United States Nuclear Regulatory Commission

Protecting People and the Environment

NUREG/CR-7151, Vol. 2

Development of a Fault Injection-Based Dependability Assessment Methodology for Digital I&C Systems

Volume 2

Office of Nuclear Regulatory Research

AVAILABILITY OF REFERENCE MATERIALS IN NRC PUBLICATIONS

NRC Reference Material

As of November 1999, you may electronically access NUREG-series publications and other NRC records at NRC's Public Electronic Reading Room at http://www.nrc.gov/reading-rm.html. Publicly released records include, to name a few, NUREG-series publications; *Federal Register* notices; applicant, licensee, and vendor documents and correspondence; NRC correspondence and internal memoranda; bulletins and information notices; inspection and investigative reports; licensee event reports; and Commission papers and their attachments.

NRC publications in the NUREG series, NRC regulations, and Title 10, "Energy," in the *Code of Federal Regulations* may also be purchased from one of these two sources.

1. The Superintendent of Documents
   U.S. Government Printing Office Mail Stop SSOP
   Washington, DC 20402–0001
   Internet: bookstore.gpo.gov
   Telephone: 202-512-1800
   Fax: 202-512-2250
2. The National Technical Information Service
   Springfield, VA 22161–0002
   www.ntis.gov
   1–800–553–6847 or, locally, 703–605–6000

A single copy of each NRC draft report for comment is available free, to the extent of supply, upon written request as follows:
Address: U.S. Nuclear Regulatory Commission
Office of Administration
Publications Branch
Washington, DC 20555-0001
E-mail: DISTRIBUTION.RESOURCE@NRC.GOV
Facsimile: 301–415–2289

Some publications in the NUREG series that are posted at NRC's Web site address http://www.nrc.gov/reading-rm/doc-collections/nuregs are updated periodically and may differ from the last printed version. Although references to material found on a Web site bear the date the material was accessed, the material available on the date cited may subsequently be removed from the site.

Non-NRC Reference Material

Documents available from public and special technical libraries include all open literature items, such as books, journal articles, transactions, *Federal Register* notices, Federal and State legislation, and congressional reports. Such documents as theses, dissertations, foreign reports and translations, and non-NRC conference proceedings may be purchased from their sponsoring organization.

Copies of industry codes and standards used in a substantive manner in the NRC regulatory process are maintained at—
The NRC Technical Library
Two White Flint North
11545 Rockville Pike
Rockville, MD 20852–2738

These standards are available in the library for reference use by the public. Codes and standards are usually copyrighted and may be purchased from the originating organization or, if they are American National Standards, from—
American National Standards Institute
11 West 42nd Street
New York, NY 10036–8002
www.ansi.org
212–642–4900

Legally binding regulatory requirements are stated only in laws; NRC regulations; licenses, including technical specifications; or orders, not in NUREG-series publications. The views expressed in contractor-prepared publications in this series are not necessarily those of the NRC.

The NUREG series comprises (1) technical and administrative reports and books prepared by the staff (NUREG–XXXX) or agency contractors (NUREG/CR–XXXX), (2) proceedings of conferences (NUREG/CP–XXXX), (3) reports resulting from international agreements (NUREG/IA–XXXX), (4) brochures (NUREG/BR–XXXX), and (5) compilations of legal decisions and orders of the Commission and Atomic and Safety Licensing Boards and of Directors' decisions under Section 2.206 of NRC's regulations (NUREG–0750).

DISCLAIMER: This report was prepared as an account of work sponsored by an agency of the U.S. Government. Neither the U.S. Government nor any agency thereof, nor any employee, makes any warranty, expressed or implied, or assumes any legal liability or responsibility for any third party's use, or the results of such use, of any information, apparatus, product, or process disclosed in this publication, or represents that its use by such third party would not infringe privately owned rights.

United States Nuclear Regulatory Commission

Protecting People and the Environment

NUREG/CR-7151, Vol. 2

Development of a Fault Injection-Based Dependability Assessment Methodology for Digital I&C Systems

Volume 2

Manuscript Completed: November 2011
Date Published: December 2012

Prepared by:
C. R. Elks, N. J. George, M. A. Reynolds, M. Miklo,
C. Berger, S. Bingham, M. Sekhar, B. W. Johnson

The Charles L. Brown Department of Electrical
and Computer Engineering
The University of Virginia
Charlottesville, Virginia

NRC Project Managers:
S. A. Arndt, J. A. Dion, R. A. Shaffer, M. E. Waterman

NRC Job Code N6214

Prepared for:
Division of Engineering
Office of Nuclear Regulatory Research
U.S. Nuclear Regulatory Commission
Washington, DC 20555-0001

NUREG/CR-7151, Vols. 1 to 4 have been reproduced from the best available copy.

ABSTRACT

Today's emergent computer technology has introduced the capability of integrating information from numerous plant systems and supplying needed information to operations personnel in a timely manner that could not be envisioned when previous generation plants were designed and built. For example, Small Modular Reactor (SMR) plant designs will make extensive use of computer based I&C systems for all manner of plant functions, including safety and non-safety functions. On the other hand, digital upgrades in existing light water reactor plants are becoming necessary in order to sustain and extend plant life while improving plant performance, reducing maintenance costs of aging and obsolete equipment, and promoting prognostic system monitoring and human machine interface (HMI) decision making.

The extensive use of digital instrumentation and control systems in new and existing plants raises issues that were not relevant to the previous generation of analog and rudimentary digital I&C systems used in the 1970's style plants. These issues include the occurrence of unknown failure modes in digital I&C systems and HMI issues. Therefore, digital system reliability/safety, classification of digital I&C system failures and failure modes, and software validation remain significant issues for the Light Water Sustainability and SMR initiatives and the digital I&C system community at large.

The purpose of the research described in volume 1 thru volume 4 is to help inform the development of regulatory guidance for digital I&C systems and potential improvement of the licensing of digital I&C systems in NPP operations. The work described herein presents; (1) the effectiveness of fault injection (as applied to a digital I&C system) for providing critical safety model parameters (e.g., coverage factor) and system response information required by the PRA and reliability assessment processes, (2) the development and refinement of the methodology to improve applicability to digital I&C systems, and (3) findings for establishing a basis for using fault injection as applied to a diverse set of digital I&C platforms. Some of the specific issues addressed in Volume 1 are:

- Fault Injection as a support activity for PRA activities.
- Development of the UVA fault injection based methodology.
- Fault models for contemporary and emerging IC technology in Digital I&C Systems.
- Requirements and challenges for realizing Fault Injection in Digital I&C systems.
- Solutions to challenges for realizing fault injection in digital I&C systems.

Volume 1 presents the findings of developing a fault injection based quantitative assessment methodology with respect to processor based digital I&C systems for the purpose of evaluating the capabilities of the method to support NRC probabilistic risk assessment (PRA) and review of digital I&C systems. Fault injection is defined as a dependability validation technique that is based on the realization of controlled validation experiments in which system behavior is observed when faults are explicitly induced by the deliberate introduction (injection) of faults into the system [Arlat 1990]. Fault injection is therefore a form of *accelerated testing* of fault tolerance attributes of the digital I&C system under test.

Volumes 2 and 3 of this research present the application of this methodology to two commercial-grade digital I&C system executing a reactor protection shutdown application.

In Volumes 2 and 3, the research identified significant results related to the operational behavior of the benchmark systems, and the value of the methodology with respect to providing data for the quantification of dependability attributes such as safety, reliability, and integrity. By applying a fault injection-based dependability assessment methodology to a commercial grade digital I&C, the research provided useful evidence toward the capabilities and limitations of fault

injection-based dependability assessment methods with respect to modern digital I&C systems. The results of this effort are intended to assist NRC staff determine where and how fault injection-based methodologies can best fit into the overall license review process.

The cumulative findings and recommendations of both applications of the methodology and application of the generalized results to broader classes of digital I&C systems are discussed in volume 4.

The digital I&C systems under test for this effort, herein defined as Benchmark System I and Benchmark System II, are fault tolerant multi-processor safety-critical digital I&C systems typical of what would be used in a nuclear power plant 1-e systems. The benchmark systems contain multiple processing modules to accurately represent 4 channel or division 2 out of 4 reactor protection systems. In addition, the systems contain a redundant discrete digital input and output modules, analog input and output modules, inter-channel communication network modules, other interface modules to fully represent and implement a Reactor Protection System. The application Reactor Protection System software was developed using the benchmark systems software development and programming environments.

To establish a proper operational context for the fault injection environment a prototype operational profile generator tool based on the US NRC systems analysis code TRACE [NRC 2011] was developed. This tool allowed generation of realistic system sensor inputs to the Reactor Protection System (RPS) application based on reactor and plant dynamics of the simulated model. In addition, the tool allowed creation of accident events such as large break LOCAs, turbine trips, etc., to stress the RPS application under the various design basis events.

Bibliography

[NRC 2001] Commission, U.S. Nuclear Regulatory. *Computer Codes.* April 2011. http://www.nrc.gov/about-nrc/regulatory/research/comp-codes.html (accessed 2011).

[Arlat 1990] J. Arlat, M. Aguera, et. al. "Fault Injection for Dependability Evaluation: A Methodology and Some Applications." *IEEE Transactions on Software Engineering*, February 2 , 1990.

FOREWORD

As discussed in the NRC Policy Statement on Probabilistic Risk Assessment (PRA), the NRC intends to increase its use of PRA methods in all regulatory matters to the extent supported by state-of-the-art PRA methods and data. Currently, I&C systems are not modeled in PRAs. As the NRC moves toward a more risk-informed regulatory environment, the staff will need data, methods, and tools related to the risk assessment of digital systems. Fault injection methods can provide a means to estimate quantitatively the behavior model parameters of the system. The quantification of these parameters (in a probabilistic sense) can be used to produce more accurate parameter estimates for PRA models, which in turn produces more a accurate risk assessment to inform the risk oversight process.

A challenge for evaluating system reliability relates to relatively undeveloped state of the art methods for assessing digital system reliability. Quantitative measures of digital system reliability are available for digital system hardware, but procedures for evaluating system level reliability (both hardware and software) are not well defined in current industry literature. However, comprehensive use of fault injection techniques for providing critical data toward evaluating digital system dependability may reduce software reliability uncertainties.

The conduct of fault injection campaigns often yields more information than just quantifying the fault tolerance aspects of a system; it also is a means to circumspect and comprehend the behaviors of complex fault tolerant I&C systems to support overall assessment activities for both the developer and the regulator. Fault injection experiments cannot be performed without gaining a deeper understanding of a system. The process itself is a learning experience, providing richer insights into how a system behaves in response to errors arising from system faults. The inclusion of fault injection information into review processes and PRA activities can enlighten the review processes of digital I&C systems. Finally, the process of conducting fault injection testing allows two very important pieces of information to come into direct connection with each other: what the system is supposed to do, and what it actually does. This information is essential for anticipating system behaviors, performing verification and validation (V&V) activities, and conducting methodical system evaluations.

This report describes an important step toward developing a systematic method of evaluating digital system dependability. Volume 1 presents a broad and in-depth development of a digital system dependability methodology, and the requirements and challenges of performing fault injections on digital I&C systems. The process developed in this research project was applied to two digital systems that modeled nuclear power plant safety functions. The results of this phase of the research are described in volume 2 and volume 3. The cumulative findings and recommendations of both applications of the methodology and application of the generalized results to broader classes of digital I&C systems are discussed in volume 4.

TABLE OF CONTENTS

Section Page

ABSTRACT iii
FOREWORD v
LIST OF FIGURES x
LIST OF TABLES xi
ACRONYMS AND ABBREVIATIONS xiii

1. INTRODUCTION 1
   1.1. Background 1
   1.2. Purpose 1
   1.3. Background and Motivation 1
   1.4. Relevance of Research with Respect to Regulatory Guidance 2
   1.5. Project Organization and Timeline 6
   1.6. Organization of this Report 6
   1.7. Digital and Computer Based I&C Systems: Overview 7
   1.8. Overview of Fault Injection 10
   1.9. References 22

2. RESEARCH METHODOLOGY 25
   2.1. Overview 25
   2.2. Identification and Selection of Appropriate Fault Injection Methods for Benchmark System I 25
   2.3. Development of the RPS Application 25
   2.4. Design and Development of the Fault Injection Environment for Benchmark System 26
   2.5. Development of the TRACE-based Operational Profile (TOP) Generator Tool. 26
   2.6. Development of Pre-Fault Injection Analysis Techniques to Support Fault List Generation 27
   2.7. Conduct Fault Injection Campaigns on Benchmark System I 27
   2.8. Conclusions, and Recommendations 28
   2.9. References 28

3. DESCRIPTION OF BENCHMARK SYSTEM I AND RPS CONFIGURATION 29
   3.1. Introduction 29
   3.2. Benchmark System I 29

4. IDENTIFICATION AND SELECTION OF FAULT INJECTION TECHNIQUES FOR BENCHMARK SYSTEM I 41
   4.1. Introduction 41
   4.2. Identification of Fault Injection Methods for Benchmark System I 41
   4.3. IEEE 1149.1 JTAG-based Fault Injection 44
   4.4. ICE-based Fault Injection 46
   4.5. Software Implemented Fault Injection (SWIFI) 47
   4.6. X-bus Communication Module 48
   4.7. Identifying and Selecting Fault Injection Techniques for the X-bus Communication Modules 48
   4.8. Summary of Fault Injection Techniques for Benchmark System I 49
   4.9. Development of Fault Injection Techniques for Benchmark System I 51
   4.10. Development of the ICE-based Fault Injector 51

TABLE OF CONTENTS
(continued)

Section Page

4.11. X-bus Fault Injector 58
4.12. JTAG Fault Injection Module 73
4.13. References 78

5. DEVELOPMENT OF THE UVA PLATFORM INDEPENDENT FAULT INJECTION ENVIRONMENT 79
5.1. Introduction 79
5.2. Motivation and Background 79
5.3. Requirements for Platform Independent Fault Injection Environment 80
5.4. Overview of UNIFI 82
5.5. Configuring the UNIFI Tool to a Target System 84
5.6. Set up of Fault Injection Campaigns 88
5.7. Fault Injection Set Up 90
5.8. UNIFI Master Fault Injection Controller and Observation GUI 91
5.9. Configuring the Benchmark System for Fault Injection 93
5.10. Integrating the Benchmark System into the UNIFI Environment 95
5.11. Fault Injection Process for Benchmark System I: Operational Perspective 100
5.12. References 103

6. TRACE-BASED OPERATIONAL PROFILE GENERATION TOOL 105
6.1. Introduction 105
6.2. Operational Profiles for Real-Time Systems 105
6.3. Characterization of Real-time Operational Profile for Fault Injection 105
6.4. TRACE Modeling Tool 108
6.5. Big Picture View of TOP Modeling Tool 108
6.6. Conclusions 114
6.7. References 116

7. PRE-FAULT INJECTION ANALYSIS AND FAULT LIST GENERATION METHODS 117
7.1. Introduction 117
7.2. Pre-fault Injection Analysis 117
7.3. Related Work on Pre-fault Injection Analysis 119
7.4. Pre- Fault Injection Analysis to Improve Fault Injection Efficiency 120
7.5. Development and Implementation 130
7.6. Experimentation and Results 131
7.7. Applying Dynamic Pre-Fault Injection Analysis to the Benchmark System 137
7.8. Results of Applying Pre-Injection Analysis to Benchmark System I 137
7.9. Other Techniques: Map File-based Fault List Generation 138
7.10. Conclusions 139
7.11. References 140

8. APPLICATION OF FAULT INJECTION TO BENCHMARK SYSTEM I: RESULTS 141
8.1. Introduction 141
8.2. System Test Configuration 141
8.3. Typical Fault Injection Sequence For Benchmark System I 143
8.4. Experiment Definition 145

TABLE OF CONTENTS
(continued)

Section Page

8.5. Processor-based Fault Injection Experiments 146
8.6. Pre-fault Injection Analysis Verification 148
8.7. Digital Output Response and Output Disabled Experiments 148
8.8. Fault and Error Latency Analysis 149
8.9. Addressing No-response Faults 158
8.10. X-Bus Fault Injections 159
8.11. References 168

9. SUMMARY, FINDINGS, AND CONCLUSIONS 169
9.1. Summary of Key Activities and Results 169
9.2. Conclusions 172
9.3. References 172

LIST OF FIGURES

Figure Page

1-1 Phases and activities of the research effort 6
1-2 Generic digital I&C system architecture model 8
1-3 Fault injection model for digital I&C 11
1-4 Fault injection experiment 12
1-5 Operational view of the fault injection based dependability assessment process 14
1-6 Fault model classes for benchmark digital I&C systems 18
1-7 Basic architecture of a fault injection environment 21
3-1 Benchmark System I architecture 30
3-2 Benchmark System I processing 33
3-3 Benchmark System I fault tolerant features 36
3-4 RPS configuration for Benchmark System I 38
4-1 ICE-based fault injection for Benchmark System I 54
4-2 ICE machine pod inserted into the Benchmark System 55
4-3 HiTOP view of the target software executing on the Benchmark System 57
4-4 Illustration of token passing mechanism in a multi-master system 60
4-5 Corruption of token when Master 1 attempts to pass token to Master 2 62
4-6 X-bus transmission packet (PTP) 64
4-7 X-bus fault injector 66
4-8 Sampling of the X-bus traffic by the FPGA 67
4-9 Block diagram representation of the FPGA X-bus fault injection module 68
4-10 JTAG TAP controller test logic diagram 74
4-11 JTAG TAP controller state machine 75
4-12 Block diagram representation of the JTAG fault injection module 77
5-1 UNIFI fault injection environment 83
5-2 Functional representation of the I/O interface module 84
5-3 UNIFI physical interfaces to the target system 86
5-4 UNIFI interface for fault injection 88
5-5 Screenshot of Master Controller window 89
5-6 Process for generating a fault list using UNIFI 91
5-7 Screenshot from single fault injection trial performed by UNIFI master GUI 92
5-8 Configuration 1 of Benchmark System I 93
5-9 Configuration 2 of Benchmark System I 94
5-10 Integrating the fault injectors into Benchmark System I 96
5-11 Integrating UNIFI/LabView environment into Benchmark System I 98
5-12 ICE machine fault injection control script 100
5-13 Fault injection operation for Benchmark System I 101
6-1 TRACE-based operational profile generation tool 109
6-2 SNAP plant representation 110
6-3 AptPlot tool 111
6-4 Configuring the Trace AptPlot output file 112
6-5 Combining the steady and transient output runs for a complete profile 113
6-6 Excel screenshot used to generate the operational profile 115
7-1 Venn diagram representation of fault space 120
7-2 Fault activation for different workloads 121
7-3 Levels of analysis 122
7-4 Flow chart representing fault list generation using dynamic analysis 125
7-5 Populating the data structures with source and destination operands 126
7-6 Illustration of the window of opportunity 127

LIST OF FIGURES
(continued)

Figure Page

7-7 Generating the fault list 127
7-8 Error propagation window of opportunity 128
7-9 Cross referencing in IDA Pro® 130
7-10 Results obtained from fault injection experiments 132
7-11 Comparison of ACE Bits obtained between random and directed fault injection experiments 133
7-12 Frequency of invoked functions for each of eight inputs 134
7-13 Integration of pre-fault injection analysis algorithms into Benchmark System I and execution trace files generated 137
7-14 Snip of a map file 138
8-1 Benchmark system configurations 142
8-2 Fault Injection sequence for Benchmark System I 144
8-3 Example of fault and error latency 149
8-4 Error log transcript from the SMS server 150
8-5 Fault latency of memory fault injections 152
8-6 First 50 variable locations of the memory fault injection campaign 153
8-7 Distribution of memory-based fault latency 154
8-8 Cumulative fault latency distribution 155
8-9 Fault latency of register-based fault injections 156
8-10 Distribution of register-based fault latency 157
8-11 Halt latency of injected processor 158
8-12 Structure of a token message 159
8-13 Structure of a variable length data message 160
8-14 Jamming signal output correctly, but not applied to the X-bus circuit 163
8-15 Jamming signal applied to X-bus thereby corrupting transmission 164
8-16 Token message fault response graph 167
8-17 Data message fault response graph 167

LIST OF TABLES

Table Page

4-1 Fault injection techniques for Benchmark System I. 50
4-2 Performance delay times for ICE-based fault injection of a few models. 58
6-1 Example composition of an operational profile for Benchmark System I. 106
7-1 Number of fault injection experiments. 124
7-2 Results of fault injection experiments in SimpleScaler. 131
7-3 Figure 7-12 function invocation and cross reference count. 135
8-1 Summary of experiments run on Benchmark System I. 145
8-2 Details of registers used in fault injection experiment. 147
8-3 Summary of results from memory based fault injection experiments. 148
8-4 Results from UNIFI experiments to detect CPU register faults 149
8-5 Synchronization times following uncorrupted and corrupted token in X-bus. 165
8-6 Data message response for long duration fault injections. 168

ACRONYMS AND ABBREVIATIONS

ACE	Architecturally Correct Execution
A/D	Analog to Digital
API	Application Programmer Interface
ASIC	Application Specific Integrated Circuit
AVF	Architectural Vulnerability Factor
BDM	Background Debug Mode
BSC	Boundary Scan Chain
BWR	Boiling Water Reactor
CCITT	Commite' Consultatif International de Telegraphique et Telephonique. (International Consultative Committee on Telecommunications and Telegraphy).
CFR	Code of Federal Regulations
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DA	Destination Address
D/A	Digital to Analog
DC	Direct Current
DFWCS	Digital Feedwater Control System
DLPDU	Data Link Protocol Data Unit
ED	Ending Delimiter
EDM	Error Detection Mechanism
EEPROM	Electrically Erased Programmable Read-Only Memory
ESFAS	Engineered Safety Features Actuation System
FARM	Faults, Activations, Readouts and Measures
FDD	Fixed Data
FDIM	Fault Detection, Isolation and Mitigation
FMEA	Failure Modes and Effects Analysis
FPGA	Field Programmable Gate Array
FW	Firmware
GUI	Graphical User Interface
GOOFI	Generic Object Oriented Fault Injection
HiTOP	In-Circuit Emulator Tool
HMI	Human Machine Interface
I&C	Instrumentation and Control
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IC	Integrated Circuit
I/O	Input/Output
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
LMS	List of Master Stations
LOCA	Loss of Coolant Accident
MI	Monitor Interface
MTTF	Mean Time to Failure
NDD	No Data
NFTAPE	University of Illinois Fault Injection Tool
NPP	Nuclear Power Plant
NRC	Nuclear Regulatory Commission
NS	Next Station
OCD	On-Chip Debugger

OP	Operational Profile
PC	Program Counter
PRA	Probabilistic Risk Assessment
PS	Previous Station
PTP	X-bus Transmission Packet
PWR	Pressurized Water Reactor
PXI	National Instruments Data Acquisition Controller
RAM	Random Access Memory
ROM	Read Only Memory
RPS	Reactor Protection System
RTE	Runtime Environment
RTL	Register Transfer Level
SA	Starting Address
SCADA	Supervisory Control and Data Acquisition
SCIFI	Scan Chain Implemented Fault Injection
SD	Start Delimiter
SMR	Small Modular Reactor
SNAP	TRACE Graphical User Interface
SoC	Systems on a Chip
SWIFI	Software Implemented Fault Injection
TAP	Test Access Port
TCK	Test Clock
T_{com}	Communication Cycle Time
TDI	Test Data In
TDO	Test Data Out
TMS	Test Mode Select
TOP	TRACE-based Operational Profile
TRACE	TRAC/RELAP Advanced Computational Engine
TRST	Test Reset
UNIFI	Universal Platform Independent Fault Injection
UVA	University of Virginia
VDC	Volts DC
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VI	Virtual Instrument
V&V	Verification and Validation
VLDD	Variable Length Data

1. INTRODUCTION

1.1. Background

This report is Volume 2 of a multi-volume set of reports that present the cumulative efforts, findings, and results of U.S. Nuclear Regulatory Commission (NRC) contract JCN N6124 – "Digital System Dependability Performance". The reports are organized as follows:

Volume 1 – Presents a broad and in-depth development of the methodology, the requirements and challenges of realizing fault injection on digital instrumentation and control (I&C) systems.
Volume 2 – Presents the application of the methodology to Benchmark System I.
Volume 3 – Presents the application of the methodology to Benchmark System II- employing the lessons learned from Benchmark System I.
Volume 4 – Presents the cumulative findings and recommendations of both applications of the methodology and generalizes the results to broader classes of digital I&C systems.

1.2. Purpose

This report (Volume 2) presents the findings of applying a fault injection based quantitative assessment methodology (presented in Volume 1) to a processor-based digital I&C system for the purpose of evaluating the capabilities of the method to support NRC probabilistic risk assessment (PRA) and review processes for digital I&C systems. The further purpose of this work is to help inform the development of regulatory guidance processes for digital I&C systems and potential improvements to the licensing process for digital I&C systems in nuclear power plant (NPP) operations. The work described herein broadly presents; (1) the development of the fault injection methods and techniques that were applied to the Benchmark system, (2) the development of a fault injection environment for digital I&C systems (3) development of pre-injection analysis methods for automatically generating fault lists for digital I&C systems, (4) results of the application of the fault injection method to benchmark systems, (5) the challenges to applying fault injection to contemporary digital I&C systems, and (6) the findings for addressing these challenges and establishing a basis for implementing fault injection to digital I&C platforms.

1.3. Background and Motivation

In recent years significant effort has gone into improving safety critical system design methodologies, assessment methods, and the updating of regulatory industry standards and NRC regulatory guidelines to ensure that digital I&C systems can be designed and assessed to the high safety requirement levels required of highly critical applications. Of particular interest recently are quantitative dependability assessment methodologies that employ fault injection methods to ensure proper compliance of digital I&C system fault handling mechanisms [Arlat 1993; Yu 2004; Smith 2000; Elks 2009(a); Aldemir 2007; Smidts 2004]. The goal of a dependability assessment methodology is to provide a systematic process for characterizing the safety and performance behavior of embedded systems (e.g. digital I&C systems) in the presence of faults.

Dependability evaluation involves the study of failures and errors and their potential impact on system attributes such as reliability, safety, and security. Often the nature of failures or system crashes and long error latency often make it difficult to identify the causes of failures in the operational environment. Thus, it is particularly difficult to recreate a failure scenario for large, complex systems just from system failure logs alone. To identify and understand potential failures, the use of an experiment-based or measurement based approach for studying the

dependability of a system is gaining acceptance in the nuclear industry for better understanding the effects of errors and failures to promote an informed understanding of risk. Such an approach is useful not only during the concept and design phases, but also during licensing review activities.

From a practical point of view, most digital I&C systems are designed as safety critical systems employing extensive fault detection/tolerance and design diversity features to ensure proper operational and fail safe behavior in the event of a system failure. For example, Fault Detection, Isolation, and Mitigation (FDIM) software and online diagnostic functions of the benchmark systems in this research effort account for as much as 40 to 50 percent of the executable system software code [Barton 1990; Palumbo 1986; Young 1989]. This code is rarely challenged during normal operations because faults and failures are an infrequent occurrence. This FDIM code is vital toward system dependability and safety compliance, and can only be effectively tested and validated by realistic fault injection campaigns.

1.4. Relevance of Research with Respect to Regulatory Guidance

The NRC has a comprehensive set of regulatory guidelines for reviewing and assessing the safety and functionality of digital I&C systems. The NRC PRA technical community has not yet agreed on how to model the reliability of digital systems in the context of PRA and the level of detail that digital systems require in reliability modeling. Nonetheless, it is clear that PRA models must adequately represent the complex system interactions that can contribute to digital system failure modes. The essential research aim of the PRA technical community is to accurately model digital I&C system behaviors to take into account interactions of the system fault handling behaviors, coverage of fault tolerance features, and the view of the system as an integrated software and hardware system.

Fault injection is a formal-based process to collect evidence to gauge the dependability of safety functions associated with I&C systems that has an underlying mathematical theory (with explicitly stated assumptions) that allows one to place stronger justification or refutation on claims of the overall safety of an I&C system. Fault injection as part of a quantitative assessment process is a robust testing process that can support verification and validation (V&V) and quality assurance activities to gather evidence that the digital I&C system can perform its safety functions in the presence of faulted and failure conditions in compliance with NRC regulations. In addition, those aspects of Appendix B of Title 10 of the Code of Federal Regulations (CFR), Part 50 (10 CFR 50), the NRC Standard Review Plan (NUREG-0800), and other relevant guidelines that address requirements for testing processes, methods and evidence to support safety function operational effectiveness are clear candidates for the application of fault injection methods.

1.4.1. Relationship to NRC Research Activities

The research conducted under this contract was done with the consideration of previous and on-going research efforts related to the safety and reliability assessment of digital I&C systems. Accordingly, the research effort was attentive of complementary research efforts and how those efforts could benefit from the work accomplished through this effort. Specifically, the researchers recognized that the products developed from this research could have the potential to be used in other research efforts. Therefore, the researchers endeavored to catalog research findings in way that promoted broader relevance and helpful information for other research efforts.

1.4.2. Research Objectives

The overall objective of this work was to develop a body of evidence to inform the development of regulatory guidance processes and potentially improve licensing processes for digital I&C systems in NPP operations. In support of this objective the research investigated the effectiveness of fault injection applied to digital I&C systems for providing critical parameters and information required by PRA and reliability assessment processes. The results and findings of this effort are aimed at assisting NRC staff in determining when, where and how fault injection based methodologies can best fit in the overall PRA and license review process. The major goals of the research effort are listed below:

Objective 1
Demonstrate the effectiveness of the University of Virginia (UVA) quantitative safety assessment process on commercial safety grade I&C systems executing reactor protection applications with respect to a simulated NPP safety system design.

Objective 2
Identify, document, and develop improvements to the process that make it easier and more effective to apply to a wider spectrum of digital I&C systems. Document the limitations, sensitive assumptions, and implementation challenges that would encumber the application of fault injection processes for digital I&C systems.

Objective 3
Document the quantitative and qualitative results that can be obtained through application of the assessment process, and provide the technical basis upon which NRC can establish the regulatory requirements for safety-related digital systems, including acceptance criteria and regulatory guidance documents.

Secondary Objective 1
Assess the level of effort and cost to implement the fault injection capability in a vendor or licensee environment.

Secondary Objective 2
Identify and develop innovative fault injection methods that would make fault injection more efficient and easier to adopt by NRC and the nuclear industry.

The scope of this work is targeted at safety critical digital I&C systems, but applies to non-safety related systems as well. The target benchmark systems were configured to be representative of a four-channel Reactor Protection Systems (RPS) system, but were limited in scale due to budget constraints on equipment availability. Therefore, the systems lacked some redundant hardware modules that would normally be found in an actual RPS. The overall complexity and configuration of the system, however, was sufficient to stress the methodology, which was the overall objective of the research effort. The specific benchmark system data results obtained from the study should be interpreted with respect to the benchmark system configuration described in this report unless stated otherwise.

Further, the methodology that was developed and applied in this research effort is part of a larger comprehensive assessment and review process, and is not intended to be interpreted as a “replacement” for existing processes. Rather the methodology is viewed as a complementary method in an effort to establish more efficient, repeatable and objective design assessment and review processes.

1.4.3. Work Tasks for Phase II

The basic research plan of the project is presented in Section 9 of Volume 1, which is used as a guide to realize the UVA fault injection-based dependability assessment methodology on the benchmark systems. This plan has two categories of tasks. The first task category includes items that require additional research and development to determine their potential for implementation. As such, the researchers realized early that this would be on-going work for the project. The second category are tasks that needed little or no additional research effort to implement to determine their overall effectiveness. The second category should be viewed as items that needed to be accomplished in order to support the overall research objectives.

1.4.3.1. Research Oriented Tasks

The research oriented tasks listed below are specifically tied to key challenges identified in Section 8 of Volume 1. The challenges are:

High performance fault injection – The need for fault injection techniques to support various fault models for fault injection that can be implemented in a manner that is minimally intrusive, controllable, repeatable and reproducible is critical to the application of fault injection to digital I&C systems. The purpose of this task is to investigate, design, develop and implement new methods to achieve these goals. This task is considered to be a high risk effort and, as such, less risky back up fault injection methods could be considered in parallel to this effort.

Data collection and analysis – In support of better measurement practices, the needs required for data collection from the benchmark systems should be investigated. This investigation includes developing a thorough understanding of the relationship between the error messages from the target benchmark system and the underlying error detection and fault tolerant mechanisms in the target benchmark systems. Prior experience has shown that vast amounts of data are the norm during long fault injection campaigns. Finding methods to manage the data, establish relationships between the data sets, and reduce the data sets to essential attributes is a key goal for developing an effective analysis process.

Fault list generation and pre-injection analysis – Being a statistical experiment, fault injection testing may require a large number of experiments to be conducted in order to guarantee statistically significant results. Thus, efficiency of the fault injection testing is important. Generating fault lists for a fault injection experiment or campaign is a critical activity for fault injection. Pre-injection analysis is a method to guide the fault injection process to produce more effective and efficient results. It is a means to reduce or eliminate the “no-response” problem associated with fault injection. The goal of this research objective is to develop a methodology for pre-analyzing the binary listing of the target benchmark system I&C system to reduce no-response fault injections, accelerate error propagation, and improve efficiency.

Operational profile generation – Operational profiles and workloads of the target system are required to set the operational and environmental context of the system. The operational profiles must be representative of the different system configurations and workloads that would be experienced in actual field operations. In order to provide a diverse and representative set of operational profiles for the target system, the use of high fidelity NPP simulator tools to generate nominal, off-nominal, and accident event profiles is a promising approach. To support this task, the research effort developed a process for integrating TRAC/RELAP Advanced Computational Engine (TRACE) thermo-hydraulic NPP simulator results into the UVA fault injection environment so that real time process data from the simulator could be used to drive the inputs of the target benchmark system under various conditions and modes.

1.4.3.2. Research Support Tasks

Research tasks are necessary to support research activities using Benchmark System I. As seen in previous fault injection process efforts, the amount of time to design, develop, test, and integrate various fault injection components together and then interface these components into the target Benchmark system can be significant. These research support tasks are:

Training and experience – To effectively apply fault injection to complex digital I&C systems of the type found in the benchmark systems, the research staff at UVA required professional training by the respective vendors of the benchmark system platforms. Once this training was completed, the staff required time to gain additional experience on the systems to fully understand the details of the system platforms from various points of view.

Fault injection environment – A well-formed fault injection environment is one of the most important aspects of a fault injection-based methodology to support credible, repeatable, and controllable fault injection campaigns. Furthermore, the fault injection environment plays a crucial role in the data collection and measurement of the responses, which are important to produce the measures of dependability (e.g. coverage, error latency, etc.). Further, the user must have the ability to manage the types of faults to be injected into the system, where they are injected, how they are injected, and when they are injected. Additionally, the responses to the fault injections must be acquired in a manner that allows the responses to be traced back to the faults so that any fault injection trial can be repeated as needed to reproduce the system response.

Over and beyond the basic functional requirements for a fault injection, effective fault injection environments must also be practical to implement and use, adaptable to changing technology, and supportable. Several development goals for the fault injection environment to allow for adaptability for different I&C systems include:

- Flexibility for a wide variety of applications
- Easy to use and familiar to the engineering culture
- Industry-grade, supportable, and open source
- Modular
- Extensible
- Evolutionary

To achieve these goals, the National Instruments™ LabVIEW® toolset was selected to develop the basic architecture of the fault injection environment. Proven technology and industry acceptance made this choice obvious. Due to the complex nature of fault injection and need for tight coordination of several processes (e.g. data acquisition, operational profile sequencing, fault injection, data logging, etc), a cross-platform tool was most effective to support these functions.

Application code development – The benchmark systems were not delivered to UVA with an application embedded on them. Therefore, the researchers built a representative reactor protection system (RPS) application as the benchmark application.

Integration and testing – Integrating the various components of the fault injection environment, the data acquisition system, and the target benchmark systems required substantial skills and knowledge. The most prominent of these tasks was the integration of the fault injector into the target I&C system. This task required considerable modifications to the fault injector to effectively integrate the fault injector into the target system.

1.4.4. Scope of Study

Fault injection-based methodologies are but one part of a comprehensive process of estimating the reliability of a digital system (hardware and software) for PRA applications. From the highest level perspective, the essential information needed for reliability estimation are (1) knowledge of the likelihood of faults (software or hardware), and (2) the consequence of activating these faults in the system context. Fault injection methods are most useful in characterizing system responses to activated faults - the second requirement. That is, providing empirical knowledge on the triggering, detection, tolerance, and propagation of errors due to software or hardware faults in the system. How a digital I&C system responds to a fault and mitigates the fault are essential elements for accurate system reliability modeling. As such, the methodology developed and presented in this report is aimed at providing empirical data in support of developing system fault response data, such as fault detection, error propagation, fault latency, timing delays, etc.

1.5. Project Organization and Timeline

This project was carried out over a 3.5 year period beginning in 2007. Three phases of work were conducted during this timeframe. The first phase, which is principally reported in this volume, developed and refined the methodology so that it could be successfully applied to the benchmark systems. This effort lasted about 12 months. The second phase of the work was applying the methodology to the first benchmark system, based on the recommendations and plan of action from the first phase of the work. The third and final phase of the work was applying the methodology to a second benchmark system based on the lessons learned from the first and second phase of the work. Figure 1-1 shows the progression of this effort through the lifecycle of the project.

