

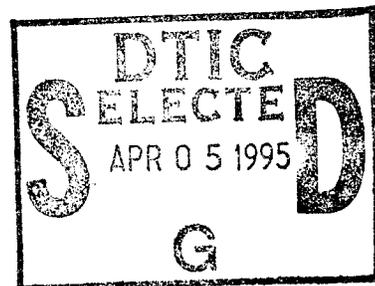
RL-TR-94-212
Final Technical Report
December 1994



MISSION CRITICAL FAILURE EFFECTS ANALYSIS USING QUANTITATIVE TECHNIQUES

Research Triangle Institute

Mark Royals, Rahul Kapoor, and Nick Kanopoulos



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

19950404 161

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-94-212 has been reviewed and is approved for publication.

APPROVED:



WARREN H. DEBANY, JR., Ph.D., P.E.
Project Engineer

FOR THE COMMANDER:



JOHN J. BART
Chief Scientist, Reliability Sciences
Electromagnetics & Reliability Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (ERDA) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE December 1994	3. REPORT TYPE AND DATES COVERED Final Feb 93 - Feb 94	
4. TITLE AND SUBTITLE MISSION CRITICAL FAILURE EFFECTS ANALYSIS USING QUANTITATIVE TECHNIQUES			5. FUNDING NUMBERS C - F30602-93-C-0009 PE - 62702F PR - 2338 TA - 01 WU - PM	
6. AUTHOR(S) Mark Royals, Rahul Kapoor, and Nick Kanopoulos			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Research Triangle Institute Center for Digital Systems Engineering P.O. Box 12194 Research Triangle Park NC 27709			10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-94-212	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory (ERDA) 525 Brooks Rd Griffiss AFB NY 13441-4505			11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: Warren H. Debany, Jr./ERDA/(315) 330-2047	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This report addresses the problem of assessing the criticality of faults in a large digital system. In particular, it addresses simulation-based methods for determining the effectiveness of built-in-test. The approach is based on automated fault injection in a VHDL model of the system, and statistical analysis of the resulting behavior of the system. The behavioral entities of the VHDL model would correspond to Line Replaceable Units (LRUs) or Line Replaceable Modules (LRMs). This report provides the top level design requirements and rationale for the detailed design of this capability.				
14. SUBJECT TERMS Fault effects, Criticality analysis, Fault injection, Built-in-test, System simulation, Design tradeoffs (see reverse)			15. NUMBER OF PAGES 106	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Accession For	
NTIS	CRA&I <input checked="" type="checkbox"/>
DTIC	TAB <input type="checkbox"/>
Unannounced <input type="checkbox"/>	
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Contents

1	INTRODUCTION	1
1.1	Technical Rationale	2
1.2	Motivation	3
1.3	Technical Approach	5
1.4	Modeling	8
1.5	About this Report	9
2	A PROPOSED TOOL STRUCTURE	12
2.1	Tool Outline	12
2.2	Tool Structure	14
2.2.1	Kernel	14
2.2.2	Fault Injector	16
2.2.3	Criticality Detector	17
2.2.4	Built-In Test Evaluation	17
2.2.5	Interactions:	18
3	FAULT INJECTION	19
3.1	Objective	19
3.2	Proposed Fault Injection Mechanism	19
3.3	Some Experimental Results	27
3.3.1	Statistical Models	29
3.4	Benefits of the Proposed Approach	31
3.5	Test Bench Generator	31
3.5.1	The Fault Selection and Injection Test Bench	33
3.5.2	The Fault Simulation Bench	36
4	CRITICALITY ANALYSIS	38
4.1	Determination of Fault-Effect Criticality	38
4.2	Methods for Criticality Analysis	40

4.3	Quantized Specifications	41
4.4	System Attributes Needed for Performing Criticality Analysis	44
4.5	Attribute Association	46
4.6	How Simulation Output Can Contribute to Theoretical Evaluation	48
4.7	Summary	49
5	BIT EVALUATION	53
5.1	BIT Evaluation Approach	53
5.2	BIT Measures	55
5.3	Recommended BIT Insertion Techniques	60
5.3.1	Survey of Automatic BIT Insertion Techniques	60
5.4	Tracing Critical Faults	66
6	TOOLS	68
6.1	The Kernel	69
6.2	Queries	72
6.3	Simulator	77
6.4	Fault Injector	79
6.5	Criticality Detector	80
6.6	BIT Evaluation Tool	82
6.7	Online Help and Tutorial	83
6.7.1	Level 1	83
6.7.2	Level 2	85
6.8	Enhancements and Options, User Interface Manual	85
	References	86

List of Figures

1.1	The Basic Approach	7
1.2	Hierarchical Structure of a System	10
2.1	The Tool Structure	13
2.2	The Concept of Kernel	15
3.1	Fault Injection Model	24
3.2	Statistical Fault Generation	28
3.3	Statistical Fault Generation	30
3.4	User Specified Fault Model	32
4.1	Basic Methodology	39
4.2	Criticality Analysis Methods	41
4.3	Fields for Specification	43
4.4	Sampling Period Can Vary	44
4.5	Attributes to be Specified for Criticality Analysis	47
4.6	Criticality Detector as a Comparator	50
5.1	The BIT Evaluation Tool	56
5.2	The BIT Evaluation Tool with Multiple Performance Measures	61
5.3	TIGER: Testability Insertion Guidance Tool	63
5.4	Tracing a Critical Fault	66
6.1	Graphical Interface Incorporating the Proposed Tool Set in the Design Cycle	70
6.2	The Design Cycle	71
6.3	Parts in Fault Injector	79
6.4	Criticality Detector	81

List of Tables

3.1	Initial File Format for Fault List	34
3.2	Format for Fault List in Single Fault Mode	35
3.3	Format for Fault List in Multiple Fault Mode	36
4.1	Default Values for Signal Attributes	48
4.2	File Format for Fault-Free System Logfile	51
4.3	File Format for Faulty System Logfiles	51
5.1	File Format for BIT Logfiles	54
6.1	Recommended Set of Queries	73
6.2	File Format for BIT Logfiles	83

List of Recommended Tools

QuickSimII is a registered trademark of Mentor Graphics Corporation, 8005 S.W. Boeckman Road, Wilsonville, Oregon 97070.

DesignArchitect is a registered trademark of Mentor Graphics Corporation, 8005 S.W. Boeckman Road, Wilsonville, Oregon 97070.

Falcon Framework is a registered trademark of Mentor Graphics Corporation, 8005 S.W. Boeckman Road, Wilsonville, Oregon 97070.

AMPLE is a registered trademark of Mentor Graphics Corporation, 8005 S.W. Boeckman Road, Wilsonville, Oregon 97070.

DesignManager is a registered trademark of Mentor Graphics Corporation, 8005 S.W. Boeckman Road, Wilsonville, Oregon 97070.

Disclaimer: The mentioning of commercial tools in this report does not constitute an endorsement by the Air Force for those tools. These tools are recommended by the authors of this report because they are determined to be suitable for the proposed tool environment.

Acknowledgements

This report was prepared for the Rome Laboratory (RL) under Contract No. F30602-93-C-0009. The work was performed at the Research Triangle Institute (RTI) by Mark Royals, Rahul Kapoor, Nick Kanopoulos (Project Manager), report editing performed by Ingrid Agolia. The technical aspects of this project were monitored by Dr. Warren Debany (RL) who also provided valuable technical direction to the project team.

1. INTRODUCTION

The design specifications for modern DoD systems include test and diagnostic requirements for on-line and off-line detection of 100% of mission-critical faults. However, the definition of mission-critical faults (i.e., faults that cause loss of life, property, and mission abort) describes only the effect of their manifestation on system operation at the overall weapon system level. For a system developer to comply with test and diagnostic requirements, it is necessary to identify quantitatively the faults in system operation, at different levels of system hierarchy (i.e., component, board, module, subsystem), that can cause the effect described by the "mission-critical fault" definition. Furthermore, the same capability is required by the Government for auditing system developer compliance with requirements. The identification of mission-critical faults in a quantitative manner will allow the system designer to properly allocate test and diagnostic resources early in the design phase so that a required maintenance approach can be effectively implemented. The capability of fault injection, while the system is performing its operation, will allow the designer to evaluate and demonstrate the effectiveness of system-level BIT and fault-tolerance features. These capabilities do not exist today as part of a CAD environment, however, this project outlines the requirements for such a system. These capabilities are very much needed by weapon system developers who are using ad-hoc approaches to perform some of the cited functions and resort to extensive, costly field testing to evaluate the effectiveness of BIT and fault-tolerance features.

For a new tool development, it is important to consider possible interfaces to existing tools that can be used synergistically to aid the designer in different aspects of the system design process.

Tool development will be guided by user needs. Tool effectiveness in performing intended functions will be evaluated using a real-life system as a benchmark. Close interaction between

the tool development team and a weapon system development team will guarantee relevance of tool output in making decisions for the design of near-future systems. Furthermore, close interaction with the end-user will insure an interface compatible with widely acceptable design practices and standards. This, in turn, will facilitate the transfer of the developed technology to weapon system developers.

1.1. Technical Rationale

Modern DoD electronic systems rely heavily on advanced technology to achieve mission objectives while maintaining high levels of availability and survivability. To accomplish these features, the design specifications of these systems include stringent requirements addressing fault tolerance, integrated diagnostics, and Built-In Test (BIT) capabilities. For example, the requirements for continuous BIT in the INEWS system (i.e., an electronic warfare system to be used by several aircraft) read: *“C-BIT shall detect all mission-critical faults, be non-interfering, have a false alarm rate less than 1%, enable reconfiguration, and report mission-critical and non-mission-critical faults to the air crew via a resource manager. Faults reported to the air crew shall be identified as being either mission-critical or non-critical”*. The major issues that have to be resolved before designing according to this specification are the following:

1. Identification of mission-critical faults in a quantitative manner.
2. Test resource allocation for implementing BIT and fault tolerance for detection and recovery of faults, system reconfiguration, and air crew reporting.
3. Evaluation of the capability of the BIT to perform its function at the level required by the specification.

The objective of the work performed in this project was to develop a methodology whereby mission-critical faults can be identified for a given application through the use

of simulation, and to define a CAD tool environment that can implement this methodology.

1.2. Motivation

A large number of techniques have been developed in the past for performing Failure Mode Effects Analysis (FMEA). Traditional techniques initially developed are: the *Tabular* approach [1], the *Matrix* techniques [2,3] and the *Bayesian approach* [4,5]. The tabular technique is a “worksheet” approach which provides a simple inductive methodology, viable at any design level. Supplementary techniques which are based on tabular FMEA output have also been developed. These are the failure combination method [6] which explores the effects of multiple, and externally induced failures, and the Hardware/Software interface analysis [7].

The matrix technique provides a grided-plot format of the effects of failures in inputs, outputs, connections, and parts, at the lower level of the design. In this technique, the lower levels of analysis propagate to the upper levels following a bottom-up approach. A more advanced approach that is based on matrix techniques calculates probabilities of failures, in addition to the failure effects. This technique was automated in [8,9].

The Bayesian FMEA is a statistical approach that lends itself to automation. The primary inputs of the Bayesian FMEA are a reliability table and a criticality table, which relate what component(s) are likely to have failed given a system output failure.

Another major category of FMEA techniques is based on the fault-tree analysis [10,11]. In fault-tree analysis a specific undesirable system failure is defined, and a fault-tree of lower-level faults that caused the top-level failure is constructed using Boolean algebra. Fault-tree analysis is suitable for the analysis of fault-tolerant designs. Supplementary techniques based on fault-trees include the *Sneak Circuit Analysis* [12,13], *System Phase Modeling* [14], and the *Event-Circuit Analysis* [15].

Finally, the last class of existing FMEA methods includes techniques that consist of a combination of the previously mentioned approaches. One example of these techniques is the *Tabular Systems Reliability Analysis* [16] which combines aspects of tabular FMEA, fault-tree analysis, and Markov chain theory. This technique is well suited for evaluation of fault-tolerant distributed systems. Another example is the *Integrated Critical Path Analysis* [17], which examines failures that are caused by hardware/software interfaces. This technique combines aspects of tabular FMEA, fault-tree analysis, and sneak circuit analysis.

The existing FMEA methods can be divided into two major classes: The first class, which includes the most of the traditional FMEA techniques (i.e., tabular, matrix, fault trees, etc.), requires a significant involvement from a prospective user in terms of detailed knowledge of the examined system and the effects of failures in all the system components. The second class consists of the techniques that use analysis methods. These techniques require less user involvement, but they are highly specialized and demand large amounts of computing resources. Finally, the automation of such techniques is not considered to be feasible [18].

A common characteristic of all these techniques is their basis of analysis using first-order dependencies, topological information for the system, and qualitative information about failure modes. The detail involved in performing traditional FMEA techniques also causes the analysis to be overly time-consuming and costly. Failure effects analyses also tend to be unattractive from a user's point of view because typically a human must systematically categorize all the failure modes of the system. The analysis methods that are used to assess systems which depend on software activity is a limiting factor for all these techniques. The performance of these methods deteriorates even further when they are applied to complex computer systems. Because of all these problems, the use of FMEA techniques has been reduced to fulfilling contractual or quality assurance obligations [19].

In addition, the analysis of failure effects in preliminary design stages, where the system architecture has not been finalized, requires a high-level functional approach. Automation

tools capable of performing this type of analysis do not exist at this time. Moreover, the criticality of fault effects is mission dependent, but is not considered by any of the traditional FMEA approaches.

The proposed project will develop a method for determining quantitatively the effect of component failures to system operation by simulating the system architecture with the application code. The criticality of the fault effects will be determined by comparing their magnitude to user-specified thresholds defining mission-critical system operation margins. The advantage of the proposed method is that it provides results that relate the criticality of fault effects to a mission, thus allowing the user to design and evaluate system features such as BIT and fault tolerance, based on their contribution to fulfilling system mission requirements.

Finally, the automation of the proposed method with prototype CAD tools and their interface with other existing tools will create a unique design environment where the user can evaluate the performance, the testability, and the reconfigurability of a system, in addition to the FMEA analysis.

1.3. Technical Approach

To meet the objectives of the program, the work can be described by the following tasks:

1. Develop an overall methodology for performing meaningful fault injection, simulation of a system architecture, and identification of the faults which are deemed mission-critical.
2. Develop a technique to inject faults within a system in a statistical manner which adequately maps observed failure modes to faulty components. The user has the choice of injecting particular faults, which occur based on an assumed probability distribution function, and they can be injected at any interface between components.

3. Describe a methodology for comparing the response of the faulty system to the fault-free case and determine the criticality of the error response based on user-supplied specifications. Here the issue is the development of a framework for the user to specify the criticality of faults and for a tool to use this specification in making decisions.
4. Develop methods for evaluating the effectiveness of Built-In Test schemes which have been implemented within the overall design. This is the stage where a complete simulation model is necessary so that a tool can evaluate a chosen BIT scheme.
5. Define a set of requirements for a proposed design tool set which can implement the methodology steps described above. These tools are parts of an overall CAD framework that uses the existing tools in the framework to implement some of the functions outlined, and is pointed to wherever possible. There is a need to develop software to implement parts of the methodology which find no implementation in form of existing tools, and also for the interface between various tools.

The block diagram in Figure 1.1 outlines the approach used in this study:

This figure illustrates how the system designer may design a system using a system component library based on VHDL code for different modules. After such a description is completed, the designer can simulate the circuit until satisfied with the functionality. At this stage the designer may have the first prototype model with possibly some BIST circuitry included. The next step is to identify mission-critical faults, and to assess if the used BIST circuitry is able to trap the identified mission-critical faults. This can be accomplished by injecting faults in the circuit. The injected faults are a subset of all possible faults, and are designed to identify the mission-critical faults. Once such faults have been identified, it should be seen whether all such faults are being flagged off by the BIST circuitry. If not, the BIST scheme needs to be refined or its placement needs to be modified. To aid in selecting an appropriate subset of the total fault set failure rate, reliability data may be required, and

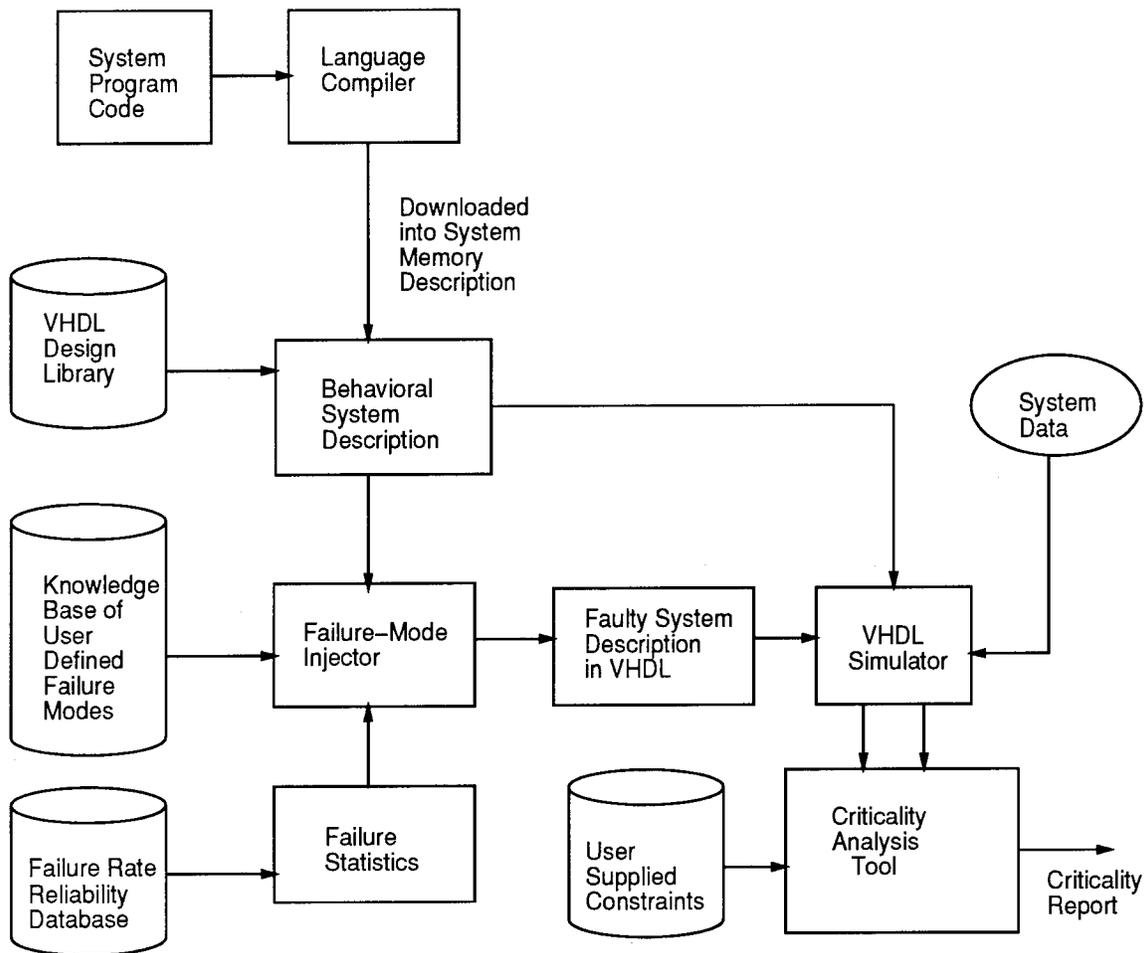


Figure 1.1. The Basic Approach

a knowledge base for user-defined failure modes may be of great assistance.

After the VHDL simulation of the faulty circuit and the comparison of its responses with those of the fault-free system, a criticality report can be prepared and used for further analysis.

The steps followed in developing the technical approach depicted in the figure are the following:

- Research available databases and the open literature to determine the current state-of-the-art techniques for performing one or more of the tasks listed above.
- Because of the emergence of VHDL as a standard method of describing the behavior/structure of a digital design, the proposed approach should leverage the wide variety of CAE tools which have been implemented to support the use of VHDL.
- Utilize a knowledge base concerning likely failure modes and their error manifestations for each of the described VHDL entities. Failure rates for each entity should also be provided from either external sources or best-guess estimates.
- User-controlled statistical parameters based on various probabilistic distribution functions can be utilized to govern the severeness and frequency of the failure injection process.
- Develop techniques to quantitatively evaluate the faulty simulation response with the fault-free response based on user-supplied threshold characteristics.

