Table 5.1(b). Execution time using different techniques

Circuit	# pairs (Million)	Exact	Algorithm 1
Z5xp1	0.36	2 min	**6 sec**
clip	0.46	4 min	**34 sec**
inc	0.25	1 min	**4 sec**
apex4	83	> 1 day	**85 min**
rd84	0.23	2 min	**6 sec**
ex1010	109	> 1 day	**4 hours**

The running time of the ATPG-based exact technique is extremely high for large designs as shown in Table 5.1(b). The entries marked with '—' are the cases where the exact technique ran for more than a day without producing any result. It follows from our discussions in Sec. 3 that the number of control test points is twice the number of observation test points. In the columns of Table 5.1(b) show the execution time required for finding the non-self-testable fault pairs using Algorithm 1 and an ATPG-based approach. All the programs were executed on a Sun Ultra-Sparc-2 workstation.

The following observations can be made from the data presented in Tables 5.1(a) and 5.1(b). It is overwhelmingly clear from Table 5.1(a) that, by adding only a very few test points in a duplex system with different implementations, we can make all the fault pairs (and all modeled CMFs) self-testable. The number of test points needed for duplex systems with different implementations is orders of magnitude lower than those needed for duplex systems with identical implementations. For minimizing aliasing (and hence, reducing the number of test points to be added), we recommend using both the MISR and the counter for calculating fault signatures in Algorithm 1. As shown in Table 5.1(b), we obtain significant speedup using Algorithm 1 compared to an ATPG-based exact technique.

6. Conclusions

In this paper, we demonstrated the advantages of using diverse implementations in enhancing the self-testability of common-mode and multiple failures in duplex systems. This result is significantly useful in the context of adaptive computing systems that enable easy instrumentation of design diversity. Our technique for finding the non-self-testable fault pairs shows orders of magnitude improvement in execution time compared to other competitive techniques. An interesting extension to our solution will be to preprocess a given circuit to identify the subset of inputs that decide the testability of a given fault. This preprocessing will be useful for circuits having a large number of inputs where each output depends on only a very small subset of the inputs. We have also described test point insertion techniques to detect all modeled common-mode and multiple failures. This enhancement helps in increasing the system data integrity and availability. There are further opportunities to reduce the number of test points using fault equivalence relationships [Mitra 00]. The test point insertion techniques reported in this paper can be combined with other test point insertion techniques used in the context of digital testing [Touba 96] to reduce the test length and detect different fault pairs more efficiently.

7. Acknowledgments

This work was supported by Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-97-C-0024.

8. References

[Abramovici 90] Abramovici, M., M. Breuer and A. Friedman, *Digital Systems Testing and Testable Design*, IEEE Press, 1990.

[Armstrong 72] Armstrong, D. B., "A Deductive Method of Simulating Faults in Logic Circuits," *IEEE Trans. Computers*, Vol. C-21, No. 5, pp. 464-471, May 1972.

[Avizienis 84] Avizienis, A. and J. P. J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *IEEE Computer*, pp. 67-80, August 1984.

[Eichelberger 83] Eichelberger, E. B. and E. Lindbloom, "Random-Pattern Coverage Enhancement and Diagnosis for LSSD Logic Self-Test," *IBM Journal of Research and Development*, Vol. 27, No. 3, pp. 265-272, May 1983.

[FIPS 77] FIPS PUB 46, "Data Encryption Standard," *FIPS Publication, US Department of Commerce/National Bureau of Standards*, National Tech. Info. Service, Springfield, Virginia, 1977.

[Hughes 84] Hughes, J. L., E. J. McCluskey and D. J. Lu, "Design of Totally Self-Checking Comparators with an Arbitrary Number of Inputs," *IEEE Trans. Computers*, Vol. C-33, No.6, pp. 546-50, June 1984.

[Kraft 81] Kraft, G. D. and W. N. Toy, *Microprogrammed Control and Reliable Design of Small Computers*, Prentice Hall, 1981.

[Lala 94] Lala, J. H. and R. E. Harper, "Architectural principles for safety-critical real-time applications," *Proc. of the IEEE*, vol. 82, no. 1, pp. 25-40, Jan. 1994.

[LSI 96] *G10-p Cell-Based ASIC Products Databook*, LSI Logic, May 1996.

[McCluskey 56] McCluskey, E.J., "Minimization of Boolean Functions," *Bell System Tech. Journal*, Vol. 35, No. 5, pp. 1417-1444, Nov. 1956.

[Mitra 00] Mitra, S., N. R. Saxena and E. J. McCluskey, "Fault Escapes in Duplex Systems," *Technical Report, Center for Reliable Computing, Stanford Univ., CRC-TR-00-1*, 2000 (http://crc.stanford.edu).

[Pradhan 96] Pradhan, D. K., *Fault-Tolerant Computer System Design*, Prentice Hall, 1996.

[Reed 97] Reed, R., *et al.*, "Heavy ion and proton-induced single event multiple upset," *IEEE Trans. Nuclear Science*, Vol. 44, No. 6, pp. 2224-2229, July 1997.

[Saxena 00] Saxena, N. R., *et al.*, "Dependable Computing and On-Line Testing in Adaptive and Reconfigurable Systems," *IEEE Design and Test of Computers*, Jan-Mar. 2000.

[Sentovich 92] Sentovich, E. M., *et al.*, "SIS: A System for Sequential Circuit Synthesis," *ERL Memo. No. UCB/ERL-M92/41*, EECS, UC Berkeley, CA 94720.

[Sohi 89] Sohi, G. S., M. Franklin and K. K. Saluja, "A Study of Time-Redundant Fault-tolerance Techniques for High-Performance Pipelined Computers," *Proc. FTCS*, pp. 436-443, 1989.

[Touba 95] Touba, N.A., and E.J. McCluskey, "Synthesis of Mapping Logic for Generating Transformed Pseudo-Random Patterns for BIST," *Proc. Intl. Test Conf.*, pp. 674-682, 1995.

[Touba 96] Touba, N.A., and E.J. McCluskey, "Test Point Insertion Based on Path Tracing," *Proc. VLSI Test Symp.*, pp. 2-8, 1996.