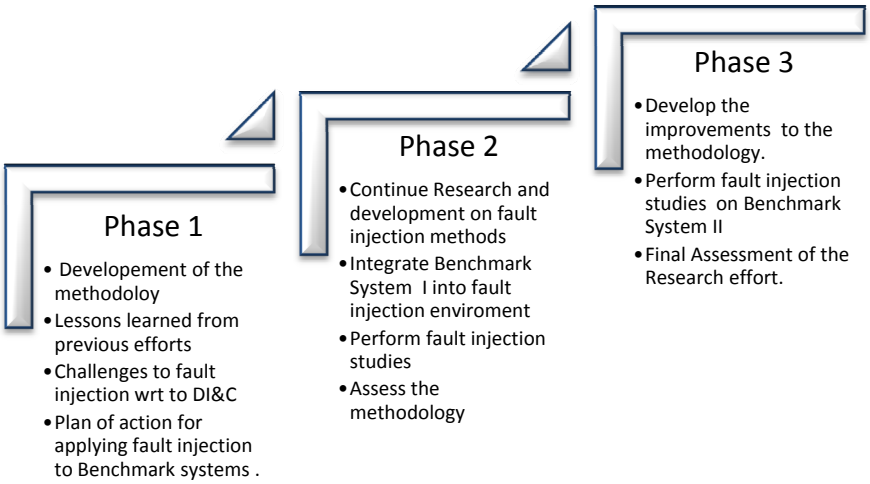


Figure 1-1 Phases and activities of the research effort

1.6. Organization of this Report

This report provides a contemporary and comprehensive perspective on fault injection for digital I&C systems for the NRC staff. In addition, this report also serves the greater digital I&C community by providing solid and deep perspective of fault injection with specific focus on digital I&C systems. This report is organized around three main themes: (1) selection of appropriate fault injection techniques, (2) development of an configurable fault injection environment for

digital I&C systems, (3) application of the fault injection based dependability assessment methodology to Benchmark System I, and (4) results, outcomes and challenges associated with the application of fault injection to the benchmark systems. The Sections each build on and connect to previous Sections. Sections 1 and 2 provide an overview of the fault injection based dependability assessment methodology, the research methodology of this phase of the research effort, and a solid and deep foundational understanding of the concepts of fault injection with respect to digital I&C systems. Section 3 presents a detailed overview of the Benchmark System I. Section 4 presents an overview of the RPS code development. Section 5 presents analysis and selection of candidate fault injection techniques for Benchmark System I. Sections 6 and 7 describe the design and implementation of the configurable fault injection environment that is used to apply fault injection campaigns to Benchmark System I. Section 8 presents the development of pre-injection analysis methods to generate fault lists for efficient fault injection. Sections 9 and 10 discuss the results and outcomes of applying the fault injection to Benchmark System I. Section 11 discusses the findings, insights, and lessons learned from this research effort.

1.7. Digital and Computer Based I&C Systems: Overview

In order to provide relevance beyond the benchmark systems that were evaluated, the research developed a generic representation of an I&C system based on the evaluation of several current digital I&C systems being proposed in new reactor applications and the emerging technologies that may be utilized in new I&C systems. The characterization that seemed most suitable is illustrated in Figure 1-2.

Modern digital I&C systems characteristic of the systems addressed in this research are neither strictly embedded systems nor general purpose computing platforms. Rather they fall into a special class of embedded computing platforms called *adaptive or configurable embedded computing*. That is, the hardware and software architectural elements of the platform allow the architecture to be tailored to specific constraints of the application domain. To achieve such flexibility the architecture may trade-off attributes like optimal performance, simplicity, and cost with respect to a fully custom embedded system. Most I&C systems being considered for nuclear power plant applications fall into this class of systems.

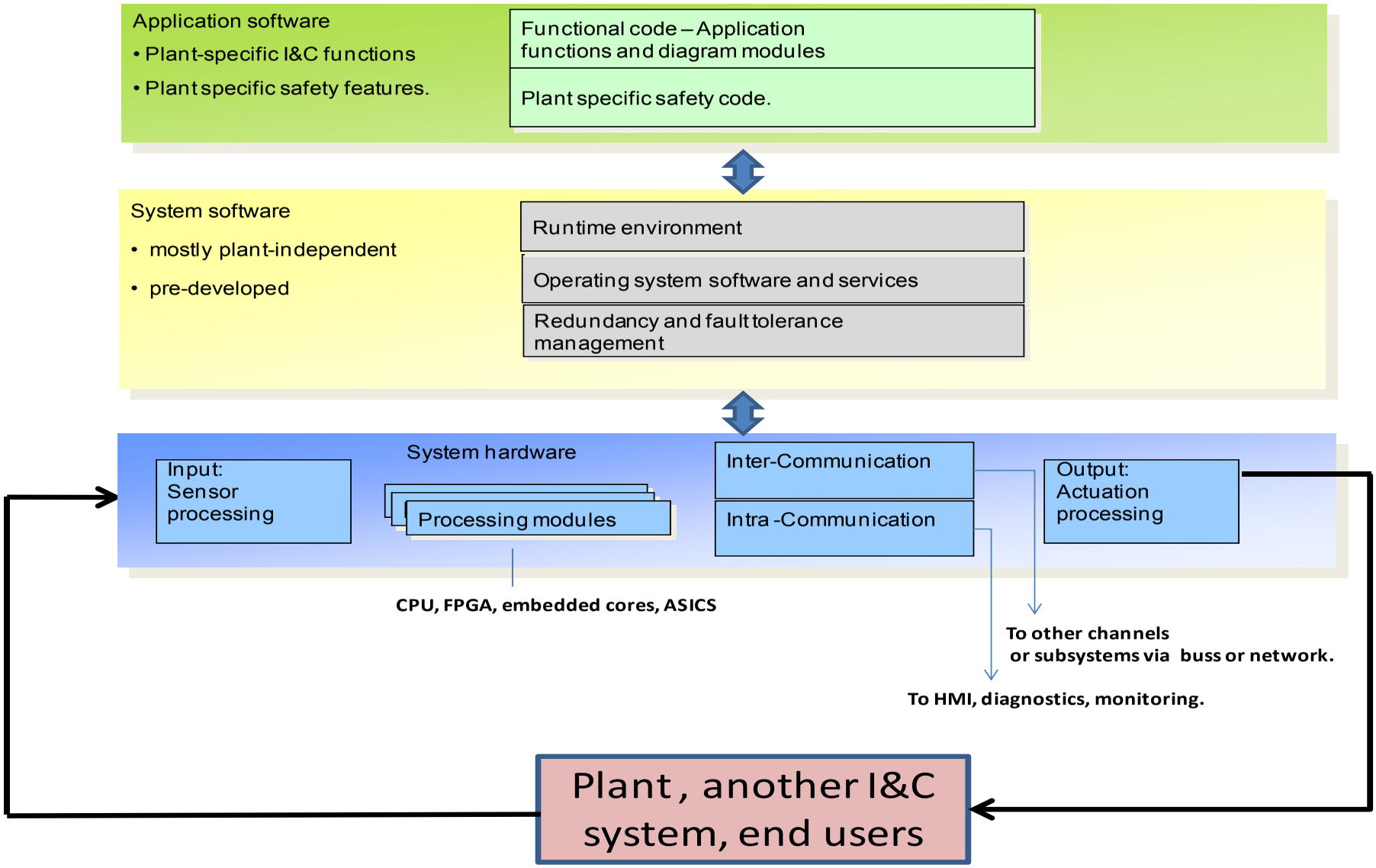


Figure 1-2 Generic digital I&C system architecture model

Figure 1-2 illustrates the several important concepts of modern I&C systems, specifically:

- Application adaptive functionality: Configurable to different plant designs, ability to change parameters and programming to optimize performance and safety.
- Layered Architecture: Separation of Application independent functions and Application dependent functions.
- Intra-communication and Inter-communication functionality to facilitate integration of I&C system operational information to other plant systems and personnel.
- Redundancy and diversity to support fail-safe operation and/or degraded operation in the presence of faults and failures.
- Interfaces and sub-systems to support health monitoring of I&C operations and system behavior.
- Interfaces to support operator monitoring and actions.

Digital I&C systems depicted in Figure 1-2 serve a variety of functions within NPP operations. The generic digital I&C architecture provides mapping of function to form, and implementations to realize the functionality. Digital I&C systems are used in NPP systems such as safety systems, plant process control systems, monitoring systems, data communication systems, and sensor processing systems.

The wide range of uses illustrates the importance that digital I&C systems are not just characterized by their internal form and function, but also by their interaction context with the environment in which they operate. Context is important. These types of systems may interact with other systems through communication systems and may indirectly interact with other systems through plant dynamics. Context establishes the basis for what a system is supposed to do, and what a system is not supposed to do. Context establishes the basis for what reasonable assumptions should be made concerning the assessment of a digital I&C system and what relevant conclusions can be made from the assessment process. The research described in this report attempted to keep this principle in mind as the methodology was developed and tested.

Another overarching aspect of modern digital I&C systems (and embedded systems) is their use of programmable elements throughout the design and implementation. Traditionally, embedded systems were viewed from two perspectives: the software and the hardware. This view is largely driven by the development processes that realize the functionality of the system. Software enables the system to perform its intended functionality in the context of its environment. Hardware provides the necessary programmability to allow software to be flexible to different applications. In actuality, neither view by itself is realistic of real behavior. Software does nothing without hardware to animate it. Hardware is just an organization of digital functions and signals. The digital I&C system should be viewed from a unified perspective, seeing them not as completely different domains but rather as an integrated system to achieve a purpose that is more representative of the functionality that the I&C system implements. This unified view of the I&C system is often represented at the object code level or register transfer level (RTL) in digital I&C systems. It is here that the interactions of software and hardware take place. This is an important concept because the failure of a hardware function can adversely affect the functionality and reliability of the software relying upon that function. In like manner, a design flaw in a software function or improper programming fault can produce different errors or

failures depending on the hardware architecture and organization. Digital I&C assessment methodologies, therefore, should be flexible enough to allow this multi-level view of the application.

Recent trends in digital technology advances have strengthened this integrated view to the point where there is less distinction between hardware and software. Integrated circuit and microelectronic capacities have increased to the point that both software processors and custom hardware now commonly coexist on a single integrated circuit (IC) package – these Systems on a Chip (SoC) have become very common. This is particularly true of the field-programmable gate array (FPGA) technology in which embedded processor cores, network and bus protocol engines, analog to digital (A/D) and digital to analog (D/A) conversion, and memory management functions are often mapped into a single FPGA structure. FPGA and SoC technologies are hardware that acts like software. Users can change the hardware organization at any time during the design, development, and field operation of FPGAs to meet the changing needs of the customers. This technology is already finding its place in digital I&C systems – as both digital I&C systems used in this research effort employed FPGA technology.

1.8. Overview of Fault Injection

Section 3 of Volume 1 presented a detailed discourse of fault injection concepts and theories as needed for the development of fault injection for digital I&C systems. This section presents a brief overview of fault injection to reacquaint the reader with the principle of fault injection.

Consider the digital I&C system in Figure 1-3, which is referred to as the target system. When fault injection is applied to the target system, the input domain corresponds to following sets: a set of faults ***F*** taken from a class of faults "F_{class}" a set of activations "***A***" that specifies the domain used to functionally exercise the system; an output domain corresponding to a set of readouts "***R***", and a set of derived measures "***M***". Together, the Faults, Activations, Readouts and Measures (FARM) sets constitute the major attributes that can be used to fully characterize fault injection:

F = faults ={Faults injected into the system}
A = input activations = {Pin = inbound communications messages, D = Inputs}
R = Readouts = {U = outputs, Y= current state, Z= global state, Pout = outbound communication]
M = Measures = {fault coverage, fault latency, responses, etc..}

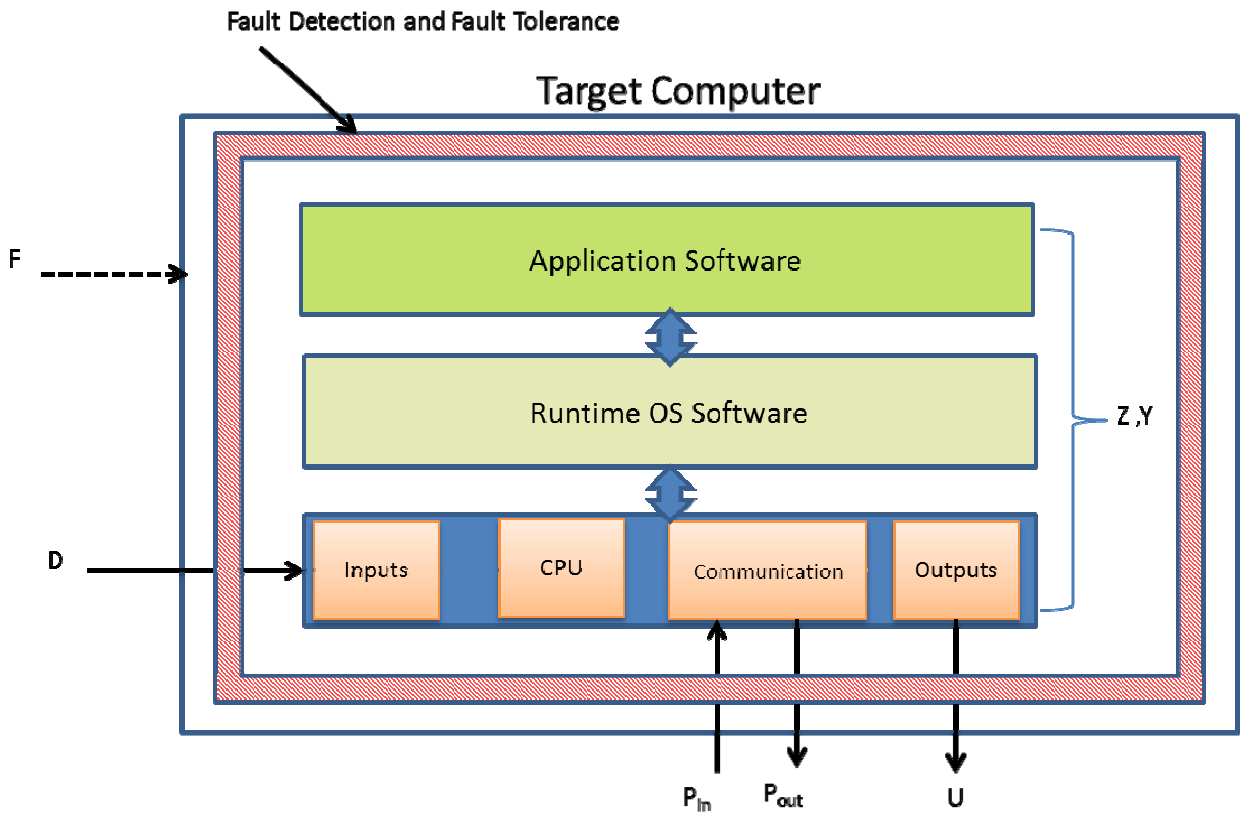


Figure 1-3 Fault injection model for digital I&C

Fault injection is a formal experiment based approach. For each experiment, a fault f is selected in F and an activation trajectory a is described in A. The reactions of the system are observed and form a readout r that fully characterizes the outcome of the experiment. An experiment is thus characterized by the triple ordinate < f, a, r >, where the readouts, r, for each experiment form a global set of readouts R for the test sequence and can be used to elaborate a measure in M. A campaign is a collection of experiments to achieve the quantification of a measure M.

Consider a test sequence of n independent fault injection experiments; in each experiment, a point in the {F x A} space is randomly selected according to the distribution of occurrences in {F x A} and the corresponding readouts collected. Expanding the F to include the fault space dimensionality of time, location, value, and fault type, yields six parameters that define a fault injection experiment:

a = the set of external inputs
Δ = is the duration of the injected fault
t = fault occurrence time, or when the fault is injected into the system
l = fault location
v **=** value of the fault
f_m = a specific fault type as sampled from fault classes

Figure 1-4 illustrates the basic concept of a fault injection experiment. Specifically, Figure 1-4 shows that faults from ***F*** are sampled from the fault space (discussed in Section 3.7 of Volume 1). These faults are elaborated by their fault type f_m, the fault duration Δ, the fault location *l*, value of the fault mask, time of occurrence *t,* along with the set of inputs *a* to characterize a set of experiments. The fault experiments are applied to the target computer, and a set of readouts

(the R set) is used to derive the M set (coverage estimation) by statistical estimation. More importantly, from a practical perspective, the parameters of the coverage equation serve as the essential requirements in the development of any fault injection methodology or tools to support fault injection. Fault injection frameworks of any type must address the control of these parameters and the observable responses of a system to these parameters as they are sampled. The following sections discuss the statistical theory behind the coverage estimation and the dependent parameters of coverage.

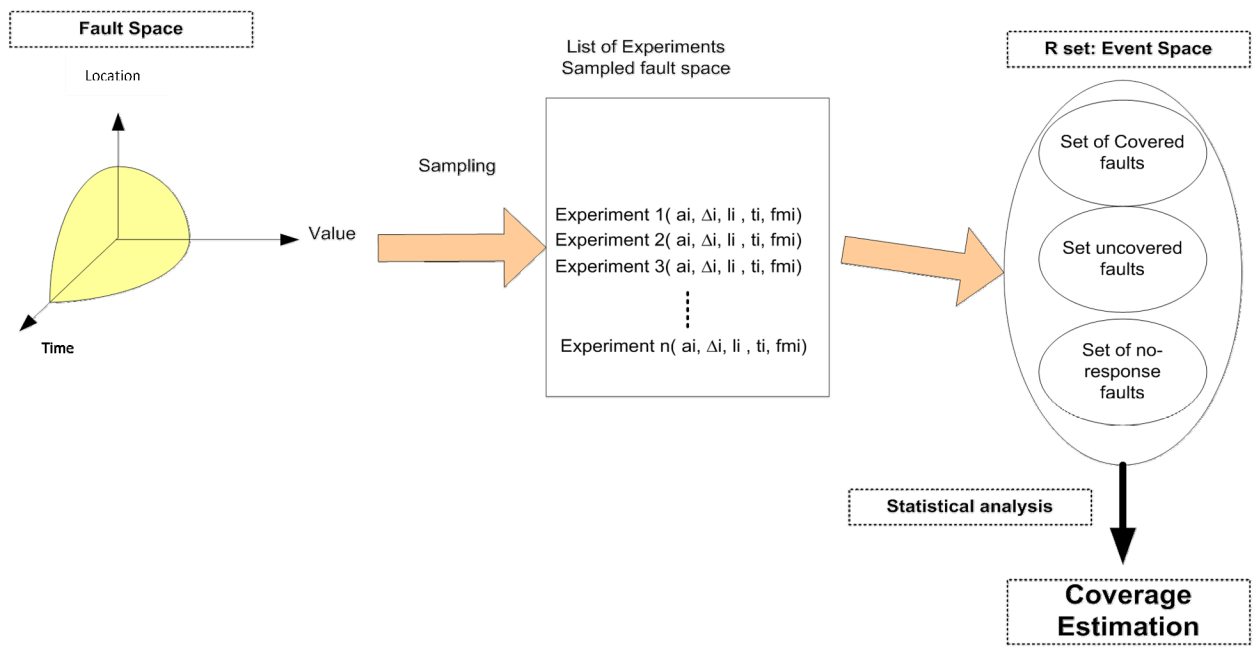


Figure 1-4 Fault injection experiment

1.8.1. Overview of the Fault Injection Based Methodology

This section provides an overview of the dependability assessment methodology. Fault injection has been extensively used in many industries to aid in the assessment of fault tolerant system and safety critical system over the past 30 years [Benso 2002], and is widely used in the software development and testing community for improving software quality and protection against cyber threats. In addition, International Electrotechnical Commission (IEC) 61508 "*Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems"* highly recommends the use of fault injection to determine the effects of faults and their mitigation. Thus, fault injection as technique to aid in the dependability evaluation of safety critical systems is not novel concept, but one of continued maturation and acceptance.

The UVA fault injection-based dependability assessment methodology was developed realizing that a fault injection approach may serve different goals and purposes as those stated above. Thus, the methodology was designed to be as flexible as possible to the needs of the assessor and designer. The goal of the dependability assessment methodology described in this report is to provide a generic, formal, systematic means of characterizing the dependability behavior of digital I&C systems and their input/output interactions in the presence of anomalous behaviors, faults, and failures. The goal (for all parties) is a methodology that provides practical means for characterizing digital I&C system/plant dependability attributes that will facilitate developers in improving V&V processes, while helping regulatory entities make informed confirmatory decisions about licensing I&C systems for critical plant operations.

Figure 1-5 shows the basic conceptual framework of the fault injection based dependability assessment process. In this depiction, the process is driven by the needs of PRA modeling efforts to estimate more accurately parameters for PRA modeling activities. Statistical sampling principles are used to guide the parameter estimation process. Then, representative fault models are selected with respect to the target I&C system. After the faults are injected into the system, the data is post-processed to produce new estimates of model parameters, and these are instantiated back into the PRA models to enhance better prediction of the PRA models. The methodology is described in more detail in Section 4 of Volume 1. The characteristics of each step in the process are described in the following discussion.

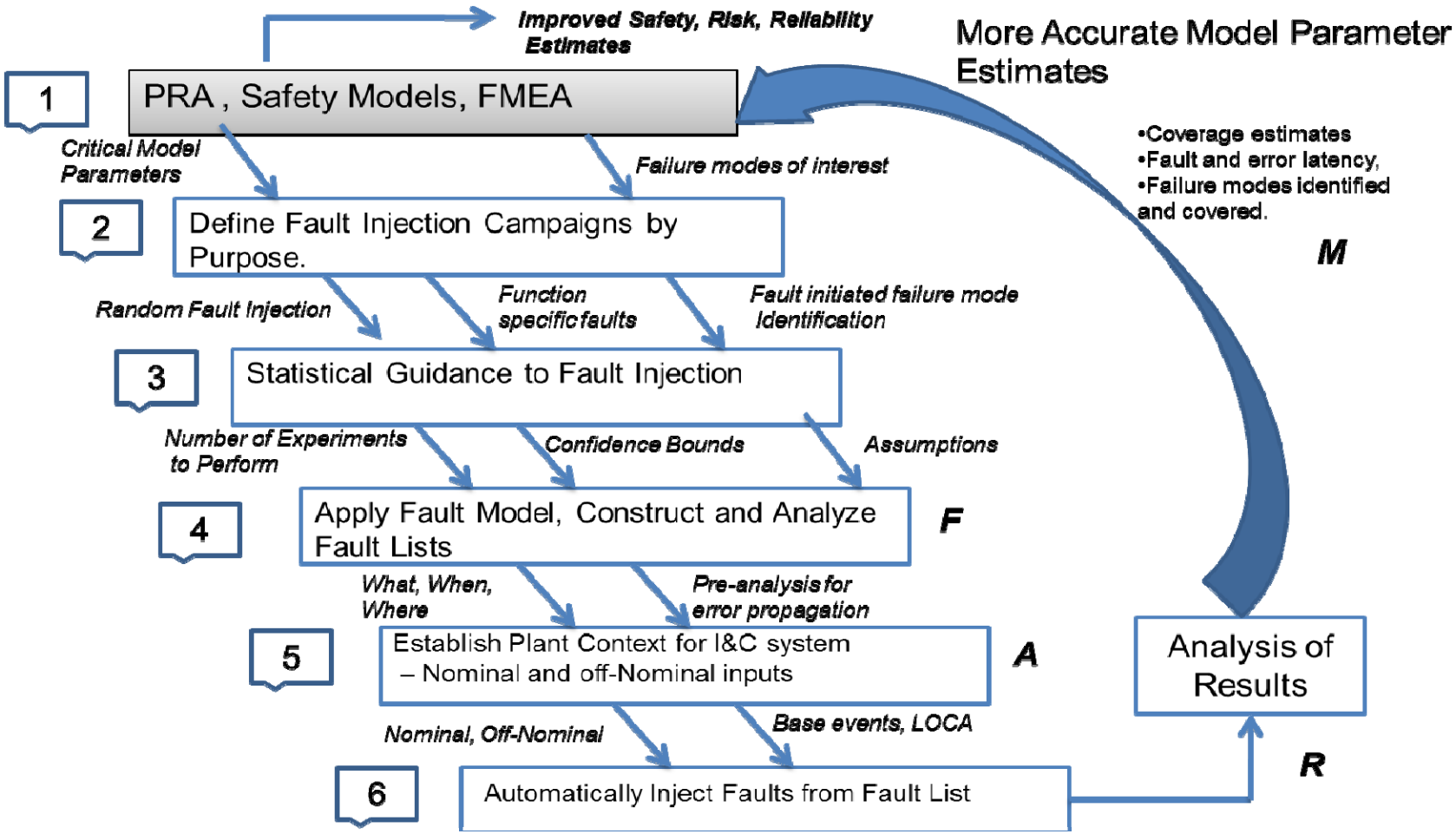


Figure 1-5 Operational view of the fault injection based dependability assessment process

1.8.2. Step 0: Defining the Dependability Metrics

The assessment process begins with defining or selecting the dependability metric of interest that serve the PRA activities. The metrics that can be used in I&C systems include but are not limited to, system reliability, probability of coincident failure, system safety, probability of failure on demand, mean time to system failure, mean time to unsafe system failure, and steady state unsafe system failure. For instance, an actuation system such as a Reactor Protection System would be more accurately characterized by probability of failure on demand, and an instantaneous availability metric rather than a mean time to failure (MTTF) or a system reliability metric. So, how the system is employed in the context of the plant is very important to the selection of an appropriate metric. In the case of the RPS, it is a reactive system. A reactive system is characterized by its ongoing interaction with its environment, continuously accepting requests from the environment and continuously producing results [Wieringa 2003]. In reactive systems, correctness or safeness of the reactive system is related to its behavior over time as it interacts with its environment. Unlike functional computations, which compute a value upon termination, reactive system or computations usually, do not terminate. If they do terminate, it is most often due to the fact that an exception event has occurred. Example applications of reactive systems include process control systems, actuation systems, operating systems, and telecommunication protocols.

1.8.3. Step 1: Support for PRA Activities

Referring to Figure 1-5, the starting point in the methodology is to understand what is needed from the PRA process. The purpose of reliability and safety assessment process is to ensure a system will meet its reliability and safety requirements, show that risk mitigation measures produce reliability and safety improvements, and the unreliability risk is controlled to an acceptable level. A *probabilistic* safety and reliability safety assessment process usually begins with asking three basic questions: (1) what can go wrong, (2) what is the likelihood, (3) what are the consequences?

The PRA modeling process usually begins with defining the scope of the analysis and a set of hazard states of interest and then models (e.g. fault trees, Markov models) are used to characterize the behavior of the system as sequences of events/actions that could lead to the hazard state. Often measurement based attributes that are appropriate toward informing the risk assessment process are used to define the end state probabilities. These typically include reliability, unreliability, safety as a function of time. In a typical PRA process there may be several dependability attributes that are used to characterize the system risk. In digital I&C system reliability assessments, measures such as probability of system failure, probability of coincident failure, probability of failure on demand, mean time to system failure, mean time to unsafe system failure, and steady state unsafe system failure are common.

The important point to make here is that PRA activities employ modeling methods like fault trees, event trees, and Markov models to assist in the determination of risk. These models have parameters that represent attributes of the system, such as physical failure rates, detection capability, capability to tolerate faults, fail-safe capability, repair capability, etc… Fault injection methods provide a means to quantitatively estimate the behavior model parameters of the system. A behavioral model parameter is a measure of how the system behaved or responded with respect to a stimulus (e.g. a fault or set of inputs or a disturbance or all). The important coverage factor parameter presented in Section 3.6 of Volume 1 is a behavioral parameter in the PRA model. Equally important is stating the assumptions the models or model parameters make in light of incomplete knowledge of the systems. Since fault injection provides response information that can be used to statistically estimate these parameters, the quantification of these parameters (in a probabilistic sense) can be used to produce more accurate parameter

estimates for the PRA models which in turn produces more accurate risk assessment to inform the oversight.

1.8.4. Step 2: Fault Injection by Purpose and Type

It is not uncommon to use fault injection for different purposes in order to get a complete picture of the system's behavior response. As indicated in Section 3, Section 3.4 of Volume 1, fault injection is used in both validation processes of digital I&C systems and design processes of digital I&C systems.

From a broader stance, fault injection is viewed as a measurement based process that provides important experimental techniques for assessment and verification of fault-handling mechanisms. It allows researchers and system designers to study how computer systems react and behave in the presence of faults. Fault injection is used in many contexts and can serve different purposes, such as:

- To support on-line monitoring so that systems can react in an appropriate way.
- Assess the effectiveness, i.e., fault coverage, of software and hardware implemented fault-handling mechanisms.
- Study error propagation and error latency in order to guide the design of fault-handling mechanisms.
- Provide evidence to support the resilience of the system to unexpected faults and failures.

Since fault injection can be used for many purposes, is it necessary to identify as early as possible the type of fault injection and the measurements that will be used and whether it will be applied to a physical system or a model of the physical system. All fault injection techniques have specific drawbacks and advantages as indicated in Section 5 of Volume 1. The comprehensive survey and characterization of fault injection methods and techniques presented in Section 5 of Volume 1 serve as a guide toward selecting fault injection for a target digital I&C system.

1.8.5. Step 3: Statistical Guidance to Fault Injection

The purpose of the statistical model is to provide a formal basis for (1) conducting fault injection experiments and (2) providing a statistical model for a estimating the measures of a fault injection experiment, such as coverage. As developed in Section 3 and Appendix A of Volume 1, the statistical model supports four specific needs of the fault injection based dependability assessment methodology:

- Characterize the fault injection experiment in formal statistical framework.
- Quantify and characterize the uncertainty of model parameters.
- Characterize and define the assumptions of the estimation process.
- Statistically estimate, based on the assumptions of the model and model parameters, the numbers of observations required to estimate a parameter to a known confidence level.

1.8.6. Step 4: Fault Model Selection

Digital I&C systems are subject to faults and failures from a variety of sources, and can manifest themselves in many ways as was discussed in fault taxonomy of Section 1.7.3 of Volume 1 [Avizienis 2004]. Fault models are abstract representations of real faults. For example, single event upset caused by a power surge or a cosmic particle strike can be modeled by the bit-flip fault model. Fault models allow assessors to evaluate the effectiveness of fault detection, diagnostic tests, and fault tolerance mechanisms with respect to the faults that are anticipated to arise in the operation of a digital I&C system. Applying these fault models to I&C systems and observing the responses is a key component of fault injection based assessment processes. Selecting the appropriate fault model for a fault injection campaign is a crucial decision.

In Figure 1-6 are the fault classes selected to apply to the benchmark systems based on our research on fault and failure behavior of Digital I&C systems. These fault models and their justifications were discussed in detail in Section 4 Section 4.5 of Volume 1.

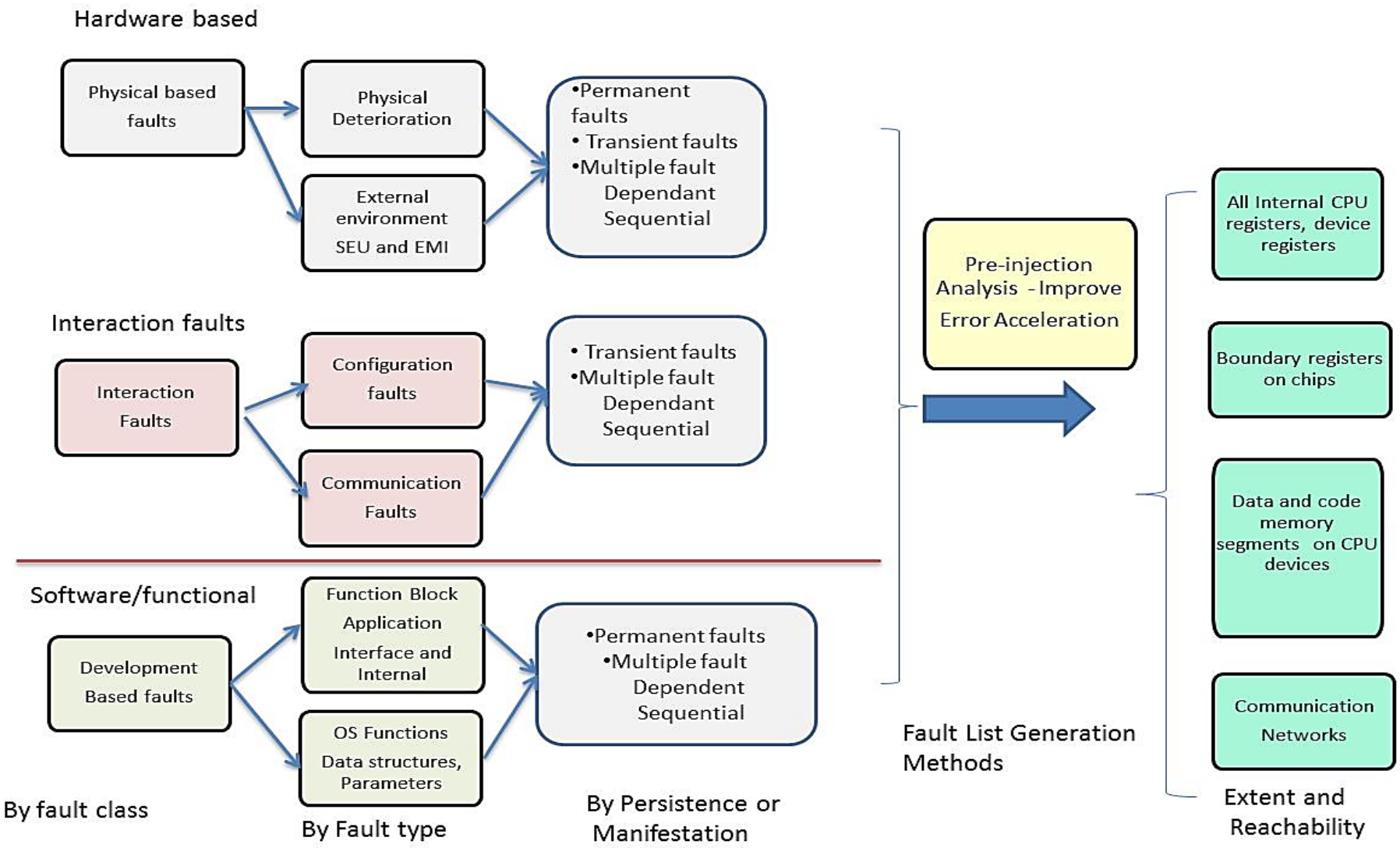


Figure 1-6 Fault model classes for benchmark digital I&C systems

Finally, the output of a fault model selection process should produce a set of faults that is relevant to a particular digital I&C system, but more importantly the process of fault model selection produces an audit or evidence trail so the assumptions, factors for determining the fault models, can be assessed during the licensing and review activities.

After a representative set of fault models has been selected, the next step is to determine a means for organizing and applying these faults to the digital I&C system. This activity is called *fault list generation*. Generating fault lists for a fault injection experiment or campaign is a critical activity for fault injection. A *fault list* is a sample set of faults taken from the fault space of the target I&C systems. Specifically, for a single fault notation in a fault list, each entry identifies.

- The type of fault to be injected – Governed by the fault model selection.
- Where the fault is to be injected – Where the corruption is to take place with respect to program execution behavior or component use.
- When the fault is injected – At what time the injection takes place, either relative to an event or when a resource is in use, or randomly selected.
- How long the fault is injected – The persistence of the fault with respect to the time domain.
- The error mask of the fault – What values represent the fault injection process with respect to a resource in use or a component.

The fault list can be thought of as a set of directives to the fault injector apparatus. Each of the directives is under experimental control of the experimenter. The fault list is used to instruct the fault injection process according to a particular campaign purpose. The fault list is strongly tied to the fault injection environment and its capabilities to emulate the faults of concern.

An important aspect of fault list generation is improving the efficiency and effectiveness of the fault injection process. Improving the efficiency and effectiveness of fault injection is often called *error acceleration* [Chillarege 2002] or more recently *pre-injection analysis* [Sekhar 2008; Barbosa 2005]. Pre-injection analysis is method to guide the fault injection process to produce more effective and efficient results.

Pre-injection analysis is defined by a set of rules that forces fault injection experiments to push the limits of the measurement on the probability of systems failure. Section 8 of this report presents new fault list generation methods that produce efficient and effective fault injection results for digital I&C systems.

1.8.7. Step 5: Establishing Operational Profile and Workload

An *operational profile* (OP) is a quantitative representation of how a system will be used within its use environment [Musa 1998]. It models how users interact and use the system, specifically the occurrence probabilities of system and user modes over a range of operations. Traditionally, it is used to generate test cases and to direct testing to the most used functions thus the potential for improved reliability with respect to the use environment is achieved. It associates a set of probabilities or weighting factors to the program input space and therefore assists in the characterization of possible behaviors of the program or collection of programs that represent a system.