1.4. Modeling

As emphasized above, a model for the system to be evaluated is needed for quantitative fault model and effect analysis. The model should be expressed in VHDL, and certain attributes should characterize the model as discussed below:

- There should be a general way for developing a model for most of the systems of interest (i.e., the designer must be able to insert both design changes and BIT circuitry with relative ease).
- The model should be hierarchical in nature so that the designer can check or modify the design at any level. Figure 1.2 shows how a system (the largest rectangle) can be visualized as consisting of several subsystems (smaller rectangles inside the largest one). The system and each of the subsystems include certain attributes which help describe the system better. Such attributes are shown by ellipses in the figure. This model is especially true of VHDL descriptions of systems. This is because the VHDL itself describes systems in a hierarchical manner. The entities can include other entities in a hierarchical manner.
- Well-defined structure or quantized fields for ease in handling the data provided by the user. This also helps the user convey requirements to the tool easily.
- Flexible in the sense that it can be used at any level of design abstraction, (i.e., at the behavioral level, structural level, etc.).

1.5. About this Report

This report initially gives the reader a general idea of the functionality and the structure of a proposed tool set for performing mission-critical failure effects analysis, with this chapter as the introduction and the next chapter describing the general structure of the tool set and how various tools fit together.

The individual modules of the tool set are then discussed in detail, including their function, emphasising how the decisions were made, and some alternative options which may be functionally viable for the objectives of this project.

THE ELLIPSES DENOTE THE FIELDS.

THE RECTANGLES ARE COMPONENTS AT DIFFERENT LEVELS (e.g., SYSTEM, SUBSYSTEM, COMPONENTS).

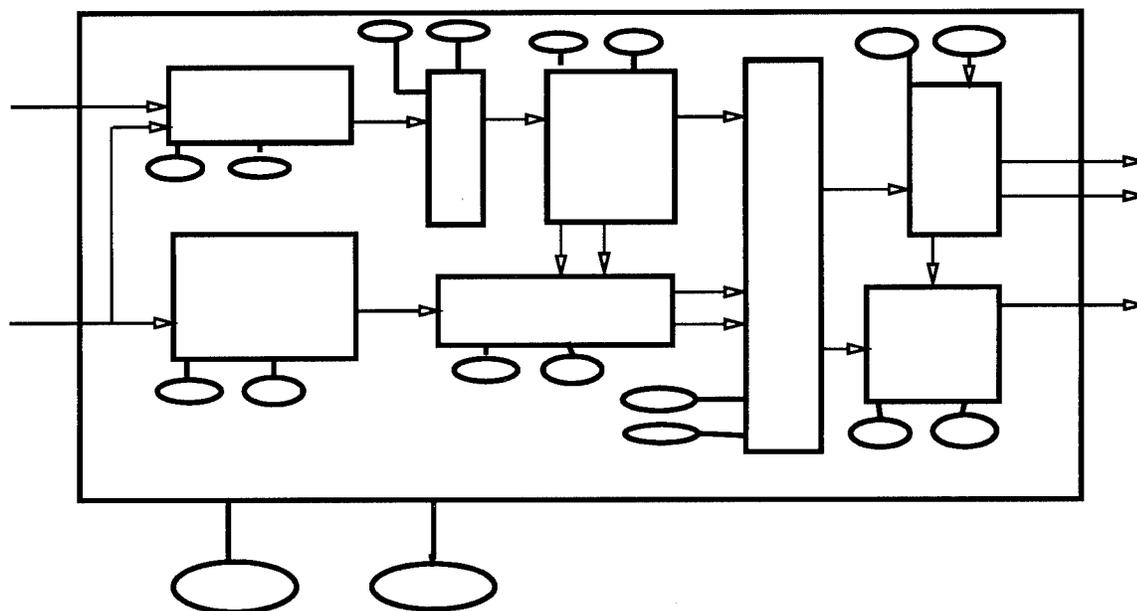


Figure 1.2. Hierarchical Structure of a System

After conceptually explaining each tool, the issue of implementation is addressed. Various details; such as the interfacing tools and their requirements, the suggested tool set, and how each fits in the general framework, are discussed.

2. A PROPOSED TOOL STRUCTURE

2.1. Tool Outline

The proposed tool consists of three parts as shown in Figure 2.1. These parts perform fault generation and injection, criticality analysis and BIT evaluation, respectively. In Figure 2.2 the dependencies and dataflow have been deliberately omitted for the sake of brevity. The emphasis is on clarifying the high-level concept of the tool function. The tool requirements and structure can be summarized as follows:

Functional Requirements:

The input to the tool will be the VHDL model of the application system which the designer is simulating. While simulating the application, the designer has certain specifications about the system behavior which need to be met. This implies that the designer can define a certain deviation of the system from the intended "normal" behavior, which may be considered non-critical to the system mission. The tool is required to check the effect of failures on the intended behavior of the system, and more specifically, if failures are mission-critical according to the user specified criterion.

Inputs to the Tool:

From the requirements of this tool, it is simple to see that the user has to specify the application code of the application to the tool. This code must be in VHDL, and the entities used by the designer have some of the attributes which the designer must specify to aid the tool in analyzing the criticality of failure (an example of these attributes can be whether the component being modeled by the entity description in VHDL is synchronous or not). The description of these attributes is deferred to the chapter describing the criticality tool.

Another set of inputs expected by the tool are the specifications of criticality. Through these specifications, the designer specifies the tolerance levels of the system. These tolerance levels are actually specified in terms of the outputs of the system. For example, the designer

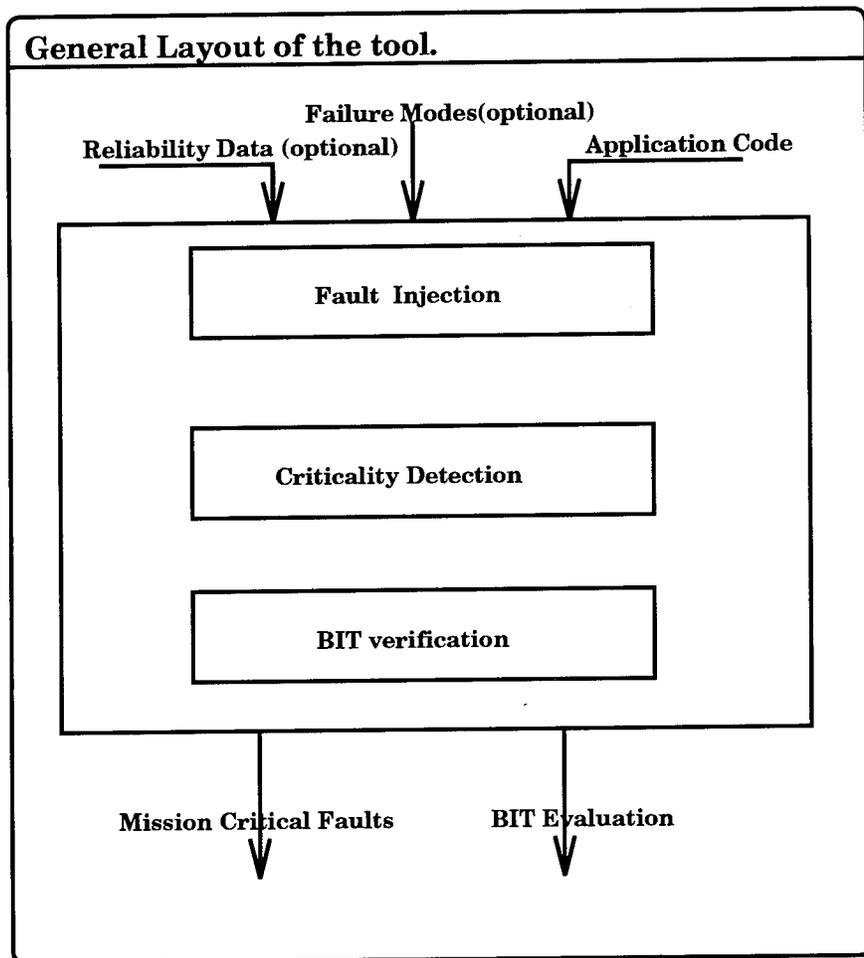


Figure 2.1. The Tool Structure

might specify that if a particular output (say x) differs from its fault-free expected value by 10 percent, then the criticality is to be flagged. In effect, the designer specifies what is considered to be mission-critical in terms of the quantitative effect of faults.

The third input (optional) is the reliability data of various components being used to model the system. These data indicate the probability of failure of these components, and can be especially helpful in the fault injection stage. If these data are not present, then equal probability of failure is assumed for each component.

Output of the Tool:

The tool will identify all the mission-critical faults out of those faults which were selected for simulation (this number can be arbitrarily large, limited only by simulation time and available computing resources).

The tool will also check the efficacy of the BIT (Built-In test) approach being used by the designer in terms of its capability in detecting mission-critical faults.

2.2. Tool Structure

First, a brief description of the structure of the tool components is given along with their interactions.

2.2.1. Kernel

The tool has three individual components with a simple dataflow, so that each component can be used alone. Despite this fact, there is a need for the central design manager which can control the dataflow and act as an interface between the user (who is designing the application) and the tool. This interface can be called the *Kernel*. The kernel can be thought of as the common database through which all the different tools can talk to each other and with the user. Such a view is presented in Figure 2.2.

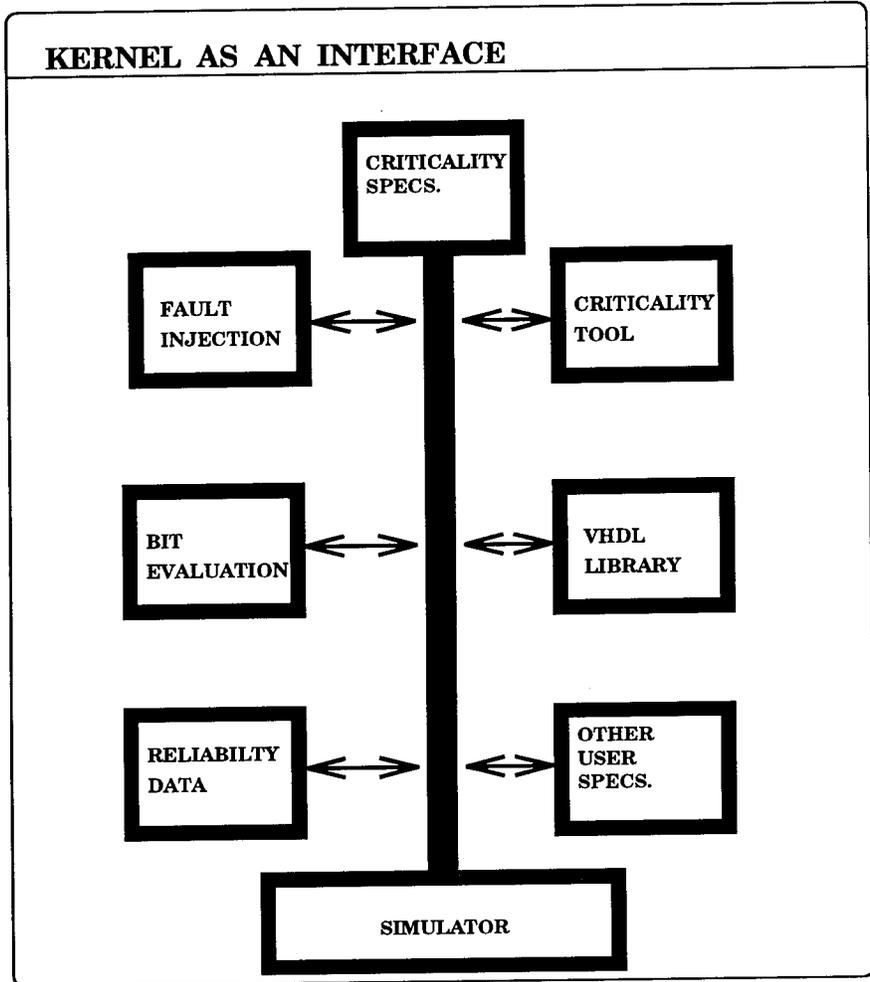


Figure 2.2. The Concept of Kernel

In this figure, two new elements are introduced. They are the VHDL library from which the designer can choose components, and the other user specifications which may be used by the kernel and/or tools. As an example, the user may want to specify whether the simulation has to be carried out in the Max, Min or Typical mode. Other things which can be specified by means of the kernel may be:

- Whether warnings are to be flagged.
- Should the attributes of the children be carried to the parent (i.e., inheriting attributes from the components).

2.2.2. Fault Injector

Methodologies for determining, in a quantitative fashion, the effects of faults at different levels of the system hierarchy, as far as mission requirements is concerned, are discussed here. This will be achieved through a simulation approach whose applicability will depend on the design level. Techniques that generate hardware failures in a statistical manner and map these failures to the different levels of the system hierarchy are a part of the total methodology. The statistical properties of the failures and the mapping could vary depending upon the application task and the operating environment of the system. The fault injector will perform the following functions:

- Parse the input VHDL description of the design and create superentities from the VHDL entities to do the fault injection with stuck-at fault model.
- Find which faults to simulate depending on user specifications and component reliability data.
- Direct the simulation.

- Should be compatible with a fault simulator.

Dependencies of the Fault Injector: The tool function depends on the component reliability database and on the statistical properties of the particular fault(s) the user wants to simulate.

2.2.3. Criticality Detector

The function of the tool is to determine the criticality of the effects of the injected faults. Based on simulation results, the quantitative effects of the faults on the system operation can be compared to user-specifications indicating what is considered a critical operation margin. This function of the tool will be performed using the following two steps:

- After getting inputs from the criticality specifications and the fault injector, the tool compares the obtained output with the fault-free expected output. Criticality is determined based on the error introduced by a fault versus an error margin that may be acceptable to the user based on the criticality specification.
- If critical errors are detected, they are logged along with warnings which are issued for potentially critical faults.

Dependencies of the Criticality Detector: The tool function depends on simulator output, fault injector, and the user-specified criticality parameters.

2.2.4. Built-In Test Evaluation

Procedures for evaluating the effectiveness of BIT schemes when the system is executing application code under faulty conditions.

- Define BIT counters at each of the BIT monitoring points. These counters keep count of various mission-critical and potentially critical faults in the system.
- Define two global counters, one for critical faults and the other for non-critical faults.
- After the simulation, observe the values of BIT counters to see if all critical errors have been detected.

After the simulation pass concludes (or upon first detection of the criticality of the fault), the values in the error counters at the BIT monitoring points can be added to one of two global counters for each location. These two global counters keep a cumulative tally of the *error counts* for the faults deemed critical, and for those that are found to be non-critical. After all of the target faults have been injected and simulated, the global counters can be used to order the candidate BIT sites in terms of their *efficiency* in detecting the faults deemed mission-critical. The non-critical faults have their own set of error counters which may provide useful information to the designer about the testability of the design, and provide feedback as to where he may want to put testpoints for non-BIT tests.

Dependencies of the BIST Evaluator: Depends on simulator output and output from criticality detector.

2.2.5. Interactions:

The interactions are fully explored in the chapter on tool implementation. A description of the files produced and used by each tool is given.

3. FAULT INJECTION

3.1. Objective

One of the most important components of the tool system described in Chapter 2 is the fault injector, which may be based on fault injection techniques described in this section. The objective of the fault injection is as follows:

Develops a technique that generates hardware failures in a statistical manner, and maps these failures to the different levels of the system hierarchy. The statistical properties of the failures and the mapping could vary depending upon the application task and operating environment.

3.2. Proposed Fault Injection Mechanism

Based on the failure probability of a component (which can be computed from the MIL-STD-217E data or estimated by the user), traditional FMEA techniques identify and analyze failure effects using primarily a topological model of the system and first-order dependencies among the components on the level of design hierarchy considered. We propose to extend this approach and map it to our failure injector mechanism such that a simulation approach can be utilized to determine the quantitative effect of the injected fault.

Whether or not the system is defined at a piece-part level of detail or at a higher level of abstraction, the failure injection method must be applicable at all of these levels. Our approach is to create a fault table for the VHDL-described system containing all of the output ports of all entities used in the functional system. This table can be created by parsing the VHDL description of the system and storing the output ports associated with each individual entity within a table. These entries need to be classified according to the signal type (e.g., whether they are single bits, multiple bit buses, integer representations

of buses, enumerated types, etc.) because in VHDL, description signals can be of different types. The next problem is to choose an appropriate fault model. Although some work has been done in defining higher-level fault models based on a VHDL behavioral description [81], it will be difficult to apply these fault models in a consistent manner. Furthermore, these models may be of questionable value for our purpose, although for test generation purposes they may be acceptable. Thus, we propose that at this time, the fault models be limited to the following:

- Pin stuck-at values
- Bus at wrong value

We propose that a weighting scheme be applied to the fault population in order to prioritize the fault injection process for simulation and analysis. We have identified several possible factors which can be assigned numerical quantities, and can thus be used as weights. In the following discussion, the objective is to find the weight w_i for each fault in the fault population of the system. These weights are found as follows:

1. **Failure rate λ** - The failure rate of a VHDL-described component can be used to identify which components are the most likely to fail within the specified mission time. These reliability data can be extracted from known piece-part reliability databases such as those found in MIL-HDBK-217E, or any other known field data sources. These databases associate a probability of failure with each component, say λ_i . The value λ_i is available for each component in the system, and from these values a weighting factor is obtained. This weighting factor is applied to all of the applicable failure modes for the specified component (e.g., all failure modes associated with a 74181 ALU would be weighted by the probability that this component will fail in the overall system). The actual weighting factor for the failure modes associated with component i is found as:

$$w_i = \lambda_i / \sum_{j=1}^n \lambda_j$$

where n is the total number of components in the system model. Note that the failure modes are the ways in which the specified component can fail, and thus correspond to the potential faults. Our concern is to find the likelihood of these failure modes. Also, it is noteworthy that the λ_i s must be expressed in the same units. So after applying this factor, all the failure modes of a specified component have the same weight, w_i .

2. **Failure mode probability-** Given that component i fails, some reliability databases may contain a conditional probability based on the applicable failure modes of the component. For example, if the 74181 ALU is selected for fault injection, then there may be conditional probabilities assigned to its individual failure modes. As an example, the probability that a failure in the ALU results from an error on the Cout pin may be 0.1, while 90 percent of the observed failures for the ALU result from the 4-bit output bus. Obviously such probabilities are of direct use to us. Assuming that for a specified component the weight as found in the previous step (from the λ_i s) was w_i . Then if we also have the failure mode probabilities for it, say β_i , the weight w_i for this mode is further modified, i.e.,

$$w'_i = w_i * \beta_i$$

Here w'_i is the new weight for the failure mode (or the fault). For the purpose of this report we are interested only in stuck-at faults.

3. **Fanout** of a signal may be weighted accordingly. If an output port of a component fans out to more than one location, the effects of a failure on that output port are likely to affect more than one destination port. Consequently, any faults injected on that

port will propagate to more than one unit and will be more likely to result in a critical error (since the multiplicity of faults may occur). A rough guess for an appropriate weighting factor would be a factor of two for every fanout point (i.e., if a signal has a fanout of three, then the weight is $2^{3-1} = 4$).

4. **Observability** - One factor which could be used as a negative weight would be the observability of the outputs of a given component. This score should be used only as a small weighting factor. Its purpose would be merely to give a higher priority to output ports which are deeply embedded in a design where the fault effects might be difficult to detect.
5. **Control/datapath functions** - A higher weighting factor may be desirable for control-oriented and/or state machine functions rather than datapath-oriented faults. Because the likelihood of the system being in an incorrect state when a critical processing cycle is required is more critical than a small miscalculation in the datapath, a higher priority may be wanted for these function types. As a default, these are the faults which might get the highest priority of all possible faults.

After creating the necessary weighting factors and applying them to the fault occurrence sites, an ordered list of these sites will be identified. Some of the properties of the fault injection should be:

- Faults can be injected based on user-defined triggering conditions.
- Bus-transfer faults can be injected based on the functional difference fault model.
- Interconnect-level faults can be modeled using stuck-at or stuck-opposite values.
- Development of an error tracing mechanism would allow the error effects to be traced through the system, thereby allowing for the identification of potential BIT locations

and the identification of equivalent faults which can be removed from the list of failure modes.

Two types of error assumptions should be used to adequately model system failures:

1. Faults occurring in the interconnect between two or more components (or between a primary I/O and a component)
2. Faults occurring within a design entity which may result in more than one error being present at the entity's output

Faults will be injected by modulating the output data being passed through the ports of an entity and may be assumed to be permanent.

The system will first be simulated within the VHDL simulation environment in order to determine the fault-free response of the system using the specified input vectors and/or application code running on the system test data. Because we are interested in determining the criticality of the system response, the simulation response for each output must be saved for future comparison with the responses of the faulty system. Furthermore, because of the BIT evaluation and recommendation facility, it will be necessary for the designer to identify the candidate nodes for BIT resource allocation. This is because we will also need to save the "fault-free" response at the monitoring nodes so that we can determine fault criticality at these nodes during the fault-injected simulation runs. As is the case with the system outputs, a signal trace mechanism needs to be used to store the simulation responses of the BIT candidate nodes during the fault-free simulation pass.

Faults will be injected within the VHDL description of the system by defining a super-entity which contains the original entity as an instance (Figure 3.1). The output ports of the original entity will be Exclusive-ORed (see discussion below) with a MASK value generated by the test bench (though described later, here it can be taken to mean that part of system

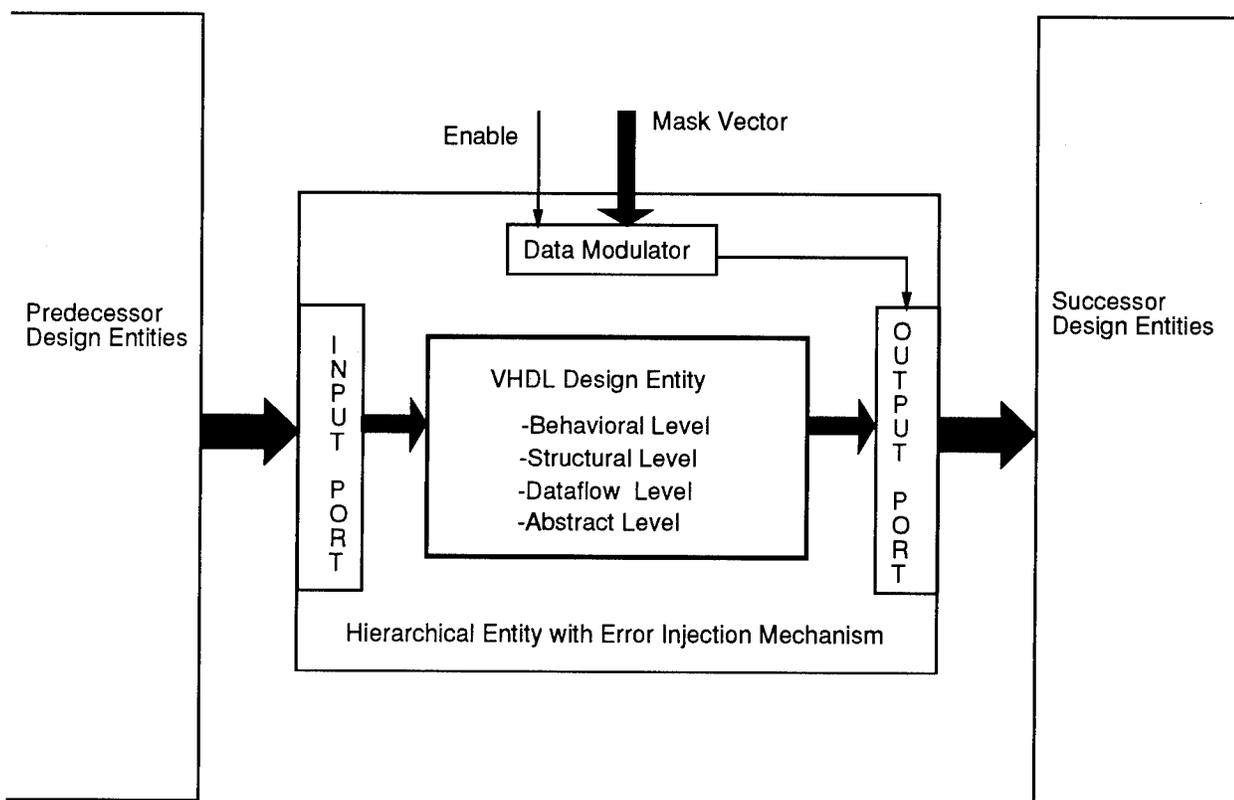


Figure 3.1. Fault Injection Model

which generates MASK values for fault injection). Several different scenarios could occur for the fault injection:

1. The MASK value is permanently fixed (based on the fault model selected for injection) and is applied permanently to the super-entity (e.g., it is always enabled). For permanent stuck-at type faults, the entity output and the fixed MASK value should be OR'd for a stuck-at-1 or ANDed for a stuck-at-0 fault. This type of fault injection can be used to model board or backplane interconnect faults where a constant signal value may occur as a result of a fault.
2. If the particular fault occurs in a cone of logic which feeds an output port of an entity, the failure is not always present. This can, again, be modeled using a fixed MASK value which is exclusive ORed with the entity output whenever the maskENABLE signal is set to one. Thus, we have the following behavior in VHDL for a fixed MASK value

```
if(ENABLE == TRUE) then
    Dout <= entity_out ^ MASK;
else
    Dout <= entity_out;
```

The intermittent nature of this type of fault is defined by the probability density function used for specifying the value of the ENABLE signal. Based on a user-defined pdf (i.e., binomial, exponential, log-normal, etc.), a random deviate can be generated from the pdf and be used to specify the value of the ENABLE signal. Thus, this will cause specific failures to be injected within the simulation at times specified by the pdf chosen beforehand.

3. The mask value is variable and is recomputed during each clock cycle of the system simulation. As in Scenario 2, the injection of failures in this manner makes use of

a probability density function. Based on the results of our experiments with various combinational and sequential off-the-shelf components, the number of faults occurring during a given simulation clock cycle have roughly a Poisson distribution. Most of the time, the injected fault within a component is masked and does not produce an error at the output. The next most frequent occurrence is for one error to occur on the component output. From a user-specified pdf, we can generate a random deviate during each clock cycle which can be used to specify the number of bit errors in an output bus for the "bus at wrong value" fault model. The number of errors in a bus must then be mapped to specific locations for each cycle. For example, suppose we generate a random deviate to inject two bit errors into a 4-bit output bus. The errors must be mapped to specific bits which can include any of the following cases (the following are the mask values where 1 in a bit indicates that the error is to be injected at that bit position),

- (a) 0011
- (b) 0101
- (c) 0110
- (d) 1001
- (e) 1010
- (f) 1100

If no information is available about the grouping of errors (which is probably the nominal case), then a uniform distribution will be assumed, and each one of these six cases will be equally likely to occur. However, if we can make use of information from a co-variance matrix that is created when a gate-level fault injection of a component is performed, then we may know beforehand that, say, bits 2 and 3 are likely to be faulted together, whereby bits 0 and 1 are completely independent and are unlikely to both

be in error during the same clock cycle. However, in most cases, it is anticipated that the functionality of the entity will be specified using either a behavioral or dataflow description, and the structural information needed to determine the covariance of the outputs will not be available.

In this scenario, the test bench must assume the responsibility of calculating the mask value on the fly during simulation. An alternative approach would be to generate the mask values and merge them with the incoming system data stimulus.

Upon injection of a fault, the simulator will be invoked with the output responses being monitored and stored for later use in the criticality analysis phase. Furthermore, it is necessary to keep track of the system responses at all of the candidate BIT locations that were identified previously. If the system does not contain rollback or instruction retry (which would cause the faulty simulation to be out of step with the fault-free simulation run) then it may be possible to build in the BIT evaluation facility within the simulation run. A comparator can be generated for each of the candidate BIT monitoring points whose function is to compare the value produced in the current clock cycle of the faulty simulation with the same cycle of data extracted from the fault-free simulation. In this manner, the fault-free data are essentially stored in an entity whose address is synchronized with the current clock cycle. If the comparator flags a mismatch between the two data streams, then it can enable a counter module which keeps a running total of the fault coverage at that location. The comparator and counters can be automatically generated by the test bench based on the number of bits in each monitoring point, their location, and the number of clock cycles that will be simulated in each pass.

3.3. Some Experimental Results

In order to examine the validity of the fault model suggested, some experiments were carried out. Several bus-structured MSI circuits were modeled at the gate level and subjected to

Cumulative Distribution of Bit Errors Observed

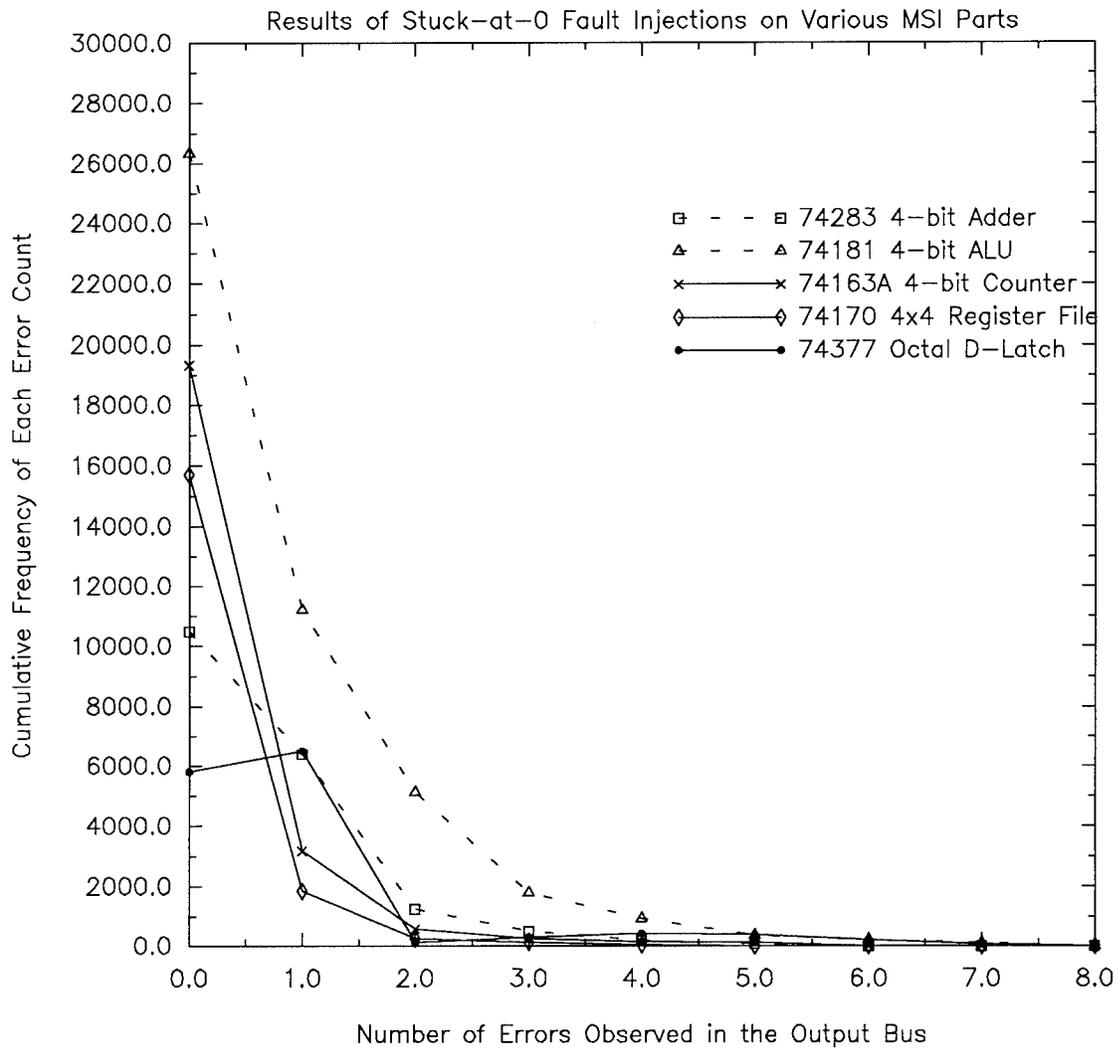


Figure 3.2. Statistical Fault Generation

exhaustive single stuck-at faults using a near-exhaustive set of vectors. The cumulative distribution of errors at the device outputs, as well as the cumulative total of the number of erroneous bits was collected. The results of such an experiment are depicted in Figures 3.2 and 3.3.

The results obtained suggest that in absence of any other information, the fault model can be used in a statistical model to generate faults. The statistical parameters can be varied, thereby leading to the error occurrence following a defined probability distribution function. These faults can then be used to simulate the system. The implicit assumption in this case is that the system cannot be exhaustively simulated, in general, due to computer time constraints. From Figure 3.2 we see the number of bits that are erroneous follow the classical Poisson distribution. Also the errors are uniformly distributed over the output bits which are related (i.e., a single bus). Thus, bus-transfer errors can potentially be modeled using statistical properties for error injection.

3.3.1. Statistical Models

As hinted, various models can be used to generate the faults for the proposed fault models, namely stuck-at and bus-at wrong value. Hence, supported by the experiments performed at RTI, there is reasonable evidence that it is better to generate errors following a statistical model, than random error generation not adhering to some specific statistical process. To this end, there is a need for the tool to provide a mechanism for incorporation of a statistical process for fault generation. The processes which have been identified have the Poisson and binomial distributions. One of these can be the default model. These two models are considered to be of special importance because they seem to fit most of the experimental data obtained during the course of this work.

Cumulative Distribution of Bit Errors Observed

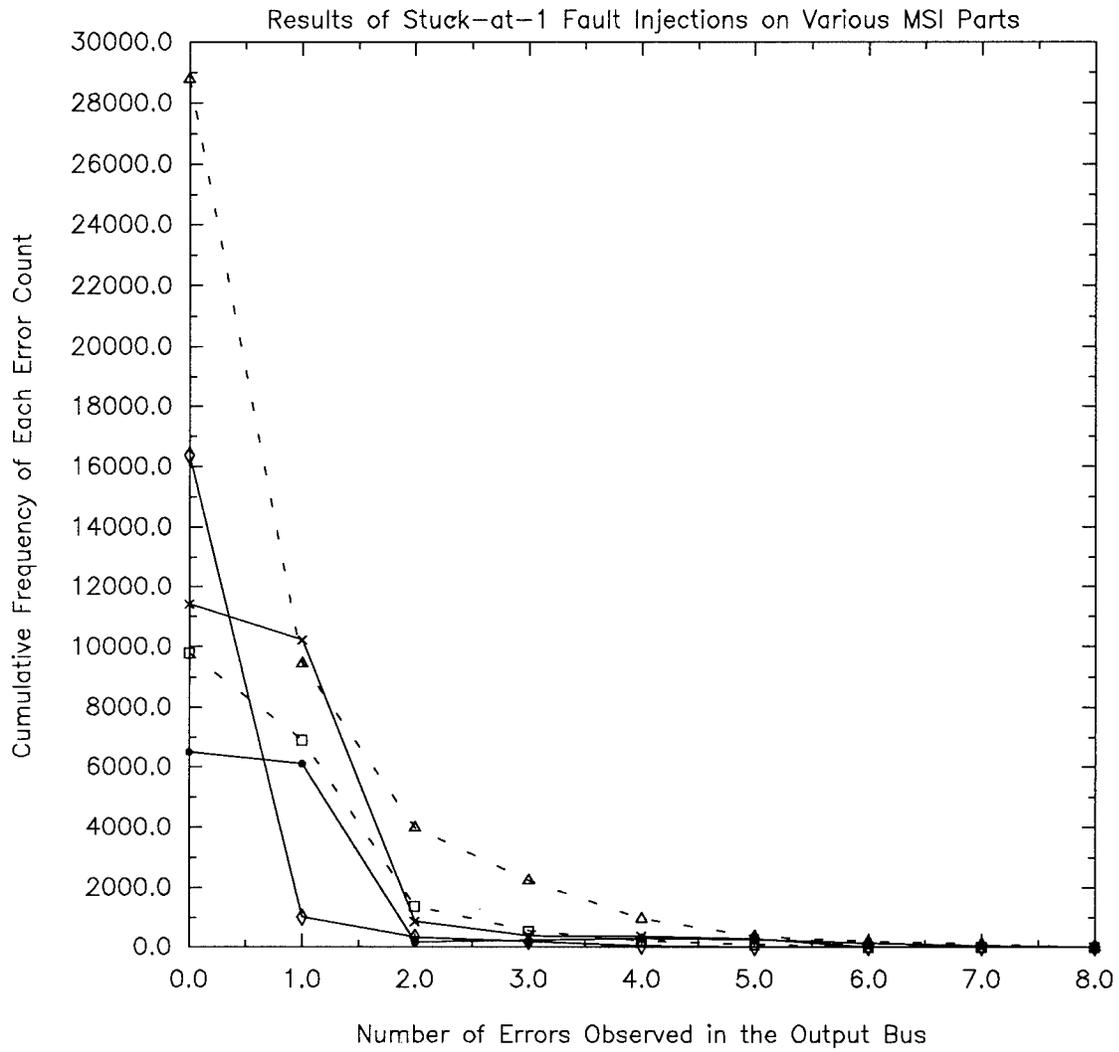


Figure 3.3. Statistical Fault Generation

3.4. Benefits of the Proposed Approach

There are some distinct benefits from the proposed approach:

- The sensitivity of the system to component failure modes can be determined quantitatively.
- The approach is applicable to multiple levels of design abstraction.
- The approach provides some meaningful insight into error manifestation sites in the system which can be used for the allocation of BIT resources to maximize their effectiveness.
- The same methodology can be used to verify BIT effectiveness once BIT modules have been added to the design.
- The effects of faults on a combined hardware/software model can be determined.

There are some issues, however, that need to be further investigated. For example, when should error injection be initiated during a simulation run? This will need to be related to the mission times of interest and/or the life-cycle of the system. Also a provision to incorporate an alternative fault model needs to be provided as shown in Figure 3.4.

3.5. Test Bench Generator

All previous ideas can be used to make a comprehensive test bench generator. It is so called because this tool is responsible for deciding which faults to simulate (i.e., which tests to run) and also for generating the associated mask values. It will automate the generation of files required to perform fault injection and simulation. The test bench is the top-level design unit required to perform both of these functions in the proposed tool environment. Conceptually,

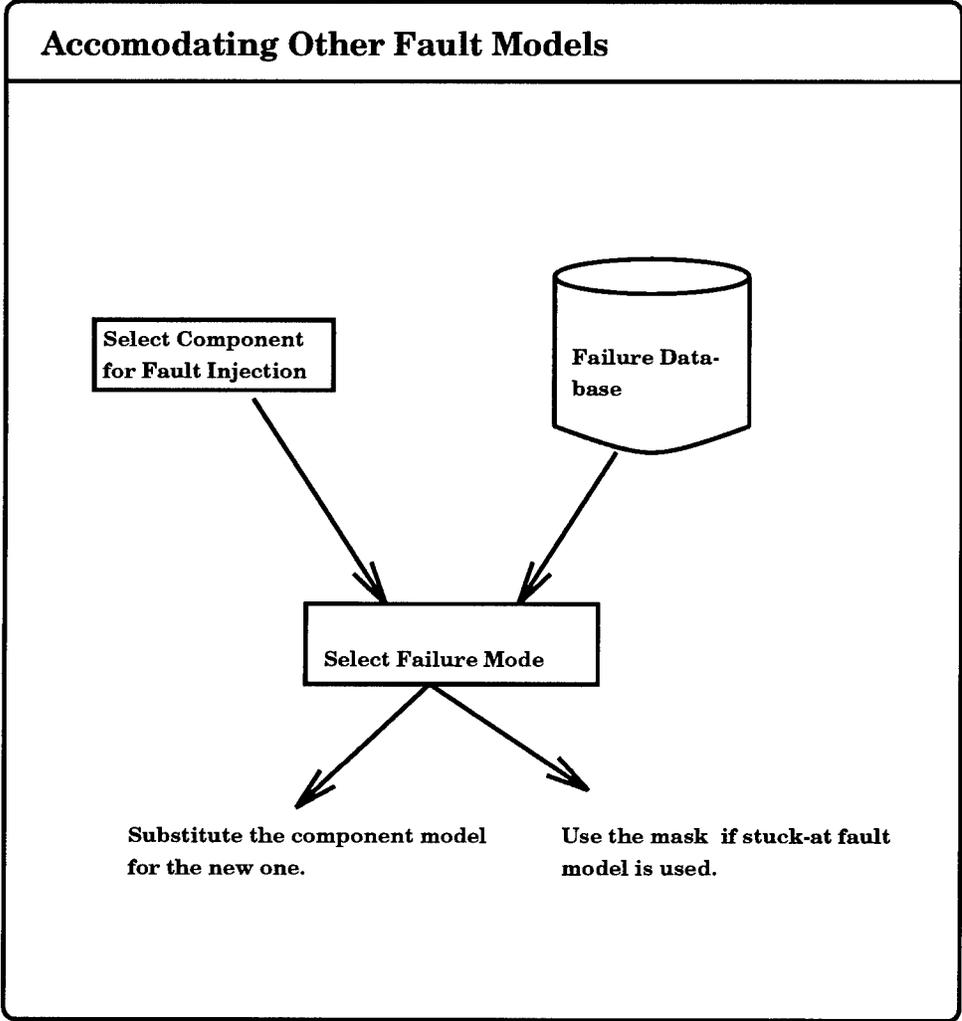


Figure 3.4. User Specified Fault Model

the test bench is a software implemented breadboard. As with traditional breadboards, the circuit components are wired together, the stimuli applied, and the responses are collected. All of the flexibility of the traditional breadboard is present. In addition, as with the case with fault injection, there is an additional facility of replacing components with their faulty counterparts and seeing what effect the fault or faults have on the circuit on this software implemented breadboard. The default model being "stuck-at-0 or 1," the faults are equivalent to tying the relevant nodes to ground or the supply voltage.