As discussed in Section 4 of Volume 1, digital I&C systems that are real-time and reactive operate on a deterministic time-triggered basis. The difference between an OP for general purpose computing and a real-time OP is that general purpose OPs typically represents many customer or user domains, while real-time OPs are specific to a particular application (user) and its environment. In this effort, an operational profile is defined in the context of its application specific nature (Reactor Protection System).

Real time operational profiles to be used in the fault injection experiments must be selected to be representative of the system under various modes of operation and configuration. Digital I&C System configurations may invoke different hardware and software modules in response to real time demands, and it is important that the fault injection assessment include sufficient combinations of these to ensure a thorough evaluation of their behavior in the presence of faults.

1.8.8. Step 6: Injecting Faults into the Target System

Figure 1-7 shows the essential components of a fault injection environment. There are a number of fault injection techniques and tools that are available to the designer for dependability validation. Section 5 of Volume 1 provides a detailed survey, classification of the various fault injection techniques that are applicable to digital I&C system.

Referring to Figure 1-7, the generic representation of the fault injection environment combines the key elements of a fault injection methodology into a setting where the theory of fault injection can be realized. In Figure 1-7, the *target system* is the system that the fault injection is applied to. The target system executes tasks assigned from the application workload environment. The *application workload(s)* of the system are representative programs which the target system typically executes in its application domain. The *operational profile or inputs* define the input domain for the target system with respect to the various workloads that the system may execute. *The fault library* contains lists of faults which will be injected into the target machine by the fault injector. These include faults which are judged to be representative of fault classes that are expected to be encountered. The *monitor* globally keeps track of execution on the target and initiates data collection when necessary. The *data collector* should be capable of capturing the effects of faults as they propagate through the execution on the target. The effectiveness of the data collector would determine the quality of the results obtained upon analysis of the collected data.

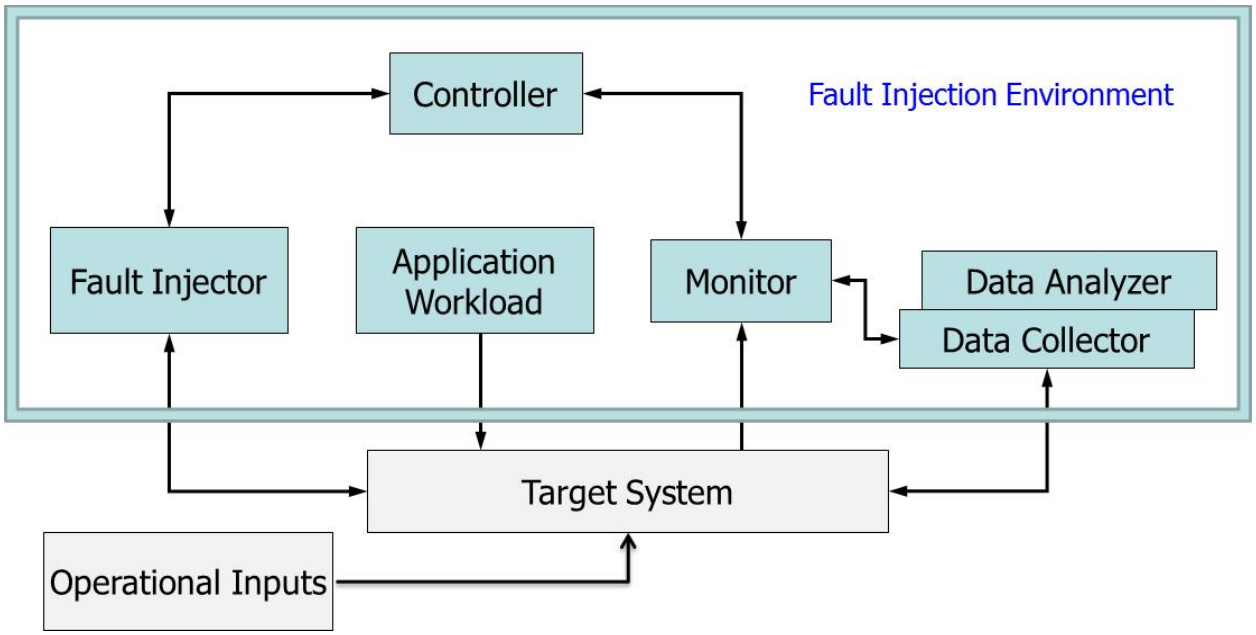


Figure 1-7 Basic architecture of a fault injection environment

Section 5 of Volume 1 developed a set of requirements and a implementation of those requirements for fault injection environments that are particular to the architectures of digital I&C systems, and that satisfy the needs of the FARM model. These requirements pertain to providing capability for automation of fault injection, and some others are concerned with ability to represent different fault models so that a wide variety of failure modes can be tested.

Support for Representative Fault Models – An effective fault injector must be able to emulate various fault models or fault classes so that the assessor of the dependable system can test the fault tolerance mechanisms under the effect of various types of faults that digital I&C system may subject to over its operational life. The ability to accommodate various fault models or fault classes using the same fault injection environment is a valuable feature. Furthermore, the ability to use several different fault injection techniques from a single environment certainly aids in the overall usability of fault injection from one platform to another.

Support for Precise Fault Activation – This essential feature is related to the requirements of the FARM model. Recall, the fault space $\mathcal{F}$ has three basic dimensions; (1) location of fault activation, (2) time of fault activation, and (3) duration of fault activation. In fault injection, often there is concern with the ability to inject a fault based on these dimensions that might be applied to emulate a certain fault. For example, a fault may need to be injected at a random point in time to emulate a transient fault. Or, a fault may be injected when a certain mode or input condition or certain event occurs on a given variable. The ability to set up composite timing and triggering constraints is an essential feature that ensures various operational modes of the target system can be exercised effectively. Being able to precisely control the time, location, and duration of when a fault is to be injected improves the controllability of the fault injection process, and thus improves the repeatability of the fault injection experiments.

Support for Experiment Control – The design of experiments consists of deciding a number of controllable parameters such as fault location, fault value, fault dependence, fault timing, and persistence of faults. The fault injection environment must provide the capability to easily decide these parameters thereby allowing easy setting up of experiments.

Support for Automation – Being able to automate fault injection is essential to be able to collect large amounts of data so that a good deal of confidence can be placed in the parameters that are determined during statistical estimation. A number of capabilities need to come together to enable automation of fault injection. Some of these are described below. When integrated into a complex microprocessor based system, the fault injector will need to communicate with the system in order to know when to inject a fault, or to convey status messages back to the data collection subsystem or any other part of the fault injection environment. The fault injector must have output signals that it can issue and also input signals that it can detect. Being able to issue and detect these signals is crucial for communication between the target system and the fault injector at different levels. The fault injector should also have commands that can be issued from a remote host computer to perform various tasks, such as the ability to compile and execute command scripts to enable automatic set up of tasks to support fault injection. These various tasks include detection of events, setting up, enabling or disabling software or hardware breakpoints, performing memory/register corruptions, and halting and resuming the central processor unit (CPU) through low level commands.

1.9. References

[Arlat 1993] Arlat, J, A Costes, Y Crouzet, J-C Laprie, and D Powell. "Fault Injection and Dependability Evaluation of Fault-Tolerant Systems." *IEEE Trans. on Computers*, no. 42 (1993).

[Avizienis 2004] Avizienis, A, J-C Laprie, B Randell, and C Landwehr. "Basic concepts and taxonomy of dependable and secure computing." *IEEE Transactions on Dependable and Secure Computing*, no. 1 (2004): 11-33.

[Barbosa 2005] Barbosa, R, J Vinter, P Folkesson, and J Karlsson. "Assembly-level preinjection analysis for improving fault injection efficiency." 2005.

[Elks 2009(a)] C. Elks, B.W. Johnson, M. Reynolds. "A Perspective on Fault Injection Methods for Nuclear Safety Related Digital I&C Systems." *6th International Topical Meeting on Nuclear Plant Instrumentation Control and Human Machine Interface Technology.* Knoxville, TN: NPIC&HMIT, 2009(a).

[Smidts 2004] C. Smidts, M. Li. *Validation of a Methodology for Assessing Software Quality.* NUREG/CR-6848, Washington, D.C.: NRC, Office of Nuclear Regulatory Research, 2004.

[Chillarege 2002] Chillarege, R., Goswami, K., Devarakonda, M. "Experiement Illustrating Failure Acceleration and Error Propagation in Fault-Injection." *IEEE International Symposium on Software Reliability Engineering*, 2002.

[Smith 2000] D. Smith, T. DeLong, B.W. Johnson. "A Safety Assessment Methodology for Comples Safety Critical Hardware/Software Systems." *International Topical Meeting on Nuclear Plant Instrumentation, Controls, and Human-Machine Interface Technology.* Washington, D.C., 2000.

[Barton 1990] J.H. Barton, E.W. Czech, Z.Z. Segall, D.P. Siewiorek. "Fault Injection Experiments Using FIAT." *IEEE Transactions on Computers*, 1990: 575-582.

[Benso 2003] A. Benso. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation,* Springer, 2003. [Musa 1998] Musa, J. *Software Reliability Engineering.* McGraw Hill, 1998.

[Palumbo 1986] Palumbo, D.L., Butler, R.W. "Performance Evaluation of a Software Implemented Fault-Tolerant Processor." *AIAA Journal of Guidance and Control, vol.3, no.6*, 1986: 175-185.

[Sekhar 2008] Sekhar, M. *Generating Fault Lists for Efficient Fault Injection into Processor Based I&C Systems.* Charlottesville, VA: University of Virginia, 2008.

[Aldemir 2007] T. Aldemir, M.P. Stovsky, J. Kirschenbaum, D. Mandelli, P. Bucci, L.A. Mangan, D.W. Miller, A. W. Fentiman, E. Ekici, S. Guarro, B.W. Johnson, C.R. Elks, S.A. Arndt. *Reliability Modeling of Digital Instrumentation and Control Systems for Nuclear Reactor Probabilistic Risk Assessment.* Regulatory Guide NUREG/CR-6942, NRC, 2007.

[Yu 2004] Y. Yu, B.W. Johnson. "Coverage Oriented Dependability Analysis for Safety-Critical Computer Systems." *The International System Safety Conference (ISSC).* System Safety Society, 2004.

[Young 1989] Young, S.D., Elks, C.R. "Performance Evaluation of a Fault Tolerant Processor." *Proceedings of AIAA Computers in Aerospace Conference.* AIAA, 1989. 625-635.

[Wieringa 2003] Wieringa, R.J. "Design Methods for Reactive Systems, 1st ed." Morgan Kauffman, 2003

2. RESEARCH METHODOLOGY

2.1. Overview

The research methodology for this report (Volume 2) consists of six main steps and is described below:

- Identification and Selection of Appropriate fault injection methods for Benchmark System I.
- Developing the RPS software for Benchmark System I.
- Design, development and Implementation of the universal platform independent fault injection (UNIFI)-based fault injection environment for the Benchmark systems.
- Develop the TRACE based Operational Profile Generator Tool.
- Develop Pre-injection analysis methods for fault list generation.
- Conduct fault injection campaigns according to methodology.
- Findings, Conclusions and Recommendations.

Each of these steps will be discussed in detail in the remaining Sections of this report.

2.2. Identification and Selection of Appropriate Fault Injection Methods for Benchmark System I

Realization and application of fault injection for digital I&C systems is a complex process of determining what types of faults to inject into system, how to inject the faults into the system, establish the context of the fault injection process. There are many different types of fault injection techniques which may or may not be suitable for physical based fault injection of a real digital I&C system. Using the survey results from Section 5 of Volume 1, knowledge of the benchmark system critical safety processing components, and the fault models of interest, a set of fault injection techniques are selected that are applicable to the benchmark system. The feasibility of implementing these fault injection techniques are assessed from the perspective of (1) the working knowledge of the system, and (2) the technical working knowledge of the digital I&C system manufacturer. This step of the research process supports step two and five of the assessment methodology. This step in the research process also helped narrow the best candidates for fault injection for not only the benchmark systems. The result of this step is a set of candidate fault injection techniques for the benchmark system.

2.3. Development of the RPS Application

The UVA research team along with the NRC technical manager selected a RPS multi-dimensional trip function that uses a number of reactor variables. The RPS function that was developed for this research is similar to the function used in [Smidt 2004], in that it is a reduced model. All of the typical reactor measurements for the trip function were not used and integrated as in a NPP system. The RPS function used in this research used three process variable measurements: reactor coolant system flow, hot leg pressure, and steam generator pressure. These reactor process variables are monitored to prevent power operation in an off-nominal basis as would be in an event such as a Loss of Coolant Accident (LOCA).

The application for both systems was modeled on the basis of a typical nuclear power industry protection system trip function. The RPS function was developed using the software development tools and environment for the benchmark system.

The purpose of this work was aimed at developing fault injection methodology for digital I&C systems, and not to produce high quality, high assurance software for the RPS function as would be typically done for licensed digital I&C system. The RPS was developed using the function block oriented auto code generation tools from the vendors of the benchmark systems which are qualified to produce code compliant with the NRC standards.

2.4. Design and Development of the Fault Injection Environment for Benchmark System

In order to provide a systematic process of conducting fault injection campaigns that would be representative of a fault injection testing environment found in industry, a platform independent fault injection environment was developed to implement the methodology.

Most fault injection tools have been developed with a specific fault injection technique in mind targeting a specific system, and using a custom designed user interface. Extending such tools with new fault injection techniques, or porting the tool to new target systems is usually a cumbersome and time-consuming process. Since one of the objectives in this research was to apply fault injection to digital I&C systems of the type found in NPP operations, the need for a flexible and portable fault injection environment was a requirement for efficient application of the UVA fault injection based dependability assessment methodology. Most importantly, the work on researching and developing appropriate fault injection techniques and environments for digital I&C systems produced a body of work that the NRC and the nuclear industry can use to establish a basis for the development and standardization of fault injection. The work presented in this Section has as its aim to explore, develop and prototype such tools to provide a better understanding of how physical fault injection can be effectively and efficiently deployed to contemporary digital I&C systems.

2.5. Development of the TRACE-based Operational Profile (TOP) Generator Tool.

Context is important in fault injection. For a fault injection based assessment methodology, the operational profiles must represent the input conditions and system interactions that can occur not only nominal operations, but also in off-nominal operations and more importantly during “accident” event scenarios. Gathering profile real plant data across all of these domains of operations is challenging task. Not all plants in operation have experienced accident events. Data may be limited due to proprietary sensitivities. In order to provide a diverse and representative set of operational profiles for the benchmark systems, the use of high fidelity NPP simulator tools to generate nominal, off-nominal, and accident event profiles is a most promising way forward. The challenges in this approach are (1) determining how to integrate the thermo-hydraulic modeling tools like TRACE [Commission 2011] into the fault injection environment to act as operational profile generator for the target system, and (2) how to coordinate the selection of the operational profiles to the fault injection process.

This task developed a preliminary tool based on the TRACE thermo-hydraulic NPP modeling simulator to extract operational profiles from model runs. TRACE is a high-fidelity simulator developed for the NRC that is capable of solving complex fluid dynamics and heat transfer problems in components typical of a nuclear power plant – e.g. pipes, valves, boilers, pumps, etc. TRACE models are developed to represent real-life reactor systems and thus are able to capture important interactions between the various systems within the plants. The TOP tool

parses input files from TRACE and produces a set of process variables that are normally used as sensor inputs to digital I&C system. These sensor inputs are organized in *profile file* to represent an operational profile for the digital I&C system. The profile file is then used as input to the sensor acquisition modules of the digital I&C system via the I/O data acquisition of the fault injection environment.

2.6. Development of Pre-Fault Injection Analysis Techniques to Support Fault List Generation

The fault list of fault injection process can be thought of as a set of directives to the fault injector apparatus. Each of the directives is under experimental control of the experimenter. The fault list is used to instruct the fault injection process according to a particular campaign purpose. The fault list is strongly tied to the fault injection environment and its capabilities to emulate the faults of concern.

An important aspect of fault list generation is improving the efficiency and effectiveness of the fault injection process. Improving the efficiency and effectiveness of fault injection is often called error acceleration [Chillarege 2002] or more recently *pre-injection analysis* [Sekhar 2008; Barbosa 2005]. Pre-injection analysis is method to guide the fault injection process to produce more effective and efficient results.

Pre-injection analysis is a means to reduce or eliminate the “no-response” and the long fault latency problem associated with fault injection. Being a statistical experiment, fault injection testing may require a large number of experiments to be conducted in order to guarantee statistically significant results. Thus, efficiency of the fault injection testing is important.

With random fault injection experiments (e.g. with no regard to when and where a fault is injected), a large fraction (up to 90%) of fault injection experiments may have no-response outcomes [Sekhar 2008; Barbosa 2005].

A large percentage of these “no-response” outcomes resulting from fault injections are due to non-use of the corrupted data by the executing program. For example, a randomly generated fault could be injected into a memory location that is not used by an application, or could be injected into a processor register that is not in use by the application at the time of fault injection. These instances in which the system would not respond to an injected fault do not convey meaningful information about the fault tolerance capabilities of the system under test. Since time has an associated cost value, if the efficiency of the fault injection campaign is low, then the cost of the fault injection campaign is increased. This research task develops a pre-fault injection analysis method for physical fault injection to improve the efficiency and effectiveness of fault injection.

2.7. Conduct Fault Injection Campaigns on Benchmark System I

This research step applied the UVA fault injection-based dependability assessment methodology to the benchmark system. Using the fault injection environment and the fault injection techniques developed for Benchmark System I, a number of fault injection experiment campaigns were conducted to assess the capability of the methodology to support PRA modeling activities and supply system dependability information to the regulatory review of digital I&C systems. The fault injection experiments are conducted using operational profiles generated from the TRACE based operational profile generation tool. The fault injection was applied to RPS application memory locations, operating system memory locations, registers in the CPU of the processing module, dual port memory of the CPU of the processing module, and to the X-bus data and token protocol. Approximately 8,000 fault injections were conducted on

processing modules, and 10,000 fault injections on the x-bus protocol. Statistical estimation of parameters of interest include fault coverage factor, latency in fault detection, unknown error codes, faulted token times of X-bus, and detection of corrupted data packets on X-bus.

2.8. Conclusions, and Recommendations

The final step in the research methodology for this report is to reflect on the overall research effort and findings with respect to the objectives of this project; specifically, the challenges that might impede or limit the use of fault injection in contemporary digital I&C systems. The overall utility of the methodology was assessed to provide information on the digital I&C PRA assessment process. Also, the feasibility of fault injection by the digital I&C systems community was assessed as part of the overall V&V effort.

2.9. References

[Barbosa 2005] Barbosa, R, J Vinter, P Folkesson, and J Karlsson. "Assembly-level preinjection analysis for improving fault injection efficiency." 2005.

[Smidts 2004] C. Smidts, M. Li. *Validation of a Methodology for Assessing Software Quality.* NUREG/CR-6848, Washington, D.C.: NRC, Office of Nuclear Regulatory Research, 2004.

[Chillarege 2002] Chillarege, R., Goswami, K., Devarakonda, M. "Experiement Illustrating Failure Acceleration and Error Propagation in Fault-Injection." *IEEE International Symposium on Software Reliability Engineering*, 2002.

[Commission 2001] Commission, U.S. Nuclear Regulatory. *Computer Codes.* April 2011. http://www.nrc.gov/about-nrc/regulatory/research/comp-codes.html (accessed 2011).

[Sekhar 2008] Sekhar, M. *Generating Fault Lists for Efficient Fault Injection into Processor Based I&C Systems.* Charlottesville, VA: University of Virginia, 2008.

3. DESCRIPTION OF BENCHMARK SYSTEM I AND RPS CONFIGURATION

3.1. Introduction

The last Section of Volume 1 introduced by way of overview the architectural features of the benchmark systems use in this study. This Section supplements the overview Section in volume with additional information on the operation of Benchmark System I, in particular the self-testing and fault tolerance features of the system. These features are noteworthy of discussion because they are explicitly tested by the developed and implemented fault injection methods.

3.2. Benchmark System I

Benchmark System I is a safety grade qualified digital I&C system specifically developed for safety or high reliability functions in nuclear facilities. The benchmark system received from the NRC for this study is a scaled version of a typical 4 division RPS. Due to non-disclosure and proprietary agreements the make and model of the target system cannot be disclosed. The salient features of the target system are its ability to be adaptable to plant-specific requirements, with almost varying degrees of redundancy. Its scalability permits development of solutions for a spectrum of safety-related tasks within the NPP systems. Typical applications include RPS and Engineered Safety Features Actuation System (ESFAS) functions.

The benchmark systems used in this effort were testing platforms to exercise the methodology. In that regard the benchmark systems represent the complexity of RPS processing and fault tolerance from both a hardware and software perspective. Typical in-plant RPS digital I&C systems are considerably more enhanced in their fault tolerance and diversity attributes than the representative benchmark systems used in this study. Therefore, results of this study are intended to be a reflection on the ability of the methodology to accommodate fault injection on digital I&C systems, and not be construed as a result on the performance and suitability of the benchmark systems for RPS applications.

3.2.1. Architecture and System Description of Benchmark System I

3.2.1.1. Overview

Figure 3-1 shows the architecture of the Benchmark System I. The system is comprised of 4 separate processors each acting as a processing channels or division for the RPS application. Data exchange via fiber-optic bus systems distributes information to each processing channel such as sensor values, fault messages, and process parameter messages. The Bus protocol for the data exchange network is an IEC standard supervisory control and data acquisition (SCADA) protocol which will be referred to as X-bus. The inter-channel X-bus network is usually configured as a point-to-point topology, but can be configured as a linear bus or ring topologies. In our configuration, it was configured as a point-to-point. Communication between channels on X-bus is deterministically upper bounded by the synchronous circulating token nature of the X-bus protocol, meaning there is an upper bound for message delivery between processing channels. However, the benchmark system as a whole is not clock synchronized among the processing channels; processing channels operate independently and asynchronously from each other in their execution of a task. The Runtime operating system run time environment (RTE) operates as a deterministic static scheduler for application tasks with several prioritized rates groups. All processing within a rate group is bounded by the cycle time of the rate group. Because of the repetitive cyclic nature of the processing, the execution time skew between processing channels is bounded.

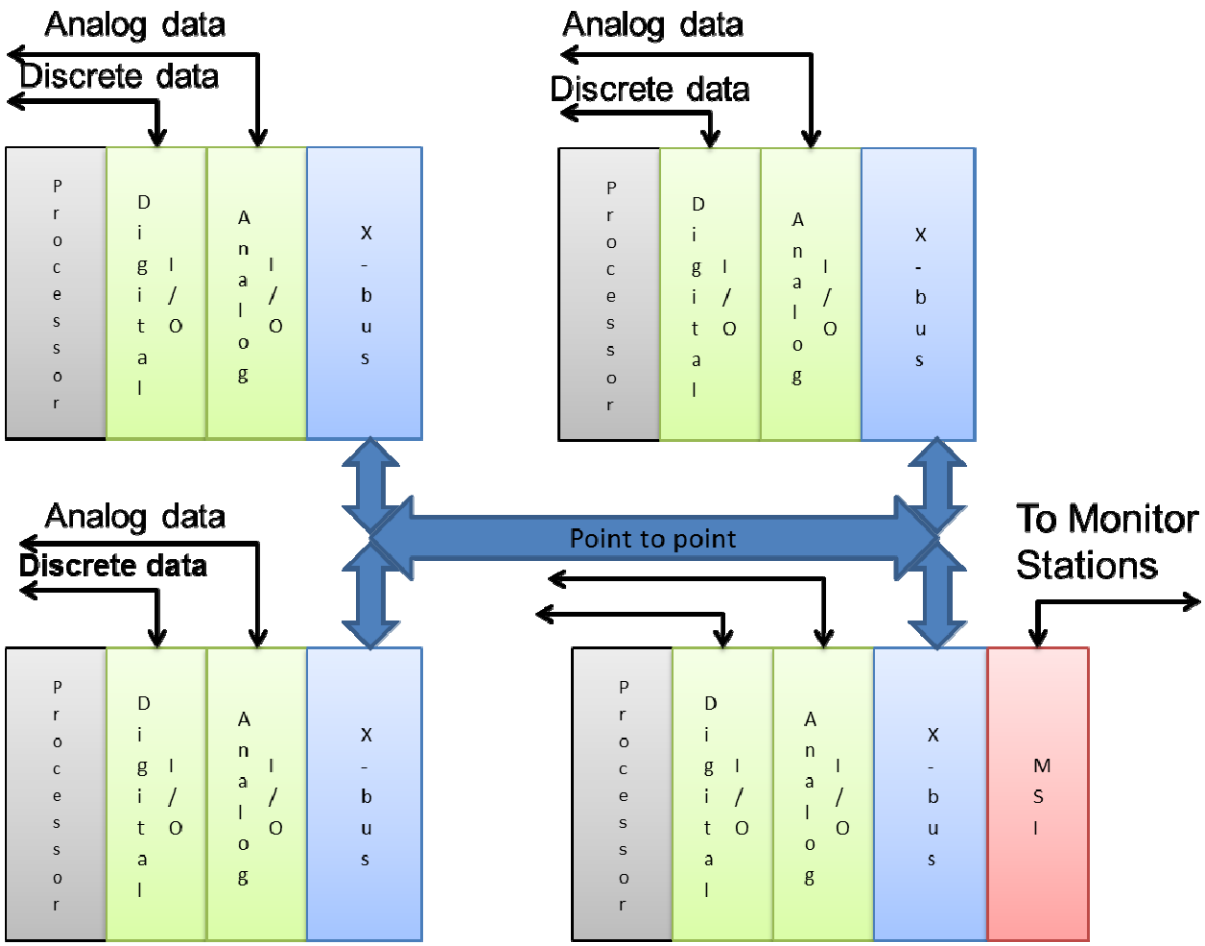


Figure 3-1 Benchmark System I architecture

Each channel typically has its own I/O. This includes multiple modules of digital input, output, analog input, and analog output. Fault masking features for the I/O sub-systems include detection of invalid signals due to known failure modes to improve the availability of the safety I&C functions. These fault masking features of the system include majority voting schemes (typically 2-out-of-4 for binary signals and 2nd minimum/2nd maximum selection for analog signals).

3.2.1.2. Software

The Benchmark System I software is comprised of (1) off-line software development and (2) on-line software to support task reliable execution and fault tolerance capability. The off-line code development environment designs application from a function diagram editor. Function Diagrams are built by selecting and connecting the appropriate function block modules available from a function block library. For each processing module the application software code is compiled and auto generated from this specification (function diagram modules) and then linked to the RTE system software resulting in a set of real-time tasks for the application.

The interface between application functions (function diagram modules) and the system software is generated by the software development code generator tool. It automatically

creates the call and data interface to the function diagram modules and describes the I/O and communication activities which have to be performed by the processing module.

The design of the processing cycle is one of the key preconditions for ensuring deterministic system behavior by maintaining strictly cyclic operation of each processor in a distributed I&C system independently of the status of the plant process. Each processor module runs three tasks being scheduled by RTE on the basis of absolute task priorities:

Priority 1 (highest) – Cycle task: The cycle task operates with a predefined, constant cycle time. It handles all communication via messages, the input and output signals, and the cyclic processing of the application functions. Having the highest priority of all three tasks, it ensures that the cyclic operation of the application functions is always completed within the specified cycle time.

Priority 2 – Service task: The service task processes service commands received from the Service Unit. When no service commands are pending it is suspended. It is reactivated by the cycle task each time a new service message has been received. After processing of the service message the service task is suspended again. There are two types of service requests which can be received from the service unit:

> ***Type 1:*** Such service requests are fulfilled without interruption cyclic processing, such as reading and acknowledgement of system messages, tracing of signal data, on-line modification of operation parameters.
>
> ***Type 2:*** Requests for diagnostic data for fault diagnosis of the CPU or the performance of tests which require the processor to be in the special operation modes TEST or DIAGNOSIS.

Priority 3 (lowest) – Self-test task: The self-test task has the lowest priority of all tasks and is only processed when the service task and the cycle task are not active. The self-test task continuously performs tests of all relevant hardware components on the processor board (RAM-test, ROM-checksums, watchdog-test, etc.). This endless loop of tests consumes all “idle” time of the processor.

Through above separation of functionality approach, the interactions between application, runtime executive, and system services level becomes more structured compared to a combination of all the functionality within a single task.

3.2.1.3. Fault Tolerance and Self-Testing Features

The fault tolerance features of the target system are both application independent and application dependent. Depending on the degree of redundancy needed for an application, the user can configure the system as an *n* out of *m* voting scheme. Where *n* is the number of channels that are in agreement with all other channels, and *m* are the total number of channels in the system. In addition to these application dependent fault tolerance features, the system executes a number of application independent fault detection mechanisms, such as runtime diagnostics, self-tests in the background of the RTE and at start up to detect latent faults in the system.

The basic self-monitoring and self-test features of Benchmark System I are:

- Hardware is tested using extensive self-tests and is monitored (at start-up as well as cyclically)

- The processing modules and the I/O modules monitor the elapsed execution time of their cyclic programs and signal an excessive computation time.

- An independent hardware watchdog monitors cyclic operation of every processor signals a failure independently from the monitored processor.

- Hardware circuitry, independent from the microprocessor and its software, controls the shutdown of the outputs in case of failure.

- Processing modules and I/O modules observe the communication between communication members and check the integrity and validity of received data.

- In case failures are detected, the communication processor marks affected messages and signals as faulty which can be used by application level error detection mechanisms to enforce fail-safe operation.

Figure 3-2 shows the typical end-to-end data processing with respect to the tolerance and self-testing functions that are engineered in Benchmark System I. To begin with, data inputs from the plant are acquired the input processing modules. The Input/Output modules are typically mapped to a processing module on a one to one basis. The input module performs a number of diagnostics and self-tests to ensure the integrity of the incoming data. The self-tests are partially implemented by the firmware (FW) executed in the modules CPU, and partially by the support hardware onboard the I/O modules, independent from CPU. All self-tests are executed in cyclically manner, and at start-up.

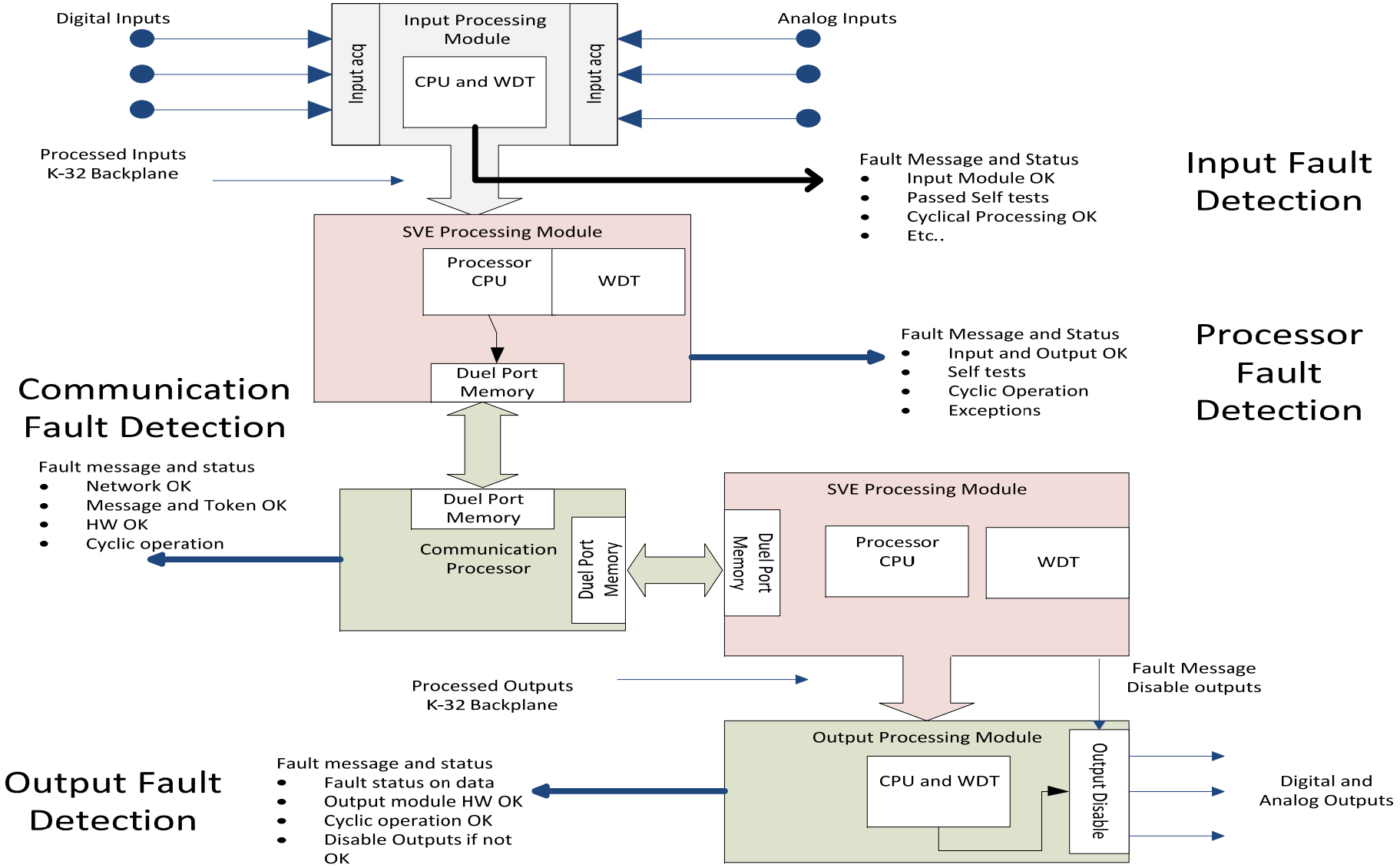


Figure 3-2 Benchmark System I processing

When an internal failure is detected by self-tests affecting the I/O module, the module enters error operation/status mode during which all outputs are switched off (presumed fail-safe state). When an external fault is detected (such as overload or short circuit or open circuit) by the I/O module, the module continues operation but indicates the signals affected by the fault and marks them with the ERROR status. This ERROR status flag can be used by the application software in the processing module to exclude the affected signal from further processing.

Referring to Figure 3-2, the next step of data processing occurs at the processing module. The processing module executes the application software, in the case the RPS software. The processor self-tests are comprised of two parts. The first set of self-tests are executed once during every boot-up sequence, and the second set of self-tests are cyclical -processed repeatedly during normal operation of the processor module. The cyclic self-tests are performed in background mode in the RTE. The cyclic self-tests are executed repeatedly during the cyclic processing as an RTE task with the lowest priority. Thus, the RTE schedules the cyclic self-tests only if no other task with higher priority is pending (like application tasks and service commands). If the cyclic self-test detects an error, it activates the exception-handler and passes error information to it. The exception-handler then executes a reset or shutdown of the processor module.

In addition to the onboard self-tests, the processor modules are equipped with an independent hardware watchdog timer. The monitoring time of the hardware watchdog is defined by the cycle time of the runtime environment. This time is typically set to (RTE cycle time) + 50ms, but can be changed according to the application requirements. The hardware watchdog must be re-triggered by the RTE system software running on the processor before the countdown time expires. If the software fails to do so, a timeout error is assumed and the hardwired WDT signal goes to “low”. This hardware signal is used to signal a processor module failure, and can be used to switch off the I/O modules’ power supply to ensure fail-safe behavior of the outputs.

Intra channel communication between processors is performed cyclically with a fixed communication cycle time $\mathbf{T_{com}}$. The fixed communication cycle time is bounded by the token rotation time of the X-bus protocol and is the same for all processing modules in the system. X-bus messages are sent once every communication cycle. The receiver of an X-bus message performs a series of checks on communication message, these include message header check, message age, cyclic redundancy check (CRC), destination and address checks, frame checks. These checks are described below.

Message Header Check The message header check process checks the following for every message:

- sender ID and address
- receiver ID and address
- message ID
- message type
- message length
- frame check

This information is checked based on the configuration data generated by the code-generator.

Message Age Monitoring The message age is monitored by the RTE cycle counter, which is included by the sender in every sent message. In case one message does not arrive in time, the values of the message of the previous cycle are allowed to be reused. If for two consecutive communication cycles no new and valid message has been received in time, the data included in the message are marked with the ERROR-status.

Data Message CRC Every data message is checked with a 16 bit Commite' Consultatif International de Telegraphique et Telephonique (CCITT) CRC. The sending processor calculates this CRC and appends it to the message packet. The receiving processor recalculates the CRC for the received message data and compares the two CRCs.

If one of the above listed checks fails, the affected data are marked with the ERROR-status. An error message is issued and sent to the monitor service-unit. These checks are performed by the RTE on processing modules.

The importance of these self-tests, diagnostics, error detection mechanisms, and fault tolerance features with respect to fault injection are that they define the core defenses of the benchmark system with respect to expected fault classes. This can be best illustrated by Figure 3-3, where each error detection mechanism defines a barrier to error propagation. It is at (or near) the error detection mechanisms where fault injection tests should be applied to the processing functions of Benchmark System I. Errors that are properly detected and mitigated at each stage of the processing pipeline are said to be covered errors. Those that are not properly detected or mitigated are candidates for uncovered or improperly mitigated errors.