As stated previously, the test bench has two distinct functions, one of automatically generating faults, and injecting these faults into the circuit by one of the methods which have already been described. The fault list can be prioritized depending on the number of constraints imposed by the user. The tool uses a prioritized list of faults out of which it picks the specified number of faults for simulation. It injects these faults, simulates them and finds whether they are critical or not. In case of lack of information on actual fault weights, one could use a probability density function (pdf) of fault arrival to generate faults in a statistical sense. With a user defined pdf (or default pdf if there is no user defined pdf), the faults are injected one by one until the number of faults exceeds a user defined number.

There is clearly a need to separate the two functions of the test bench generator. This is so because both fault injection and fault simulation can be done in several ways. By separating these functions the user can have more choice to realize each of these functions in the desired way. The first function of fault list selection and fault injection can be done in several ways. An approach to performing these functions is described in the next section of this report.

3.5.1. The Fault Selection and Injection Test Bench

The elements needed to configure the bench are as follows:

- The **entity_name** identifier provides a device name for the model. For example an 8-bit register might have an entity name of REG8. This identifier is supplied by the user during the program initialization.
- The **Fault Selector** to be developed will parse all the entities for the input and the output ports. With this information, the stuck-at model can be used to finalize the fault list. Also after parsing, the output will be sent to a file called parse_output. This file has a column for the signals, a column for the entity name the signal corresponds to, and a column indicating whether the signal is input or output. The function of finding the weights for the faults can be performed during this parsing pass. Based on our previous discussion, there are certain weights which can be established by parsing the VHDL description of the system. For example, take the case of the weight established using fanout of a particular signal. This will be known from the VHDL description which indicates how many times this signal is directly or indirectly assigned to some other signal. Because of the available information, this task can be handled at compile time. This special parsing pass is needed for developing a prioritized fault list and the software for that has to be developed.

The output file for developing prioritized fault list of this process may have the format of Table 3.1.

Table 3.1. Initial File Format for Fault List

Port_name	Port_type	Entity	Fault	Weight
Cin	IN	ADDER	s-a-0	2
			s-a-1	1.3
A	IN	MUL	s-a-0	0.3
			s-a-1	4.1
...
...

After the required information is extracted from the user-given system description, there may be additional information available for the tool to use. As discussed previously, this information may be in the form of a reliability database for the components which make up the system. Such information must be used to refine the fault weights and, in this case, the output file from the previous stage can be used as input to this process. This additional information is optional, and a fault list will be established even without it. After processing this step, the output is again a file in the same format as before, but with the modified entries in the weight column. At this point (assuming that the user has already specified, through the kernel, his choice for faults to be simulated), the fault selector can proceed to select faults from the file and present it to the fault injector. For example, if the user specified only two faults to be simulated, after selection the file may have the format of Table 3.2.

Table 3.2. Format for Fault List in Single Fault Mode

Fault No.	Port_name	Port_type	Entity	Fault
1	A	IN	MUL	s-a-1
2	Cin	IN	ADDR	s-a-1

- The **Fault Injector** is the test bench unit which controls the mask values, used for the fault injection process. Depending on the input from the fault selector, the injector injects the faults in the system by controlling the mask and enable values. Depending on the mask scenario being used (i.e., one of the three described in section 3.2), the fault selector injects a fault every simulation cycle and invokes the simulator. It should also be possible for the user to inject several faults in a batch mode. The easiest way to do that would be to point out whether the fault injector should run in the *Single Fault* or the *Multiple Fault* mode. This can be enquired by the kernel, and if the user selects the latter, the kernel should prompt the user to indicate which faults should be

considered. This can be done by showing the fault list to the user and allowing faults to be selected together by clicking on them in a window. Continuing with our example, if the user chose to consider both the faults simultaneously, the file available for fault injection might be formatted as shown in Table 3.3.

Table 3.3. Format for Fault List in Multiple Fault Mode

Fault No.	Port_name	Port_type	Entity	Fault
1	A	IN	MUL	s-a-1
	Cin	IN	ADDR	s-a-1

- The test cases are the set of input combinations for which a designer might want to test the design under consideration. The simulation time, or what is also referred as simulation cycle in this report, depends on the number of test cases run. After this set of inputs has been exercised, the tool will inject the next fault in the list, and so forth until the chosen fault list is exhausted.

3.5.2. The Fault Simulation Bench

Under this section, the issues related to the simulation are taken up. With the typical test cases and the fault from the fault injector, the simulation test bench runs the simulation. It is possible that the user has only a few test cases for which he has already simulated the system and stored the response of the system. Alternatively, the user may want to change the test case for each simulation. Depending on which of these the user chooses, at the very outset of fault injection the user must specify whether the system should do the simulation for both the faulty and fault-free system in each simulation cycle or should just simulate the faulty system and compare the response with the stored responses. Obviously, with the stored responses there will be a saving in both time and effort. It also means that the user loses the flexibility to change the simulation inputs in each simulation cycle. That is not a

very big constraint in most situations because the designer has knowledge of the test patterns beforehand.

- The **simulator** is a VHDL simulator which can take the VHDL description (augmented with superentities) and proceed with the simulation. A commercially-available VHDL simulator can be used in this case. Before passing the circuit netlist to the simulator, those I/O signals which are specified as stuck-at by the mask values must be suitably modified.
- The **good model output signals** transfer the reference signals for the comparator to compare against. As already explained, these are the outputs of the superentities which the tool has constructed around the entities but no mask is applied, meaning that their values come out unchanged. These signals can be bits or buses, but in both cases the comparisons are performed at the bit level.
- The **faulty model input-output signals** are the results of the simulation with the injected faults in the model and are the other input to the comparator. There can be two modes of simulation: the faults can be injected one at a time, or as a group where a parameter *fault_number* specifies the number of faults injected. A fault type file would be created at runtime that would specify fault number and the type of fault injected.
- The **comparator output signals** convey the result of individual comparison operations. These signals can be used to determine whether the injected fault is critical or not. Criticality issues and how the comparison should be performed along certain guidelines is elaborated upon in the next chapter.

4. CRITICALITY ANALYSIS

4.1. Determination of Fault-Effect Criticality

Upon fault injection and simulation, the comparison will be made between the outputs of the faulty and the fault-free system under consideration. However, it is up to the user to define a quantitative threshold for fault effects beyond which a fault is deemed to be a mission-critical fault. Furthermore, the user will define when the comparisons take place during the fault injection process (i.e., after every clock cycle during simulation, or at the end of a simulation task). An assumption may be that the fault-free and faulty responses are compared every simulation step, however, other requirements may be imposed according to the system designer's view of the system function. For example, if the design under consideration is a FFT processor, the requirement may be to determine criticality based on the amount of error introduced by an injected fault in the final output only and not necessarily in intermediate results. This implies that the determination of criticality is made at the time of the comparison of system response at times specified by the user.

Figure 4.1 shows the basic methodology which incorporates the various phases for performing quantitative FMEA, including the criticality evaluation phase. It is assumed that the system simulation is complete and results are available and stored before comparisons are made to determine the criticality of an output. This is the default mode assumed for any system.

The criticality evaluation phase occurs after the simulation. Its inputs are the criticality specifications from the user and the simulation output from the simulation tool (fault-free output and system output in presence of injected faults).

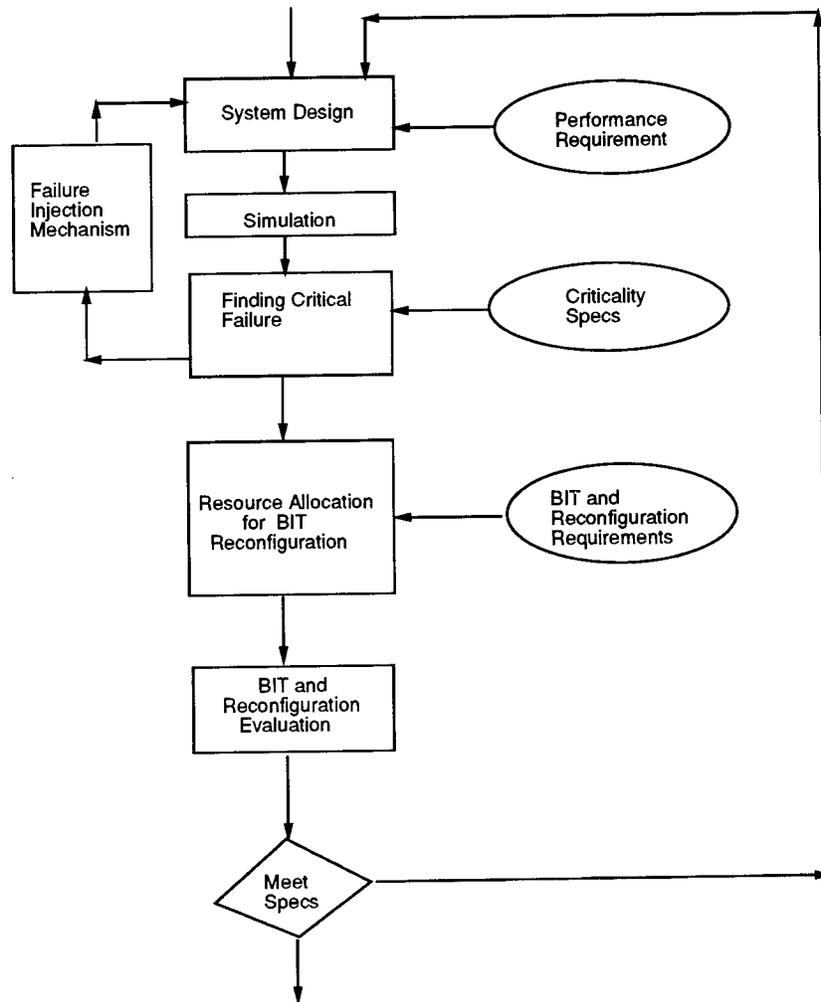


Figure 4.1. Basic Methodology

4.2. Methods for Criticality Analysis

Traditionally, the methods used for criticality analysis have not used simulation and have relied more on the usage of reliability models. These models require the user to define the criticality threshold for a component, system, subsystem, or a failure mode. Then the tool (using a database containing failure rates of components and failure modes) explores the interactions among the components from a topological point of view. Its output is traditionally described as the *criticality number*, and is indicative of the criticality of the failure mode in question. The disadvantage of this approach is that the model uses a great deal of data that are subjectively input by the user, and therefore the results are heavily skewed by the subjective judgement of the user.

In the simulation approach, the user specifies the failure to be injected and also the threshold for the criticality in terms of the quantity of error introduced by the injected failure, and the tool decides whether the failure manifests itself as critical to the system function. As described in the last chapter, the fault injection can be automated so that a representative fault set can be input. In most cases, the designer may be assumed to have a good idea of the types of faults that the system needs to be subjected to.

These two approaches are depicted in Figure 4.2. Also shown are the input requirements for each method. The traditional FMEA concept requires the knowledge of failure rates of various components (or VHDL entities in the simulation approach). The failure rates may have to be associated with a particular failure mode (i.e., for each type of failure of the system, the probability of that failure needs to be known). Obviously, this can be difficult for a large number of components. Still more difficult to know are the modification factors, such as the environmental modification factor (K_e discussed later in Section 4.6). Being unable to determine these probabilities, the user might be forced to inject some probabilities based on estimates that may not reflect the behavior of a component in a statistical sense.

In the simulation approach, a vehicle such as VHDL or any other structure specifying tool is needed as a vehicle for simulation. The VHDL model is the preferred one for such a tool, and this is the model which can be used to inject the faults and simulate the system. What is meant by quantized user specification will be further elaborated below.

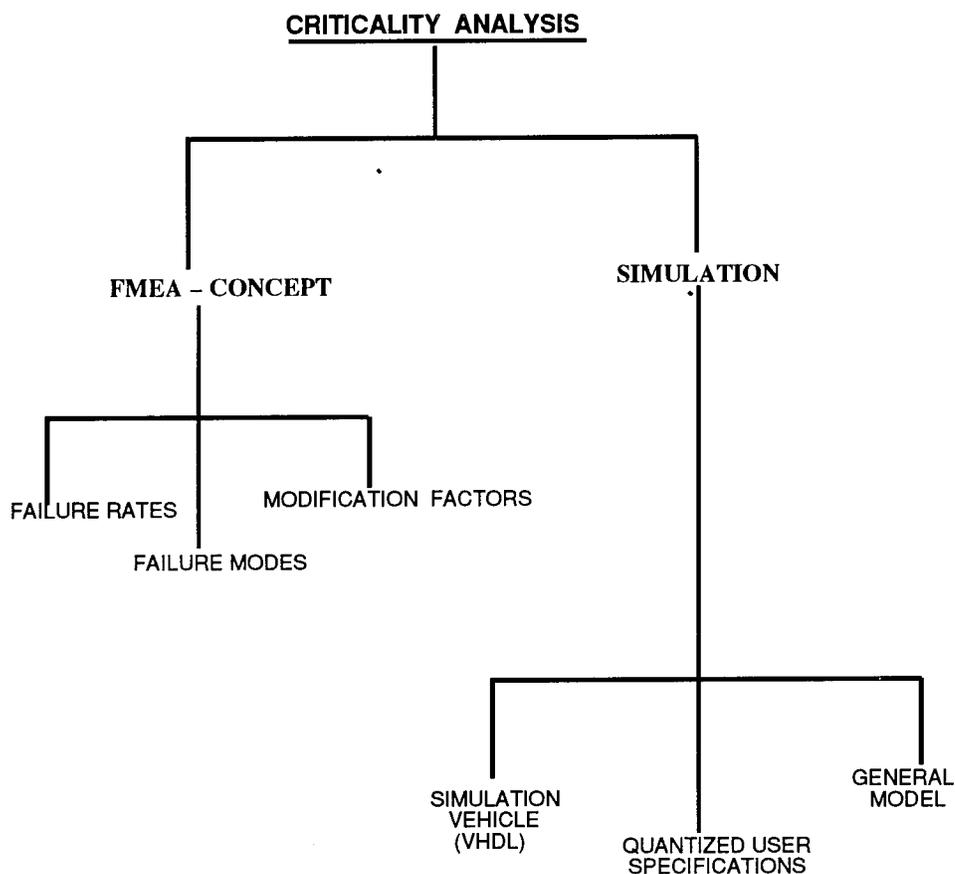


Figure 4.2. Criticality Analysis Methods

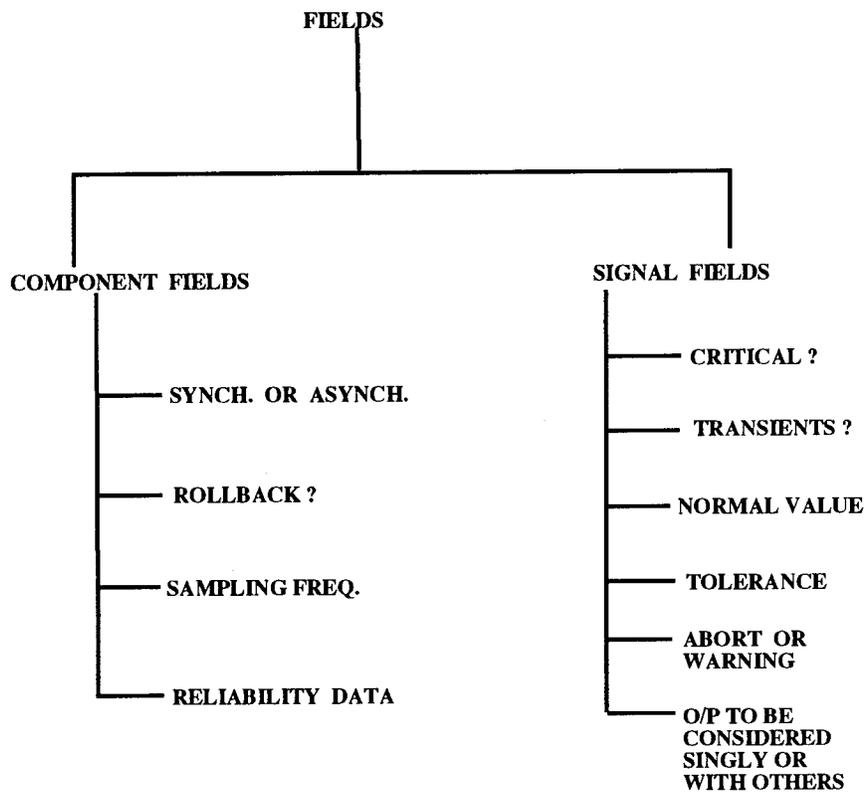
4.3. Quantized Specifications

The simulation of the system, including failures injected, produces results which have to be compared to user-defined criticality thresholds to determine if the injected failures are critical for the system mission. This comparison needs to be done with some guidelines

given by the user as to how the tool should determine whether a fault is critical or not. A technique needs to be developed that gives maximum flexibility to the user regarding how the criticality data may be specified. Criticality is manifested at the outputs of a system, subsystem, or component. Also, the criticality specifications depend on the type of system at hand. The term *quantized specifications* means that, with the help of system attributes, the user can completely instruct the tool as to what constitutes criticality for a given system mission. Figure 4.3 illustrates an example of attributes that may be used to perform criticality analysis. The aims in defining these attributes will be:

- To unambiguously define the system operational characteristics and the way they may affect the determination of criticality. The system characteristics to be considered may be synchronous or asynchronous operation, inclusion of rollback capability, etc.
- Define how the the comparison of the fault-free and faulty system response will be implemented. The comparison between the faulty response and the fault-free response might be done per clock cycle or every n th clock cycle where n is specified by the user. Another possibility is that the comparison is performed using the RMS value of the output for a user-specified number of cycles.
- Capture the diversity in the specifications by the users. The presence of transients and the like have to be incorporated as fields before being used to determine critical behavior.

Before the criticality tool proceeds to determine the failure effects, it must account for the type of components used in the system and the type of signals it is handling, in order to produce meaningful results concerning the criticality of the injected failures according to user requirements.



DEFAULT VALUES TO BE PROVIDED EVERYWHERE.

Figure 4.3. Fields for Specification

4.4. System Attributes Needed for Performing Criticality Analysis

This section provides a discussion on some of the attributes which the criticality tool will need before it proceeds with the analysis. Since the objective is to produce a tool that is general, there are a lot of attributes to be considered to take as many systems as possible in consideration. Normally all these attributes will not be needed and will be set to their default values.

The concept to be considered first is that of *sampling period*. It is an important parameter because this period determines where the comparison will take place between the normal expected data and the data returned by the simulator under fault injection. It must be noted, that this comparison determines whether the injected faults have been critical to the system or not at the end of a specified sampling period. By default, this period will be set to the clock cycle for synchronous systems, and at end-of-event simulations for asynchronous ones.



Figure 4.4. Sampling Period Can Vary

- If **TRANSIENTS** (events that occur out of synchronization with the system clock to which all other transitions are synchronized) are to be taken into account (i.e., if the output can be critical due to the transients) then the sampling period may need to be

decreased by the specified amount. By specifying this attribute the user decreases the sampling period to capture transients that may occur in short time intervals. Figure 4.4 shows how an otherwise smooth signal might have transients. On the right, the sampling period is shown to be larger because the user expected the signal to be smooth. On the other hand, if a transient was expected, the user might decrease the sampling period (as on the right of the axis) to examine the system fault effect at the time of the transient.

- The system can have the default value of the sampling frequency, but only in the case of **SYNCHRONOUS** systems. Otherwise the sampling frequency must be specified. Even in the case of synchronous system sampling frequency may have to be specified. One way to specify the sampling frequency, in the case of asynchronous systems, can be to tie it with a particular output. For example, in a system implementing a handshake protocol, the user might want the outputs to be compared when the acceptance from the remote party arrives.
- At the beginning of the process, the user could define which outputs need to be considered for criticality evaluation, and the aspects of how to measure criticality at a given output. This provides the user the capability to consider internal system nodes (in addition to input and output nodes) as well.
- If the user is interested in examining an output taken singly or an output combination (e.g., sum of two outputs), the **COMBINATION OF OUTPUTS** needs to be specified.
- Since an output might never be critical, or it might manifest itself as critical after a long latency, or it might be known to be critical, a field to keep track of these three possibilities is needed.
- A field is also needed to indicate whether the system has **ROLLBACK** points, or

not. This attribute will help the faulty and fault-free simulation to stay in step. This means that for systems that have a built-in, fault-checking mechanism; if the system rolls back upon detecting an injected fault, fault simulation will keep in step with the rollback, and normal expected value and faulty values will be compared only if the fault checking mechanism approves of the rollback values and the system is ready to move to the next stage. Note here that a complete simulation model is assumed, which gives the tool access to the flag which determines whether the system is to be rolled back or not. This implies is that the tool has access to both the external system and the internal system nodes.

- The user has to specify the **TOLERANCE** (i.e., margins of error) acceptable before the output becomes critical. Here arises the issue of whether the user might be interested in the average value, per sample value, or the RMS value of the output to be monitored. Depending on the user requirements, the tolerance needs to be appropriately defined.
- The **NORMAL** value to be used as a basis for specifying the tolerance must also be specified. Normally this is the value obtained from the fault-free simulation.
- **Degree of criticality.** There might be a need to warn the user of the effect of a particular fault if it produces an error that is within a specified margin of the normal value and to declare it critical if it exceeds that margin.

So we have a final picture of the needed attributes in Figure 4.5.

4.5. Attribute Association

An issue that needs to be addressed is how all these attributes are combined to form a comprehensive database. Some of the attributes, such as the system being synchronous or asynchronous, are derived from the system description. Other attributes, such as whether to

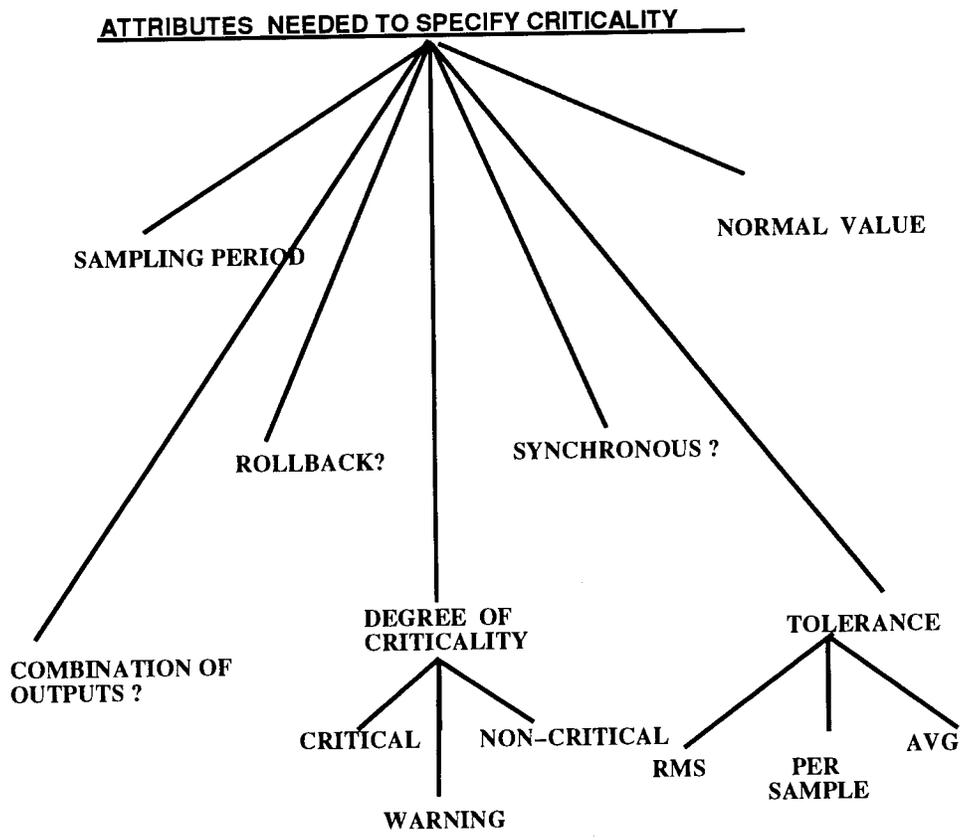


Figure 4.5. Attributes to be Specified for Criticality Analysis

consider the combination of outputs for criticality evaluation, can be neither associated with any of the components, nor with the system. Others, such as tolerance, are associated with individual signals. The degree of criticality is also associated with individual signals and it seems reasonable to associate the system attributes with the signals itself. In this case, the user may have to identify signal values of which the system will need to keep a record. These are the signal values the system will trace through the simulation.

After marking out signals of interest, the user needs to attach to these signals the attribute values that are characteristic of the system and the mission. Not doing so would lead to the use of default values. These default values may be tabulated as shown in Table 4.1.

Table 4.1. Default Values for Signal Attributes

Attribute	Default Value
Synchronous/Asynchronous	Synchronous
Sampling Period	Global Clock Cycle
Combination of Output	Individual Signals
Rollback Present	NO
Degree of Criticality	No Warnings

So the conclusion is that the attributes be attached with the signals. They can derive the relevant attributes such as synchronous or asynchronous from the components, of which they are a part.

4.6. How Simulation Output Can Contribute to Theoretical Evaluation

In the FMECA (Failure Modes and Effects Criticality Analysis) procedure, all the steps for FMEA are performed. In addition, *criticality analysis* is also performed. Criticality analysis is done by combining the occurrence of failure modes and the determination of the impact of a failure mode on the reliability of the system. As discussed before, there exists the concept of the criticality number which characterizes the failure effects on a system function using a

number of parameters which are often difficult to determine for real systems. In traditional FMECA analyses, subjective assignments are made to the values of unknown probabilities.

In the FMEA literature [78, 79, 82], the criticality number is defined as:

$$\sum_{i=1}^n \alpha * \beta * \gamma * T * K_a * K_e$$

where,

λ = Fraction of total failure rate attributable to each failure mode.

β = Conditional prob. that if failure mode occurs then the critical failure will occur.

γ = Generic failure rate for item (known).

i = Specified failure mode.

n = Total number of failure modes for component.

K_a and K_e are environmental failure rate modification factors and T is the operating time.

The values of λ , β and γ are evaluated from the data typically available from the field. If such data are unavailable, the analyst provides the missing values based on experience. Thus, a great deal of subjectivity is introduced in the determination of the criticality number. The proposed tool can reduce such subjectivity by providing the β values for various components. Since the tool simulates the failure modes and finds whether those failure modes are critical, determination of β for a given failure mode becomes a deterministic computation, which can be used for FMECA analysis.

4.7. Summary

This section charts the steps involved during the criticality detection phase. An overall picture of this phase is presented in Figure 4.6. After defining which signals are to be traced

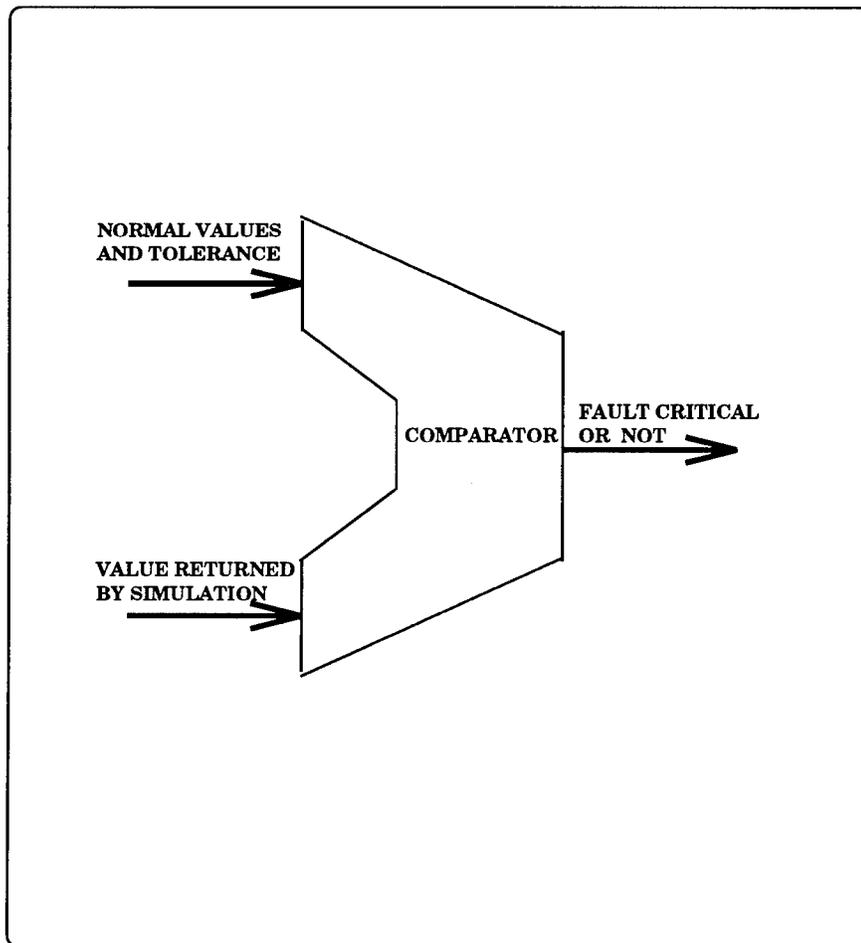


Figure 4.6. Criticality Detector as a Comparator

by the tool, the user lets the simulation run. As the simulation progresses, the results are continually stored for future comparison by the criticality detector. The sampling period attribute helps the tool decide what results are to be stored, and when. The preferred scenario would be that the simulator continues to work for the duration of the sampling period, at the end of which it stops and stores the results. The same procedure is carried out both for the faulty and the fault-free systems.

The results are stored in two logfiles. One logfile is used to store the result of the faulty simulation, and the other for fault-free case. The format for such a file can be of the form shown in Tables 4.2, 4.3.

Table 4.2. File Format for Fault-Free System Logfile

Time	signal 1	signal 2	signal 3	signal 4+5
10ns	2	3	1	.5
20ns	2.5	3.2	1	.5
30ns	2	3	1	.5
40ns	2	3	0	.9
....
....

Table 4.3. File Format for Faulty System Logfiles

Time	signal 1	signal 2	signal 3	signal 4+5
10ns	2	3.2	1.1	.5
20ns	2.5	3.2	1.0	.4
30ns	2	3.2	1.1	.6
40ns	1	2.2	1.1	.5
....
....

As shown in the tables, the logfiles chart the progression of values on the specified nodes as a function of the time. The values placed in the files are governed again by the attributes the user specified for the criticality phase. For example, they might be RMS values or, as shown in the tables, one of the columns might be a combination of signals rather than being

just a single signal. These logfiles can be updated after each simulation cycle, and later these values can be fed to the comparator which determines if and when the fault became critical. If the tool is operating in **one step** mode (i.e., stopping after each simulation cycle to determine whether the fault is critical or not), then as soon as the criticality is detected, the fault can be flagged. As an example from above tables, at 40ns the sum of signals 4 and 5 is 0.5 in the faulty circuit and 0.9 in the fault-free one. This might constitute a critical error. Still the tool may go on to finish the total simulation and do all the comparisons at the end, when it will find that first critical error occurred at 40ns. Alternatively, the simulator may be set to abort at the first instance of the critical error (**one step**). In **one step**, the comparison is carried out every simulation cycle.

5. BIT EVALUATION

The BIT evaluation tool will help the designer assess the efficacy of the BIT resources to meet system requirement. This will be done by determining the number of critical faults that the BIT detects. In Section 4.7, the concept of logfiles for storing the system response under faulty and fault-free conditions was introduced. This concept can be extended to BIT evaluation.

5.1. BIT Evaluation Approach

Since a complete VHDL model is input to the simulator, when the criticality detector detects a critical fault, the proposed tool can pinpoint both the nature and the location of the fault because the tool itself injected the fault. The objective of BIT assessment is to determine whether the fault is detected by any of the BIT schemes being used. This task will be performed by the BIT evaluation facility. After the simulation of each fault, the status of all BIT points together with the internal simulation data is stored, and then checked to see whether the BIT was able to detect the injected fault. So while simulation is in progress, an extra logfile needs to be maintained. This logfile captures the changes in BIT points as the critical errors are identified.

After a simulation pass concludes, or upon first detection of criticality of a fault, the values of the error counters keeping track of the BIT monitoring points can be added to one of two global counters. These two global counters will keep a cumulative tally of the *error counts* for the faults deemed critical, and for those that are found to be non-critical. After all of the target faults have been injected and simulated, the global counters can be used to assess the candidate BIT schemes in terms of their "efficiency" in detecting the faults deemed mission-critical. The non-critical faults have their own set of error counters which may provide useful information to the designer about the testability of the design and

provide feedback as to where test-points may be placed for non-BIT tests. A logfile associated with the BIT monitoring points might be of the type shown in Table 5.1. This file may be constructed simultaneously with the other logfiles, which record the values of specified signal points. When the signal values are written in those files, the simulator updates the status of the BIT points. If any of the BIT points have a valid flag a 1 is kept in the corresponding point in the array to depict that the BIT in that point has been able to detect the error.

Table 5.1. File Format for BIT Logfiles

	BIT_POINT1	BIT_POINT2	BIT_POINT3	BIT_POINT4
Fault1	1	0	0	0
Fault2	0	1	0	1
Fault3	0	1	0	0
Fault4	0	0	0	0

As can be seen from the table, all the faults except fault 4 are detectable at one or more of the BIT points. This implies that the used BIT scheme is not sufficient to detect all the errors due to the injected faults. However, it might be the case that the criticality detector finds that fault 4 was not critical according to the user's requirements. Then the system depicted above detects all the mission-critical faults (if any of the other faults are critical). Similarly, from the same table we see that if the fault set is limited to these 4 faults, then BIT_POINT 3 is redundant, because it does not detect any of the injected faults, and therefore it may be removed. Such tradeoffs can be explored after all the information has been gleaned and formatted for the criticality detector (comparator).

At this point, it is assumed that the critical errors have been identified, and a log of those errors has been kept during the simulation phase. While the simulation was being performed, a record was kept indicating where the critical errors occurred and whether they were detected by the BIT structures which were specified in the design. This can be thought of keeping a table of critical errors which has three entries in it:

- The injected fault, which resulted in a critical error.
- The value of the critical error.
- The BIT resource that detected the error.

It is possible that no BIT resource was able to detect a critical error. In that case, a new BIT resource has to be allocated if it is desired to detect all mission-critical failures (i.e., as in the case of INEWS system).

It must be understood that the recommendation here is to intertwine this step with the other phases because all the ground work for this phase is done during the simulation and the criticality analysis phase. It is during the simulation that all the data are collected to be collated afterwards. It is during the criticality analysis phase that the logfiles are generated. These logfiles are the ones on which the BIT evaluation phase will act and generate the measure of the efficacy of the BIT. It must be noted that, until now, only one measure of the efficacy of the BIT has been discussed, namely its capability to detect all the mission-critical resources. There can be other tradeoffs the designer might like the tool to evaluate. These will also be discussed here.

The structure suggested is shown in Figure 5.1.

5.2. BIT Measures

The initial impetus for the development of BIT techniques was to reduce the testing difficulties encountered during component production screening. As designs become more complex, with high logic to I/O ratios, the economic benefits of self-test become accentuated at both the chip and system levels. Although system-level, built-in self-test appears to be a promising solution to chip testing and system diagnosis problems, it is not without its drawbacks. Four primary disadvantages include:

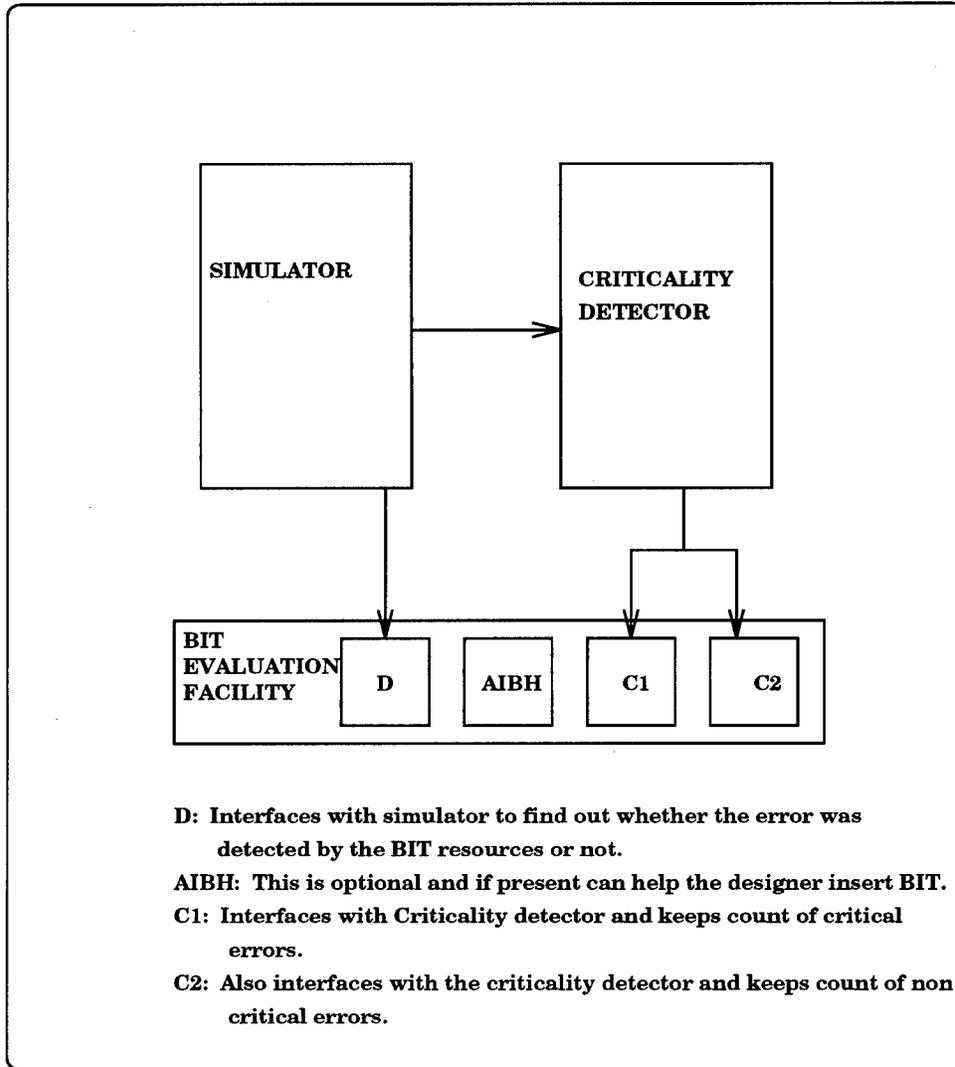


Figure 5.1. The BIT Evaluation Tool

1. Additional I/O pins and hardware overhead required.
2. Decreased reliability may be possible due to increased number of components in the design.
3. Negative performance impact due to additional circuitry (i.e., path delays through BIST components).
4. Additional design time and cost.

An obvious system design goal is to minimize the **negative impact** of BIST while maximizing its performance benefits. Traditionally, BIST approaches are evaluated with strict regards to the additional area overhead required to achieve a specified level of fault coverage. Because low area overhead is naturally preferred, the use of complex BIST techniques is generally restricted due to area minimization concerns. Since built-in self-testing approaches are being developed for the system level, this suggests the use of system-level performance metrics, in addition to the simple area overhead and fault coverage measures. The emphasis must be to have performance measures which capture the effects of BIST strategies on both system performance and reliability. By assessing the composite BIST enhanced system, optimization of the tradeoffs involved in a system design can be performed.

Test time is simply the time required to apply test patterns to a device under test. For general systems, test time will be a function of several parameters including: number of test patterns (test length), system complexity, and system test application frequency. **Fault coverage** refers to the number of faults detected compared to the number of faults assumed. The dependence of fault coverage on test length for random [56] and pseudorandom patterns [56,57] has been explored in the literature, using both heuristic and algorithmic methods.

Area overhead is a measure of the hardware cost of a BIST scheme. The BIST hardware overhead is generally expressed as a percentage. For an integrated circuit, the area overhead

is the percentage of silicon area occupied by the BIST circuitry compared to the total die area. Since device yield decrease with increasing die area, a small percentage overhead is preferable.