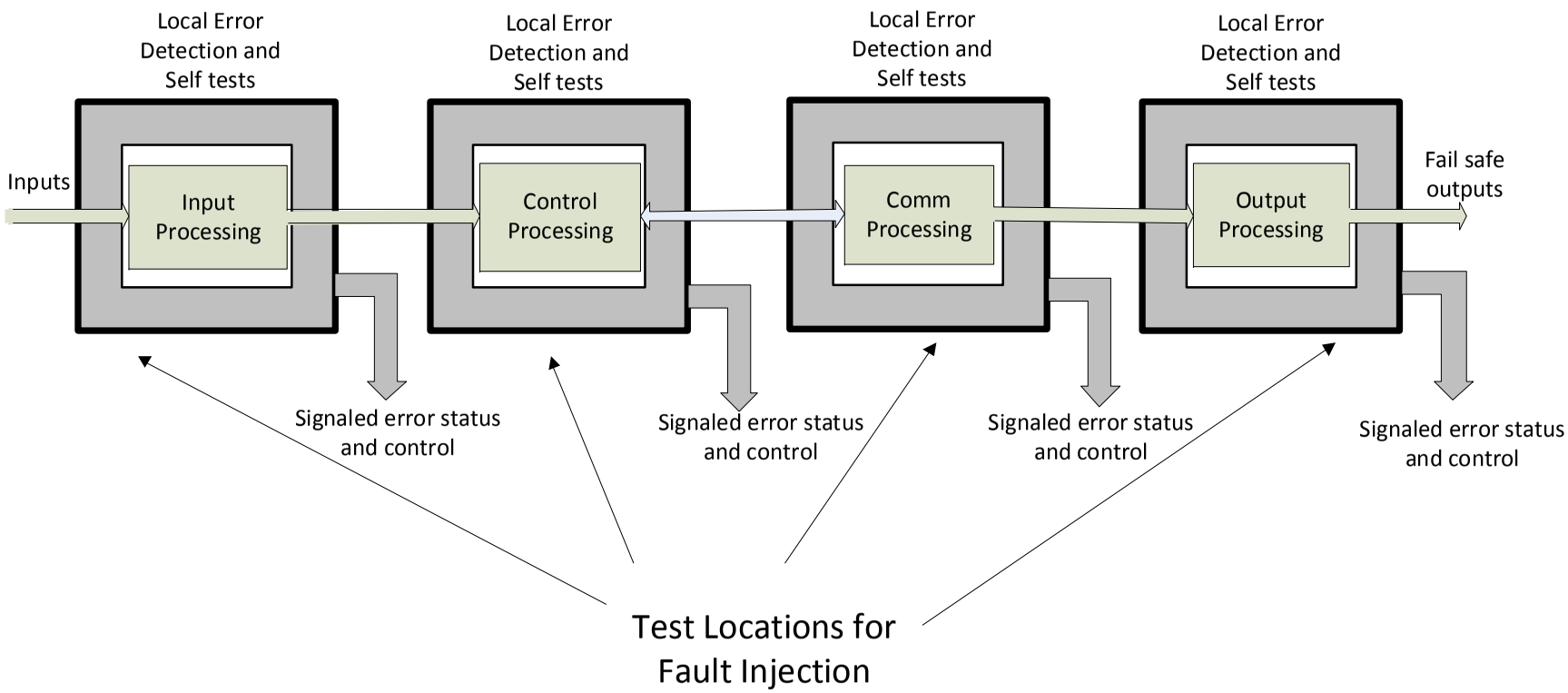


Figure 3-3 Benchmark System I fault tolerant features

3.2.1.4. Monitoring Interface

External communication to non-safety monitoring stations for purposes of monitoring the operation of the application (e.g. RPS) is facilitated by a special interface called the Monitor Interface. The Monitor Interface (MI) is responsible for gathering system level diagnostic health messages, application level messages from the Benchmark I System and forwarding this information to plant and operator monitoring stations. Referring to Figure 3-2 and Figure 3-3, each stage of processing is associated with a collection of error detection mechanisms. Any detected error at any stage of processing results in a mitigation response, and a set of error messages for that error condition are sent to the monitor interface module which in turn forwards the messages to an operator work station. In addition, the Monitor Interface serves as boundary between the safety functions onboard Benchmark System I and the non-safety interface functions that reside outside Benchmark System I. The MI and the messaging protocol is designed to be non-interfering with respect to the safety functions operating on Benchmark System I.

3.2.1.5. RPS Configuration for Benchmark System I

The RPS software development environment for Benchmark System I starts with the specification of an I&C system comprising function diagrams and hardware diagrams and is created interactively using a *function block editor*. This tool performs a series of consistency and plausibility checks on the diagrams created. This type of software compositional process typically reduces the possibilities of error in the plant-specific I&C specification. The concept behind the engineering of I&C functions with these function block code generator systems is based on the graphical "interconnection" of function blocks to produce I&C functions in the form of function diagrams. A graphical specification language is used for this which results in the following:

- A defined set of libraries containing standardized (project-independent) function blocks with specified and tested functionality.

- Implementation of the intended I&C functions by means of interconnection of these blocks (generation of function diagrams).

- The block functions can be controlled through specific parameter settings (parameterization). The function block diagrams are a standard in the nuclear I&C industry.

The software development provides a number of engineering functions in support of the overall process of creating, testing and verifying the operational functionality of the developed I&C code, including:

- Specification of I&C functions and hardware topology,
- Verification of the system specification
- Automatic code generation from I&C system representation
- Verification of generated code
- Validation of I&C functions in a simulation environment
- Compilation and linking of the software for the target system
- Loading the software onto the target system
- Testing the I&C functions on the target system
- Support for Diagnostics and system configuration

Using the software development and testing environment of Benchmark System I, the RPS configuration for Benchmark System I was configured as a two out of four voting system for three monitored reactor process signals. These signals were hot leg pressure, coolant flow, and steam generator pressure. This means that if any two channels indicate that any of the measured sensor variables from the reactor are out of safety range, the Trip Logic will initiate a shutdown command and signal to the reactor to shut down. If a channel becomes faulty and it is detected as so, Benchmark I System gracefully degrades to a two out of three voting scheme to allow continued operation in a limited capacity while maintenance and service can perform off-line diagnostics and repair of the failed channel. Figure 3-4 shows the basic processing with respect to two of the measured RPS signals.

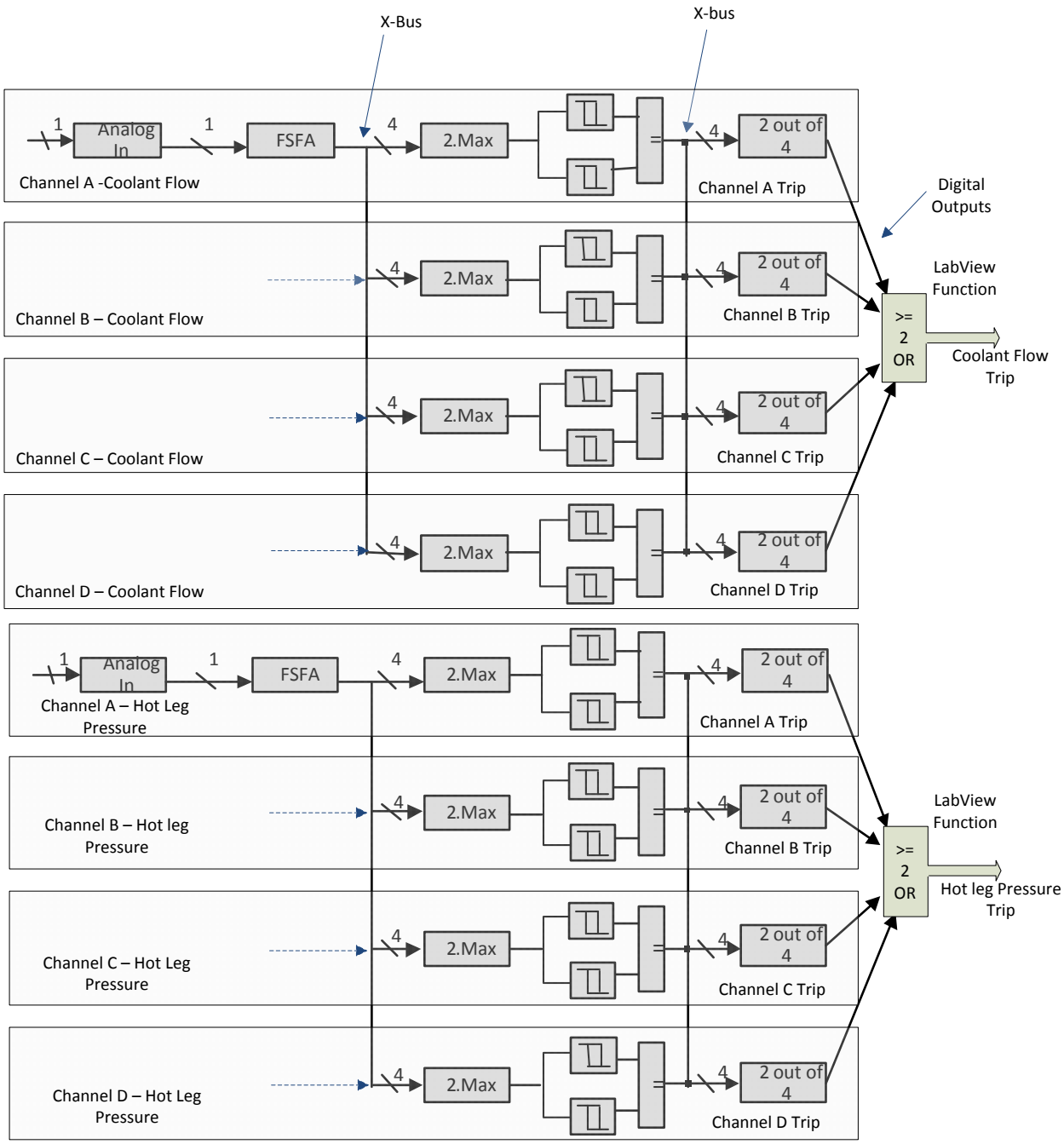


Figure 3-4 RPS configuration for Benchmark System I

Each channel gathers four redundant reactor sensor variables for each of the monitored processes: hot leg pressure and coolant flow. These values are acquired by the analog input modules converted to digital signal representation, and are distributed to all other channels (e.g. channels A, B, C, and D) by X-bus. The sensor values are preprocessed by a 2nd min/max selection function to bound the influence of any outlier sensor values. The 2nd max takes the second highest value recorded from the group of sensor readings. The 2nd min takes the second lowest reading from a group of sensors.

The output of the 2nd min/max function is then provided to a set point comparison function where the conditioned sensor values are compared to maximum and minimum set points for the safe operation of the reactor vessel with respect to the process variable. The output of the set point function is a Boolean – Reactor trip or no-trip. The outputs of the set point functions are then sent to the two out four comparator/voter block. If two or more trip indications have been noted, the two out of four function issues a trip signal. The channel trips are sent to the Labview where they are monitored. If two or more channel trips are indicated then the RPS will initiate a trip to the reactor. In a real in-plant RPS, the channel trip signals of the safety I&C functions are distributed to an additional voting process, which consists of the two computers each running as master/checker pair.

The Benchmark I System in this research did not have this additional 2nd min/max fault tolerance capability. Instead this functionality was emulated in Labview with a simple threshold OR gate.

4. IDENTIFICATION AND SELECTION OF FAULT INJECTION TECHNIQUES FOR BENCHMARK SYSTEM I

4.1. Introduction

This Section describes identification and selection of fault injection techniques for Benchmark System I. The research and development process for this task is as follows

- Identify from Section 5 of Volume I appropriate fault injection techniques for Benchmark System I.
- Determine the feasibility of implementing the identified fault injection given based on time, effort, and vendor support required to implement the technique.
- Select the appropriate fault injection techniques.

4.2. Identification of Fault Injection Methods for Benchmark System I

The previous Section noted in Figure 3-3 the indication points where fault injection should be considered with respect to Benchmark System I. These points included input processing (both analog and digital), the processor unit, X-bus communication module, and the output processing module (digital). Each of these modules instrument a number of self-tests and error detection functions to support high levels of fault detection and tolerance.

4.2.1. Input and Output Processing Modules

4.2.1.1. Overview

The input processing modules for Benchmark System I consist of (1) analog input acquisition module, and (2) a digital input acquisition input module. The analog input modules acquire analog process signals and convert them into a numerical format that can be processed by the processing module. With an analog input module, the signals of up to eight input channels in the case of differential measurement or up to sixteen input channels in the case of measurements with reference to ground can be acquired.

The process signals are applied to the front connector of the analog module. Signal processing of the acquired analog signals are accomplished by an onboard micro-controller. The analog process signals are multiplexed to the input of a programmable amplifier where they are amplified before being inputted to the A/D converter. The A/D converter samples the analog signals and converts the signals to digital representation. The binary output values of this analog-to-digital converter are transmitted to a signal buffer and stored there.

The onboard micro-controller of the analog input processing module coordinates acquisitions of the A/D circuitry, the storage of the digitized signals, and the requests for the digitized inputs values from the processing module. In addition to these processing functions, the micro-controller runs self-tests to determine if errors occurred during the signal conversion process.

The digital input module is considerably simpler in design than analog processing module. The digital input modules serve for the acquisition of 32 binary process signals with signal levels of 24 VDC. The input circuits of the S430 and the S431 suppress interferences and convert the input signals into the internal signal level (5VDC) of the module. The acquired signals can then be read by the processing module via the backplane bus. An onboard microcontroller controls

the process of storing the digital values in a temporary input buffer, and then coordinating the transfer to the processing module.

The digital output module is used in the benchmark system for the output of control commands to actuators, motors, pumps, etc. The outputs are divided into two electrically isolated sections, each of which has two channel groups and each of these has eight outputs. The supply voltage is conducted separately for each range. When the Module receives a "output inhibit" command from the processing module, then the outputs are set to zero. The digital output module uses a similar micro-controller as the input processing modules to coordinate data processing actions between the digital output processing module and processing module. As such, the same fault injection method findings that were indicated for the input processing modules hold for the digital output processing module.

4.2.1.2. Identifying and Selecting Fault Injection Techniques for the Input/Output Processing Modules

A number of key processing functions are involved in the input/output processing, as follows:

- Multiplexing of the analog input signals
- A/D conversion process of analog signals
- Coordination of the A/D conversion by the microcontroller
- Storage of the digitized signals
- Transfer of the digitized signals to the processing module
- Self-tests
- Transfer of digital output commands to the digital output processing module.

Faults can occur at each of these input and output processing functions. The key component in all I/O modules is the onboard micro-controller. It is responsible for coordinating the actions of the signal conversions, signals processing, data transfer, and self-tests. Therefore, the micro-controller in all I/O modules were identified as a component for fault injection.

The analog input processing module has no user accessible ports (e.g. serial or Ethernet) that are desired to allow access to the operations of the micro-controller and a control path for fault injection. The same is true for both the digital input and output processing module. The lack of user accessible input port on all modules rules out fault injection techniques like software implemented fault injection (SWIFI), which require a port to interface to the fault injection exception handlers that execute on the target processor.

The next alternative examined was joint test action group (JTAG) boundary scan fault injection. JTAG boundary scan is an Institute of Electrical and Electronics Engineers (IEEE) testing standard that allows the boundary of a chip (e.g. the pins) to be tested for faults. JTAG fault injection involves invoking the JTAG serial port of device and loading corruption values into the boundary registers of the device. There was no reference of JTAG test ports in the user manuals; however what appeared to be JTAG test ports were noted on both boards. W vendor was questioned about these ports, the vendor stated that the ports were proprietary test ports for which they could not share information.

Another option considered was on-chip debugger (OCD) or in-circuit emulator (ICE) machine based fault injection. With ICE machine fault injection the micro-controller is removed from the IC socket on the circuit board and replace with a CPU pod that has the same processor as the removed micro-controller. The CPU pod is controlled by an interactive debugger that resides on a host machine. With the interactive debugger and the CPU pod it is possible to perform fault injection in a similar manner as was done on the digital feedwater control system (DFWCS)

reported in Section 7 of Volume 1. The microcontroller used onboard the input and output processing modules did not have full OCD support. However, ICE machine tools were available for the micro-controller but they were beyond are allocated budgetary costs.

Unfortunately, the fault injection techniques identified for the input/output processing module could not be realized by the research effort due to (1) inability to gain access to vendor sensitive information, or (2) time and cost constraints of implementing ICE based fault injection. Nonetheless, the techniques that were identified can be investigated by the vendors for suitability and feasibility. To partially compensate for lack of true fault injection capability at the input processing module level, the digitized input values from the analog processing module were intended to be corrupted as they are received by the processor module. These fault corruptions would only emulate faults that occur during storage and transfer of the digitize signals from the input processing module to the processor module.

In addition, through the UNIFI fault injection tool the input signals to the benchmark system can be corrupted with various environmental disturbance functions such as Gaussian noise and loss of signal.

4.2.2. Processing Modules

The processing module in Benchmark System I is an x86 32 bit CPU. It is responsible for executing the application code for the RPS functions. The processing module is available in two variants: an AMD processor and a Pentium processor. The key functions of the processing module are as follows:

- Scheduling and executing all system start up self-tests
- Executing all application I&C functions
- Coordinating data transfer to and from all input and output processing modules
- Interfacing to the X-bus inter-processor communications module
- Executing cyclic self-testing

The processing modules are the heart of the benchmark system and, as such, considerable attention was directed toward identifying appropriate fault injection techniques. Section 3 describes the fault tolerance, software, and self-testing capabilities of the benchmark system. Many of these features rely on the capabilities of the processor module to operate effectively.

4.2.2.1. Identifying and Selecting Fault Injection Techniques for the Processing Modules

An effective fault injector must to be able to emulate various fault models or fault classes so that the assessor can test the fault tolerance mechanisms under the effect of various types of faults a digital I&C system may be subject to over its operational life. The ability to accommodate various fault models or fault classes using the same fault injection environment is a valuable feature. Furthermore, the ability to use several different fault injection techniques from a single environment aids in the overall usability of fault injection from one system to another.

The survey results in Section 5 of Volume 1 and knowledge of the benchmark system for processor based fault injection were used to identify several fault injection techniques that could be successful on Benchmark System I. These techniques were selected based on the following criteria:

- Applicability to the benchmark system – Techniques can be accommodated by the benchmark system as it is configured.

- Controllability – Techniques that allow precise control over the attributes of fault model parameters; time, value and location.

- Support for a variety of fault models – The faults models listed in Figure 1-6 of Section 1 serve as the fault model objective for fault injection. These fault classes include: (1) hardware based faults including transient and permanent faults, (2) interaction faults including communication faults and configuration faults, and (3) development-based faults including faults in software I&C based functions and operating system-based functions. Techniques that can maximally support these fault classes are favored.

- Support for Precise Fault Activation - This essential feature is related to the requirements of the FARM model. Being able to precisely control the time, location, and duration of when a fault is to be injected improves the controllability of the fault injection process, and thus improves the repeatability of the fault injection experiments.

Based on the above criteria, three candidate fault injection techniques for Benchmark System I were identified:

- JTAG-based fault Injection
- ICE machine-based fault injection
- SWIFI-based fault injection

Each of these techniques has its advantages and limitations; the aim was to use several of these techniques jointly in a complementary fashion to maximize the reachability and effectiveness of the fault injection studies. Each of these techniques are discussed in the following sections.

4.3. IEEE 1149.1 JTAG-based Fault Injection

Most current integrated circuits have external input and output pins linked together in a set called the *Boundary Scan Chain* (BSC). The Joint Test Action Group (JTAG) was designed to be able to access BSC by means of a virtual register (Boundary Register) connected to its input and output pins. It is possible to alter the contents of BSC and hence alter the current signals on the pin-outs by serially shifting in data into the Boundary Register. At the same time, bits from the Boundary Register are serially shifted out to the output pin of the JTAG controller. Because of the common occurrence of a JTAG port on the current processors and IC devices, there has been fairly extensive work done on attempting to perform fault injections via this technique.

One of the first works on this issue claimed that the IEEE 1149.1 (JTAG) standard, when used as a standalone technique by using the BSC architecture, cannot handle the requirements for a successful fault injection campaign and would require additional logic and functionality for satisfying performance requirements and accessibility of internal components [Santos 2003].

However, at around the same time, researchers at Chalmers University of Technology (CUT) have shown that Scan Chain Implemented Fault Injection (SCIFI) performed by accessing built in logic specified by IEEE 1149.1 can be used for dependability validation of embedded computer systems and improves the controllability, reachability and observability of the system over standard hardware fault injection techniques [Folkesson 2003]. Moreover, CUT has shown

that the SCIFI technique provides significantly faster performance than software-implemented fault injection.

Since then, researchers have performed validation of various systems by performing fault injections based on accessing the capabilities of JTAG interface. The tests were performed either by connecting the fault injector to a backplane of the system [Chakraborty 2007], or by creating a switch module for accessing all of the components [VanTreuren 2007]. Unfortunately, both of these fault injection campaigns required additional hardware to be introduced into the system to perform the fault injections. This problem was solved in [Pignol 2007] where an off-the-shelf JTAG was used in conjunction with software running on a host computer. The fault injection campaign was successful, but it was noted that real-time operation of the software was problematic and could lead to significant overhead. In addition, the experiments utilized intrusive software running on the target system. Most recent solutions for performing JTAG fault injections employ an FPGA-based fault injector that was programmed to perform experiments with a predefined set of faults [Portela-Garcia 2007]. This solution was unobtrusive as it did not require any additional hardware on the target system and the performance was not degraded by including a host computer as the driver. However, preprogramming an FPGA with the fault injection sequence deemed this solution to be specific to one campaign. Also, performance in fault injection with target real-time systems could not be established.

To summarize this review of JTAG fault injection systems:

Advantages:

- It has been shown that it is possible to perform successful fault injection campaigns by using the functionality provided by a JTAG on-chip device.
- Can provide a means for injecting faults internal to integrated circuits, processors, FPGAs and application specific integrated circuits (ASICs) when other techniques are difficult to implement.
- Usually provides good controllability and observability.

Disadvantages:

- Fault injection is limited to locations that can be reached by the boundary scan registers.
- Requires external hardware be designed or purchased to access the Test Access Port (TAP) of the device.
- The device under test is temporarily taken out of operational mode and put into test mode while fault injection is active.

The attraction of JTAG fault injection is that it complements other fault injection methods (such as (ICE-based Fault Injection or OCD-based fault injection) by providing a diverse method that reaches different portions of the internal IC under test. In addition, since most processors and complex IC devices have a JTAG port, it provides a means to gain access to the IC for fault injection when other means are not possible. This is particularly true for devices like FPGAs that are principally hardware devices and usually do not have a software component.

For these reasons, the design and implementation of JTAG fault injection module was used for the UNIFI fault injection environment, which is described latter. The JTAG fault injector could be used late in the project and therefore it could not be the principle means of fault injection.

4.4. ICE-based Fault Injection

Previous experiences with ICE-based fault injection in the DFWCS fault injection study provided mixed results. ICE-based fault injection provides a ready-made solution for integrating fault injection into a digital I&C system, however it was shown that it is less than optimal for injecting faults into real-time systems. The time delays for halting and resuming the processor to insert a fault into the processor or memory of the target system can be significant (10's of milliseconds or greater). OCD fault injection solutions are far better for minimizing time delays associated with fault injection, however, OCD solutions require that the target processor have built in OCD hardware functions to support fault injection. Unfortunately, in the case of Benchmark System I, this was not the case. Both variants of the processor modules did not support OCD functionality due to obsolescence issues (i.e., the modules were designed in the mid 1990s). Most processors designed and marketed after ~2000 have OCD support of some type. For this reason the OCD based fault injection on Benchmark System I could not be explored. In order to ensure fault injection capability on the processors of Benchmark System I, it was decided that ICE-based fault injection would be the principle fault injection technique even though it may not be optimal.

Recall from Volume 1, an ICE machine is a tool used by designers of embedded systems to debug embedded software. Debugging embedded system software is particularly challenging because embedded systems usually lack suitable user-interface devices such as keyboards and displays. Under such circumstances, ICE machines provide a `window' into the system through which the designer can exercise a good deal of control of the embedded system at a very low level (e.g. assembly code, and signals). In-circuit emulators usually have a CPU pod that plugs directly into the socket where a CPU chip is inserted. There is interface circuitry that provides a connection between an ICE machine and a terminal computer. This terminal can be used to run an interactive user interface application through which a designer can monitor the embedded system being designed.

All ICE machines use a graphic user interface (GUI)-based debugger command tool to control the ICE machine (e.g., performing functions like halting and resuming the processor and altering the contents registers and memory). For fault injection the GUI based debugger command tool is not appropriate for automation purposes. Instead, the use of command script files, which contain a sequence of shell commands that when used in sequence allow the basic steps of fault injection to be executed in a serial manner. A script file is a sequence of commands and command arguments that the ICE machine uses to perform its debugging functions. This is a key capability in the realization of fault injection using the ICE machine. These script files enable automatic fault injection.

Based on previous experiences with ICE based fault injection it was decided early on to work with the selected vendor of the ICE machine to address performance issues experienced in the DFWCS fault injection study. The principle concerns were twofold:

- Reducing the time delay impact of an injected fault
- Supporting pre-fault injection Analysis

Initial and ongoing discussions with the vendor suggested that time delays associated with fault injections into registers and memory would be no more than 10ms, and usually less. While this value is relatively large compared to the cycle time of the application (e.g. 10ms out of 100ms),

due to the non-synchronized operation of Benchmark System I, this time was considered acceptable. Breakpoint initiated fault injections have been stated to be on the same order of magnitude.

As discussed in Volume 1 and detailed in later Sections of this volume, a technique called pre-fault injection analysis was developed to maximize error acceleration and reduce no-response faults. To effectively implement the technique on a real target system, the pre-injection analysis algorithms needed extracted traces of executing code to determine when to inject a fault for maximizing the error propagation potential. Therefore, the ability to extract traces from the operating Benchmark System I by the ICE machine was needed. The vendor stated that trace collection capability was possible, but was limited to 5 to 10ms worth of program execution data. While extracting a full trace (1 cycle time or 100ms) was preferable, it was decided that some trace data was better than no trace data.

Based on these preliminary investigations and findings, it was concluded that an ICE-based fault injector for the benchmark system could be implemented.

4.5. Software Implemented Fault Injection (SWIFI)

SWIFI encompasses techniques that inject faults through software executed on the target system.

In run-time injection, faults are injected while the target system executes its application or workload. This requires a mechanism that i) stops the execution of the workload, ii) invokes a fault injection routine or Interrupt Service Routine (ISR), and iii) restarts the workload. Thus, run-time injection can incur a significant run-time overhead if the implementation of the fault injection method is not optimized. Software-implemented fault injection (SWIFI) relies on the assumption that the effects of real hardware faults can be emulated either by manipulating the state of the target system registers and memory via run-time injection, or by modifying the target workload through pre run-time injection. This assumption usually holds true for transient faults, but for permanent faults it presents some difficulty due to the repeated invocation of the fault injection exception handler every time a register or memory location is referenced.

The research interest in SWIFI type fault injection was centered on the enhanced performance it can provide compared to ICE-based fault injection. SWIFI type fault injections can execute processor register and memory fault injections in a few milliseconds or less. With appropriate internal processor support for breakpoint registers SWIFI can trap on user variables, operands, instructions, and control flow conditions, which is a desirable feature for precision fault control.

One example of SWIFI based fault injection is Xception [ref], which injects faults through an ISR executing in kernel mode. The ISR is typically added to the Operating System Software as a build or patch. The fault injection ISR typically is loaded with a fault list from a host. The ISR therefore must have an available dedicated external port on the target system to use for communication to the host. The fault injection ISR can be triggered by the following CPU events:

- op-code fetch form a specified address,
- operand load from a specified address,
- operand store to a specified address, and
- a specified time passed since start-up.

Using these triggers it is possible to emulate both permanent and transient faults.

The disadvantage to SWIFI is that it requires modifying the system software or adding software to the target system (e.g. in the way of a low level Interrupt Service Routine). This can be challenging if the complete details of the system software are not known, particularly with respect to implementing the ISR so that it is properly nested in the processor interrupt chain. In addition, a “free” serial or Ethernet port must be available on the target system to communicate with the fault injection host computer. In the case of Benchmark System I, there is a user defined serial port available. The manuals did not clearly define how to link this serial port from the operating system.

To effectively design, develop and implement a SWIFI type fault injector requires that the vendor supply information about the details of the system software so that the ISR can be integrated into the operating system properly and so that it can communicate with the host fault injection controller computer. Citing the proprietary basis of this information, the vendor would not release the information. Without assistance from the vendor, SWIFI-based fault injection for Benchmark System I was not used. However, SWIFI-based fault injection may be a good fault injection technique for Benchmark System I the vendor may want to pursue. As indicated in Volume 1 Section 5, there are number of academic tools and a few commercial tools that support SWIFI style fault injection (Xception, generic object oriented fault injection (GOOFI), and UNIFI) that may be of interest to the vendor.

4.6. X-bus Communication Module

The X-bus communication module provides communication connectivity between the processing modules in the Benchmarks System. The RPS application sends input signal information and channel alarm status information over the X-bus to the distributed processing modules. Therefore, these X-bus interactions between processing modules are critical to the fault tolerance and fail safe capabilities of the engineered I&C functions and the benchmark system. X-bus communication represents the class of interaction faults discussed in the fault model taxonomy shown in Figure 1-7.

For execution of the X-bus protocol and firmware, the X-bus communication module is equipped with a 16-bit microcontroller. The communication module is a piggyback module for the main processing module. It is used in combination with the main processor as the basic module to form communication processors for the benchmark system. Special purpose communication software executes on the processing module and the communications module to facilitate transfer of the data from the processing module to the communication module through dual port memory. X-bus messages onboard the processing module are scheduled on a cyclic basis to be transferred to the communications processor through the dual port memory.

4.7. Identifying and Selecting Fault Injection Techniques for the X-bus Communication Modules

The X-bus communication protocol is an IEC standard protocol for connecting distributed embedded systems in process control and automation applications. As such, it has a detailed open standard to reference upon. Reviewing the detailed design of the X-bus communication module suggested several complementary methods for injecting faults or failures into the X-bus communication network. The first concept investigated was using JTAG fault injection to inject faults into the main 16-bit microcontroller. For similar reasons as stated before, the details of the JTAG scan chain of the communication processor were not available to use. Due to the generation age of the processor family, neither OCD-based fault injection nor ICE-based fault injection solutions were feasible.

Since X-bus protocol is an open IEC standard, it was decided that building an X-bus protocol and message corruption injector that would intercept and corrupt messages/control tokens as they communicated between processor modules could prove to be useful. The problem with this approach is that it is limited to corruptions in the X-bus protocol as it is on the wire. Faults that occur internal to the X-bus communication module could not be represented very well or at all. Nonetheless, it was determined that the exercise of designing and using an X-bus fault injector would provide useful information to the digital I&C community on such fault injection techniques. Later Sections discuss the X-bus fault injector.

4.8. Summary of Fault Injection Techniques for Benchmark System I

Table 4-1 presents suggested physical based fault injection for Benchmark System I. These recommendations are presented from the view of the technical staff of the vendor or an independent assessor who has technical expertise and knowledge equivalent to the vendor. Thus, Table 4-1 reflects what is possible given sufficient technical information about the system.

The most prevalent and accessible fault injection technique for Benchmark System I is JTAG or SCIFI fault injection. It was assumed that the microcontrollers on the input and output processing modules, the CPU on the computational processor, and the CPU on the X-bus communication processor have accessible JTAG scan chains. ICE-based fault injection is an option on the computational processor module and the Ethernet communication processor, but with the aforementioned caveats of time delays when conducting a fault injection. SWIFI is another fault injection option for the processor module, but requires adding a special purpose exception handler to the system software. SWIFI, ICE-based fault injection and SCIFI provide the least amount of investment in terms of time and resources in that order.

Communication or interaction faults likely would be best represented by two different fault injection methods. The first would be a fault injection technique like SCIFI to introduce faults into the communication CPU of the X-bus processing module. The second fault injection technique is an interceptor or jammer module that corrupts various bit fields in the X-bus protocol as it being transmitted over the network. The fields that are bolded represent techniques that were planned to be implement based on the level of information about the system.

Table 4-1 Fault injection techniques for Benchmark System I.

Module	Suggested Physical-based Fault Injection Techniques For Benchmark System I					
	JTAG or SCIFI	**ICE Machine-based Fault Injection**	**Communication Processor Fault Injection**	**Pin Level Fault Injection**	**SWIFI**	**OCD-based Fault Injection**
Input Analog and Digital	Recommended for microcontrollers	If ICE machine is available in market	N/A	If JTAG is not available	Possible, but requires a test port	Not supported
Output Digital	Recommended for microcontrollers	If ICE machine is available in market	N/A	If JTAG is not available	Possible, but requires a test port	Not supported
Computational Processor	Recommended for main processor and support ICs	As alternative to JTAG	N/A	Not recommended	If properly integrated into operating system	N/A
X-bus Communication Processor	If supported	As alternative to JTAG	Preferred for emulating transmission errors	Not recommended	If properly integrated into operating system	N/A
Ethernet Communication Processor	Recommended for main processor and support ICs	As alternative to JTAG	N/A	Not recommended	If properly integrated into operating system	N/A

4.9. Development of Fault Injection Techniques for Benchmark System I

Based on the findings above, three fault injection techniques were selected to further develop for Benchmark System I. These techniques were (1) ICE based fault injection for the main processing module, (2) X-bus fault injector for the X-bus communication protocol, and (3) JTAG/SCIFI fault injection. The third technique, JTAG/SCIFI, was developed but ultimately not used on Benchmark System I due to inability to acquire JTAG test port information on the various modules of the benchmark system. The objective of this task was to develop a set of fault injectors to emulate the fault models presented in Section 1. The development of each technique is discussed in the next section.

4.10. Development of the ICE-based Fault Injector

This section describes the development and implementation of the ICE-based fault injector for the main processing module of Benchmark System I. Specifically, the development of the fault injector for corrupting register and memory operations of the RPS software and system software as it executes onboard the processing module is described.

4.10.1. Background: In-Circuit Emulators

In-circuit emulators are tools used by designers of embedded systems to test the implementation of a design. The tool provides a “window” into the hardware and software operations of an embedded processing system by allowing the designer to view and/or modify processor and system execution states (instructions, registers, and memory locations) to support validation of the embedded system design.

An ICE is one of the oldest embedded debugging tools, and is still widely used in the embedded systems industry in cases where CPUs do not support OCDs. It is the only tool that substitutes its own internal processor for the one in the target system. Using one of a number of hardware monitoring and control functions for the target CPU, the emulator can monitor everything that goes on in the on-board CPU, giving the user complete visibility into the target system code operation. In a sense, the emulator is a bridge between the target system and the hardware/software development environment, giving the designer both an interactive terminal for peering deeply into the target system operation and a rich set of debugging resources.

An ICE machine’s most fundamental resource is target access: the ability to examine and change the contents of registers, memory, and I/O. However, since the ICE replaces the target CPU, it generally does not need additional hardware or special software to be added or modified to the target system. This makes the ICE machine an attractive fault injection option for digital I&C systems because modifications to system hardware and software during V&V activities are usually discouraged. Breakpoint capability is another important resource useful to fault injection realization. The capability provides the ability to stop a program at precise locations or conditions (e.g., "stop just before executing line 51").

There is an important distinction between breakpoints used by software code debuggers and breakpoints used by ICE machines. Software breakpoints work by replacing the destination instruction with a software interrupt, or trap instruction. Clearly, it is impossible to monitor code in ROM or electrically erased programmable read-only memory (EEPROM) with software breakpoints. On the other hand, ICE machines generally offer a number of hardware breakpoints, which use the ICE machine internal hardware to compare the break condition

against the observed execution stream. Hardware breakpoints work in RAM or ROM/flash, or even unused regions of the processor address space.

Complex breakpoint conditions are another feature of ICE machines that enhance fault injection capability. A typical condition might be: "break if the program writes 0x1234 to the variable buffer, but only if function ‘get data()’ is called first." Some software-only debuggers (like the one included with Visual C++) offer similar power, but interpret the program at a very slow pace while watching for the trigger condition. Emulators implement complex breakpoints in hardware and, therefore, are considerably faster as compared to software based debugging tools.

Real-time trace is an important ICE machine feature that is very useful for fault injection. Trace functions capture a snapshot of the executing code to a very large memory array, called the trace buffer, at nearly full speed. Depending on the buffer size and the speed of the CPU, it saves hundreds of thousands of machine cycles, displaying the addresses, the instructions, and transferred data. The emulator and its supporting software translates raw machine cycles to assembly code or even high level language statements for viewing, drawing on target source files and the link map for assistance.

Generally, emulators use no target resources. The do not use stack space, memory, or affect the code execution speed. This “relatively” non-intrusive aspect is critical for dealing with real-time systems.

While all of the above are most favorable for debugging target system software, there are practical realities of using ICE machines for fault injection on an embedded system or digital I&C system. First, contemporary embedded system CPU speeds continue to increase, and as such these increasing CPU speeds create profound difficulties for the electrical signal connections between the ICE pod and the ICE machine host computer. For example, the machine cycle in a CPU running at 200MHz lasts 5ns. At these speeds, even an 18-inch cable between the target and the ICE starts to act as a complex electrical circuit rather than a simple wire. One solution vendors have used is to shrink the emulator, putting all or most of the unit nearer the target CPU socket. The popular OCD features present in many contemporary CPUs are a direct response to this problem. The microelectronics industry and the IEEE recognized this problem and established the IEEE Nexus On-Chip Debugger standard.