Measures appropriate for the evaluation of fault-tolerant systems include reliability, performance or reward, area utilization and **cumulative reward**. The reliability $R(t)$ of a system is defined as: "the conditional probability that the system has survived the interval $[0,t]$, given that it was operational at time $t=0$." This metric reflects the operational or mission lifetime of a system, and is inherently a function of the reliabilities of the system's components and their interconnect. Another performance measure of a system is what is referred to as **reward**, as in [80]. The definition of this metric is not unique; it is generally chosen to be the quantity that most adequately describes the performance of the system. For example, reward for a multiprocessor system may be chosen to be its computation capacity (e.g, instructions per second) as defined in [58].

Area utilization [59] accounts for additional hardware overhead (e.g., BIST circuitry) required by the system to implement its fault-tolerant features.

Prior to gracefully degrading systems, reliability measures and performance measures were evaluated separately from each other. However, with the advent of gracefully degrading systems, combined performance-related reliability measures have emerged [58]. It has since been suggested that the prior separate measures do not adequately describe gracefully degradable systems [60,61]. Consequently, other measures combining performance and reliability aspects have evolved and are termed "performability" measures. One such measure is referred to as cumulative reward. It is a function of time, and represents the total accumulated performance of a system over a given interval. Mathematically, cumulative reward is the integral of the instantaneous reward function. This measure is significant in that it provides additional information regarding the performance of a system. However, status concerning the completion of a job requiring the execution of N instructions, that was started at

time t , cannot be determined from the instantaneous reward function. However, cumulative reward reveals this information by providing the total accumulated number of instructions the system has executed from the time it began operation ($t = 0$) until time t .

In order to include the benefits of the cumulative reward measure, a new metric to quantify BIST system overhead, called cumulative area utilization, is proposed. This measure is defined as:

$$\text{cumulative area utilization} = \frac{CR}{TA}$$

where,

CR = Cumulative Reward

TA = Total Area

Thus, cumulative area utilization represents a more informative area measure analogous to cumulative reward, providing a more discerning performance measure.

Another metric that serves to unite self-testing and fault tolerant system performance measures is fault coverage. There are several types of fault coverage, depending on whether the designer is concerned with fault detection, fault location, fault containment, or fault recovery [83]. System fault coverage may be defined simply [62] as the percentage of all possible faults from which a system can recover. Thus, coverage probability is defined as the probability that a system recovers successfully, given that a fault has occurred. Coverage has been shown to be a sensitive parameter of system performance measures [62,63]. The coverage probability mainly depends on the fault coverage of the testing technique [59]. Thus, system coverage provides a critical link between the testing strategies of the BIST techniques, and the performance levels of a fault-tolerant system.

Based on the above brief discussion, it may be suggested that apart from the aspect of fault coverage (which was addressed by the first model presented in this chapter, Figure 5.1) there may be a need to evaluate the BIT resources from other angles as well. In Figure 5.1, it

is suggested that the tool should keep count of both the critical and non-critical errors. This is tantamount to tracing the fault coverage of the BIT resources inserted by the designer. That is why fault coverage is the only performance metric being employed in this default model. Efforts to use other performance measures, such as area utilization, would invariably need the description of the system, including for example, the number of extra pins, extra hardware, etc. So, in Figure 5.2 the possibility of the existence of such a model is indicated. One such model should be provided as a default (D in Figure 5.2), and the flexibility should be given to the user to interface an application specific model for BIT evaluation, if desired (M in Figure 5.2). The default model is proposed to have the metrics like cumulative area utilization, as suggested above. Also, if the user is using standard cell library, then a close estimate of hardware overhead can be incorporated. In that case, even the delays introduced by the BIT circuitry can be used to evaluate the *reward parameter*. Many such models already exist and can readily replace the default model.

Next, we look at techniques that help the designer plan his BIT resources.

5.3. Recommended BIT Insertion Techniques

In addition to the proposed BIT evaluation scheme, some representative ways of inserting BIT in hardware are now described. These may be built in the BIT evaluation tool to help the designer insert effective BIT for detecting mission-critical faults.

5.3.1. Survey of Automatic BIT Insertion Techniques

Several systems for automatic DFT have been reported in the literature. Of these, the two most relevant systems are briefly reviewed here. The first system is the Testable Design Expert System, or TDES [64]. The basic strategy of the system is to find a suitable DFT (Design For Test) scheme for each part of a circuit. These DFT schemes are called Testable Design Methodologies (TDMs). Examples of TDMs are Scan-Path, LSSD (Level Sensitive

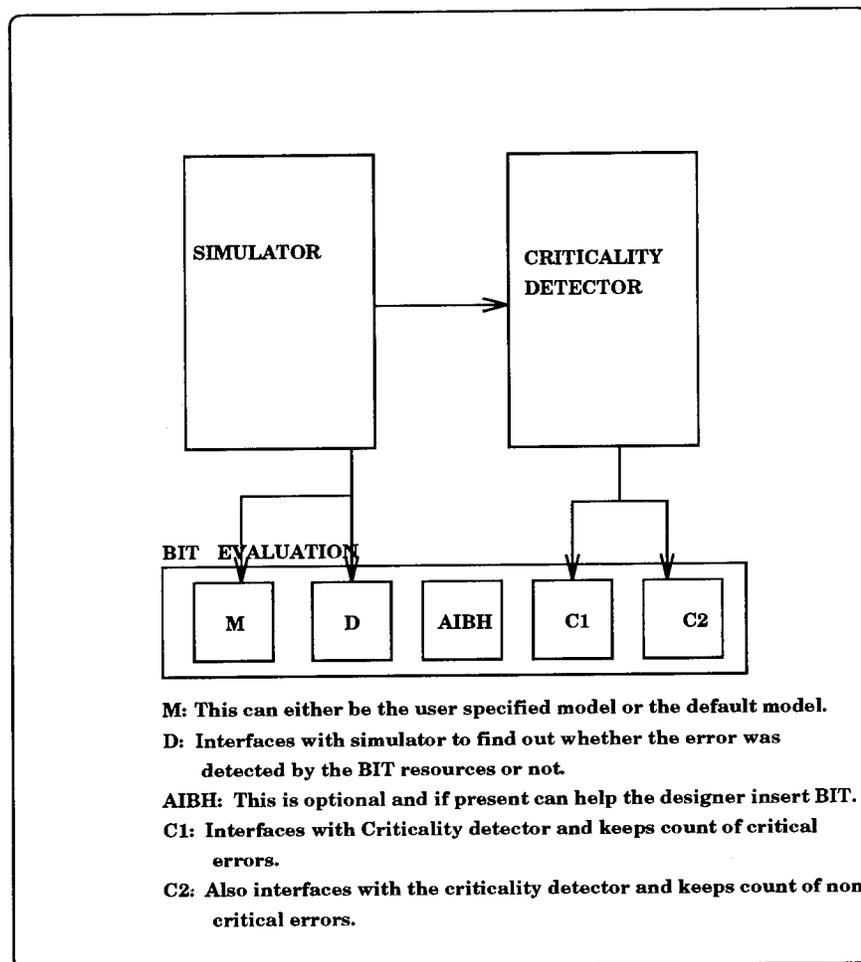


Figure 5.2. The BIT Evaluation Tool with Multiple Performance Measures

Scan Design), BILBO (Built-In Logic Block Observer), and several methods for testable design of PLAs (Programmable Logic Arrays). This system first partitions the circuit into several parts called "kernels." Then, an applicable TDM is sought for each kernel using a matching process. In the matching process, the I-path concept is used. An I-path is a path through which data can be transferred without any change. This I-path is used for identifying possible registers and paths for test pattern generation and response evaluation. If more than one TDM is applicable to a kernel, then TDM measures are used to choose between them. TDM measures are estimates of the costs associated with using a TDM. Frames are used for representing knowledge about TDMs. The TDES system has been implemented in LISP. A high level view of the configuration of this system is shown in Figure 5.3.

Another system proposed by Jones and Baker [66] uses AI techniques, including a rule-based system and planning, for high-level BIST design. This system relies heavily on BILBO-based BIST implementation. Insertion of BILBO is guided by constraints on hardware overhead and testing time, which are specified by the designer. The basic strategy employed by the system is to insert an appropriate number of BILBOs so that the given constraints on hardware overhead and testing time are satisfied. The system presents a range of test plans which meet the constraints imposed by the designer and achieve a balance between area penalty and test time. Each plan satisfies the constraints in different ways. From the set of possible test plans, the designer can select a suitable one that best matches the original design considerations. This system has been developed using the LOOPS [66] multi-paradigm programming environment which supports object-oriented, rule-based, procedural and access-oriented programming.

Although different AI techniques are used in the implementation of these two systems, there are some similarities between the two. First, both systems consider the insertion of BIT hardware at the register transfer level. Second, random pattern testing, based on the BILBO architecture, is used for the testing of Combinational Logic Blocks (CLBs). Third,

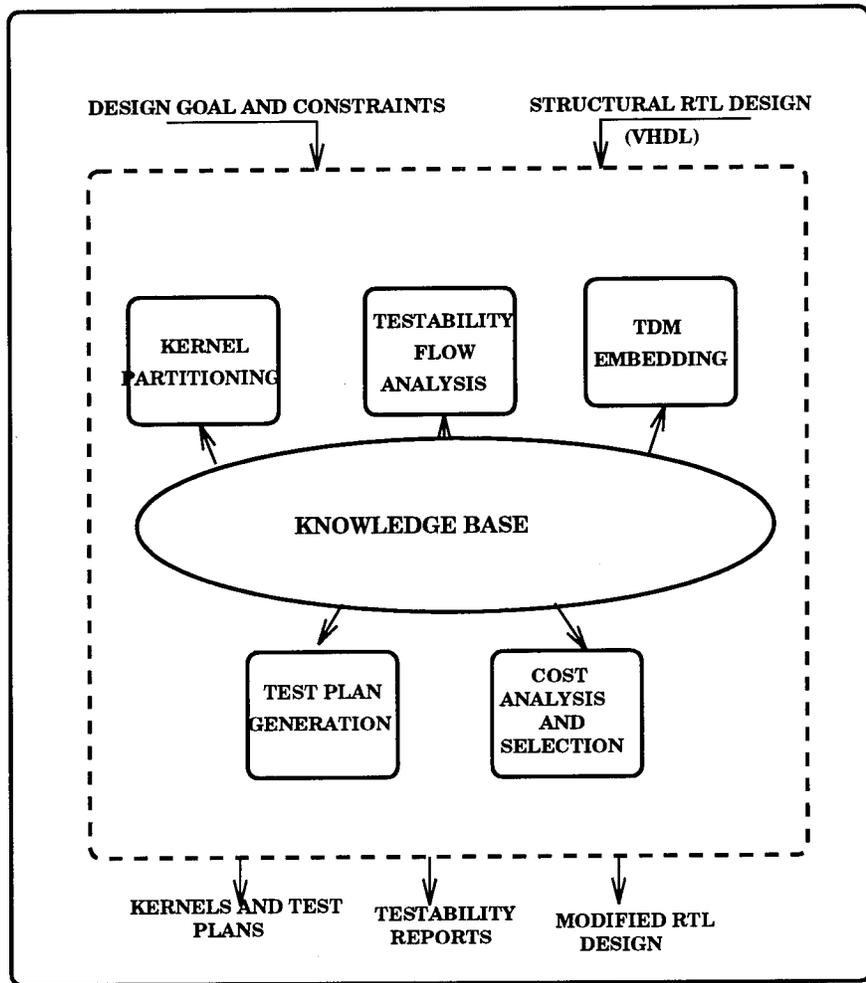


Figure 5.3. TIGER: Testability Insertion Guidance Tool

BILBO insertion is performed locally, rather than globally, in both systems. Neither of the two systems uses VHDL in the input description of the hardware. Since it has already been stated that VHDL is the preferred vehicle for modeling a design, it will be required to insert BIT resources in the VHDL model.

5.3.1.1. Tasks for Automatic Insertion of BIST Hardware

Many of the ideas used in the two systems described above can also be used in the proposed tool. However, a major difference is the use of VHDL. Given a VHDL description of a design, the structure of the design is extracted first. Then the tool should allocate testing resources such as Pseudo Random Pattern Generators (PRPGs) and Multiple Input Signature Registers (MISRs), to test CLBs in the design. Resource allocation may be followed by a scan-path organization. A scan-path can be used to collect signatures and to test sequential logic in the design. Upon finalizing a recommendation for test resource allocation, a test session for each CLB (combinational logic block) is scheduled, and control signals for distinguishing between several functional modes are distributed to the multi-functional registers in the design. In the following paragraphs, each of these tasks will be described in some detail.

To insert BIST circuitry into a design, the structure of the design needs to be extracted from the description of the hardware and represented internally in a usable database. Therefore, the first task performed should be the transition of VHDL description into a structural description.

Consider the case where a CLB is tested using BILBOs. A structural configuration, PRPG -> CLB -> MISR, should be constructed (the symbol -> implies an I-path between two entities). Here PRPG is the acronym for Pseudo Random Pattern Generators and MISR is an acronym for Multiple Input Signature Register. Since PRPGs and MISRs can be configured from existing registers, any available resource from the original design should be used. However, there may not be registers available which can form the required configuration, i.e.,

Register -> CLB -> Register. In this case, it may be necessary to add extra registers to the original design. Therefore, the second task of AIBH (Automatic Insertion of BIT Hardware) is the allocation of existing registers to act as PRPGs and MISRs for testing CLBs and indicating where hardware must be added to perform PRPG and MISR tasks that meet the test specifications for design.

This task may be followed by the formation of scan path that can facilitate the testing of sequential logic. In its simplest form, a scan path at a chip I/O can be used to facilitate a structural testing technique at the board level, namely boundary scan. This becomes significantly important due to the existence of standards that support a test interface and a bus protocol, most notably IEEE 1149.1.

The next task is test scheduling. Test scheduling is an arrangement of testing sessions so the total testing time can be minimized, given the fixed allocation of PRPGs and MISRs. The final task of AIBH is the distribution of register control signals. Since registers are used in several functional modes, control signals are needed for the proper configuration of registers in each test session.

In summary, the core tasks of AIBH should be the following:

1. Translation of VHDL description into structural information.
2. Allocation of PRPG and MISR for the chosen CLBs in a design.
3. Scan-path organization.
4. Test scheduling.
5. Distribution of register control signals.

In order to accomplish these tasks, the input description of a design must provide all of the necessary information. Details of a specific implementation of above mentioned concepts can be found in [65].

5.4. Tracing Critical Faults

Given a VHDL model of a hardware design of the system, it is easy for a parser to extract the relationship between various signals in the system. The motivation to do so arises from the fact that after identifying a critical fault, the designer may be interested in knowing how the faulty signal fans out to the modules driven by it. Also, tracing this signal back to other signals might be helpful for the designer.

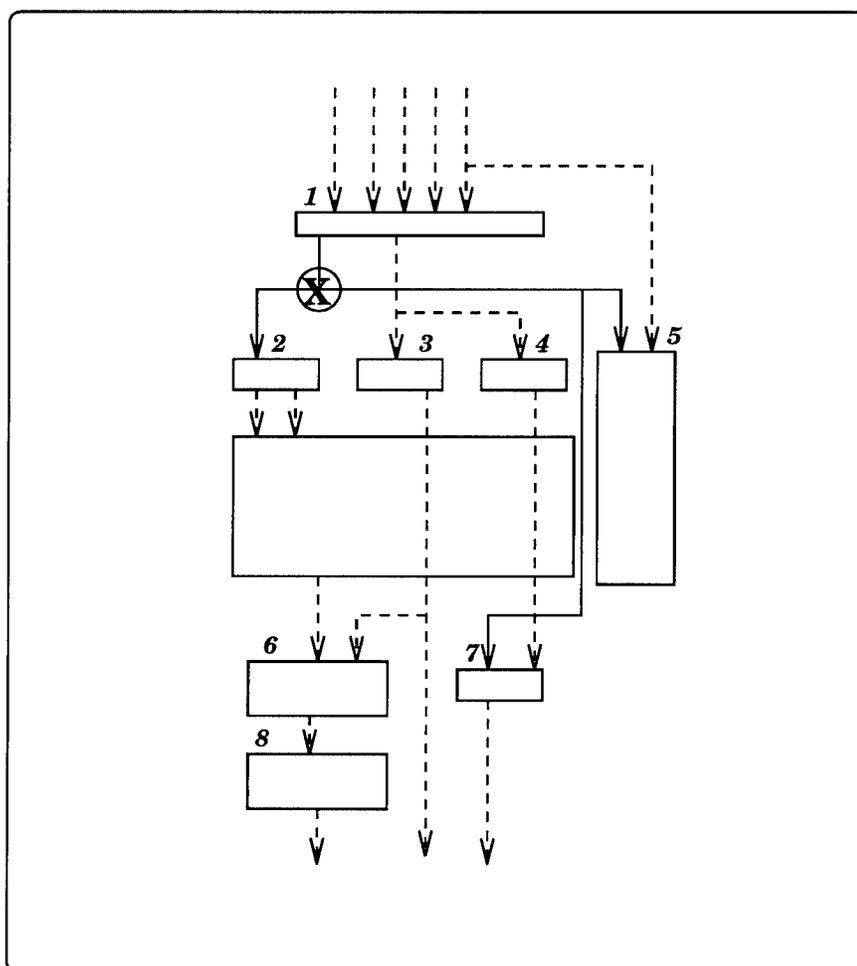


Figure 5.4. Tracing a Critical Fault

Figure 5.4 shows a system with eight modules. In this design example, the interconnec-

tions of various modules by different signals is shown with dotted arrows. Also shown is a critical fault that has been detected at the node marked by an X in a circle. The fanout from this point has been shown with the help of solid lines, so that it is obvious that the critical error can have an effect on a number of components. This is the effect the designer may want to be aware of. The parser can determine the fanout from the VHDL model and indicate to the designer (upon request) the modules affected by the occurrence of a critical fault that may be detected at any point in the circuit.

To generate fault diagnostics (used in tracing the error), many AI-based systems exist using *rules*. Each rule specifies a symptom and fault pair [69,70,71,72]. Such rules are created for each design. Given a set of symptoms, these rules give a set of possible faults for a particular design. As the complexity of system design increases, the rule-based approach becomes impractical [73,74]. Some modifications of these algorithms have been suggested to improve efficiency [75]. Some AI based algorithms for fault diagnosis use the stuck-at-fault model [70].

None of the above systems is able to perform fault diagnostics on VHDL descriptions of the system. In [67], a VHDL Fault Diagnosis Tool (VFDT) is described.

6. TOOLS

When laying down the guidelines for the implementation of the proposed tool set, an effort is made to identify some existing tools to perform required functions. Such existing tools, if incorporated in the proposed framework for the tool set, can lead to great savings in time and effort needed to implement the tool set.

In the previous chapters, a functional description of various tools were presented within the context of Quantitative FMEA (QFMEA). Here, issues regarding the implementation of various tools are discussed. The proposed tool set consists of tools implementing different functions in the QFMEA concept such as fault injection, simulation, criticality analysis, and BIT evaluation. Existing tools that can be used to perform one or more of these tasks include VHDL simulators, which exist in the market, e.g., [68]. Similarly, there have been efforts to design a system which performs fault diagnosis and simulation of systems whose architecture has been described in VHDL [67]. This system uses a VHDL simulator (VSIM), which uses a discrete-event-based compiled code simulation algorithm. To generate fault diagnostics (used in tracing the error), many AI-based systems exist using *rules*. Some of these tools can be directly incorporated in the proposed tool set and others can be used as models for implementation. The emphasis is on modifying the existing tools to fit into the framework of the proposed tool set. In general, the requirements for pre-existing tools to fit in are:

- The tool should have good performance in terms of reasonable runtime and functional correctness of the results it produces.
- The tool should fit in the overall structure. This means that it should be easy to interface it with other parts.

Tool functions which cannot be implemented with existing tools have to be developed as new tools. Also, software has to be developed to interface between the tools, and between the kernel and the various tools.

Various tools recommended to be included in the tool set are mentioned below:

- The Kernel.
- The VHDL Simulator.
- Fault Injector.
- Criticality Detector.
- BIT Evaluation Tool.
- Online Help and Tutorial.
- Enhancements and Options, User Interface Manual.

The detailed description of tools starts with the kernel which, as described is the framework where all the different subtasks take place, is the mode of communication between the user and the tool.