Another issue to contend with is the impact of time delay associated with modifying the contents of memory or registers during real-time operation. While 10’s of milliseconds or 100’s of milliseconds for read-modify-write operations on registers are not a problem for designers, this amount time can be overly intrusive to sensitive error detection mechanisms onboard the target system. All measures should be taken to reduce this time delay overhead as much as possible by using “backdoor” Application Programmer Interface (API) functions to bypass graphical interface features of the debugger and working with the ICE machine vendor to come up with innovative solutions.

Another aspect of ICE machine challenges is connectivity to the target system. Most ICE machine emulators physically replace the target processors CPU. In commercial based digital I&C systems, connection strategies more be more difficult with a soldered-in surface-mounted CPU. Some emulators come with an adapter that clips over the surface-mount processor, tri-stating the device core, and replacing it with the emulator's own CPU. In other cases, the emulator vendor provides adapters that can be soldered in place of the target CPU. As chip sizes and lead pitches shrink, the range of connection approaches will be typically limited and in some cases not feasible.

Connecting the ICE machine is sometimes difficult. Physical features of the target system and CPU placement can get in the way of some adapters, so planning for ICE insertion into the target system is something that should be done very early in the evaluation process.

4.10.2. Use of ICE Machines for Fault Emulation

As discussed above, an ICE machine can fulfill most of the requirements for realizing fault injection experiments with the noted concerns. This section describes how an ICE machine can be used to meet the four requirements of fault injection experimentation.

Emulation of fault models: Naturally occurring faults can be emulated by introducing into the affected structures time-controlled value errors such as registers or memory locations. Depending on the duration of the applied value error, the persistence of each fault model can be realistically reproduced. For example, a transient fault can be emulated by introducing a value error on a single bit by inverting it once at a random point of time during the execution of a workload. This can be achieved by halting execution at a random point in time, reading out the value in the location to be faulted, writing back a faulty value and resuming execution.

Precise fault activations: A permanent fault can be modeled by repeatedly forcing the value of a bit to a predetermined value each time it gets altered by the system (e.g. after each write operation). Such a value error can be introduced into the system using an ICE and triggering it each time a specified memory location is accessed. This can be realized using a breakpoint, which is a commonly available feature in most ICE machines. In-circuit emulators thus fulfill the need for precise fault activation to achieve emulation of realistic faults.

Controllability and observability: Using an ICE machine, it is possible to realize control of the various functions required to realize a complete fault injection experiment. That is, in addition to the ability to introduce a value error, system-level controls such as restart and monitoring of debug signals can be achieved using various features offered by common ICE machines. Similarly most ICE machines provide trace collection capabilities, which can be used to monitor and observe the propagation of a fault as an error in the system after a fault injection. This can then be compared to a ‘golden’ or fault free trace to identify difficulties of the target system in the handling of the fault.

Experiment control and automation: Most ICE machines provide a command line interface with simple scripting capabilities. These are the main features used to realize the scheduling of experiment control steps and for automation of large fault injection campaigns. This feature is an essential component to ensure that statistically significant results are obtained.

4.10.3. Realization of the ICE-based fault Injector

A prototype implementation of an in-circuit emulator-based fault injector was designed using the HiTex DProbe P5 In-Circuit Emulator, manufactured by Hitex International Development Systems. Referring to Figure 4–1 below, the HiTex DProbe in-circuit emulator hardware consists of a pod that plugs into the socket of an Intel Pentium I CPU socket of a processing module and an interface circuitry box that interfaces the pod to the host machine through a Ethernet port. Figure 4–2 is a photograph of the actual set up on the benchmark system.

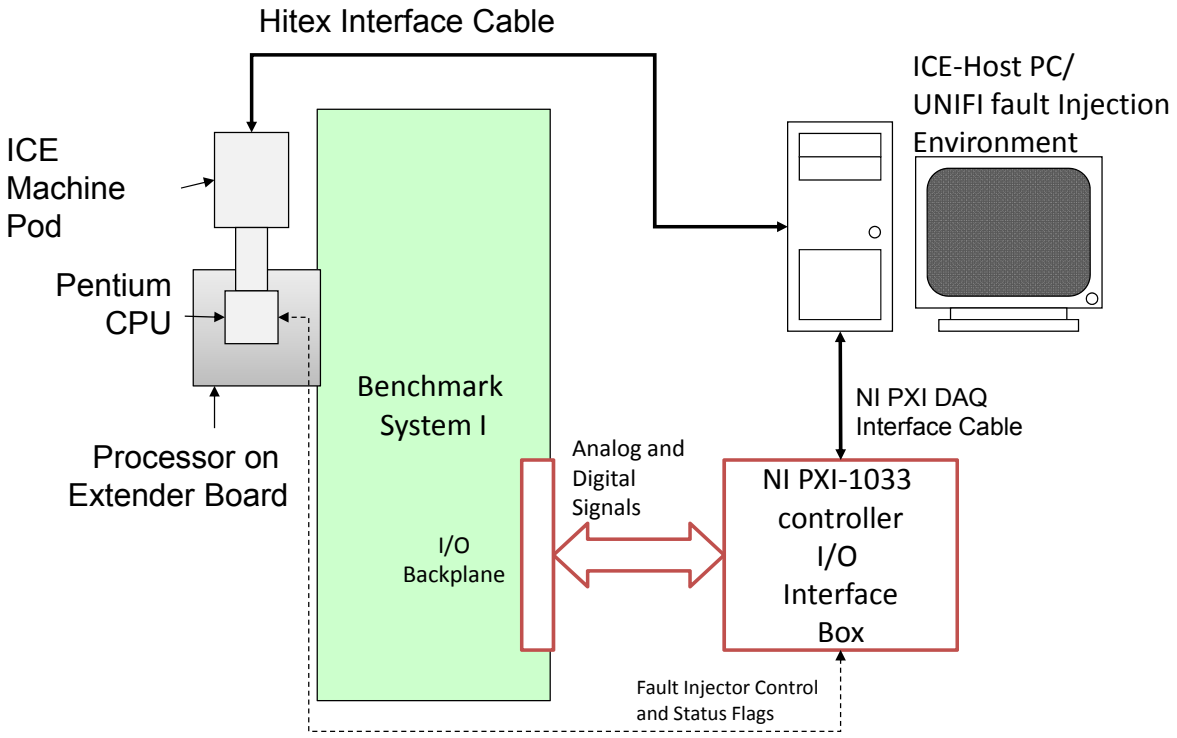


Figure 4-1 ICE-based fault injection for Benchmark System I

The CPU of the target system is removed, and the ICE-pod is plugged directly into the CPU socket of the benchmark system. This pod provides an interface to the host computer that will control execution of the target under test. When connected to the host computer, execution of the target CPU can be controlled at a very low level. For example, the target CPU can be reset, and its execution halted and resumed using commands issued from the host. While in a halted state it is possible to read and write the contents of the memory and general purpose registers in the target environment.

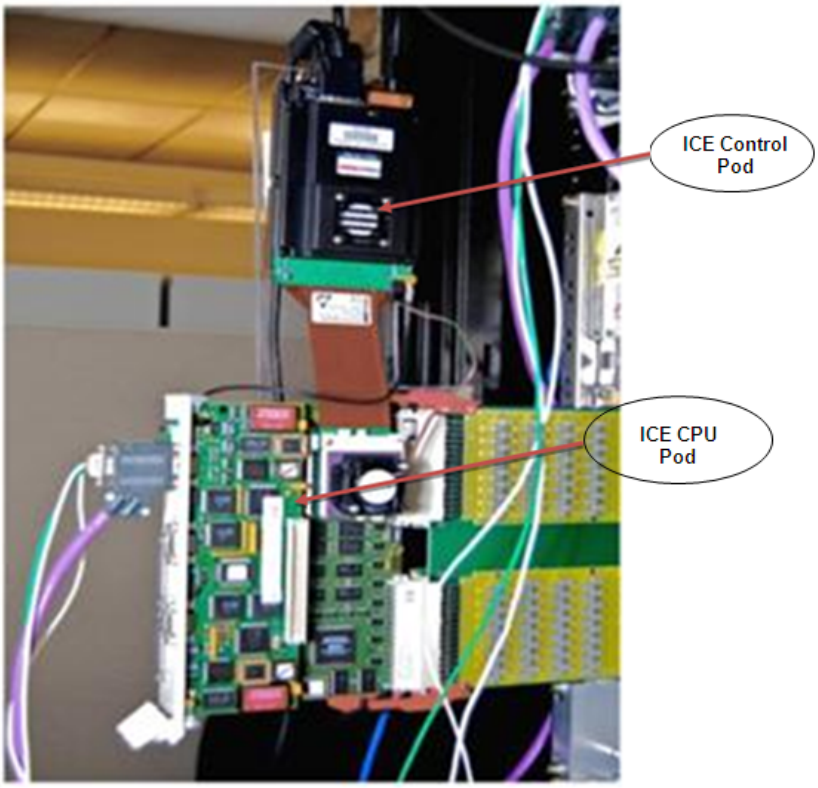


Figure 4-2 ICE machine pod inserted into the Benchmark System

Referring to Figure 4-1, the ICE machine pod is connected to a separate computer (the UNIFI host), which hosts the interactive debugger tools for applying fault injection to the target system. The interactive debugger tools are a suite of software tools called HiTOP that are provided by Hitex. The interface between the ICE machine pod and host computer is connected through a direct Ethernet connection. All communication between the host computer and the target system takes place through this interface. HiTOP offers the following key commands:

- set breakpoints,
- examine and modify variables,
- display and modify structures according to their type,
- follow any branch-list or data-trees,
- analyze high level language (HLL) expressions dynamically, and
- display detailed information on all program objects.

Since HiTOP has been tailored to work with other software tools, interfacing HiTOP into the fault injection environment is possible through a LabVIEW interface.

More importantly, HiTOP includes a macro scripting language, HiSCRIPT, which allows the user to repeatedly execute user-determined command sequences which is important for automating fault injection. Automating the injection process by using scripts built out of low-level commands is the best way to reduce the time delay overhead associated with ICE based fault injection. Having this scripting capability is also useful to interface with the fault injection environment that automates the entire fault injection experiment (reset, monitoring, data acquisition, etc.) so that long fault injection campaigns can be realized without user intervention.

Once the ICE machine pod interface and the target system have been powered up, the HiTOP application can be initiated on the host. It will give a disassembled view of the memory around the location that the program counter register points to on the target system. When halted, the disassembled view of the code running on the target machine can be viewed on the host machine. As shown in Figure 4–3, the memory locations immediately following the location currently pointed to by the instruction pointer are displayed in the window, together with the binary code of the instruction at that location and its corresponding disassembled mnemonic assembly code.

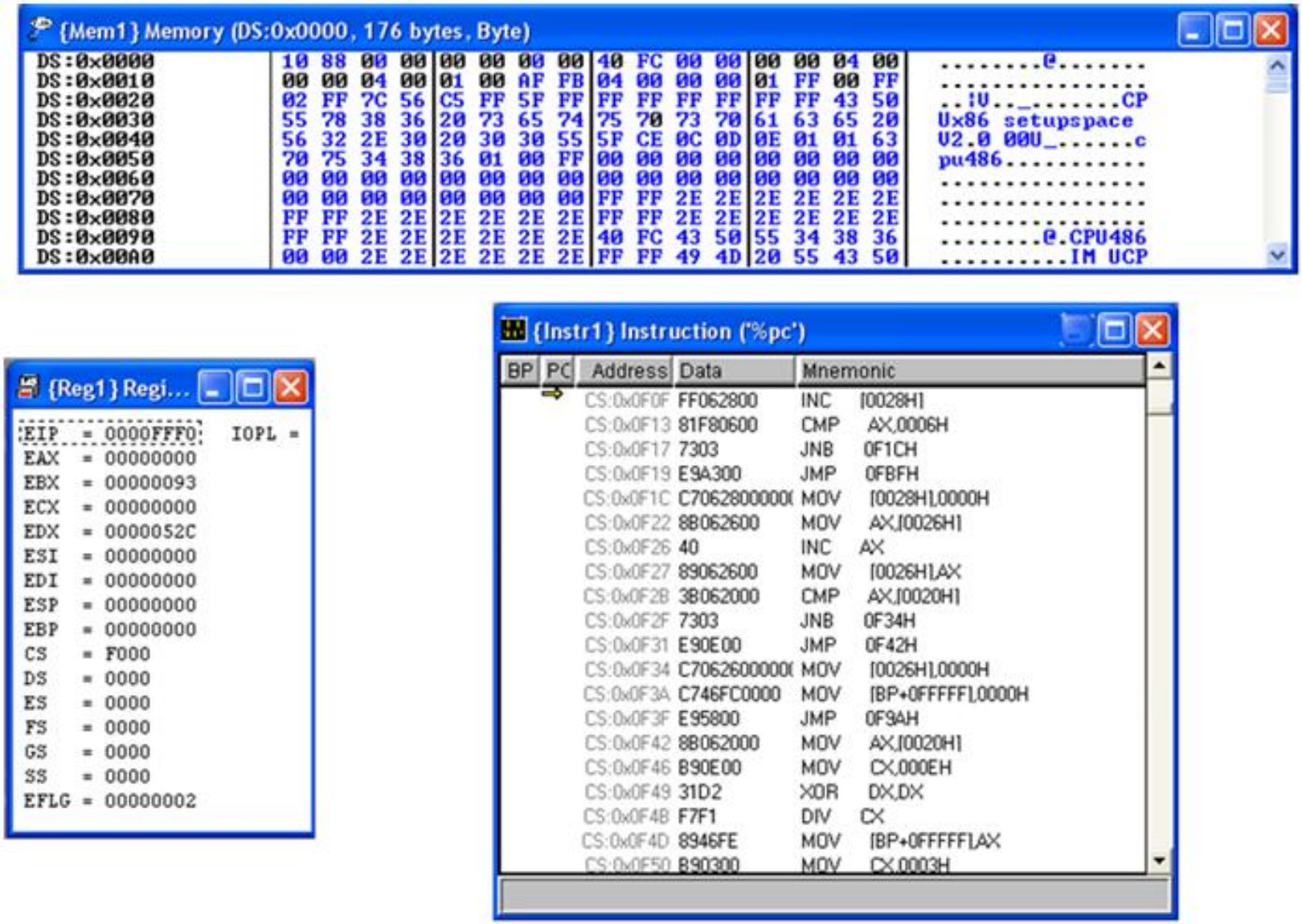


Figure 4-3 HiTOP view of the target software executing on the Benchmark System

An additional level of control provided by the HiTOP emulator is allowing external signals to trigger the ICE control pod for specific actions and to receive status information from the ICE pod. These external signals are used to facilitate enhanced control and automation of the fault injection process.

As mentioned earlier, an important consideration when using any fault injection process is to ensure that the time delay intrusion of the ICE when it attempts to corrupt a register or memory location does not falsely trip watchdog timers built into the system. For example, if the halt-read-modify-write-resume process of injecting a fault can introduce a delay that can be caught by an on-board fault detection capability (e.g., a watchdog timer), then this in turn could reset the system as a precautionary measure after identifying the delay in the CPU execution cycle. This situation is an artifact of the intrusion introduced by the fault injector. This artifact must be avoided by minimizing the overhead of the injection process. Automating the injection process overhead by using scripts built out of low-level commands is necessary to reduce the occurrence of this artifact.

As part of this effort, a few performance trials of typical low level fault injection commands were executed to emulate various fault models. To emulate transient bit-flip and multiple bit-flip models, the value in the location of the register or memory must to be read out, and a random bit in the value must be selected and inverted before it is written back. This called the *read-modify-write* transient fault model. The read-modify-write fault model typically uses a breakpoint to trigger on a memory or register access. A simpler transient model is the *write-resume* model. It was observed that a halt or break-write-resume operation could be faster. Therefore, instead of a halt or break-read-modify-write-resume operation, the 'read and modify' operation in the read-modify-write injection step was eliminated and a random value was written into the location to emulate a transient fault (a halt-write-resume operation).

The next fault model of interest is the permanent fault model. The permanent fault model is an extension of the two transient fault models. The difference is that the breakpoint or halt occurs each time the register or memory resource is active (being used). This can incur significant time delays or performance penalties if the resource is used often. As such, the use of the permanent fault model was limited to judiciously chosen memory accesses. Some preliminary performance measurements of the ICE-based fault injector were taken during development to gauge the ability to support the various fault models. These measurements are listed in Table 4–2.

Table 4-2 Performance delay times for ICE-based fault injection of a few models.

Fault Model Type	Halt Read-Modify-Write	Halt Write-Resume	Watchdog Tripped
Transient: Single bit	~ 20 to >100 ms	~ 10-15ms	Only on HRMW
Transient: Multi-bit	~ 20to > 100 ms	~ 10-15ms	Only on HRMW
Permanent: single bit	150ms	50ms to 150ms	Frequently

4.11. X-bus Fault Injector

4.11.1. Introduction

The motivation for this work was to create an X-bus fault injector that would be able to corrupt specific bits of the transmitted data stream. At first, it was necessary to gather better knowledge of the X-bus network traffic patterns at the bit level so that a successful fault injecting device

could be designed. This was achieved by understanding the transmission specifications based on the information in the X-bus manual provided by the vendor, reviewing the IEC specifications, and by capturing and observing traffic on the benchmark system with the help of a logic analyzer that was connected to an X-bus interface present on the actual nuclear digital I&C protection system. Observance of the X-bus communication traffic was imperative in the process of designing the fault injector because it greatly enhanced the understanding of the bit level data transmission characteristics. After complete investigation of the X-bus data message delivery system, the scope of this work was extended to cover fault injections into the data message packets and tokens that were being transmitted over the network. This type of fault injection was performed in a manner different than the original fault injection in order to alter the contents of the Data Message, rather than simply corrupting the transmission. The X-bus protocol both data message delivery system and token management are discussed in the following section.

4.11.2. X-bus Data Message Delivery System

X-bus is a standard IEC fieldbus specification intended to provide communication for industrial processing applications. It can offer deterministic bounds on the latency of delivery for high-priority messages. As a result, it can be used in applications having real-time communication constraints. X-bus is also robust with mechanisms specified to handle various fault conditions. As a result, it may be considered for applications requiring highly reliable communication. The X-bus response to fault conditions requires time and this time should be considered when evaluating the potential real-time response characteristics of the network in the presence of faults [Tovar 1999].

The X-bus data link layer provides a hybrid medium access control mechanism that includes both centralized and decentralized control. Decentralized medium access control is implemented through a token passing mechanism that establishes which of the multiple master stations will control the data link during each time interval. While a specific master station controls the data link it imposes centralized master-slave control to send data on the link and request that data be sent to it over the link [Gil 1997].

4.11.3. X-bus Token Management

The X-bus Token passing mechanism requires each master station to maintain a list of master stations (LMS) containing the addresses of all master stations sharing the data link. A master station's own address in the LMS is identified as this station (TS). The predecessor station in the ring is identified as the previous station (PS), and the successor station in the ring is identified as the next station (NS). Each TS receives the token from its PS and passes the token to its NS. Each master station determines the LMS, its PS, and its NS addresses using a dynamic configuration algorithm. The order of master stations in the logical ring is in ascending order of addresses with the highest master station NS pointing to the lowest master station to complete the logical token ring.

The master station that owns the token controls the data link until it passes the token to the next station. During the time that TS holds the token, it can send out different types of messages besides Tokens that X-bus DP supports. These types include No Data (NDD), Fixed Data (FDD), and Variable Length Data (VLDD). Messages of types FDD and VLDD are used for transferring data between stations, NDD message is used for discovering stations that might have been inserted into the logical ring or for reinserting stations that were removed due to some problems. However, when TS obtains the token, it is guaranteed to be able to send out only one high priority message. Only if time permits can the TS send out other messages with lower priorities. This time is computed by calculating the difference between the predefined expected time to pass a token around the ring (Ring Time) and the time that it actually took

since the TS last owned the token (Ring Rotation Time). If this time is greater than the time it takes to send one message, the TS can send out additional messages that might be in its queue. This includes the discovery message (NDD). Therefore, the presence of faults on the link could cause tightness in the scheduling of messages and could significantly delay reinsertion of X-bus stations.

The TS passes the token to NS using a particular data link protocol data unit (DLPDU) composed of a start delimiter byte to identify the DLPDU as a token followed by destination and source address bytes. The next master station detects the DLPDU as a token with its destination address and a source address corresponding to its PS. If the token header byte, the destination address byte, and the source address byte are all correct, then the receiving master station accepts ownership of the token and assumes control of the data link. The sequence in Figure 4-4 depicts the Token passing mechanism between several masters.

Any problems with the received Token DLPDU will prevent the NS from taking ownership of the Token. The master station passing the Token listens to data link traffic after it has attempted to pass the Token. If it does not detect any bus activity within specific time duration, it will then make another attempt to pass the Token. The time-out duration is called Slot Time (T_{sl}), and it is a parameter established for the particular X-bus implementation based upon the physical characteristics of the network system.

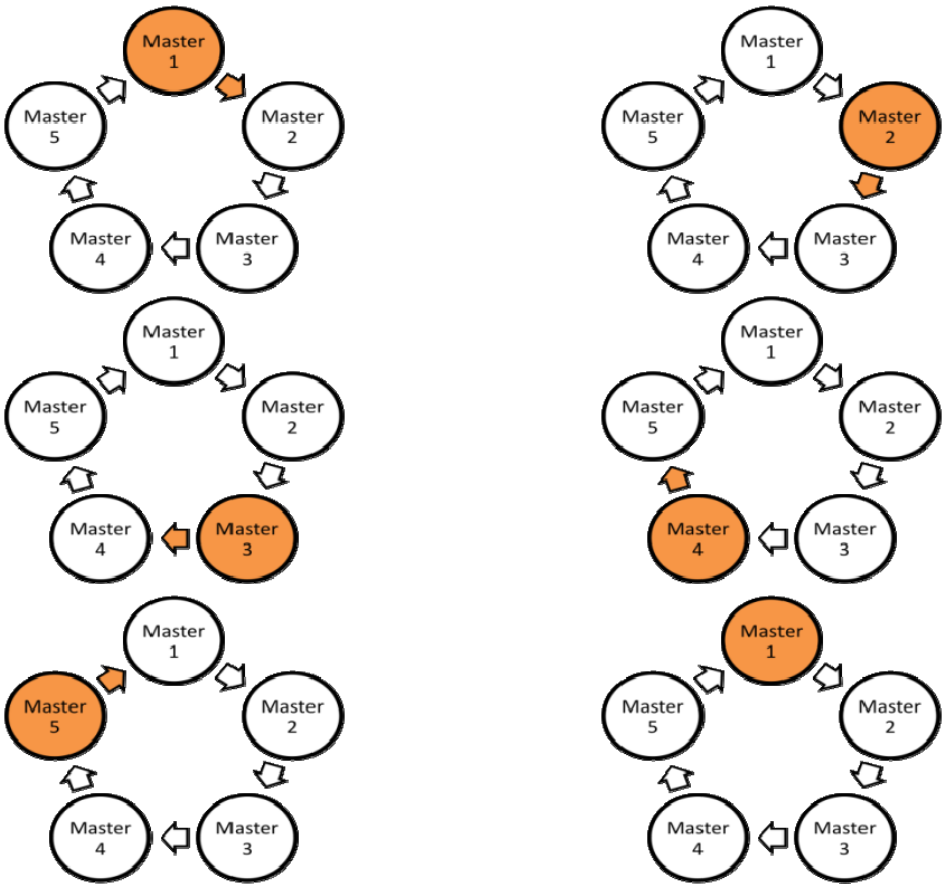


Figure 4-4 Illustration of token passing mechanism in a multi-master system

After making a second attempt to send the Token, the master station again monitors the data link traffic. A successful Token transfer will be recognized if the new master produces data link traffic within T_{sl}. If gain the TS attempting to pass the Token does not detect any bust activity

within the slot time, it will make its third attempt to pass the Token to the NS in its LMS. If the third attempt is unsuccessful, TS tries to transmit the Token to NS one last time.

After the last attempt to send the Token, the master station again monitors the data link traffic. A successful Token transfer will be recognized if the new master produces data link traffic within the slot time. If the master station passing the Token again does not detect any bust activity within the time slot, it will repeat this Token passing approach with the next sequential master station in its LMS. Thus, the master attempting to pass the Token, the current TS, assumes that the NS in its LMS is no longer available so the current TS attempts to pass the Token to the NS of its NS.

If the master station cannot pass the Token to the next station in the LMS, it will continue this process around the ring until it is either successful or until the next master on the list is itself. If it reaches the point of sending the Token to itself, it assumes that it is the only master on the data link. Figure 4-5 illustrates the process of attempting to pass a corrupted Token on the network.

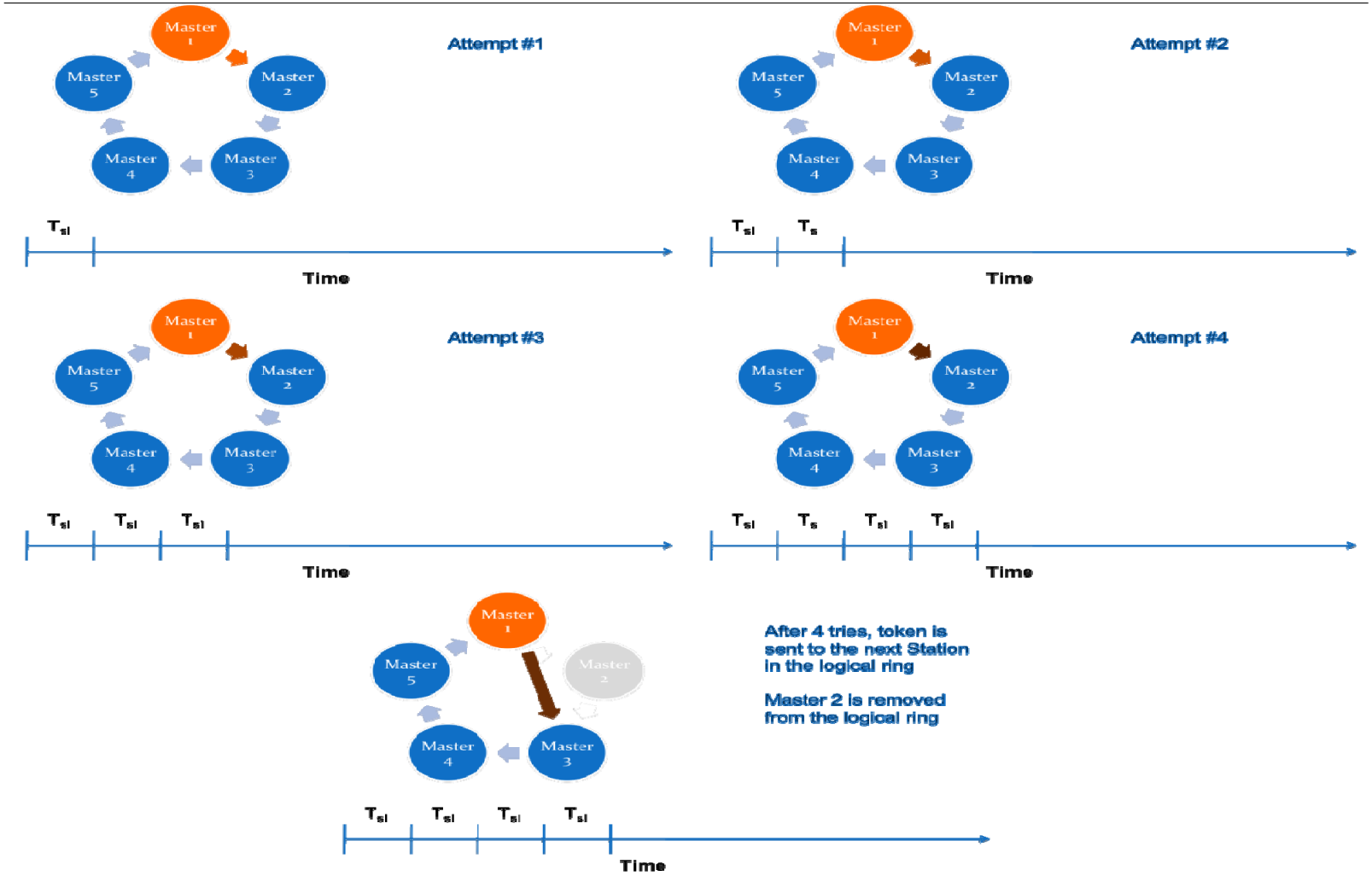


Figure 4-5 Corruption of token when Master 1 attempts to pass token to Master 2

The Token passing control nature of X-bus practically eliminates any possibility of an automatic immediate check for transmitting any other message than a token because at the time that TS holds the token, only it can transmit on the link. Therefore, if a data message was corrupted or not delivered, it can be only announced to TS when a station the message was intended will be in possession of the token. However, if the receiving station has already several high priority messages in its queue, it might not be able to announce to TS that the message TS transmitted was corrupted for at least another ring rotation. This could pose quite significant time delays in transmitting and receiving important data information between stations and could lead to failures in real-time safety-critical systems due to the late delivery of crucial information.

The first efforts at X-bus fault injection focused on disrupting the functionality of X-bus by corrupting tokens and observing the recoverability of the network and the target system. The main goal was to measure the following timeouts incurred by corrupting a Token:

Time delay represented by T_{sl} after corrupted Token.

Time required to reinsert an NS into the logical ring after it was excluded due to the inability to recognize corrupted tokens, thus being unable to take ownership of the logical ring.

After further investigation of the X-bus data message delivery system, the scope of this work was extended to cover fault injections into the data messages (VLDDs) that were being transmitted over the benchmark system X-bus network. This type of fault injection was performed in a manner different than the token fault injection in order to alter the contents of the data message, rather than simply corrupting the transmission.

4.11.4. X-bus Fault Injection

X-bus is inherently a time-triggered, asynchronous network bus. This means that the devices on the network do not have constant clock synchronization between each other. Rather, the bus functions on the basis of providing synchronization timeouts, with the start bit of each message serving as the synchronization element. The synchronization period is produced by a long sequence of logic level high (bit meaning “1’s”) that can be observed on the network. The beginning of a transmission of any message is therefore specified by a low voltage (bit meaning “0’s”), inserted by the transmitting station. When the devices on the network observe this bit, they synchronize their clocks in order to correctly receive transmissions originating from the station that currently holds the token.

The main building block of any X-bus transmission is the 11 bit X-bus Transmission Packet (PTP) illustrated in Figure 4-6. PTP consists of the following bit sequence:

- Start bit (bit value 0)
- Eight data bits carrying the information
- Even parity bit
- Stop bit (bit value 1)

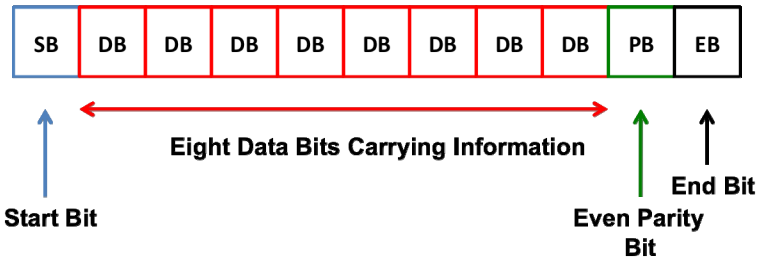


Figure 4-6 X-bus transmission packet (PTP)

The Start and Stop bits are intended for synchronization purposes, while the Parity bit is a simple check over the eight data bits for an uneven number of bit upsets. Each X-bus message consists of a predefined number of PTPs used to accomplish successful communication between stations. The structure of all message types (Token, NDD, FDD, VLDD) are very similar. The first PTP, called the Start Delimiter, carries the Header information that determines what type of message is being transmitted. The last PTP of each transmission, the End Delimiter, carries a constant that indicates the end of a transmission to the receiving stations. Each message also contains PTPs that hold the information pertaining to the source and destination addresses, which is necessary for proper network communication in multi-station systems. Additionally, data messages contain PTPs carrying information about their length, as well as the actual data that is transmitted. The data payload is usually spread over many PTPs to provide the ability to handle transmission of larger amounts of information. The proper deciphering of the data information is performed at higher data link levels of X-bus functionality. One physical constraint of the X-bus fault injector was the capability to only affect and corrupt bit values of 1 and pull them down to a bit value of 0. This limitation was given by the physical access to the X-bus network signals.

To achieve the purpose of the X-bus fault injector to execute deterministic fault injection the design had to be based on pattern matching of the Start Delimiter. This was necessary to properly identify the type of a transmission to execute the correct fault injection type. Without this feature a situation could occur where a data message would be corrupted as if it was a token and vice versa. In order to correctly identify the X-bus traffic, the fault injector would remain idle and observe the X-bus signals during the X-bus synchronization period and adjust its internal clock at the occurrence of the first start bit, and subsequently at the start bit of each PTP to attain maximum possible precision.

The required information for executing Token fault injections was obtained after determining the first few bits of the first PTP (the Start Delimiter) when it was possible to correctly identify the message type and corrupt specific bits if necessary. Data Message corruptions were performed differently, altering the PTPs specifying the length of the transmission, but the fault injection idea was the same as with Token:

- Observe the network traffic
- Compare obtained data to a predefined constant
- Inject interference to X-bus signals at a specific bi

4.11.5. X-bus Requirements

A set of requirements to achieve correctness of the implementation of the X-bus fault injection module was developed after identifying a successful implementation of the X-bus fault injection system. The design had to be minimalistic, but at the same time provide high performances so that the proper functionality could be accomplished in real-time. The following are the set of design requirements for the X-bus fault injector:

- Design must be able to perform both types of fault injections without further alterations
- Ability to synchronize with the X-bus signal
- Pattern matching mechanism for recognizing different message types
- Corruption signal must interfere with X-bus signals at an exact instance
- Corruption cannot affect any other bits than the desired ones
- Functionality for performing fault injections either manually or automatically
- Ability to perform multiple consecutive fault injections, as well as an infinite fault injection (the system keeps executing fault injections until stopped by an input)
- Implemented in Very High Speed Integrated Circuit Hardware Description Language (VHDL) on an Altera FPGA board
- FPGA design clocked at high frequency
- Final product must include capability to be controlled from an external interface for integration into automated fault injection system

4.11.6. X-bus Fault Injection Architecture

The basic architecture of the X-bus fault injector is shown in Figure 4–7. At the core of the heart of the fault injector is the FPGA based controller that performs all the X-bus traffic observations including the timing analysis of the protocol and determining the precise time when to inject a fault to corrupt a token or a message. Referring to figure 4–7, the differential signal of the X-bus is converted by a differential receiver to a single-ended bit-stream. Custom digital hardware in the receiver observes this bit-stream to detect the token start delimiter byte. The detection of the token start delimiter byte immediately following a synchronization period indicates that a token is being transmitted on the bus.

The FPGA block in figure 4–7 generates a control output at the times when fault injection should be applied to the X-bus. The control can be asserted only after the eighth data bit of a token start delimiter has been received. The actual start time and duration of this control signal can be adjusted. An additional setting determines how many sequential tokens will trigger the X-bus fault injector circuit. The control signal is applied to the fault injection circuit that applies corruption to the X-bus while the control signal is asserted. X-bus signals are captured by a logic analyzer also connected to observe the data stream from the differential receiver. Analysis of this data visually through the logic analyzer signal display can be tedious, so software was written to convert the signal waveforms to a more convenient form for analysis.

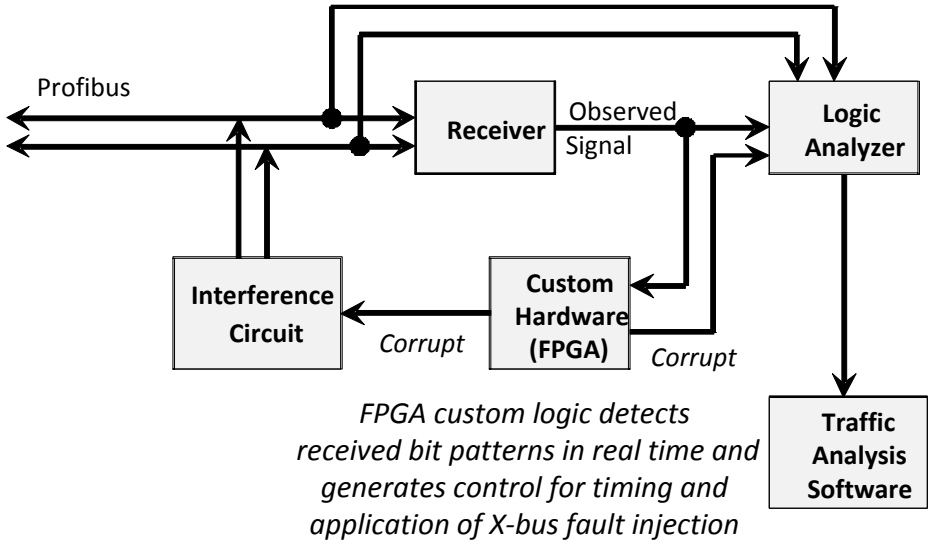


Figure 4-7 X-bus fault injector

The analysis software processes data exported from the logic analyzer. It merges the data into one long bit stream that is then partitioned based on the synchronization intervals. The start delimiter byte for each message is then recognized to place each sequence into one of these categories:

- Token, No Data (NDD),
- Fixed Data (FDD), or
- Variable Length Data (VLDD).