6.1. The Kernel

The kernel is the basic user interface and a common bus through which various parts of the tool set can communicate. The interface between the designer and the kernel can be implemented as a GUI (Graphical User Interface). Figure 6.1 shows one such example in which the GUI handles various phases of design cycle together with the proposed tool set. A GUI customized to meet the requirements of this tool will offer best performance, but a great saving in time and effort can be realized by modifying an existing GUI. The Design Manager tool, which is a part of the Falcon Framework by Mentor Graphics, is recommended as a vehicle for the GUI implementation. The Falcon Framework also offers some other distinct advantages:

- The Falcon Framework provides a user interface language AMPLE [76]. With the help of this facility, it is easy to interface various tools with the GUI provided by the Falcon Framework.
- Since the proposed tool set will be used as part of a design cycle, which may be depicted as in Figure 6.2, the proposed framework should be able to support other tools (such as schematic editor, simulation and synthesis tool etc.), in addition to QFMEA tool set. This requirement is also fulfilled by the Falcon Framework.

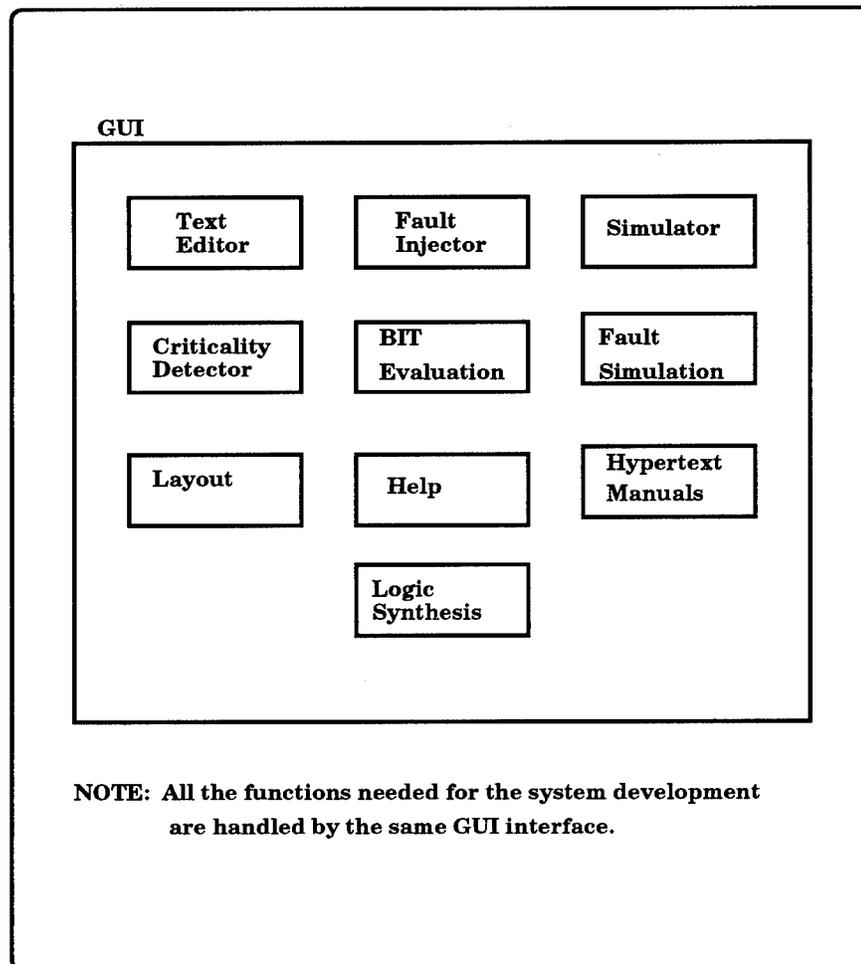


Figure 6.1. Graphical Interface Incorporating the Proposed Tool Set in the Design Cycle

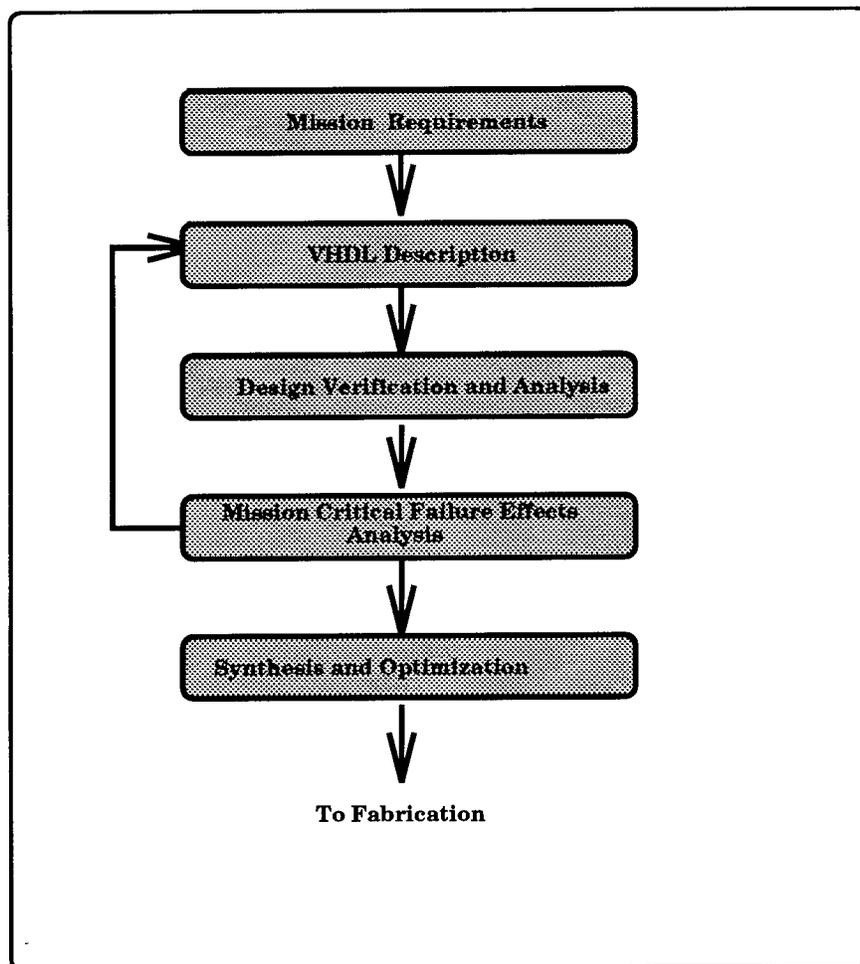


Figure 6.2. The Design Cycle

An integrated tool set is recommended because it allows the process of identifying mission-critical faults, and designing BIT capable of detecting these faults, to be viewed by the designer as part of the overall design process.

The Falcon Framework also features a number of tools that can be used to provide some of the data needed to perform the required QFMEA functions. Such tools are the VHDL simulator (System 1076), the design capture (Design Architect), and the model databases supporting these tools (e.g., VHDL models from Logic Modeling Corporation).

6.2. Queries

This section describes the information which the user should pass to the tool set through the GUI (recommended above). The designer has to specify some inputs in addition to a VHDL model. He should be able to specify these inputs (discussed below) interactively with the help of the GUI. However, for a large system it might be preferable to place such inputs in files which the system can read. These additional inputs are described next.

Along with the VHDL description the designer should provide various system attributes to be used by the Criticality Detection tool (Section 4.4). Since an output signal can be tested for criticality based on various parameters (such as RMS value or average value) and outputs can have different characteristics (such as synchronous or asynchronous), these attribute specifications must be made by the user. This will help the system to store the required values in the required format while the simulation is in progress. The attributes are listed below:

- Sampling period (i.e., the basic unit of simulation time).
- If individual outputs, or a combination of outputs is to be traced for criticality.
- Whether the system has rollback points or not.

- Whether the system needs to trace only critical faults according to the criticality criterion, or trace all faults.
- Whether the system is synchronous or asynchronous.
- Tolerance value before a fault effect becomes critical.

In addition to the attribute specification, the tool set may need to know whether the user wishes to provide specifications interactively or through the default files.

A list of recommended queries which the designer needs to resolve are listed in Table 6.1.

Table 6.1. Recommended Set of Queries

Query	User chooses from	Description
Outputs	Outputs to be monitored	The user specifies outputs to be traced
Tolerance	Tolerance value per output	User states what is critical
Tolerance Mode	RMS or average or default	Default is per sample
Synchronous	Yes or No	If yes, clock period is sampling period
Sampling Period	sampling period	Simulator knows when to store results
Output Combination	Output combination	If combination of outputs to be traced
Rollback Present	Yes or No	If rollback, tool defines sampling period
Warning	Yes or No	Warnings flagged or not
Simulator Options	As described	-
Abort	Abort or continue	Abort at instance of first critical error

It is possible that conflicts and/or omissions might exist between various user specifications. Such conflicts have to be reported back to the user. Some examples of potential conflicts are listed below:

- **Conflict 1:** If the user defines that rollback is present in the system and fails to define the “rollback point” (i.e., a bit that flags when the simulation of the system rolls back). If such a bit or an equivalent mechanism is not present, the tool has no means of keeping the faulty and fault-free simulation values in synchronization while comparing them.

- **Conflict 2:** Another conflict might arise if the user defines a signal as synchronous but fails to specify the signal (typically the clock signal) with which it is synchronous. The default can be the global clock for all such signals (provided the clock is specified by the user).
- **Conflict 3:** If the user wants a specified output or a combination to be traced on the basis of RMS or average value but fails to specify the time interval over which such operations are to be carried out.
- **Conflict 4:** If the tool is to flag warnings, then it needs to know what range of error values constitutes critical system behavior and what range should be flagged as a warning (i.e., depending on whether the warnings are to be flagged or not, the format of tolerance will be different). Failure to specify the values in the right format will lead to a conflict being reported.

The above set of queries were related mostly to the criticality analysis. An additional set of queries is related to the fault injection operation. The tool should provide the user with various options regarding which faults to simulate. These options should include as a minimum:

- The user can let the tool access the reliability and component failure database, and determine a fault list to be simulated, based on a selection procedure to be developed.
- The user may explicitly specify the faults (corresponding to the fault model being used) to be simulated.
- The user may use a statistical generation and mapping of faults, by specifying distributions of fault occurrence for the different components or the system.
- A combination of the above alternatives.

In any case, the user has to specify the number of faults, *num_faults*, that need to be simulated.

There are also queries relating to BIT evaluation. The system designer needs to specify BIT allocation points, which are the points at which the designer intends to insert BIT into the system. These BIT points are defined in the VHDL model of the design, along with a functional model of the BIT technique used to monitor these points. The user in this case has to indicate:

- The explicit location of the BIT allocation points.
- How the BIT resources allocated at these points detect and report the faults.

Indication of the BIT point can be done in the VHDL code. The resulting VHDL code should be compatible with standard VHDL. So the indication can be done inside the VHDL comments if needed. The VHDL parser can, thus, identify the BIT points and the tool can allocate a global BIT counter for each of the BIT monitoring points. Such counters keep track of the faults which are detected at a corresponding BIT allocation point. With the help of these counters, a detectability table (also described earlier, in Chapter 5) can be constructed by the BIT evaluation tool. An issue that needs to be resolved is; how a BIT resource (inserted in VHDL code by the user) will communicate to the tool that a fault has been detected. Again, this difficulty can be surmounted by letting the user decide what constitutes detection of a fault (which varies with the BIT strategy employed). When such a condition occurs, the user specifies (in the VHDL code inside the comments) *BIT_flag* <= 1. The tool can interpret this as a sign that an injected error has been detected. For example, suppose the original VHDL code is the following:

```
architecture rtl of shf10 is
    signal pre_Q : qsim_state_vector(9 downto 0) := (OTHERS => 'X');
```

```

begin
  SHIFT_REGISTER_Process: process(CLK)
  begin
    if (CLK'event and (CLK = '1') and (CLK'last_value = '0')) then
      if (LD = '1') then
        pre_Q <= DIN;
      elsif (SE = '1') then
        pre_Q(9) <= '0';
        pre_Q(8 downto 0) <= pre_Q(9 downto 1);
      end if;
    end if;
  end process SHIFT_REGISTER_Process;
  DOUT <= pre_Q;
end rtl;
}

```

If LD = '1' corresponds to the detection of a fault, the code which the user should input is:

architecture rtl of shf10 is

```

  signal pre_Q : qsim_state_vector(9 downto 0) := (OTHERS => 'X');
begin
  SHIFT_REGISTER_Process: process(CLK)
  begin
    if (CLK'event and (CLK = '1') and (CLK'last_value = '0')) then
      if (LD = '1') then  ---** BIT_flag <= 1;
        pre_Q <= DIN;
      elsif (SE = '1') then

```

```

        pre_Q(9) <= '0';
        pre_Q(8 downto 0) <= pre_Q(9 downto 1);
    end if;
end if;
end process SHIFT_REGISTER_Process;
DOUT <= pre_Q;
end rtl;

```

This code assumes that '**' after '-' for comments is used to hide information in these comments. These are the comments the tool looks for when defining the Global counters for BIT evaluation.

Presence of BIST resources can also be indicated in any of the following manners:

- An executable statement that calls a logging routine that, itself, turns logging on and off.
- An executable statement that is easily commented out when not needed. In this strategy, there may be a need to recompile and edit the code each time the designer wants to turn the BIST mechanism on or off.

All this queries can be conveniently incorporated in the Design Manager software in the Mentor Falcon Framework so these functions can be easily implemented; this is another advantage of using a standard GUI.

6.3. Simulator

After the VHDL code describing the design has been compiled, the fault injection takes place and this design is simulated for determining the criticality of fault effects based on the observed error quantities. Both the compilation and simulation can be performed by

tools provided as a part of the Falcon Framework. Compilation of code can be performed by *hdl*, the VHDL compiler. QuickSimII [77] can be used to do the required simulation. The design can be captured, in this case, using Design Architect. QuickSimII works with the Design manager and offers a number of capabilities and options that can be very useful for the purpose of the proposed tool set.

- Timing Mode of simulation can be minimum, maximum, or typical among others. This facility provides greater flexibility to the system designer.
- Constraint Mode can be off or on for messages. As already mentioned, this is a recommended feature in the proposed tool set (i.e., the ability to flag not only the critical faults but also the potentially critical faults based on the designer's need).
- Delay Model can be inertial or transport (the two methods defined in VHDL). This lets the designer create a VHDL model which better suits his need.

Since this system supports the standard VHDL (i.e., System 1076) as well as Mentor VHDL, the user has a greater choice of module libraries and components. Tracing, listing and viewing individual signals, viewing the progression of signal values over time, running the simulation and providing stimulus are easily done in the framework of QuickSimII.

It must be noted, that recommending the use of the Design Manager tool and the QuickSimII has been made with interfacing capabilities in mind. Mentor Graphics provides user interface functions for all of its higher-end tools such as QuickSimII and Design Manager. These are part of the AMPLE [76] procedure calls. With the help of these calls, the user is able to suitably modify the interfaces and build applications around these tools. As mentioned earlier one of our objectives was that the interface for the usable tools be modifiable to fit the needs of QFMEA tool set.

The simulator has to interface both with the kernel and with the fault injection tool. Interface with the fault injector is needed because the fault injector identifies the fault to be

simulated, generates the appropriate mask value in the “superentity model,” and calls the simulator. This function cannot be performed with a fault simulator resident in the Falcon Framework.

6.4. Fault Injector

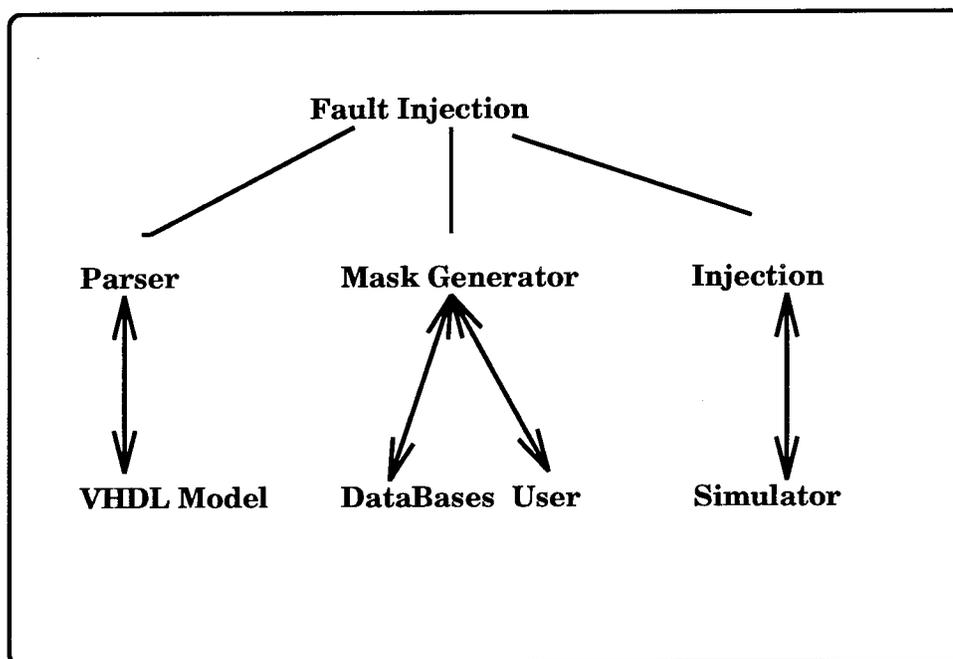


Figure 6.3. Parts in Fault Injector

The fault injection module (Figure 6.3) determines which faults, out of the defined fault population, should be simulated, based on the user-specified selection approach as discussed previously. This tool has to interface with the component reliability database and examine the structure of the design to select a subset of the fault population. The software to be developed for the Fault Injector should have the following components:

- Parser: To parse the VHDL entities and generate the masked entities called *superentities*.

- **Mask Generator:** Controls the mask values depending on the selected faults from the total fault set. This is the module which interfaces with the reliability database and the user to weigh the faults and, thus, form a prioritized subset of faults.
- **Injector:** A function which calls the simulator with the mask appropriate for the fault to be simulated.

Fault injection requires that each of the VHDL entities used in the original design be expanded into the *superentities*. These *superentities* are the original entities with a mask around them. This mask is useful to regulate the values at the output or input of these entities. If the stuck-at fault model is used, then temporary or permanent faults can be generated. This process will be handled by the fault injector, which needs to be developed.

All these parts of the fault injector have to be developed and should be compatible with the Mentor Graphics tools that have been recommended.

6.5. Criticality Detector

The purpose of this stage is to determine whether the responses of the faulty system, which have been collected during fault injection, are critical or not. In a typical scenario, the designer provides a sequence of inputs to the simulated system. The response of the fault-free system to this sequence is then recorded by the tool. Thereafter, the faults chosen are injected and the response of the faulty system is also stored for a comparison. At the end of the run (i.e., when all the faults have been simulated), the criticality detector is evoked.

The criticality detector (Figure 6.4) is a comparator which uses the files corresponding to the faulty and fault-free responses and compares the corresponding entries. Such a simplistic view for this stage can be afforded, provided all groundwork involved in the queries which let the simulator store the response data in a convenient format for comparison has been completed correctly.

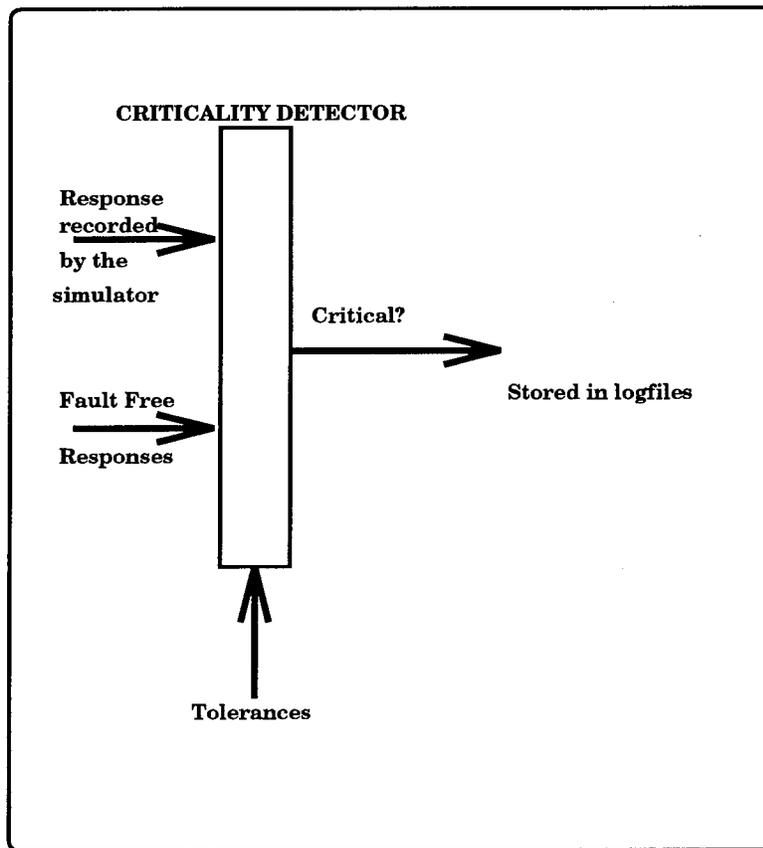


Figure 6.4. Criticality Detector

The comparison needs to be done only for the outputs which have been identified previously (as a part of the query stage). The responses are stored only for such nodes. The comparison takes into account the tolerances which have been specified by the user, records whether the fault is critical or not, and in which interval or sampling period the fault became critical. Such a record is kept in logfiles which serve two purposes:

- To inform the user of the time instances when critical errors occurred.
- For BIT evaluation, which is discussed next.