After the message is categorized, the remaining characters are decoded according to the X-bus specifications for each DLPDU field.

Finally, all data is written into a text file in a readable form to support observation and analysis of the X-bus traffic. The data input to the software is exported from the logic analyzer running in "repetitive" mode to acquire multiple sequential data frames. A simple batch script was written to copy the exported data text files to a workstation over an Ethernet link. The analysis software runs on the workstation to produce a text file description of the actual X-bus traffic. The text file presents each DLPDU sequentially in the form: identification, data fields according to the DPLDU type, and synchronization time. The text file presents all of the DLPDUs in the capture window.

4.11.7. FPGA Implementation of the X-bus Controller

The main challenge in implementing the X-bus fault injection design based on the requirements specified in the previous sections was to achieve correct synchronization with the X-bus transmissions, and correctly identify each detected bit. Unfortunately, the exact width of each bit is never exact; therefore it was important to resynchronize with the start bit of every observed PTP in order for the FPGA to remain tightly synchronized with the X-bus bit-stream to correctly identify each bit value. This was accomplished by clocking the FPGA design at a fairly high frequency (50 MHz) and by implementing several hardware counters inside the design. Based on the known baud rate of this specific X-bus implementation, the approximate duration of a single bit present on the network was determined to be 650 ns, which is significantly slower than the clock cycle of the implemented hardware counters (20 ns). By using the speed counters, it

was ensured that the design clocking would never drift far enough to misinterpret any of the 11 PTP bits.

Additionally, the network traffic bit-stream was sampled in the approximate “middle” of each bit (around 320 ns, depicted in Figure 4–8) which eliminated possible problems such as a small clock drift of sampling the current bit during its setup time, which could cause an incorrect identification of the X-bus network traffic.

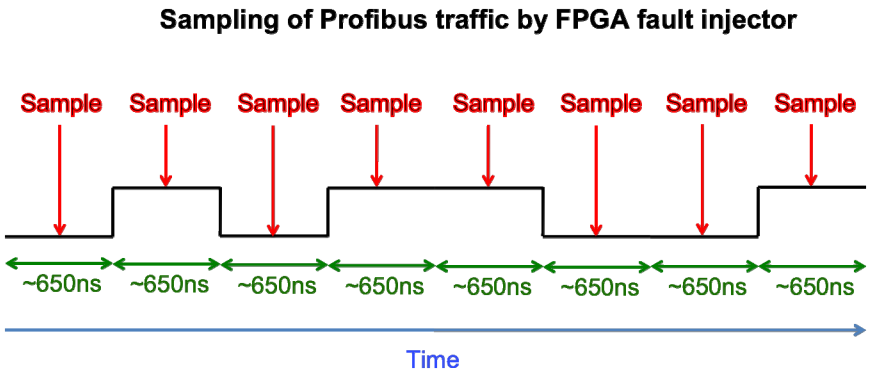


Figure 4-8 Sampling of the X-bus traffic by the FPGA

After it was determined how to correctly sample and detect the X-bus network traffic by the fault injector, the token fault injector design was implemented by incorporating a state machine for achieving correct functionality. Figure 4–9 shows the FPGA implementation of the FPGA fault injector.

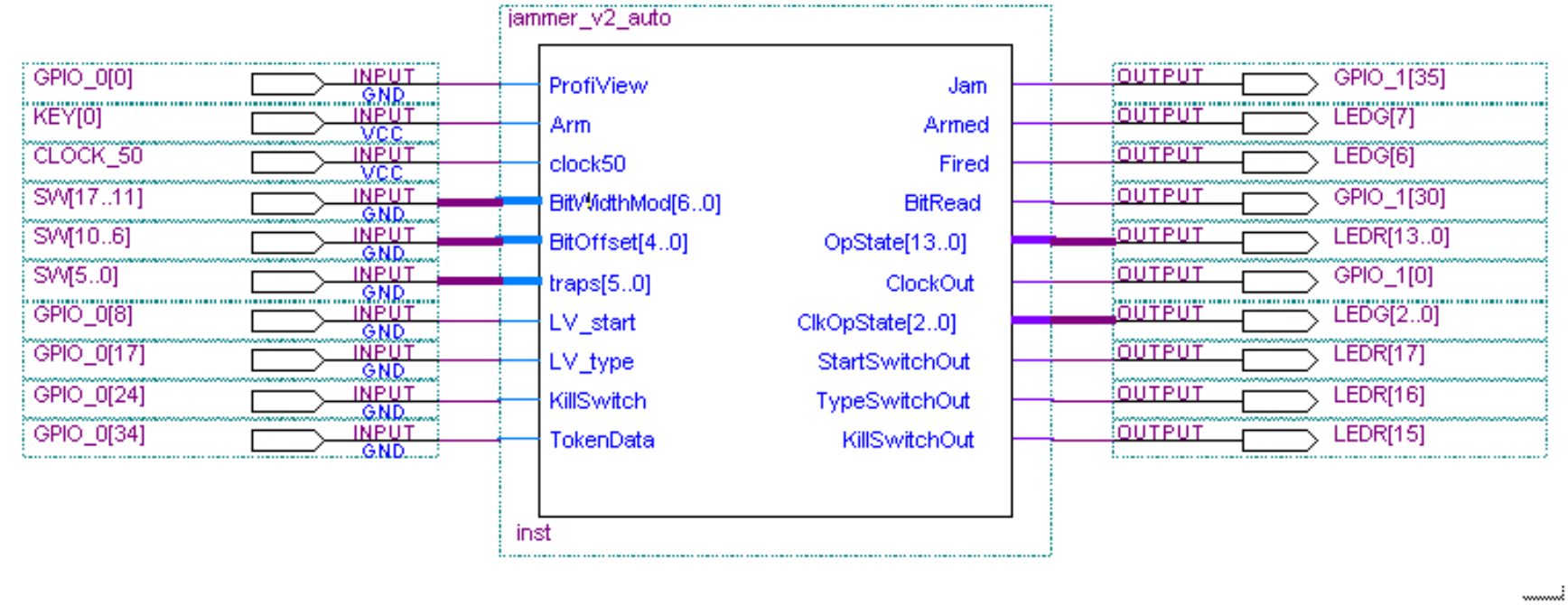


Figure 4-9 Block diagram representation of the FPGA X-bus fault injection module

The following list describes the function of each stage of the state machine as well as the inputs and outputs of the FPGA fault injection module:

Inputs

- **ProfiView** – Pin; signal coming from the X-bus network
- **Arm** – Button that activates the fault injection
- **Clock50** – 50 MHz clock internal clock
- **BitWidthMod** – Switch buttons; input value for the duration of the jamming signal
- **BitOffset** – Switch buttons; input value adjust a delay of the jamming signal
- **Traps** – Switch buttons; input value specifies then number of fault injections to be performed
- **TokenData** – Switch button; selects between Token and Data Message corruption

Outputs

- **Jam** – Pin; Jamming signal
- **OpState** – LEDs; visual observation of the current state, used for debugging
- **Fired** – LED; visual observation of a finished fault injection

State Machine
ARMING

- State in which the fault injector is initialized and is waiting for an input to begin fault injection.
- When all signals are initialized, the state machine advances to next state.

SYNCWAIT

- The fault injector is observing the X-bus traffic and waiting for the synchronization period that is by X-bus specifications identified to constitute of at least 32 consecutive high voltage bits; this value is observed by a counter; if the value reaches a specific constant the control is passed to the next state.

HEADERWAIT

- While the synchronization delay in X-bus is precisely specified, it can be adjusted for a specific X-bus network; therefore, this state continues to observe the traffic on the network and waits until it detects a first bit value of 0 that would signify the first start bit of a Start Delimiter of a transmitted message.
- Upon detection of the 0 bit value, the state machine advances to the next state.

BYTECAPTURESTART

- This state verifies that the detected bit was in fact an actual low voltage bit and eliminates the possibility of a glitch in the network by counting to the middle of the current bit and sampling the traffic again, as described earlier.

- If the first detection of 0 was just a glitch and the currently sampled value is 1, the control is returned to the ***SYNCWAIT*** state and the fault injector waits for the duration of a synchronization period again.

- If the sampling confirms that the bit value was indeed 0, the system waits for a full duration of one bit and then transfers control to the next state, which executes the pattern matching mechanism.

BYCAPTURE

- The current state captures the next 8 bits representing the data field of the start delimiter (***SD***) ***PTP*** and stores them into a temporary register.

- When the 8th bit is captured, the value of the temporary register is compared to the constant value representing ***SD*** of ***Token*** (0xDC).

- If the values are different, the control is returned back to the ***SYNCWAIT*** state, as the current ***DLPDU*** on the network is not a ***Token.***

- If the values match, the control is transferred to the next state.

JAMIT

- This state outputs a corrupting signal to the X-bus network traffic and causes an interference on the bus, thus driving the voltage low, inherently changing the bit values of the transmitted message; the counter holding the number of performed fault injections is increased. It was necessary to allow for the possibility of altering the precise position and duration of the jamming signal by accepting an input from the push switches on the board to achieve the exact position and length necessary for executing proper interference to the circuit; this was determined by observation of the traffic with the help of a Logic Analyzer.

- If the number of performed fault injections reach the desired value, the system moves into its last state; otherwise the control is returned to the ***SYNCWAIT*** state.

FIRE

- This state negates the jamming signal and reinitializes the system for the next fault injection.

To extend the capabilities of the fault injection system to allow for performing Data Message injections, some modifications had to be done to the state machine functionality:

Input Addition

- ***TokenData*** – Switch button; selects between Token and Data Message corruption.

State Machine - Data Message corruption
BYTECAPTURE

- After acquiring the whole data field, the value of the temporary register is compared not only to the ***SD*** value of ***Token***, but also the one of ***VLDD.***

- If the value matches the **Token** constant, or if it does not match anything, the state machine continues as described previously in the Token fault injection.

- If the value matches ***SD*** value of ***VLDD***, the device has to then verify the next PTP, which is accomplished in state ***HEADERWAIT_FRM1.***

HEADERWAIT_FRM1 (added)

- Performs the same function as ***HEADERWAIT***, but if successful, passes the control to ***BYTECAPTURESTART_FRM1.***

BYTECAPTURESTART_FRM1

- Performs the same function as ***BYTECAPTURE***, but passes control to ***BYTECAPTURE_FRM1.***

BYTECAPTURE_FRM1

- Performs the same function as ***BYTECAPTURE***, but compares only the first four acquired bits, because at that point the system can determine whether or not the fault injector is observing the expected Data Message traffic.

- If the traffic is correct, the control is transferred to **JAMIT_FRM1** state; however, if something is wrong and the acquired bits do not match the predefined constants, the state is changes to ***SYNCWAIT*** to start over.

JAMIT_FRM1

- Performs a corruption of the X-bus network traffic as described in JAMIT state, but corrupts the 6th and 7th bit of the data field instead of the parity and stop bit; this is due to the nature of the Data Message fault injection.

- Instead of finishing up the fault injection and transferring control to the FIRE state, the control is moved to the HEADERWAIT_FRM2 state because two consecutive PTPs must be corrupted.

HEADERWAIT_FRM2

- Same function as ***HEADERWAIT_FRM1***, but for the second PTP specifying the length; transfers control to ***BYTECAPTURESTART_FRM2***.

BYTECAPTURESTART_FRM2

- Same functionality as ***BYTECAPTURESTART_FRM1*** state, but for the second ***LE PTP***; if successful, it transfers the control to the ***BYTECAPTURE_FRM2*** state, otherwise the fault injection has to be restarted by changing the state to ***SYNCWAIT***.

BYTECAPTURE_FRM2

- Same functionality as ***BYTECAPTURE_FRM1*** state, but for the second PTP; control is transferred to the ***JAMIT_FRM2*** state.

JAMIT_FRM2

- Combination of functionalities of ***JAMIT*** and ***JAMIT_FRM1*** states.

- Outputs the interference signal into the X-bus wires during the 6th and 7th bit of the data field; the count of performed fault injections is increased.

- If the requested number of fault injections is reached, the fault injection campaign is finished and the control is transferred to the ***FIRE*** state; otherwise the control is transferred back to the ***SYNCWAIT*** state.

4.11.8. Integration with an Automated Fault Injection System Environment

Building the standalone version of the X-bus fault injector is extremely important for the purposes of testing the X-bus network and determining the responses of the real-time safety-critical system under test. However, to fully utilize the capabilities of the designed fault injection system, it had to be integrated into a complex Fault Injection Campaign Control Interface, a central system that executes different types of fault injections performed by a number of various fault injection systems at predefined times to stress the target system and achieve a robust fault injection campaign.

Therefore, it was necessary to alter the designed system to include an option for an external fault injection execution control. It was determined that the fault injections would be differentiated by their duration, and set up by the central system. Fortunately, the required alterations did not demand re-implementing the internal structure of the design, but only including some additional logic and several inputs for accepting commands from the UNIFI fault injection environment. These commands are:

Command Inputs

- **TokenData** – Changed from Switch button to Pin so that it can be driven by a signal coming from the central system

- **LV_start** – Pin; signals the design to start fault injection

- **LV_type** – Pin; determines whether to perform a single fault injection or an infinite one

- **KillSwitch** – Pin; signals the design to stop infinite fault injection and to reinitialize itself

These commands allowed the designed X-bus fault injection system to become a completely self-contained fault injector with the capability to be operated manually as well as remotely by inputs received from an automated fault injection system.

4.12. JTAG Fault Injection Module

Most current integrated circuits have external input and output pins linked together in a set called the Boundary Scan Chain (BSC). JTAG (IEEE 1149.1 standard) was designed to be able to access BSC by means of a virtual register (Boundary Register) connected to its input and output pins. It is possible to alter the contents of BSC and hence alter the current signals on the pin-outs by serially shifting in data into the Boundary Register. At the same time, bits from the Boundary Register are serially shifted out to the output pin of JTAG controller. Because of the common occurrence of the JTAG port on current devices such as CPUs and FPGAs, there has been fairly extensive work on attempting to perform fault injections via this technique. The most sophisticated solutions for performing JTAG fault injections employ an FPGA as the fault injector programmed to perform experiments with a predefined set of faults [Portela-Garcia 2007]. This solution is unobtrusive as it does not require any additional hardware on the target system. Further, the performance is not degraded by including host computer processing in the fault injection process.

To summarize the review of JTAG fault injection systems:

- Most of the solutions took advantage of implementing additional logic on the target system; this solution is obtrusive and it is not always possible to alter target system hardware.
-
- Systems that did not employ additional hardware were purpose-specific or CPU-specific; it would be hard to port these types of fault injection systems to a new system.

- None of presented solutions sufficiently addressed the problem of performing fault injections on real-time systems.

The approach taken in this research was to optimize the fault injection “scan-in” process. As a test platform the JTAG fault injection ideas used the same FPGA development board as was used for the X-bus fault injections to implement additional fault injection capability. The boundary scan register of the Actel 42MX FPGA holds the values of bits on each input and output line. By accessing the target JTAG interface a fault injection can be performed by shifting data into the interface that subsequently disrupt the boundary scan register.

The implementation of the JTAG fault injection module is driven by the idea of inserting only a few bits into the boundary scan register, instead of reading and writing the full 198 bit register then altering it and shifting it back, as this would be time consuming. In addition, the duration of an executed fault injection could trip one of the watchdog timers or violate the real-time properties of the system under test.

The boundary scan register is implemented as a serial chain, thus by shifting one bit into the target JTAG interface, the rest of the bits already located in the register are shifted as well. By this process, a very short sequence of bits could be passed to the interface that would provide modification of the required input and output pin values that are the targets of the fault injection. Therefore, the desired corruption is achieved in a fraction of the time that a typical JTAG fault injection technique would require.

4.12.1. Specifications

The JTAG interface is represented by a TAP and hardware logic that performs halting of the system, setting up breakpoints, and manipulating the boundary scan register. The TAP contains five pins: Test Data In (TDI), Test Data Out (TDO), Test Clock (TCK), Test Mode

Select (TMS) and Test Reset (TRST, active low). Since there is only one pin available for each line, the protocol is inherently serial-like. The required actions are performed based on the combinations of inputs TMS and TDI at each clock cycle. Implementation of the TAP controller state machine is illustrated in Figure 4–10. Its functionality will be described in the following paragraphs.

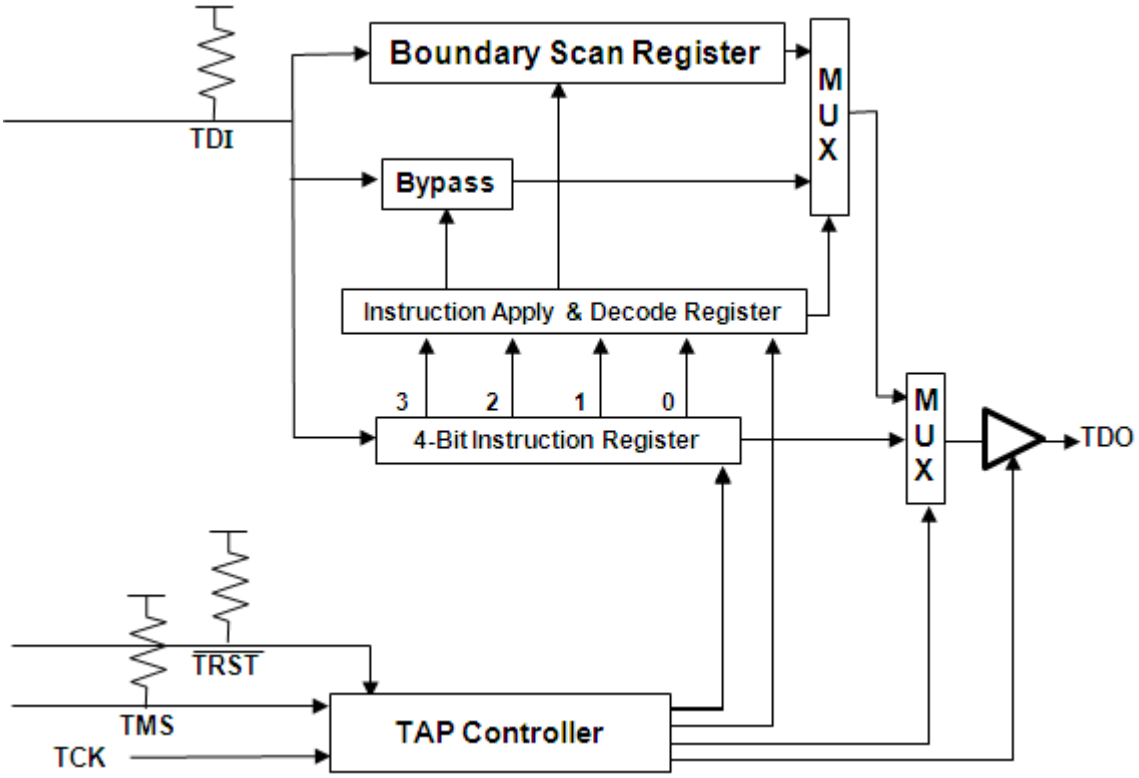


Figure 4-10 JTAG TAP controller test logic diagram

The JTAG implementation present on the Actel FPGA supports all mandatory IEEE 1149.1 instructions and consists of the TAP controller, a 4-bit instruction register, and two test registers. One test register serves as a 1-bit bypass register for connecting TDI input straight to TDO output for testing the correctness of connections within the internal components. The second test register is a 198- bit Boundary Scan Register (BSR) that contains bit values of all digital input and output signals inside the logic board. This register can be connected between TDI and TDO and bit values can be shifted into BSR through TDI, while at the same time shifted out and put on the TDO output. BSR is used for capturing data on the input signals, forcing fixed values on the output signals and selecting the direction and drive characteristics. The names of the inputs are described in a separate file [Actel 2009].

The Instruction register (IR) is a 4-bit register (without parity) that determines the function that is being performed by the TAP controller. There are five functions supported by the Actel FPGA TAP controller, but for the purposes of implementing a JTAG fault injection module only three of these functions are required.

The first function that must be loaded into the IR is the SAMPLE/PRELOAD (0001) function. It initializes BSR output cells before they can be connected to the boundary scan chain of the FPGA. It is necessary that this function be invoked before performing the bit corruptions. Otherwise, the output signals might be driven to a random state when scanning of the boundary begins.

The BYPASS function simply connects TDI and TDO through a 1-bit bypass register. It avoids connecting the 198-bit scan register and serves for checking the correctness of the connection of certain components. It is not necessary for the actual fault injection, but it might be a useful feature to be able to check for correctness of the internal connections before executing the actual fault injection.

The function that actually enables the BSR and connects it between TDI and TDO is EXTEST (0000). During each clock cycle that this function is present in the IR, the bit present on the TDI input is shifted into the BSR and the lowest bit of the register is shifted out to the TDO output pin. The operation of the TAP controller is based on a simple state machine, illustrated in Figure 4–11.

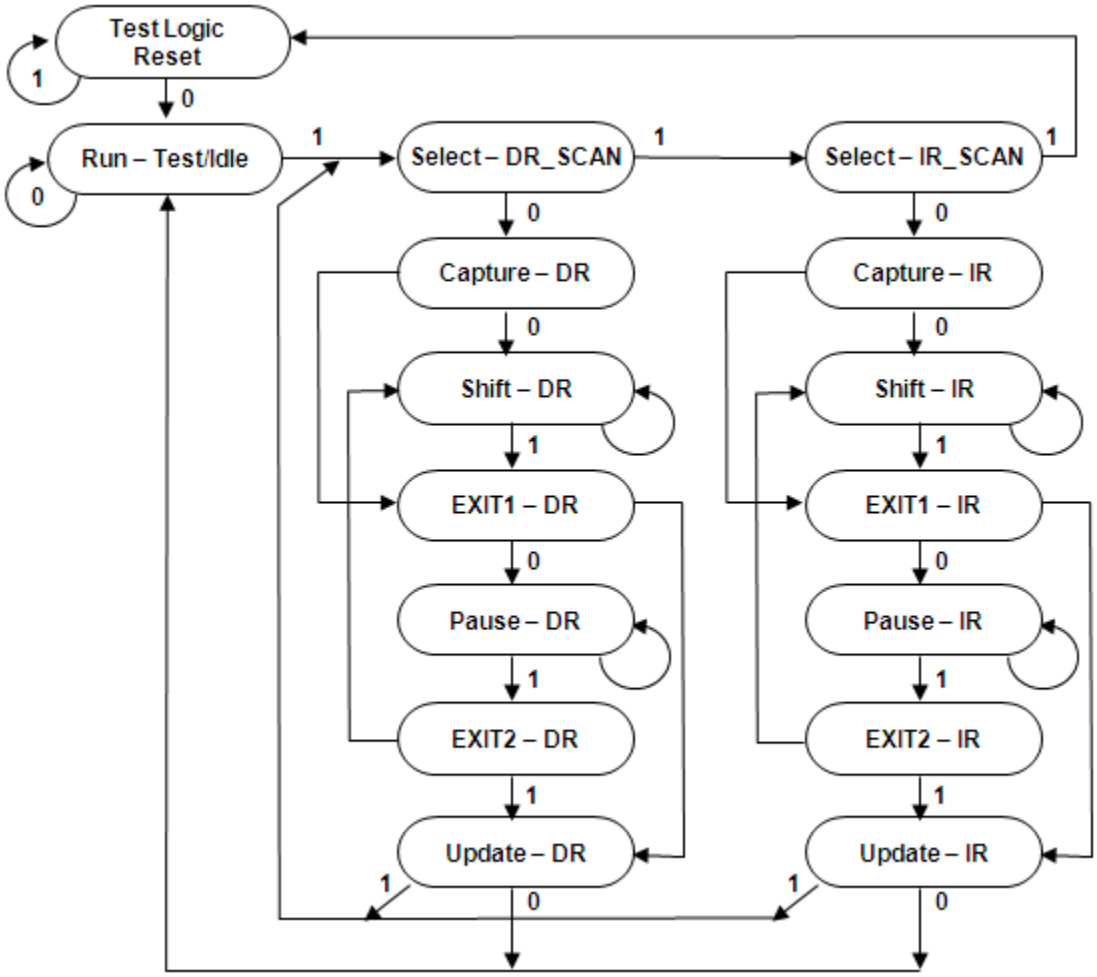


Figure 4-11 JTAG TAP controller state machine

The state machine inputs are obtained from the TMS input port on the FPGA board and determine progress through the graph. Once the graph has reached the states SHIFT IR or SHIFT DR, the input from TDI is shifted into the IR or BSR, depending on the current state. By this implementation a function code can be shifted into the IR for selecting the desired function, or a bit can be shifted in the BSR for performing the desired corruption.

4.12.2. Implementation

The design of the JTAG fault injection module must follow the specifications of the TAP state machine and execute the correct bit sequence in order to achieve the desired functionality of the target JTAG interface. From an overview perspective, at first the state machine must be reset, followed by inserting the PRELOAD/SAMPLE instruction into the IR for initialization of the BSR.

The next step is to insert the EXTEST instruction in order for the fault injection module to grant access to the BSR. At this point the fault injection can be executed. The sequence must be concluded by returning the TAP state machine to the Run–Test/Idle state.

As required, the JTAG fault injection module contains five signals for communication with the target JTAG interface:

TDI Input signal from the target JTAG interface

TDO Output signal to target JTAG interface

TMS Output signal that operates the target JTAG TAP state machine

TCK Output clock signal driving the JTAG target interface

TRST Output signal that resets the JTAG interface; redundant as the same functionality can be achieved with the TMS signal

Additionally, the JTAG fault injection module requires six signal lines to connect it to the main FPGA control module, and one signal line that is connected directly to a button on the development board, as follows:

CLK IN Input clock of the module.

START FI Input signal that initiates the fault injection.

BEATS FI Input value that specifies the number of clock cycles before the fault injection is started.

DATA TO NIOS Output value to the FPGA control module representing the data that was gathered by observing the signal values present on the output pin of the target JTAG interface.

TRST Output signal that resets the JTAG interface; redundant as the same functionality can be achieved with the TMS signal.

SENDING DATA Output signal to the main FPGA control module informing it that the fault injection has been completed and that the output data is ready.

The block diagram representation of the implemented JTAG fault injection module is illustrated in Figure 4–12. The representation contains all of the described signals, as well as additional signals that were used for debugging purposes.

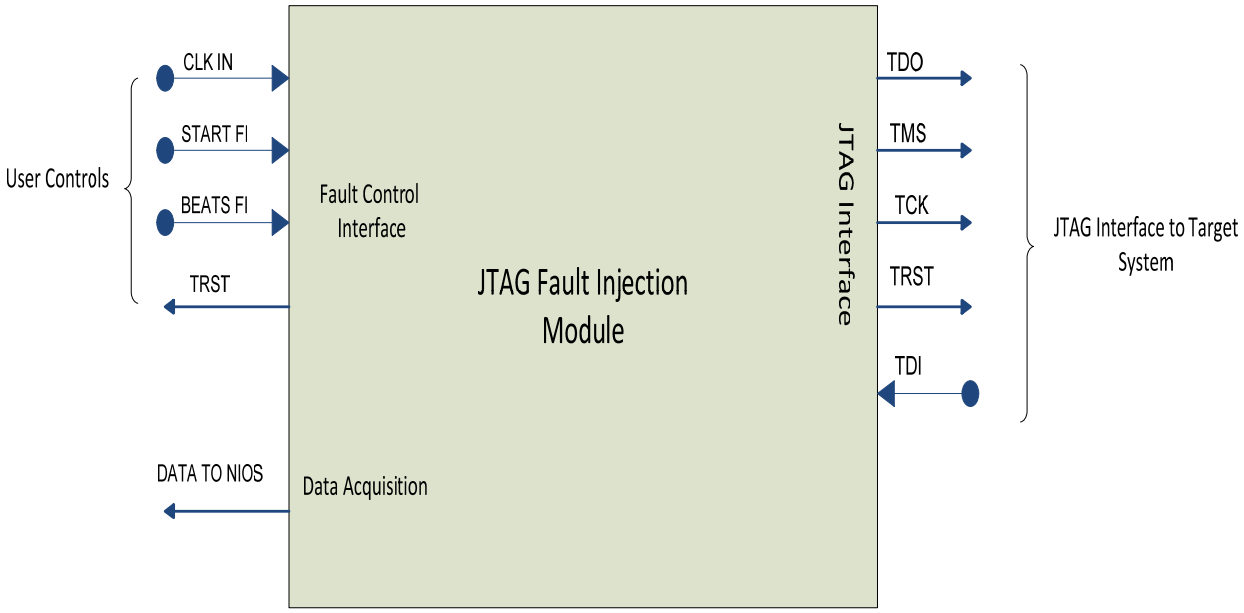


Figure 4-12 Block diagram representation of the JTAG fault injection module

The functionality of the fault injection module is based on a state machine that goes through several states (Initialize, Reset, Preload, Extest, Write, Done). The module is set up and counters are cleared during the Initialize state. When the JTAG fault injection module receives a START FI=1 signal from the test interface module, it moves to the Reset state, which resets the target JTAG interface so that fault injection can begin. This state is necessary for achieving proper initial values of all signals and variables, especially for performing multiple subsequent fault injections when the internal signals could still contain residual data. During the Preload and Extest states, the interface follows a sequence that guides it through the state machine of the host TAP controller by outputting the appropriate TMS bit signals.

At first, the system must reach the state SHIFT IR, after which the proper sequence representing the PRELOAD/SAMPLE instruction is output to TDO and shifted into the IR of the TAP controller. The sequence then continues to return to the Run Test/Idle state, after which it is directed back to the SHIFT IR, where the sequence for the EXTEST function is shifted into the IR. Afterwards, the sequence reaches the idle state again.

At this point, the target JTAG interface is set up to receive the transmission from the developed JTAG fault injection module. Therefore, the TMS signal guides the TAP state machine to reach the SHIFT DR state. During this state, the bit value presented on the TDO output signal is shifted into the BSR, while at the same time an output bit from the BSR is placed onto the TDI input signal. The interface remains in this state for the number of clock cycles represented by the number of bits contained in the corruption sequence.

The proper sequencing and synchronization of each step is achieved by utilizing internal counters implemented within the JTAG fault injection module. One counter is used for propagating through the states by selecting the proper value for the TMS signal, while a different counter is used for counting the number of bits remaining to be inserted into the IR or BSR. Each counter also serves as an index for selecting the current TDO and TMS output bit values transmitted to the target JTAG interface.

4.13. References

[Actel 2007] Actel. *Actel 42mx36 BSDL File.* 2009 . http://www.actel.com/documents/bsdl/42mx36_cqfp208.bsd (accessed 2010).

[VanTreuren 2007] B.G. VanTreuren, A. Ley. "Jtag System Test in a Microtca World." *IEEE International Test Conference.* ITC'07, October 2007. 10-10.

[Tovar 1999] E. Tovar, F. Vasques. "Real-time Fieldbus Communications Using Profibus Networks." *IEEE Transactions on Industrial Electronics* 46 (1999): 1241-1251.

[Santos 2003] L. Santos, M.Z. Rela. "Constraints on the Use of Boundary-Scan for Fault Injection." In *Lecture Notes on Computer Science*, 39-55. Berlin/Heidelberg: Springer, 2003.

[Portela-Garcia 2007] M. Portela-Garcia, L.-O. Celia, M. Garcia-Valderas, L. Entrena. "A Rapid Fault Injection Approach for Measuring Seu Sensitivity in Complex Processors." *13th IEEE International On-Line Testing Symposium.* IOLTS'07, July 2007. 101-106.

[Folkesoon 2003] P. Folkesson, J. Aidemark, J. Vinter. *Assessment and Application of Scan-Chain Implemented Fault Injection.* Technical Report, Goteborg, Sweden: Chalmers University of Technology, 2003.

[Gil 1997] P. Gil, J.C. Baraza, D. Gil, J. Serrano. "High Speed Fault Injector for Safety Validation of Industrial Machinery." *8th European Workshop on Dependable Computing.* Goteborg, Sweden: Chalmers University of Technology, 1997.

[Pignol 2007] Pignol, M. "Methodology and Tools Developed for Validation of Cots-Based Fault-Tolerant Space-Craft Supercomputers." *13th IEEE International On-Line Testing Symposium.* IOLTS'07, July 2007. 85-92.

[Chakraborty 2007] T.J. Chakraborty, C. Chen-Huan, B.G. Van Treuren. "A Practical Approach to Comprehensive System Test and Debug Using Boundary Scan Based Test Architecture." *IEEE International Test Conference.* ITC'07, 2007. 1-10.

5. DEVELOPMENT OF THE UVA PLATFORM INDEPENDENT FAULT INJECTION ENVIRONMENT

5.1. Introduction

As described in Section 5 of Volume 1, fault injection can be used at various abstraction levels depending on the information available about the system and at which stage of the design process the method is applied. Fault injection techniques can be divided into simulation-based and physical techniques depending on whether faults are injected into a model of a system, or into an actual physical system or prototype. The advantage of simulation-based fault injection is that it can be used early in the development process before the actual system is available, which facilitates early discovery of design deficiencies. Physical fault injection is important since it allows the actual implementation of the system to be tested.

While fault injection as a dependability assessment method has been applied to many systems over the past 30 years, the effort described in this report is relevant and notable for several reasons. First, the methods and techniques UVA developed were specifically designed to be applicable to a variety of digital I&C system technologies. Second, the benchmark I&C system used in this study was not designed or developed with fault injection in mind; therefore, the system presents the same challenge an independent assessor would encounter if employing a fault injection methodology on a comparable digital I&C system.

Most fault injection tools have been developed with a specific fault injection technique in mind targeting a specific system, and using a custom-designed user interface. Extending such tools with new fault injection techniques, or porting the tool to new target systems is usually a cumbersome and time-consuming process. Since one of the objectives in this research was to apply fault injection to digital I&C systems of the type found in NPPs, the need for a flexible and portable fault injection environment is a requirement for efficient application of the UVA fault injection based dependability assessment methodology. Most importantly, the work on researching and developing appropriate fault injection techniques and environments for digital I&C systems produces a body of work that the NRC and the nuclear industry can use to establish a basis for the development and standardization of fault injection methods. The work presented in this Section has as its aim to explore, develop and prototype such tools to provide a better understanding of how physical fault injection can be effectively and efficiently deployed to contemporary digital I&C systems.

5.2. Motivation and Background

Recent tools have addressed the issues of extension and portability to different target systems, but none to digital I&C systems. The GOOFI tool [Aidemark 2001] is the most advanced portable fault injection environment found in the fault injection survey (Section 5 of Volume 1). GOOFI is designed to be adaptable to various target systems and different fault injection techniques and is highly portable between different host platforms since it relies on the Java programming language and a SQL compatible database. The most recent version of the GOOFI framework supports four different techniques for fault injection. They are 1) software implemented fault injection, 2) scan-chain implemented fault injection, 3) fault injection via two on-chip debug interfaces known as background debug mode (BDM) and Nexus (a recently introduced standard interface), and 4) NFTAPE [Stott 2000], a University of Illinois fault injection tool that relies on available lightweight fault injectors, triggers, monitors and other components to facilitate porting the tool to new target systems as well as adapting it for different fault injection techniques. The Xception tool [Kanawati 1995(b)] is implemented using a modular design, and has recently been extended to include different types of fault injection techniques.

While these tools embody significant fault injection and analysis capabilities (in particular GOOFI), they are not optimally designed for industrial-based digital I&C systems. In particular they are (1) inadequate with regard to the interface needs of industrial I/O digital and analog signals that are used by the digital I&C systems (e.g. 24 volt digital I/O signals), and (2) lack the means to effectively and efficiently accomplish system integration tasks that are required for automated fault injection. Based on integration experiences with the DFWCS (reported in Section 7 of Volume 1), systems integration was a significant task in the overall work effort.

Reviewing the architectural characteristics of Benchmark System I and Benchmark System II, a portable and flexible target independent fault injection environment for digital I&C systems was a valuable asset in this exploratory research effort. Because this was an exploratory research effort, it was recognized from the outset that having a fault injection environment that is flexible and modular allows unexpected engineering problems that may be encountered to be resolved without significant redesign. The Universal platform Independent Fault Injection (UNIFI) fault injection environment was developed in this research. A major objective of the UNIFI framework is to provide a user-friendly fault injection environment with a graphical user interface and an underlying generic architecture that assists the user when adapting the tool to new digital I&C target systems and new fault injection techniques.

5.3. Requirements for Platform Independent Fault Injection Environment

As noted in previous Sections, the main purpose of a fault injection environment is to provide the necessary functional environment to perform controllable, repeatable, and automated fault injections in accordance with the fault injection methodology and the governing FARM model. The necessary functional requirements are:

- Support for various types of fault models
- Support for operational profile generation
- Accurate control, timing and measurement
- Fault list generation
- Data acquisition and analysis of results.

In addition to the basic functional requirements for fault injection it was recognized that effective fault injection environments must also be practical, adaptable to changing technology, and supportable. Early in the development of the fault injection environment several development goals were outlined for the fault injection environment to allow for technology transfer to a variety of industries. These goals were:

Flexible to a Wide Variety of Applications – Digital I&C systems and supporting communication networks are pervasive and varied in NPPs. The ability to adapt to different systems via modular plug-ins and use of pre-defined libraries is a desirable attribute for acceptance testing.

Easy to Use and Familiar to the Engineering Test Culture – Testing and fault injection environments for which the engineering community is unacquainted have little chance of technology transition beyond the academic and research world. Therefore, adopting a standard or an open source model that is widely used by the engineering community is needed if fault injection is to be used by the nuclear Industry.

Modular – There must be support for a variety of modules that are most often used in configuring target systems to fault injection test environments. These modules include functions

such as signal interfaces, file operations, sequencing of event triggers, timing triggers, data recording, and data filtering.

Managed Under Configuration Controls – The price paid for automated fault injection and test environments is large amounts of diverse data. Therefore, a means to manage the data and establish test configurations, assign relational operators, and retrieve data according to its relational properties are all requirements for effective management of fault injection. Especially, the ability to repeat and reproduce the effects of fault injections result by reloading the environmental and fault injection conditions onto the fault injector controller to allow "interesting" results to be confirmed with further testing.

Support a Variety of Fault Injection Techniques – Digital I&C platforms of different make and type may require different fault injection techniques depending on the technology used, the accessibility of system software, and the type of faults that are germane to the system. A fault injection environment that allows a "plug-in" template application to accommodate different fault injection methods is an appropriate feature to aid the user in configuring and using the fault injection environment for their specific needs.

Due to the complex nature of fault injection and the need for tight coordination of processes that are used in automated fault injection testing (e.g. I/O interfaces to the target system, data acquisition from the target system, initialization and program loading on the target computer, error logging, etc), a cross platform toolset that currently provides support for instrumentation of digital embedded systems would be the most effective path to ensure portability to different digital I&C systems. To achieve the above goals, the National Instruments™ LabVIEW [Corporation 2011] tool was selected as the basic toolset for the UNIFI environment. The intrinsic cross-platform capability of LabVIEW allows for a generic architecture comprised of components to perform the aforementioned tasks. Switching between different target systems involves modest effort, which is primarily focused on I/O issues. Because different systems and technologies may use a variety of fault injection techniques that vary between target system chip architectures, some customization is necessary, but the flexible nature of UNIFI and the LabVIEW interface reduces the amount of time from conception to implementation.

LabVIEW is a program development application, much like various embedded programing languages like C, C++, or JAVA development systems. However, LabVIEW is different from most development systems in one important respect. Other program development systems use text-based languages to create lines of code; LabVIEW uses a graphical programming language (G) to create programs in block diagram form. LabVIEW uses terminology, icons, and ideas familiar to scientists and engineers and relies on graphical symbols rather than textual language to describe programming actions. LabVIEW has extensive libraries of functions and subroutines for most programming tasks. More importantly, LabVIEW includes libraries of functions and development tools designed specifically for instrument control, data measurement, and acquisition. LabVIEW programs are called virtual instruments (VIs) because their appearance and operation emulate actual instruments. They are analogous to functions from conventional language programs. These VIs have both a function interface and a source code equivalent, and accept parameters from higher-level VIs.

With these features, LabVIEW promotes and adheres to the concept of *modular programming or function block programming,* which is widely used in digital I&C programming environments. An application is divided into a series of tasks, which can be further subdivided until a complicated application becomes a series of modest subtasks. A virtual instrument (VI) is built to accomplish each subtask and then these Vis are combined with VIs on another block diagram to accomplish the larger task. Because each sub-VI can be executed by itself apart from the rest of the application debugging is much easier. Furthermore, many low-level sub-VIs often

perform tasks common to several applications, so that a specialized set of sub-VIs can developed for future applications.

In summary, the feature rich graphical programming nature of LabVIEW, its extensive libraries of digital, analog, file I/O, signal processing, measurement modules and its broad platform support meet the fault injection requirements listed above. In addition, LabVIEW support for a variety of analog and digital input and output hardware system provides widespread connectivity to embedded digital I&C systems.

5.4. Overview of UNIFI

The objectives of UNIFI are to provide 1) a user-friendly fault injection environment, and 2) support for adaptation to new target systems and new fault injection techniques. To achieve the first goal, the UNIFI graphical user interface has been designed to be more or less self-explaining such that fault injection experiments with different fault injection plug-ins are carried out in a consistent manner.

The second objective is achieved by providing a plug-in-based framework. New techniques and target systems are added through the UNIFI-LabVIEW plug-in interface. A major advantage of this architecture is that a new plug-in can be added to UNIFI without the need of a regression test since the old system will not be affected by bugs in the added plug-in. UNIFI does not have to be recompiled when a new plug-in is added and the new plug-in will automatically be found when UNIFI is restarted in the LabVIEW environment.

Figure 5-1 shows the UNIFI tool with different plug-ins and how UNIFI interfaces with a target system and target system software development environment. In the UNIFI framework, the various fault injection plug-ins, the database that stores information, and results from the experiment are located within the host computer as shown in Figure 5–1.

The *operational profile generator* module takes a special pre-processed input file from the TRACE thermo-hydraulic simulation tool that provides all of the sensor data that the target system would acquire in its operational setting. This includes sensor data for nominal, off-nominal, and accident scenarios.

The *experiment set up and control* plug-in function selects fault injector(s), configures the fault injectors, and initializes the UNIFI tool for a fault injection campaign.

The *fault list generation plug-in module* generates a fault list that is parameterized with fault models of interest, locations of fault injection on the target system, type of fault injection, and when the fault is injected. This fault list is then loaded into a file for the experiment control plug-in to access during a fault injection campaign. The inputs to the fault list generation plug-in module are dependent on the type of fault injection selected. Typically for processor based fault injection the map files from the target system compile and link process are used as the inputs. For JTAG fault injection the boundary scan registers map from the IC vendor are used. For communication-based fault injection, the control and data packet structure of the communication messages is used to identify where and when to corrupt message traffic.

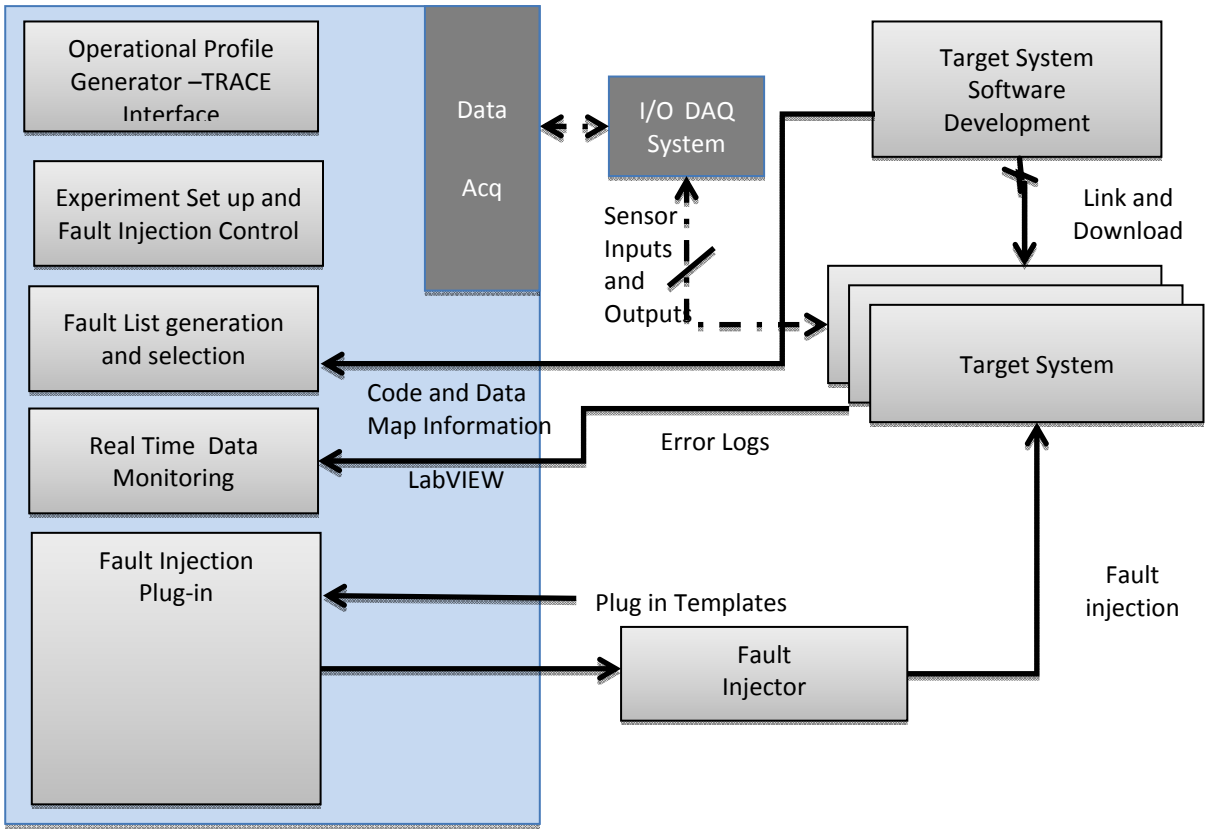


Figure 5-1 UNIFI fault injection environment

The *real time data monitoring and collection module* interfaces to the target systems diagnostics and error monitoring server to collect error messages and error logs after the fault is injected into the target system. These error logs and timing files are stored in a database that allows the error log to be correlated with the experiment control information such as the type of fault that was injected, the operational profile conditions, time the fault was injected, where the fault was injected, etc. This allows experiments to be reproduced consistently if needed. In addition, the outputs and feedback loops of the target system are sampled by LabVIEW to get a complete time response of the target system for each fault injection experiment.

The *fault injection engine plug-in module* allows different types of fault injection techniques to the adapted to the UNIFI tool. At this phase of research plug-in modules have been developed for ICE based fault injection of a Pentium Processor (Benchmark System I), the X-bus fault injector for corrupting message and control traffic on the Benchmark System I.

Switching between different target systems involves minimal effort, which is primarily focused on fault injection plug-ins, data monitoring, and the I/O subsystems. Because UNIFI does not support all target system chip architectures, some customization (e.g. designing a new plug-in module) may be necessary. However, the flexible nature of the UNIFI tool reduces the amount of time from conception to implementation.
Fault injection requires the joining together of several processes and coordinating these tasks to achieve the overall goal of automated fault injection. An example is the coordination of processes required for configuration, fault injection set-up, fault injection campaign management, and data acquisition and monitoring. With UNIFI, generic templates have been

designed for important processes to assist the user in adapting the UNIFI tool to their target system.

5.5. Configuring the UNIFI Tool to a Target System

The configuration phase of UNIFI involves adapting the UNIFI tool to the target system. UNIFI uses LabVIEW building block structures in the graphical user interface to aid the user in the definition and adaptation of UNIFI to a particular target system. There are three steps to adapting a new system to UNIFI.

(1) Interface the digital I&C system to UNIFI through the I/O interface plug-in.

(2) Describe the characteristics of the target system to UNIFI. This typically involves enumerating the processors used in the system, and defining the programmer's processor model to UNIFI.

(3) Select and configure the fault injection modules to be used on the target system.

(4) After these steps have been completed the target system will be in a controlled test environment. It is important to note that additional tasks are required to set up and conduct a fault injection campaign.

5.5.1. Step 1: Configuring the I/O Interface Module

The configuration phase is for adapting the UNIFI to the target system. The I/O Interface module provides the building blocks to connect UNIFI to the Target system. The I/O Interface module can be generalized as consisting of three functional components in UNIFI (shown in Figure 5–2).

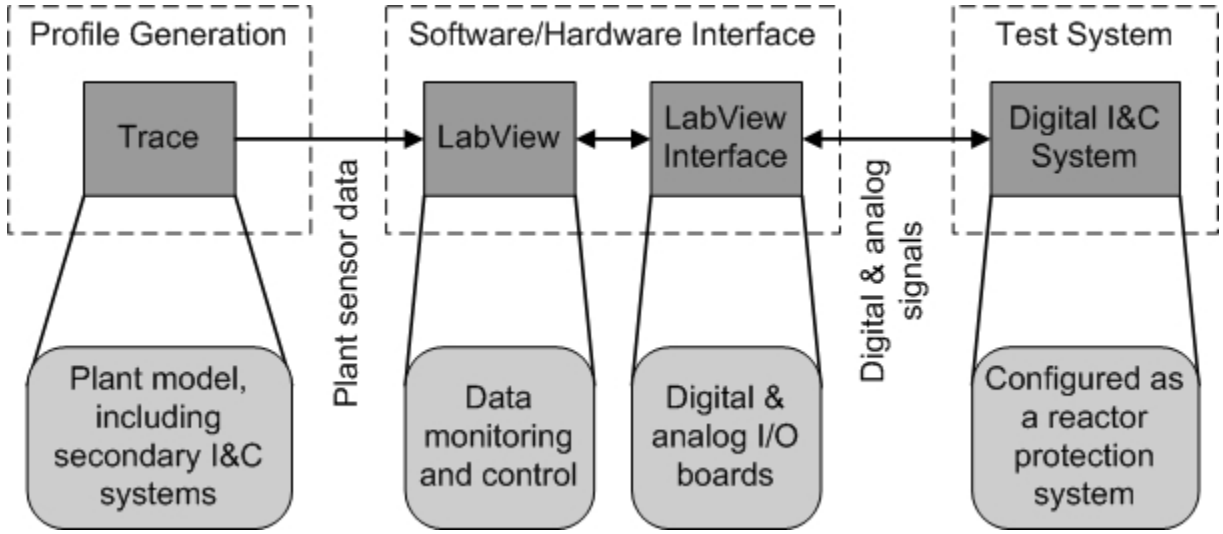


Figure 5-2 Functional representation of the I/O interface module

The I/O interface module or template is a collection of LabView block diagrams that provide a software/hardware interface between UNIFI and the target I&C system. As shown in Figure 5-2, the module performs two functions. First, it converts the sensor and signal data from the operational profiler tool into digital and analog signals, which are fed into the target system test system. Second, it collects and logs response data such as trip alarms, failure event flags,

feedback signals from the benchmark I&C system. These signals and events are recorded using the recording functions in the measurement and recording plug-in. These tasks are accomplished using a special library of VI functional blocks provided by the LabView vendor, National Instruments™. By using these functional I/O blocks it is possible to interface with a wide variety of digital I&C systems through the types of digital and analog signals typical of a plant environment.

With UNIFI the National Instruments™ PXI-1033 data acquisition system was used to provide a physical interface between UNIFI and the target benchmark systems. Figure 5–2 shows the physical connections between UNIFI host computer and the PXI-1033 data acquisition system and the target digital I&C system. The PXI-1033 performs the typical D/A and A/D conversion functions, variable signal sampling, as well as the industrial digital input and output conversions (see Figure 5–3). The UNIFI environment does not require the XI-1033 data acquisition system; consequently, the end-user can employ their particular data acquisition system. The only requirement with UNIFI is that data acquisition systems must have a LabView VI interface. This is because in UNIFI there is a tight coupling between the hardware (PXI-1033 DAQ boards) and I/O function block modules (I/O data acquisition VI modules) in LabVIEW.

Since LabVIEW monitors all signals coming in and out of the D/A and A/D boards, UNIFI provides a non-intrusive means of observing sensor and feedback signals. There are also built-in safeguards that prevent accidental system damage (e.g. exceeding signal voltage levels). The sampling rate of the I/O data can be varied to match or oversample the sampling rate of the digital I&C system under test to ensure no loss of data and accurate operational environment.

5.5.2. Step 2: Configuring UNIFI for a Specific Processor Type

In UNIFI the information about the target processor is independent of the target application. For a target system, a list of processors is downloaded to the UNIFI database. More than one processor type can be downloaded into UNIFI for a given target system, thereby allowing a family of processors to be defined for a given target system. For a given processor plug-in, a textual description of the target processor registers are downloaded into the plug-in database table when the database is created for the first time. The processor information is grouped and defined by: <location>, <CPU name>, <register type>; <register name>; < read access>; <write access>.

Location refers to where the processor is in the target system, and what its function is. For instance, the same processor may be used for different functions in target system, in one case it might be a computational processor, and in another case it might be an I/O processor.

CPU name specifies the make and model of the processor (e.g. Intel Pentium II).

Register Type refers to how the register or resource is labeled in the processor architecture. *Types* are typically user, hidden, privileged, memory mapped I/O, etc.

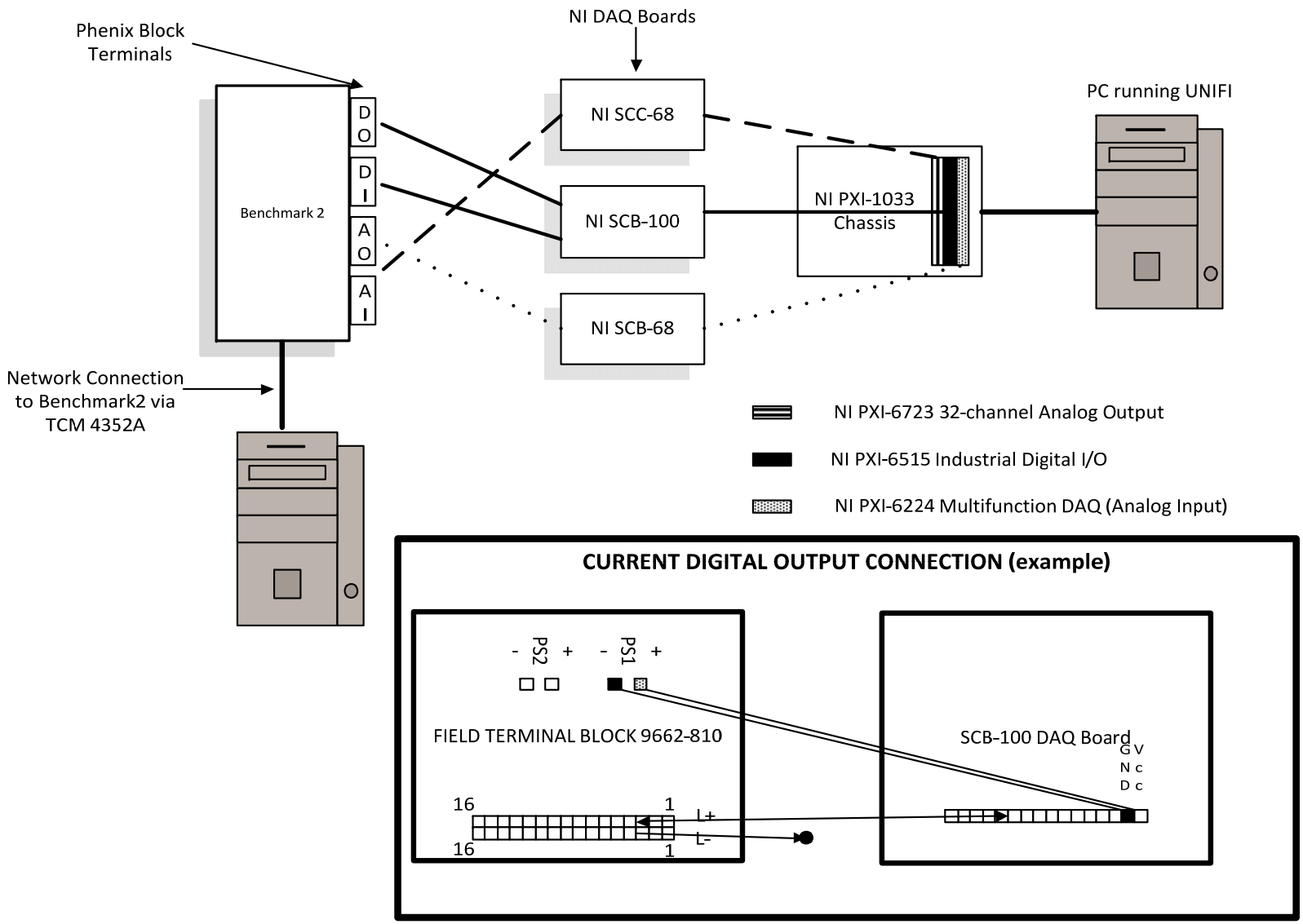


Figure 5-3 UNIFI physical interfaces to the target system

Register name refers to the mnemonic name of the register (e.g. R1, EDX, etc.).

Read access and ***write access*** refer to read and write accessibility of the registers during normal operation.

Information about where the code and data segments of the application are located in memory is defined in the target system linker file and map file. The memory map file contains information about the different memory segments used by the target application (e.g., at which address interval the program code is located). The linker file and map file also contain information about memory addresses and names of data variables in the target application. The fault list generation and selection plug-in uses this information together with the target system memory layout map to produce a structured view of the processor, I/O, and memory resources on the target system that can be used in a fault injection campaign.

By using all of this processor-specific information stored in the plug-in database together with the memory map, the UNIFI GUI presents all of the necessary information to execute a fault injection campaign.

5.5.3. Step 3: Configuring and Selecting a Fault Injector

The final step in configuring UNIFI for a target system is to select and interface an appropriate fault injection technique for the target system. As noted in Volume 1 Section 5, there are many different fault injection techniques one could use on a digital I&C system. Therefore, the fault injection environment must provide necessary functions and an API to allow the end-user to efficiently interface a fault injection technique into UNIFI. Experience with integrating fault injectors into digital I&C systems indicates that a generic template can be designed to assist the user in this task. The plug-in template has been designed with necessary modules to aid the user in the design of the interface.

Figure 5–4 shows the basic interface used in UNIFI. The modules in Figure 5-4 are available as LabView functions. These modules typically are file open, file close, file return, string to array, call function, and return function. Referring to Figure 5–4, the modules enclosed by the dashed line frame are functions typically used in the fault injector API. The *command script function* takes ASCII commands from the UNIFI interface and converts them to fault injector specific commands. *The fault list* is the target specific fault list for the selected fault injector. The fault injector API function provides the appropriate signaling interface, command interface, return status, buffering functions with respect to the fault injector API.

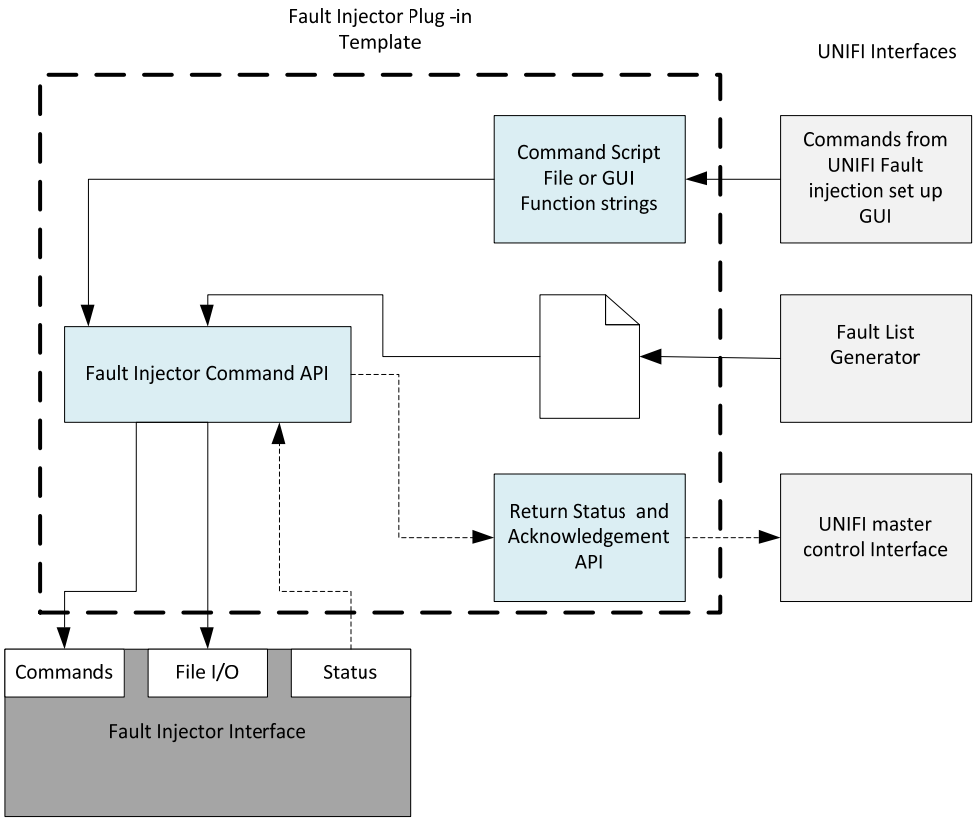


Figure 5-4 UNIFI interface for fault injection

UNIFI uses basic decision tree data structures in the graphical user interface to aid the user in defining a fault injection campaign during the set-up phase. From the GUI commands and click boxes, locations to observe and inject faults in can be selected. The user must create these tree structures in the configuration phase by providing information about the target processor (e.g. accessible registers) and the target application (e.g. where the application is located in memory).

5.6. Set up of Fault Injection Campaigns

The set-up phase is used for setting up fault injection campaigns and generally involves three steps in UNIFI. In the first step, the user enters data about the campaign in the campaign setup tab in the master controller window (see Figure 5–5). Then, specific information about where and when faults should be injected are defined in the fault injection setup tab. Finally, the registers and memory positions the user wants to observe are defined in the observation setup tab.

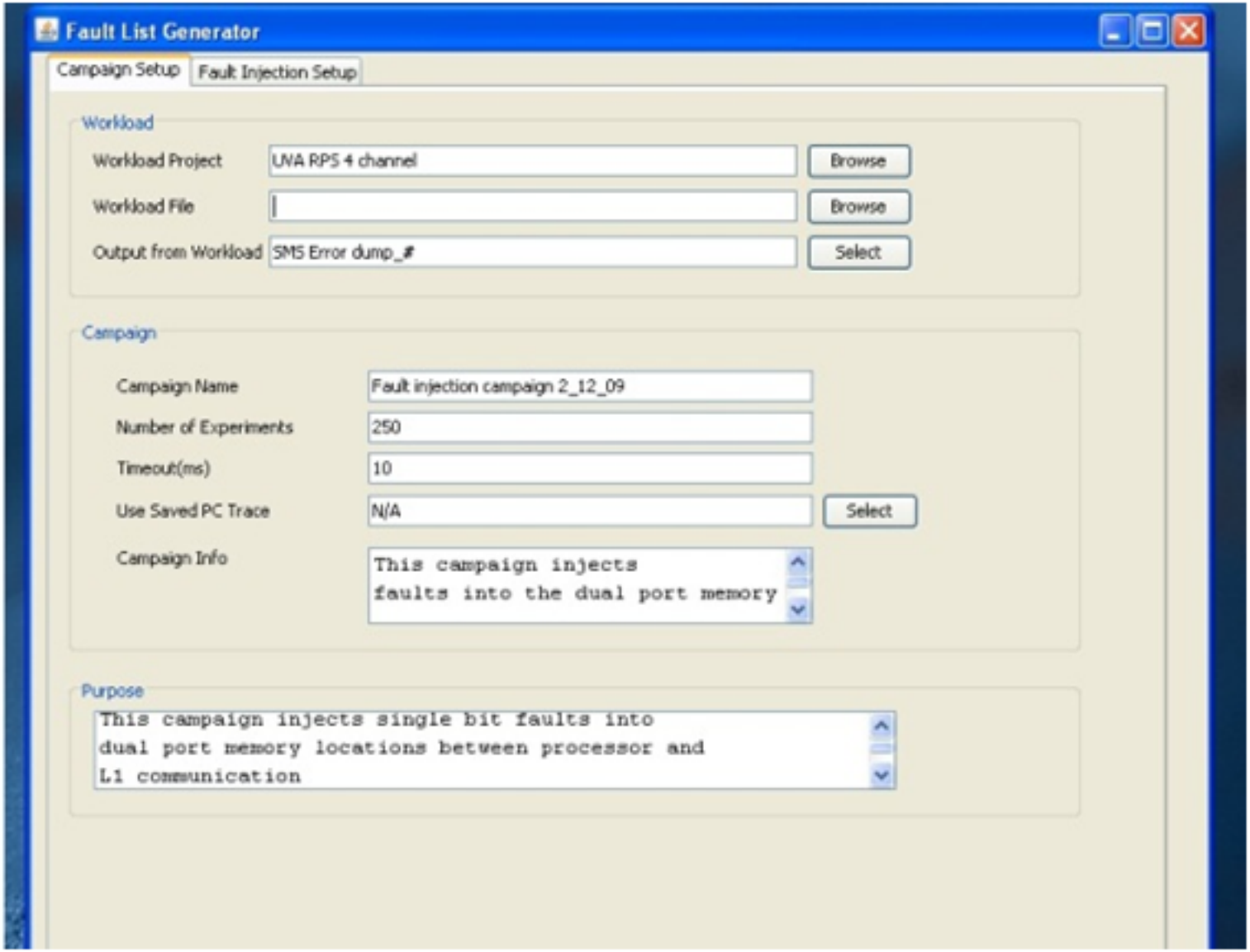


Figure 5-5 Screenshot of Master Controller window

From the menus in the GUI, fault injection campaigns can be configured by starting the corresponding plug-in for a chosen target system and fault injection technique. The campaign name, the number of experiments in the campaign, and the time-out value for the experiments must also be entered. A fault injection experiment can be terminated when a time-out value has been reached, an error has been detected or the execution of the workload ends, whichever comes first. The workload may consist of a program that either terminates or is executed as a cyclical task.

A fault injection campaign requires a reference run (fault-free run). A reference run from an earlier campaign can be reused by pressing the 'Select' button to the right of 'Use Saved PC' Trace. The user can select a reference run from a campaign using the same workload and settings as the one being configured. When the campaign is saved, the program counter (PC) trace (the values of the program counter logged during the execution of the reference run) and logged registers from the old reference will be copied to a new reference experiment belonging to the new campaign. It should be noted that the PC trace function requires some form of real time trace extraction from the running target system. This is usually accomplished by using an ICE machine or interactive debugger tool.

In some cases, target I&C systems that employ older processors do not support interactive debugger tools or the use of interactive debugger compilers. In these cases, the PC trace function can be used to store error log information from the target system. In these instances, the target system error log buffer space is cleared before a fault injection, and a clean error log file is created before the fault injection campaign begins. These reference fault free error log files are used to compare against error logs where fault injection occurred.

The user can choose between three fault injection modes:

Normal User-selected memory is saved after each control loop and user selected registers are saved at the end of the execution.

Normal and Trace The program flow is saved in addition to the Normal mode.

Detailed The program flow and user-selected registers and memory locations are saved after each executed instruction.

At present, only the normal mode has been implemented and tested. It should be noted that a significant amount of data is stored and transferred with Trace mode and Detailed mode, thus they have the potential to impact real time performance of the target system.

The user may also choose a pre-injection analysis to improve the efficiency and maximize the error acceleration of the fault injection experiment. The user can also choose whether to inject single or multiple bit-flip faults. Pre-injection analysis and error acceleration are discussed in detail in Section 8.

5.7. Fault Injection Set Up

The fault injection set up tool is used to create the detailed fault lists for the fault injector. Figure 5–6 shows the process for generating a fault list in UNIFI. As shown, there are two basic modules to the fault injection set up tool: a front end GUI for defining the fault injection parameters, and a backend parser module to parse the map files from the target system compiler. The fault injection set up tool was designed with two separate modules to enhance portability between digital I&C platforms.

The front end GUI is the high level interface that contains the relatively established fault injection parameters that are used on most digital I&C systems. These include type of fault, fault mask, memory locations, etc. Occasionally, some modifications may be necessary to the front end GUI for particular target processor. In these cases, the GUI is easily modified due to its Java based designed. The open source free-ware Net-Beans Java creation tool was used to create the front end GUI.

The back-end module is specific to the target system native object code and map file format that is generated from the compiler. This information nearly always changes from one digital I&C system to the next.

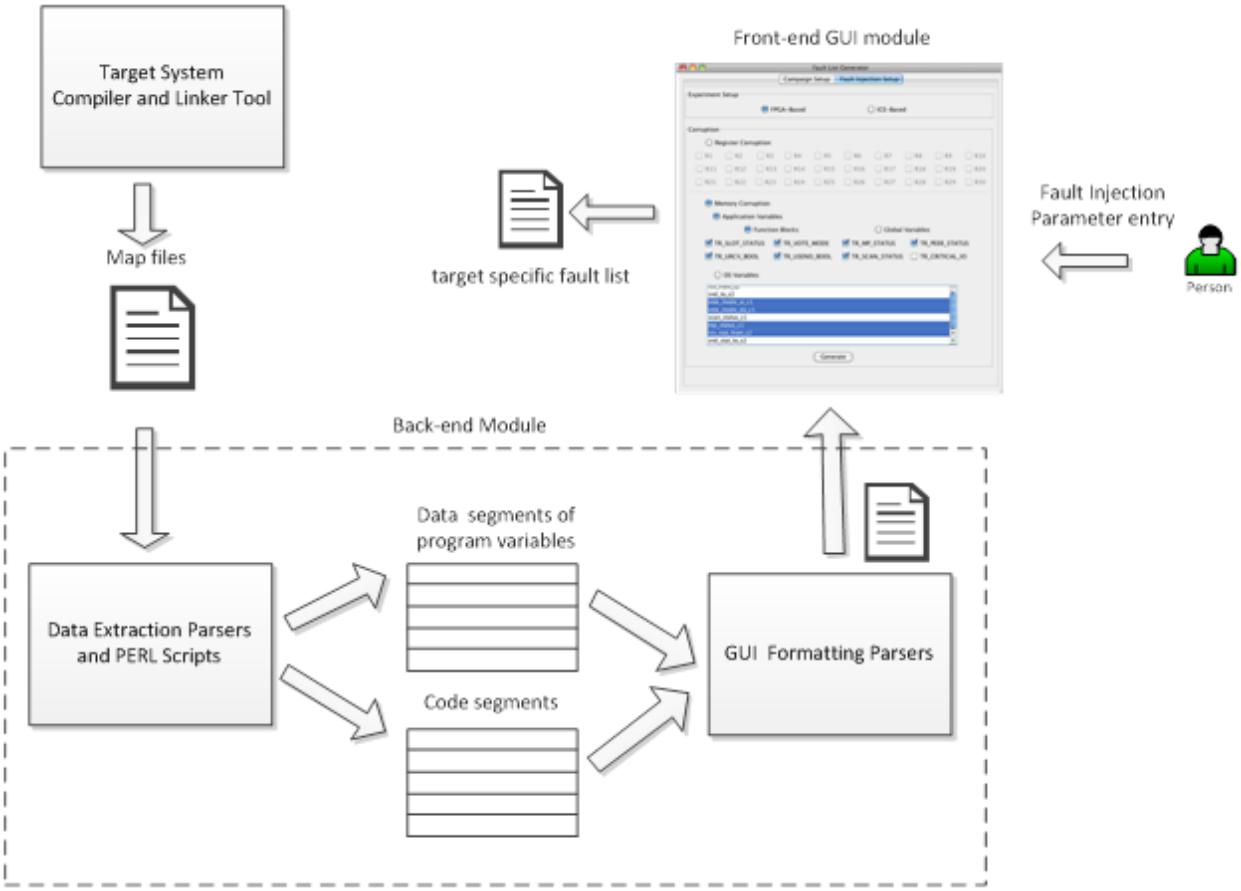


Figure 5-6 Process for generating a fault list using UNIFI

5.8. UNIFI Master Fault Injection Controller and Observation GUI

The master fault injection controller (see Figure 5–7) is where fault injection campaigns are initiated and executed, and the target system responses are monitored. The “master” controller takes information from the other fault campaign and fault injection set up GUIs to provide the necessary information to execute a fault injection campaign. The Master GUI provides the following functionality:

- Target system controls - Controls the noise levels on the sensor inputs, censor time of the fault injection experiment, and start and stop of fault injection experiment.

- Target system input and output monitoring – Graphical display of the monitored digital outputs and inputs, analog sensor inputs, and recording.
- Fault list selection – Select a generated fault list file from a directory of fault lists.
- Fault injector status information – Provides health status on the fault injector, progress on the fault injection campaigns, and when a fault is injected.
- Operational profile data files – Selects an operational profile file from a directory of generated profiles.