Since all the results of simulation are being retrieved in this stage, some work related to BIT evaluation can be completed. Whenever a critical fault is identified, all the global counters associated with the BIT monitoring points, where BIT resource are located, are checked to determine if the fault has been detected. The status of all global BIT counters is updated each time a critical fault is identified.

6.6. BIT Evaluation Tool

All the data needed for the BIT Evaluation Tool are collected in the criticality detection phase. The function performed at this stage is to use these data and make decisions about the efficacy of BIT resources in the system.

The methodology for this phase has been elaborated in the chapter on BIST evaluation. For this level, all the software has to be developed. This tool will read the logfiles mentioned above and the values of the global counters, and will collate these responses. The purpose of doing this is to create a table in the format shown in Table 6.2.

This table is a summary of the activity at the monitoring points where BIT resources are allocated, and also a report of how effective these points and their associated BIT resources have been in detecting mission-critical faults. This table is used as a feedback to the

Table 6.2. File Format for BIT Logfiles

	BIT_POINT1	BIT_POINT2	BIT_POINT3	BIT_POINT4
Fault1	1	0	0	0
Fault2	0	1	0	1
Fault3	0	1	0	0
Fault4	0	0	0	0

designer, to inform him of the effectiveness of the BIT resources employed. This task is relatively simple in terms of software development. However, since a subset of faults and fault combinations will likely be used for a complex design, the BIT assessment has to use some statistical post-processing of the data to produce confidence levels for the BIT effectiveness resulted from a chosen fault injection experiment.

6.7. Online Help and Tutorial

While designing a system, simulating it, injecting faults, or for that matter in any other design activity, the user may need help with different facets of the tools. A help facility is recommended for the tool set and it can be arranged in several levels.

6.7.1. Level 1

In previous paragraphs, the Falcon Framework was proposed to be used as a basis for developing the kernel. The proposal to use the Falcon Framework as the overall framework in which the QFMEA tool set takes into consideration the ability to use the resident help facility, which can be easily accessed at any point as the user interacts with the framework. This help facility should be augmented to provide online help for the QFMEA tool set. The help can be organized in different ways. At the very outset (i.e., when the designer presents his VHDL design to the QFMEA tool), the designer must be able to look at the *tool summary and functions*. This is the first level of the help function, and should necessarily be kept

simple. This level is for a novice user of the tool, and should serve the following functions:

- Introduction to the tool set as a whole.
- Introduction to the tool structure.
- Introduction to the individual tools, their functions, and recommended sequence of use.

6.7.1.1. Tutorial

At the first level, the user should be able to go through a brief tutorial, which will illustrate, through an example, how the tool works. A relatively simple system can be selected to illustrate the functioning of the proposed tool set. As a first step, mission requirements should be clarified. The user is then notified of the fault model being used and the possible fault set for the chosen system. Then, a preset database is used to show the user how the tool selects faults for injection. Also, the user must be made to realize how those outputs whose criticality is to be examined can be chosen, and how to specify criticality requirements for those outputs. The tool then proceeds to simulate a user-defined set of faults or the faults injected by the tools, and shows which are found to be critical.

After identifying these mission-critical faults, the tutorial should point out to the user how the BIT resources perform versus the specifications. Poor BIT performance can be highlighted by identifying some mission-critical faults which the BIT resources have not been able to detect. At this point, one iteration of the use of QFMEA tool may be complete and the next starts after the tool shows a possible modification of the BIT resources and goes through the cycle again, which results in a better BIT performance.

6.7.2. Level 2

The second level of help is the actual online help when the user is actually using the tool. At all points in a tool session, the user must have help available, so this level is distributed over individual tools in the tool set. At all times, the tool keeps track of what the user is doing and provides relevant help. This level is incorporated in the Falcon Framework and can be expanded to include help on various features of the QFMEA tool set.

6.8. Enhancements and Options, User Interface Manual

Throughout this report, it has been pointed out that the user might be inclined to use features different from the ones provided by default. The main example can be that of the fault model being used by the tool. Although the stuck-at model seems to be the most convenient choice, and is the one recommended for this tool, the user may prefer to use an alternative model. Also, there may be enhancements over the default models which the user may want to incorporate in the tool. The idea being emphasized here is that of flexibility of the tool. The tool developer must define the interfaces to all tools and produce a user interface manual.

Wherever alternate options are allowed, a comprehensive way of realizing those options must be provided to be used by either the user or a third party. In any case, the user interface manual must list and explain all the submodules that can help a user in enhancing the capabilities of the tool, and modifying them to tailor his needs. Also, in this manual all the known bugs of the designed system, the error messages and corresponding remedies must be listed.

References

- [1] H.B. Dussault, "The Evolution and Practical Applications of Failure Modes and Effect Analyses," RADC-TR-83-72, March 1983.
- [2] G. Babour, "Failure Mode and Effect Analysis by Matrix Method," *1977 Proceedings Annual Reliability and Maintainability Symposium*, January 1977, pp 114-119.
- [3] S. Herrin, "System Interface FMEA by Matrix Method," *1982 Proceedings Annual Reliability and Maintainability Symposium*, January 1982, pp 111-116.
- [4] H. Ohlef, W. Binroth, and R. Haboush, "A Bayesian Approach to a Failure Mode Effects Analysis," Technical Report, BRL/TR-76-8161, Bendix Research Laboratories, Southfield, MI, September 1976.
- [5] H. Ohlef, "A Computer System Specification for a Bayesian Failure Mode Effect Analysis Technical Report," BRL/TR-76-8186, Bendix Research Laboratories, Southfield, MI, September 1976.
- [6] F. Hedin, A. LeCoguiec, L. LeFloch, M. Llory, and A. Vilemeur, "The Failure Combination Method," *1981 Proceedings Annual Reliability and Maintainability Symposium*, pp 163-172.
- [7] W. L. Bunce, "Hardware and Software: An Analytical Approach," *1980 Annual Reliability and Maintainability Symposium*, pp 209-213.
- [8] M. J. Legg, "Computerized Approach for Matrix-Form FMEA," *IEEE Transactions on Reliability*, Vol. R-27, No. 4, October 1978, pp 254-257.
- [9] P. L. Goddard, and R. Davis, "Automated FMEA Techniques," RADC-TR-84-244, December 1984.

- [10] R. J. Schroder, "Fault Trees for Reliability Analysis," *1970 Proceedings Annual Reliability and Maintainability Symposium*, pp 198.
- [11] J. B. Russell, G. J. Powers, and R. G. Bennetts, "Fault Trees - A State of the Art Discussion," *IEEE Transactions on Reliability*, Vol. R-23, No. 1, April 1979, pp 51.
- [12] R. C. Clardy, "Sneak Circuit Analysis, Reliability and Maintainability of Electronic Systems," Arsenault and Roberts, ed., Computer Science Press, 1980, pp 223-41.
- [13] D. L. Burratti, and S. G. Goody, "Sneak Analysis Application Guidelines, RADC-TR-82-179, June 1982.
- [14] B. Tiger, "Evaluating System States in Their Mission Phases," *1969 Proceedings Annual Reliability and Maintainability Symposium*, pp 18-26.
- [15] T. W. Yellman, "Event-Sequence Analysis," *1975 Proceedings Annual Reliability and Maintainability Symposium*, pp 286-291.
- [16] R. K. Thatcher, J. L. Easterday, and F. R. Raylor, "An Integrated Predictor of System Reliability," *Proceedings Annual Reliability and Maintainability Symposium*, pp 254-260.
- [17] F. Tuma, "Software/Hardware Integrated Critical Path Analysis (ICPA)," *1980 Proceedings Annual Reliability and Maintainability Symposium*, pp 384-387.
- [18] H. B. Dussault, "Automated FMEA - Status and Future," *1984 Proceedings Annual Reliability and Maintainability Symposium*, pp 1-5.
- [19] J. H. Wensley, et al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings of IEEE*, Vol. 66, No. 10, October 1978, pp 1240-1255.
- [20] Kenyon, R. L., and R. J. Newell, "FMEA Techniques for Microcomputer Assemblies," *1982 Proc. of Annual Reliability and Maintainability Symposium*.

- [21] Chandler G., et al, "Failure Modes/Mechanism Distribution." Reliability Analysis Center, Griffiss AFB, NY.
- [22] Dhillon, B. S., "FMEA Analysis-Bibliography, Microelectronics and Reliability," Vol. 32, No. 5, pp 719-31.
- [23] Iyer, R. K., and G. S. Choi, "Experimental Validation of Real Time Fault-Tolerant Systems," NASA-CR-190985, October 1992.
- [24] Monaghan T., "The Concept Exploration Stage of a Dependable Avionic System," *IEEE/AIAA Digital Avionics Systems Conference*, Los Angeles, CA.
- [25] Hesterberg, T. C. "Advances in Importance Sampling," Ph.D. dissertation, Department of Statistics, Stanford University, 1988.
- [26] Hammersley, J. M., and Hanscomb, D. C., *Monte Carlo Methods*, London: Methuen, 1964.
- [27] Goyal A., Heidelberger P., and Shahabuddin P., " Measure Specific Dynamic Importance Sampling for Availability Simulations," *Proceedings of the 1987 Winter Simulation Conference*, pp 351-357, 1987.
- [28] Beckman, R. J., and McKay, M. D., "Monte Carlo Estimation Under Different Distributions Using the Same Simulation," *Technometrics* 29, pp 153-160, 1987.
- [29] Booth, T. E., "A Monte Carlo Learning/Biasing Experiment with Intelligent Random Numbers," *Nuclear Science and Engineering* 92, pp 465-81, 1986.
- [30] Conway, A. E., and Goyal, A., "Monte Carlo Simulation of Computer System Availability/Reliability Models," *Proceedings of the Seventeenth Symposium on Fault-Tolerant Computing*, Pittsburgh, PA, pp 230-235, 1987.

- [31] Hesterberg, T. C., "Importance Sampling in Multivariate Problems," *Proceedings of the Statistical Computing Section, American Statistical Association 1987 Meeting*, pp 412-417, 1987.
- [32] Hopmans, A., and Kleijnen, J. P. C., "Importance Sampling in Systems Simulation: a Practical Failure?" *Mathematics and Computers in Simulation* 21, pp 209-220, 1979.
- [33] Kahn, H., "Nucleonics," 6(5), 27-37 and 6(6) pp 60-65, 1950.
- [34] Kahn, H., and Marshall, A. W., "Methods of Reducing Sample Size in Monte Carlo Computations," *Journal of the Operations Research Society of America*, 1, pp 263-278, 1953.
- [35] Kioussis, L. C., and Miller, D. R., "An Importance Sampling Scheme for Simulating the Degradation and Failure of Complex Systems During Finite Missions," *Proceedings of the 1983 Winter Simulation Conference*, pp 631-639, 1983.
- [36] Kloek, T., and Van Dijk, H. K., "Bayesian Estimates of Equation System Parameters: An Application of Integration by Monte Carlo *Econometrica*," 46(1) pp 1-19, 1953.
- [37] Moy, W. A., "Sampling Techniques for Increasing the Efficiency of Simulations of Queuing Systems," Ph.D. dissertation, Industrial Engineering and Management Science, Northwestern University, 1965.
- [38] Murthy, K. P. N., and Indira, R., "Analytical Results of Variance Reduction Characteristics of Biased Monte Carlo for Deep-Penetration Problems," *Nuclear Science and Engineering* 92, pp 482-487, 1986.
- [39] Siegmund, D., "Importance Sampling in the Monte Carlo Study of Sequential Tests," *The Annals of Statistics* 4, pp 673-684, 1976.

- [40] Stewart L., "Multiparameter Univariate Bayesian Analysis," *Journal of the American Statistical Association*, 74, pp 684-693, 1979.
- [41] Stewart L., "Multiparameter Bayesian Inference Using Monte Carlo Integration Q Some Techniques for Bivariate Analysis," *Bayesian Statistics 2*, eds J.B. Bernardo, M. H. DeGroot, D. V. Lindley, A. F. M. Smith, Elsevier Science Publishers B.V. (North-Holland), pp 495-510, 1983.
- [42] Therneau, T. M., "Variance Reduction Techniques for the Bootstrap," Technical Report No. 200, (Ph.D. Thesis) Department of Statistics, Stanford University, 1983.
- [43] Tukey, J. W., "Configural Polysampling," *SIAM REVIEW* 29, pp 1-20, 1987.
- [44] Wilson, J. R., "Variance Reduction Techniques for Digital Simulation," *American Journal of Mathematical and Management Sciences* 4, pp 277-312, 1984.
- [45] Danner, F., and W. Consolla, "An Objective Testability Rating System. *Proc. of 1979 IEEE Test Conf.* pp 23-28, Cherry Hill, NJ, 1979.
- [46] Takasaki S., M. Kawai, S. Funatsu, and A. Yamada, "A Calculus of Testability Measures at the Functional Level," *Proc. of 1981 IEEE Test Conf.* pp 95-101, 1981.
- [47] Z. Segall, D. Vrsalovic et al., "FIAT-Fault Injection Based Automated Testing Environment," *Proc. FTCS-18*, Tokyo, pp 102-107, 1988.
- [48] "Fault Simulation at the Architectural Level," *Proc. of 1984 ITC*, IEEE, pp 669-679, 1984.
- [49] "A Simulation Approach for Computing System Reliability," *Microelec. Reliability* Vol. 27, No. 3, pp 463-467, 1987.
- [50] Padilla P. A., "Fault Recovery Characteristics of the Fault Tolerant Microprocessor," *IEEE/AIAA/NASA Digital Avionics Conference*, 9th Virginia Beach, 1990.

- [51] Vaanmoorsel, AAD P.A., Haverkort, Boudewijn R., Niemgeers, I. G., "Fault Infection Simulation: A Variance Reduction Technique for Systems with Rare Events," *2nd Intl. Working Conf. on Dependable Computing for Critical Appln.*, Tucson, AZ, 1991.
- [52] Choi, G. S., Iyer, R. K., Carreno V. A., "Simulated Fault Injection - A Methodology to Evaluate Fault Tolerant Microprocessor Architectures," *IEEE Trans. on Reliability*, Vol. 39, pp 486-491, October 1990.
- [53] Barton, J. H., Czeck, E. W., and Segall Z. Z., Siewiorek, D. P., "Fault Injection Experiments Using FIAT," *IEEE Trans. on Computers*, Vol. 39, pp 575-82, April 1990.
- [54] Baker, R. L., Magnum, L. S., and Scheper, C. O., "A Simulation-Based Fault Injection Experiment to Evaluate Self Test Diagnostics for a Fault Tolerant Computer," *AIAA/IEEE Digital Avionics Conf.*, San Jose, CA, pp 220-226, 1988.
- [55] Hummel, R. A., "Automated Fault Injection for Digital Systems," *Annual Reliability and Maintainability Symposium*, Los Angeles, CA, pp. 112-117, 1988.
- [56] Chin, C. K., and McCluskey, E. J., "Test Length for Pseudorandom Testing," *IEEE Trans. Computers*, Vol. C-36, No. 2, pp 252-256, February, 1987.
- [57] Williams, T. W., "Test Length in a Self Testing Environment," *IEEE Design and Test*, Vol.2., pp 59-63, April 1985.
- [58] Beaudry, M. D., "Performance Related Reliability Measures for Computing Systems," *IEEE Trans. Comput.*, Vol. C-27, pp 540-547, June 1978.
- [59] Koren, I., and Breur, M. A., "On Area and Yield Considerations for Fault Tolerant VLSI Processor Arrays," *IEEE Trans. Comput.*, Vol. C-33, No. 1, pp 21-27, January 1984.

- [60] Meyer, J. F., "On Evaluating the Performability of Degradable Computing Systems," *IEEE Trans. Comput.*, Vol. C-29, No. 8, August 1980.
- [61] Meyer, J. F., "Closed Form Solutions of Performability," *IEEE Trans. Comput.*, Vol. C-31, No. 7, pp 648-657, July 1982.
- [62] Arnold, T. F., "The Concept of Coverage and its Effect on the Reliability Model of a Repairable System," *IEEE Trans. Comput.*, Vol. C-22, pp 251-254, March 1987.
- [63] McGough, J., et al., "The Conservativeness of Reliability Estimates Based on Instantaneous Coverage," *IEEE Trans. Comput.*, Vol. C-34, pp 602-609, July 1985.
- [64] M. Abadir, J. Newman, D. D'Souza, and S. Spencer, "Partitioning Hierarchical Designs For Testability," *International Test Conf.* pp 174-183, 1991.
- [65] K. Kim, J. G. Trent, and D. S. Ha, "Automatic Insertion of BIST Hardware Using VHDL," *25th ACM/IEEE Design Automation Conference*, pp 9-15, 1988.
- [66] M. A. Jones, and K. Baker, "An Intelligent Knowledge Based System Tool for High Level BIST Design," *Proc. IEEE International Test Conf.*, pp 743-746, 1985.
- [67] V. Pitchumani, P. Mayor and N. Radia, "A System for Fault Diagnosis and Simulation of VHDL Descriptions," *28th ACM/IEEE Design Automation Conference*, pp 144-150, 1991.
- [68] QuickSimII User's Manual, Mentor Graphics Corporation, 1993, 8005 S.W.Boeckman Road, Wilsonville, Oregon 97070.
- [69] R. T. Hartley, "CRIB: Computer Fault Finding Through Knowledge Engineering," *IEEE Computer*, pp 76-83, March 1984.
- [70] L. Apfelbaum, "An Expert System for In-Circuit Fault Diagnosis," *Proceedings of Int. Test Conf.*, pp 868-874, 1985.

- [71] O. Grillmeyer, and A. J. Wilkinson, "The Design and Construction of a Rule Base and an Inference Engine for Test System Diagnosis," *Proceedings of Int. Test Conf.*, pp 857-867, 1985.
- [72] "Self Diagnostics on System Level By Design," *Proceedings of Int. Test Conf.*, pp 1921-1927, 1986.
- [73] M. R. Genesereth, "The Use of Design Descriptions in Automated Diagnosis," *Artificial Intelligence*, pp 347-40, December 1984.
- [74] R. Davis, "Diagnostic Reasoning Based on Behavior and Structure," *Artificial Intelligence*, pp 347-410, Dec. 1984.
- [75] "Diagnostic Reasoning in Digital Systems," *Proceedings of the 18th International Symposium on Fault Tolerant Computing*, pp 286-291, 1988.
- [76] AMPLE Reference Manual, Mentor Graphics Corporation, 8005 S.W. Boeckman Road, Wilsonville, OR 97070, 1993.
- [77] QuickSimII User's Manual, Mentor Graphics Corporation, 8005 S.W. Boeckman Road, Wilsonville, OR 97070, 1993.
- [78] P. L. Goddard, and R. Davis, "Automated FMEA Techniques," Hughes Aircraft Company, RADC-TR-84-244, Final Technical Report, December 1984.
- [79] Rasmuson, D. M., et al., "Common Cause Failure Analysis Techniques: A Review and Comparative Evaluation," EG&G, September 1979.
- [80] D. L. Landis, and D. C. Muha, "Evaluation of System BIST Using Computational Performance Measures," *1988 International Test Conference*, pp 531-536.
- [81] P. C. Ward, and J. R. Armstrong, "Behavioral Fault Simulation in VHDL," *1990 Design Automation Conference*, pp 587-593.

- [82] H. B. Dassault, "The Evolution and Practical Applications of Failure Modes and Effects Analyses," RADC-TR-83-72, Rome Air Development Center. Air Force Systems Command, Griffiss Air Force Base, NY 13441.
- [83] B. W. Johnson, "Design and Analysis of Fault Tolerant Digital Systems," Addison-Wesley Publishing Company Inc., ISBN 0-201-07570-9, 1989.

***MISSION
OF
ROME LABORATORY***

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

- a. Conducts vigorous research, development and test programs in all applicable technologies;
- b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;
- c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;
- d. Promotes transfer of technology to the private sector;
- e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.