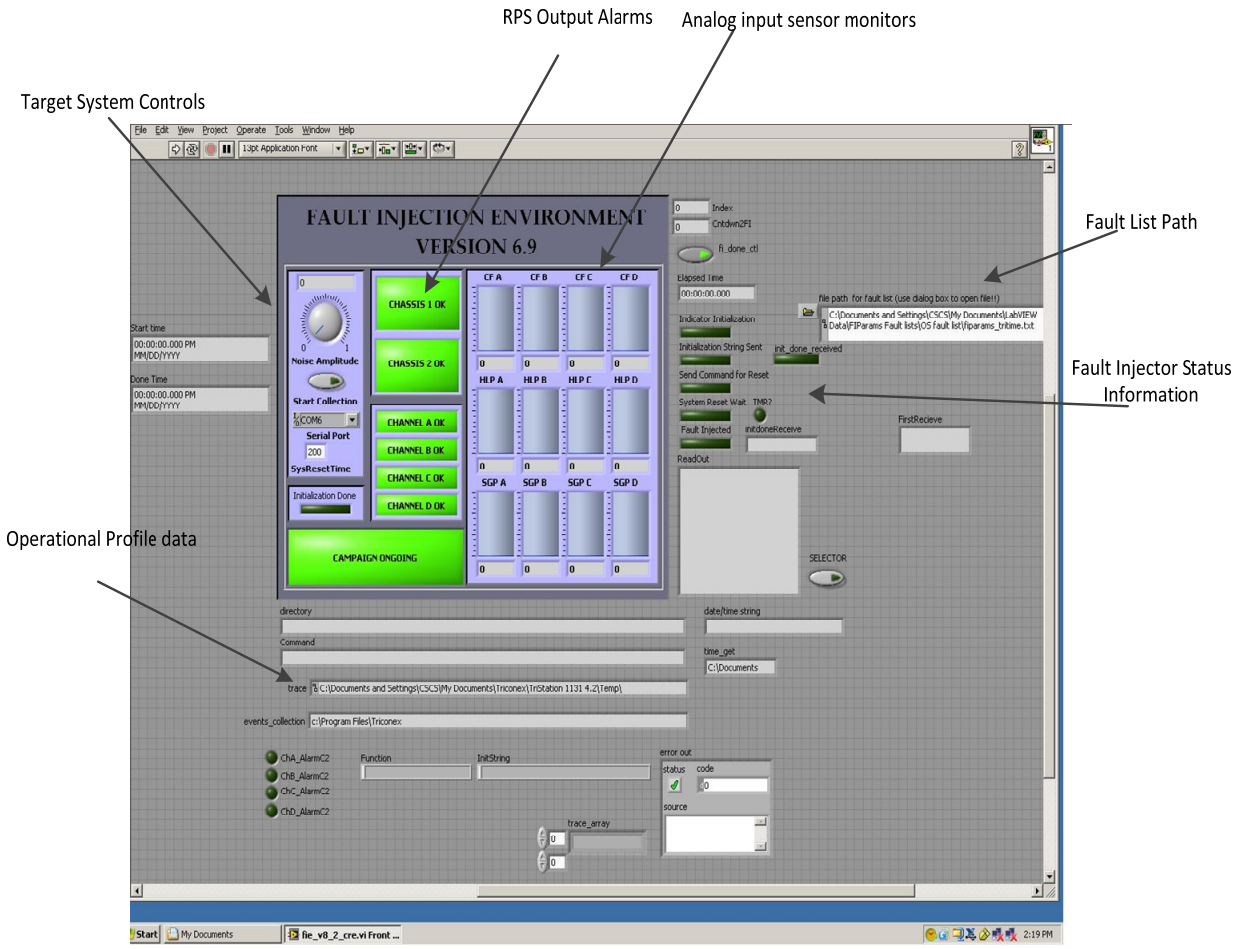


Figure 5-7 Screenshot from single fault injection trial performed by UNIFI master GUI

In addition to these basic functions, the Master GUI provides timers and trigger functions to assist the user in coordinating the sequencing of processes to conduct a fault injection trial. File manipulation functions like file open and file close are used to read and write system response data and error logs from the target system.

5.9. Configuring the Benchmark System for Fault Injection

The modular and function block nature of UNIFI allows various system integration and configuration tasks to be executed in an incremental manner allowing testing of each integration task. This development effort consisted of two separate tasks. The first task was configuring the benchmark system for the RPS mode of operation. The second task was integrating the benchmark system into UNIFI environment to ready the system for fault injection.

5.9.1. Benchmark System Test Configurations

The benchmark system was organized in two different configurations to implement the RPS application described in Section 3. The first configuration shown in Figure 5–8 was suggested by the vendor. The second configuration shown in Figure 5–9 was implemented to more accurately reflect the distributed nature of a four processor RPS configuration. Both configurations are described in the following sections.

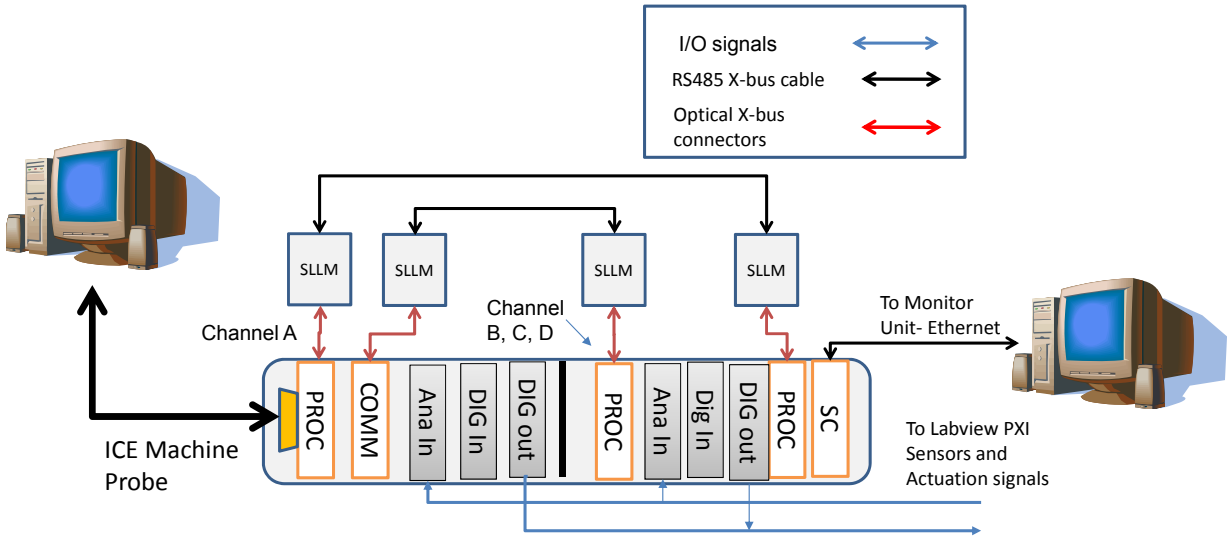


Figure 5-8 Configuration 1 of Benchmark System I

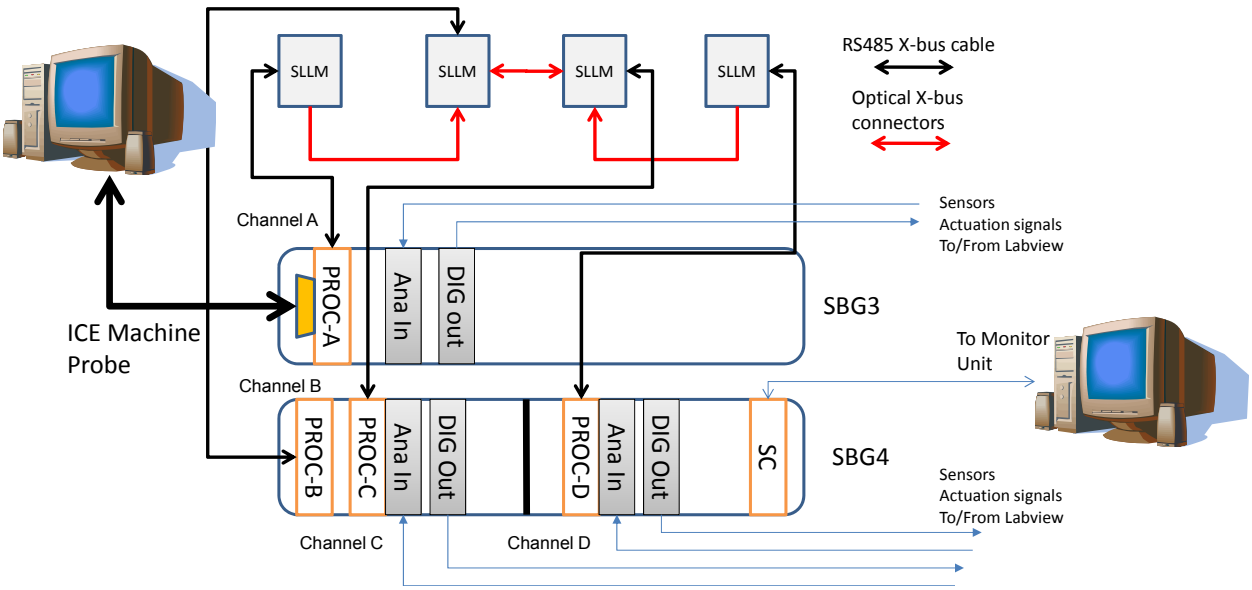


Figure 5-9 Configuration 2 of Benchmark System I

5.9.1.1. Benchmark System I Configuration 1

In configuration 1, the RPS channel A processor is isolated from the other channel processor by a split component rack. The split component rack is supplied by independent power on the left and right side. RPS Channel A has a communication processor that distributes the channel A input sensor signals to channels B, C, and D by X-bus.

RPS channel A has its own set of analog inputs for the coolant flow, SG pressure, and Hot leg Pressure process variables. These inputs operate on a 4-20ma current loop, which is typical for NPP I&C systems. There is one analog signal for each process variable for a total of three analog inputs. Digital inputs were not used in this configuration.

The digital outputs for RPS channel A are the trip signals for each monitored process variable (i.e., coolant flow, hot leg pressure, and steam generator pressure) resulting in three digital outputs. The digital output signals are 24V. The digital outputs and analog inputs are mapped to the channel A processor control. In the event channel A becomes faulty and is detectable, the outputs of both I/O modules are disengaged.

On the right side of the split rack, a single processing module emulates RPS channels B, C, and D. Analog inputs were mapped to each of the emulated channels resulting in three analog inputs for each channel for a total of nine analog signals under B/C/D processor control. Digital outputs were mapped in similar fashion as channel A. Each emulated channel produces three digital trip alarms - one for each monitored process variable. There are nine trip alarm digital signals for channels B, C, and D.

In addition, a communication processor that interfaces to the service monitor unit was located in the right side of the rack. This communication processor receives error messages from the benchmark system and, under certain operating modes, allows service messages to be commanded to the benchmark system. The service monitor allows observation of the benchmark system under test. Monitoring involves observing the benchmark system on the basis of the consistency of the observed behavior with the expected behavior of the system (or

module). Typical information that can be observed by the service monitor is fault and failure messages, signal values, tracing events, and signals.

Another important function of the service monitor is related to the operational control of the benchmark system. In certain operating modes (parameterization or test mode) it is possible to modify certain parameters in the RPS function blocks, change operating modes, set the values of signals and messages, read/write from memory, and read data from the I/O modules.

As shown in the Figure 5–8, the channel A processor was the target for fault injection by ICE based fault injection. X-bus bus fault injection is not shown, but was applied on each leg of the RS485 X-bus cable.

5.9.1.2. Benchmark System I Configuration 2

In configuration 2 (see Figure 5-9), the RPS application is distributed over two component racks in the benchmark system chassis. The reason for this configuration option is that each channel of the RPS has its own processing module and its own separate X-bus communication channel – much like a conventional four-channel RPS. Sub-rack 1 contains the components for RPS channel A. The processing module for channel A executes the function diagrams for channel A.

The Channel A analog inputs are forwarded to channel A from the channel B analog input module. The reason for this is that the test platform did not have a third analog input card to use with channel A. The same is true for the digital output module of channel A. As in configuration 1, the channel A analog inputs are the same three monitored process variable of the RPS – coolant flow, hot leg pressure, and steam generator pressure.

The second sub-rack is a split component rack. The left side contains the processors for channels B and C. The analog inputs and digital outputs for channels B and C are under control of processor B. Each channel receives independent analog signals for each of the monitored process variables. The right side of the split rack contains the processor for channel D, the analog input and digital output modules for channel D, and the communication processor for the service monitor.

The X-bus communication network is mapped as follows. Channel A is networked to Channel B, Channel C is networked to Channel D, and Channel B and C are connected together to form a complete network. Normally, a full point to point network topology would be used in a RPS configuration, however, there were not enough X-bus SLLM connection modules to achieve a point to point configuration.

5.10. Integrating the Benchmark System into the UNIFI Environment

The integration of the benchmark system into the UNIFI fault injection environment was aided significantly by the input and output data acquisition function modules of the LabView libraries. The integration is shown in Figure 5-10.

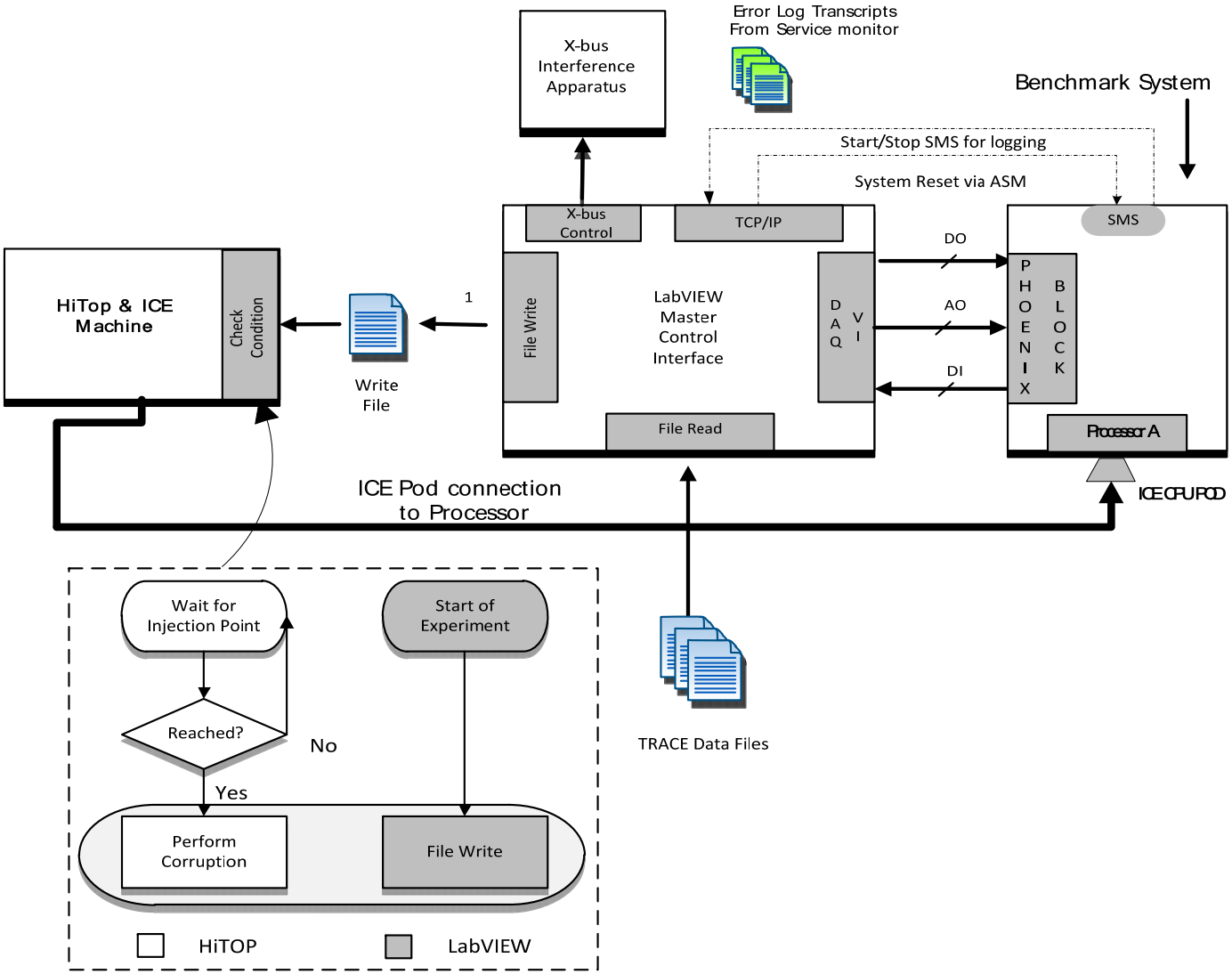


Figure 5-10 Integrating the fault injectors into Benchmark System I

A significant reduction in manpower effort was required as compared to the integration effort for the DFWCS described in Section 7 of Volume 1.

There are two principle tasks associated with the integration effort:

- Connecting and configuring the analog and digital signals between the Benchmark system and UNIFI.

- Integrating the fault injectors into the UNIFI environment.

5.10.1. Integrating the UNIFI I/O data acquisition system to the Benchmark System

The UNIFI I/O Interface module or template is a collection of LabView block diagrams that provide a software/hardware interface between UNIFI and the target I&C system. Referring to Figure 5–11, the benchmark system I/O backplane, which consists of an array of Phenix block connectors, is wired to the National Instruments™ PXI data acquisition signal connection breakout boxes. These breakout boxes form the connection between the benchmark system and the data acquisition modules in the PXI-1033 controller. The signal wiring conforms to the benchmark system interface standard as shown in the digital output example box in the figure.

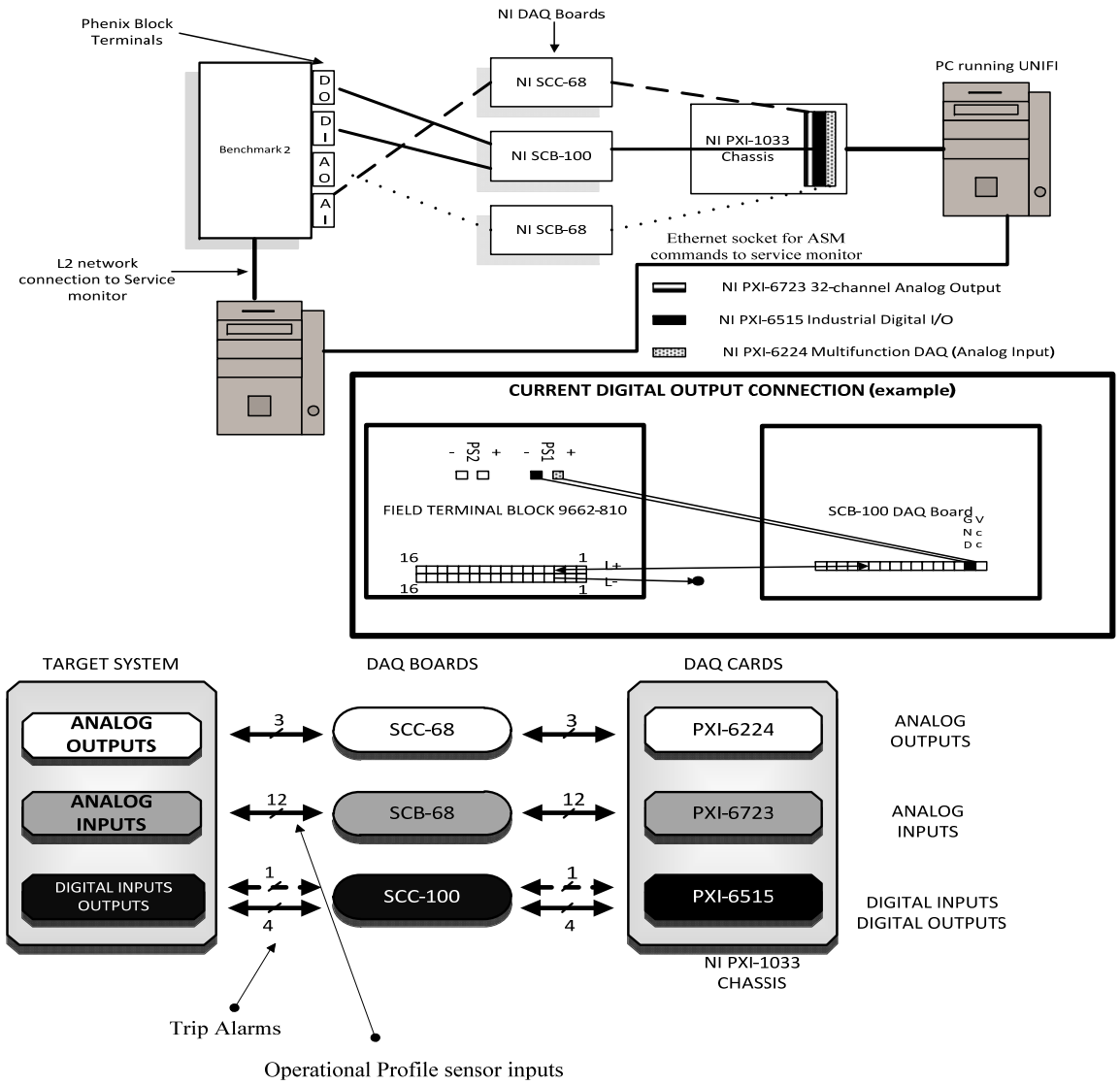


Figure 5-11 Integrating UNIFI/LabView environment into Benchmark System I

Analog input signals are connected to the SCC-68 module, digital input and output signals are connected to SCB-100 module. Both of the breakout boxes are connected to PXI-1033 controller by a wiring harness.

The PXI-1033 controller chassis contains A/D, D/A, and signal conversion boards to interface all I/O signals entering and exiting the benchmark system. The PXI-1033 controller chassis is connected to the UNIFI host computer by a PXI express connection to a PXI port on the host computer. The LabView design environment recognizes the PXI-1033 and loads the drivers for the PXI-1033 chassis onto the host machine so Labview can recognize the controller.

Control and configuration of the data acquisition cards in the PXI-chassis is accomplished through the LabView interface and libraries. All of the data acquisition, control and measurement of the signals are accomplished by UNIFI LabView function blocks.

Analog input signals (sent to the benchmark system) are generated from the TRACE operational profile generator tool (described in the next Section) and placed into a profile file. LabView reads this file and converts the sensor and signal data from the operational profile into digital and analog signals, which are fed into the target system test system by the PXI-1033 controller.

Response data from the benchmark system (digital outputs) is collected, time-stamped, and logged. The response data includes trip alarms, failure event flags, feedback signals from the Benchmark I&C system. These signals and events are recorded using the recording functions in the measurement and recording plug-in. These tasks are accomplished using the special library of VI functional blocks provided by LabView.

Since LabVIEW monitors all signals coming in and out of the D/A and A/D cards of the PXI-1033 controller, UNIFI provides a non-intrusive means of observing sensor and feedback signals.

5.10.2. Integrating Fault Injectors into UNIFI

The fault injectors described in Section 4 were integrated into UNIFI using the standard plug-in module shown in Figure 5–4 in Section 5.5.3. The interaction between the ICE based fault injector and UNIFI is handled through simple file I/O operations using the fault injector plug in module. The HiTek DProbeP5 ICE has the capability of being triggered by events based on file-write operations. As such, the UNIFI fault injector plug-in module performs a file write at specific time intervals to initiate fault injection (e.g. memory register corruption). Corruption is performed when the fault injection time point is reached, and at that time a “File Write =1” operation is performed. The HiTOP fault injector is triggered by this event. This allows HiTOP (debugging software that interfaces with the ICE machine) and UNIFI/LabView to run simultaneously and asynchronously. The fault injection experiment termination is also executed according to an external file write operation when the TRACE based operational file reaches the end.

The HiTOP ICE-based fault injector user interface is passed to the fault list from the fault list generator. This script contains the fault list data (e.g. register, memory, process variables, fault mask, triggers, etc) and instructions for triggering the injections. Once executed, the script will run through all scheduled injections automatically. An example script is shown in Figure 5–12. On Benchmark System I, a user can run approximately 200 to 500 processor-based injections each day.

The X-bus fault injector relies on a file-read operation to start the process as well as to end it. Since the file I/O code is reusable there is no additional overhead to perform this operation. A simple text file containing Boolean values that control the apparatus’ digital switches (e.g. start,

continuous token corruption, end, etc.) handles the start and termination of the X-bus fault injector process.

```
            // Memory level fault
injection
> WAIT TRIGGER T1   // Wait for
signal
> CHANGE DS:0x0000 = 0xFF  // Corrupt
location
            // Register level fault
injection
> WAIT TRIGGER T2   // Wait for
signal
> DISABLE NMI
> HALT
            // Bit flip in EAX
register
> CHANGE EAX = EAX XOR $MASK
> GO
> ENABLE NMI
```

Figure 5-12 ICE machine fault injection control script

5.11. Fault Injection Process for Benchmark System I: Operational Perspective

Referring to Figure 5–13, the process for conducting a fault injection campaign for the benchmark system starts by opening UNIFI in the LabView environment. Using the pull down GUIs, (see Figure 5–5) the user sets up the campaign parameters for the experiments to be run. The user then starts the campaign process by enabling the “run” button on the Master controller GUI (see Figure 5–7).

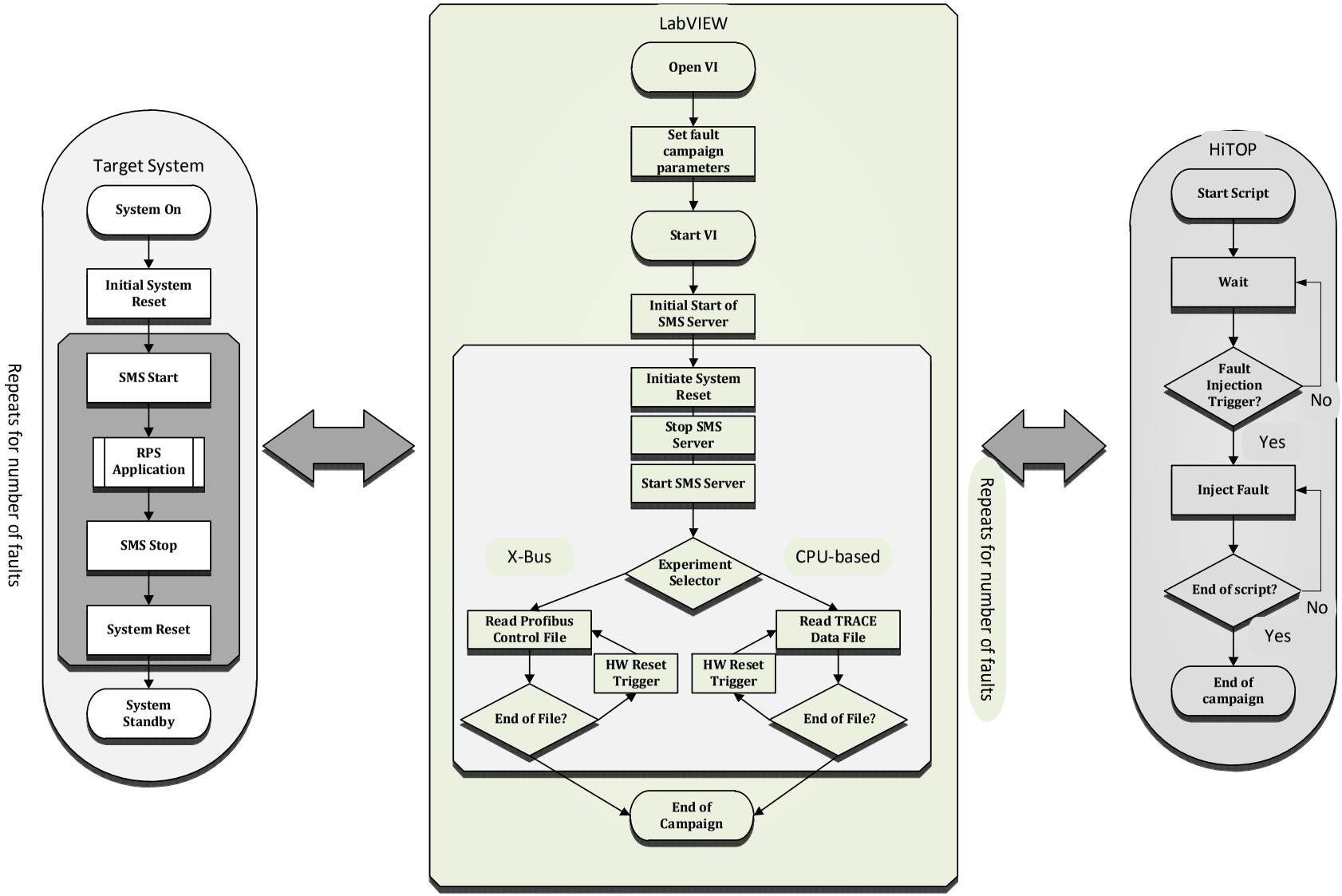


Figure 5-13 Fault injection operation for Benchmark System I

The master controller begins by initializing the parameters for the fault injection campaign. To collect error log data from the benchmark system, a TCP/IP connection socket is established to the SMS service monitor unit of the benchmark system.

The next step in the sequence is a reset initiated to the benchmark system to ensure that the system is in an error free state. UNIFI receives status information back from the benchmark system to ensure the boot-up was successful and the system is in cyclic normal operational mode. An additional step is necessary to properly establish a TCP/IP socket to the SMS server, after which the SMS server is stopped and started again.

Once these steps have been completed the UNIFI master controller enables sensor inputs to benchmark system and supplies the sensor inputs from the TRACE based operational profile data file. Once the benchmark system is running and the inputs are applied, the RPS test bed is fully operational.

The RPS system takes analog signals in and returns digital alarms indicating that signals have either exceeded the upper bound or have fallen short of the lower bound. These rules are defined by the RPS code running on each of the four processors. Since the data logging capabilities of the RPS are used, the only user interaction is the starting of the TCP/IP clients necessary to perform the system resets and logging of data in separate data files. Once the benchmark system is operating with proper sensor inputs, the steps to automatically inject faults into the benchmark system are initiated by UNIFI.

The next step is to initiate the fault injection campaign process. Three choices are possible, a processor-based fault injection, an X-bus based fault injection, or an external I/O based fault injection. The fault injection process is initiated by invoking the HiTOP fault command script described in the previous section. The script initializes the ICE-based fault injector, reads the fault list, and then executes the fault list.

This fault injections occur at the processor-level. To activate an X-bus fault injection, the process is similar. The X-bus fault injector relies on a file-read operation to start the process as well as end it. Since the file I/O code is reusable, there is no additional overhead to perform this operation. A simple text file containing Boolean values that control the apparatus digital switches (e.g. start, continuous token corruption, end, etc.) controls the start and termination of the X-bus fault injector process.

I/O-based corruption is performed by either adding noise to the incoming signals (via data acquisition hardware) or replacing the signal values with desired values. In this way, permanent faults occurring at the signal-level may be simulated.

After these steps are completed, fault injection campaigns can be executed without any further human interaction. In a 4 month period, 8,000 processor-based injections or over 10,000 network-based injections running the campaigns on intermittent basis can be performed.

5.12. References

[Corporation 2011] Corporation, National Instruments. *NI LabVIEW Technical Resources: Getting Started, Support, and Downloads.* 2010. www.ni.com/labview/technical-resources/ (accessed 2010).

[Stott 2000] D.T. Stott, B. Floering, D. Burke, Z. Kalbarczyk, R.K. Iyer. "Nftape: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors." IEEE, 2000. IEEE International Computer Performance and Dependability Symposium (IPDS 2000).

[Kanawati 1005(b)] G. Kanawati, N. Kanawati, J. Abraham. "FERRARI: A Flexible Software-Based Fault and Error Injection System." *IEEE Transactions Computers*, 1995(b).

[Aidemark 2001] J. Aidemark, J. Vinter, P. Folkesson, J. Karlsson. "GOOFI: Generic Object-Oriented Fault Injection Tool." *IEEE/IFIP International Conference on Dependable Systems and Networks.* Goteborg, Sweden: IEEE, 2001. 83-88.

6. TRACE-BASED OPERATIONAL PROFILE GENERATION TOOL

6.1. Introduction

An *operational profile* (OP) is a quantitative representation of how a system will be used within its use environment [Musa 1998]. It is a model of how users interact and use the system, specifically the occurrence probabilities of the system and user modes over a range of operations. Traditionally, it is used to generate test cases and to direct testing to the most used functions thus increasing the potential for improved reliability with respect to the use environment. Determining the OP of the non-trivial system is a challenging part of dependability assessment in general [Shukla 2004].

Another often used term that is used interchangeably with Operational Profiles is the *Workload* of a system. While the terms are similar, they are not exactly the same. A *workload* is a set of tasks or functions and their respective activation input space that reflects the *processing capacity and demand* on an embedded digital system. These tasks are typically application specific, real-time in nature. The workload on the system can vary depending on the configuration of the system, or its operating state. Thus, a workload is sub-set of an operational profile.

6.2. Operational Profiles for Real-Time Systems

Most digital I&C systems such as Benchmark System I and Benchmark System II are *reactive real-time systems*. A *reactive system* is characterized by its ongoing interaction with its environment, continuously accepting requests from the environment and continuously producing results [Wieringa 2003]. In reactive systems, correctness or safeness of the reactive system is related to its behavior over time as it interacts with its environment. Unlike, functional computations, which compute a value upon termination, reactive programs usually, do not terminate. Digital I&C systems that are real-time and reactive operate on a deterministic time-triggered basis. The software that runs on the target computer consists of a set of concurrent tasks all governed by a real-time kernel. Each task is represented as a finite sequence of events with respect to the operating system task scheduling. Tasks are scheduled on cyclical basis.

The difference between an OP for general purpose computing and a real-time OP is that general purpose OPs typically represent many customer or user domains, while real-time OPs are specific to a particular application (user) and its environment. In this effort, an operational profile is defined in the context of its application specific nature (i.e., the RPS).

Real time operational profiles to be used in fault injection experiments must be selected to be representative of the system under various modes of operation and configuration. Digital I&C system configurations may invoke different hardware and software modules in response to real time demands, and it is important that the fault injection assessment include sufficient combinations of these modules to ensure a thorough evaluation of their behavior in the presence of faults.

6.3. Characterization of Real-time Operational Profile for Fault Injection

The first step in characterizing an operational profile is to establish a use profile the digital I&C system uses according to its various operational modes. As shown in Table 6–1, a typical digital I&C system used in a safety critical plant application has at least four defined operating

modes. These modes are (1) initialization mode, (2) normal mode, (3) test mode, and (4) parameter change mode.

Table 6-1 Example composition of an operational profile for Benchmark System I.

Mode	**Sub-System Activity**					**Time Interval**
	Processors	Fault Tolerance	I/O	Communications	Service Unit Interface	
Initialization	Diagnostic test patterns, configuration checks, com checks.	May be temporarily disabled or diminished during testing	Diagnostic checks, Connectivity. I/O disengaged	Protocol initialization, diagnostics, and connectivity.	Diagnostics, connectivity checks, com checks.	2-3 minutes
Normal	Safety Function or control law operational. Various of modes of operation depending on plant configuration	Full system error and fault detection.	Acquisition of plant specific Inputs for the safety functions or control laws. Outputs signals are sent to the plant interfaces.	Data and health status traffic is passed between operational units.	System health and performance messages are sent to the operator service and monitoring station	10 -18 months
Test mode	Special Diagnostic routines, and run-time monitors are available to run concurrently with safety and control law functions	May invoke special diagnostics to enhance the detection, and isolation of a fault	Plant specific Inputs, output(s) may be disengaged.	Data and health status traffic is passed between operational units	Special diagnostic messages are sent to the operator and monitoring station	As needed for testing (~24 hours - 48 hours)
Parameter Change	Ability to re-calibrate plant parameters in the safety function and control laws.	Should have full system error and fault detection.	Plant specific Inputs and outputs, possibly special test inputs to validate the parameter change	Data and health status traffic is passed between operational units	System health and performance messages are sent to the operator service and monitoring station	Plant dependent (~8 hours)

In the initialization mode, diagnostic self-tests and health checks are executed before the system is transitioned to the normal operating mode. During this phase, the digital system does

not receive inputs from the nuclear power plant system. The average initialization time duration is around 2-3 minutes. A system initialization would most likely occur after plant outages or after a reactor shutdown event.

During normal operation, the digital I&C system monitors or controls the plant system to ensure safe and reliable operation for the prescribed safety envelope. The normal operating mode is the mode where the safety functions or control algorithms would be required to execute. The functional modes for normal mode operation are application dependent, but they always relate to the operating state of the plant. For instance, at an NPP the reactor could be operating in normal power mode, transitional mode, start-up mode, low-power mode, or manual mode. The functional modes within the safety functions or control algorithm would be a part of the operational profile make-up.

Another mode often seen is the test mode. The test mode allows for a part (or all) of the digital I&C system to be placed in a special mode where operational aspects of the system can be measured for surveillance monitoring purposes. In this mode the safety functions may be executing, but the actuation outputs of the digital I&C may be disengaged. Special monitoring diagnostics are usually invoked to monitor the performance of various sub-systems, error reports, and trends. Test mode operation usually occurs as part of a planned outage, during an unplanned outage event, and during scheduled surveillances.

The parameter change mode allows specific parameter changes to the control or safety function software. This mode of operation may occur during normal operating conditions where the controller must be tuned or calibrated to compensate for slow dynamic changes occurring within the plant. Like the test mode, this mode of operation is fairly infrequent as compared to the normal operating mode. A parameter change is usually a planned action by the operating staff.

After the operational modes have been identified and characterized, the next step is to define how the operational modes will be used in the testing environment. Since workload and the input stimulus to the system in various modes of operation can have significant impact on the estimation of parameters such as coverage [Folkesson 1999], it is important to represent the operation of the system accurately. In this research effort the primary concern was how the *normal operating mode* of the benchmark systems interact with the reactor plant model under nominal and accident-based or transient-based conditions. Another, important factor was how the testing operational profile could differ from real system operational profile data. The testing profile could be a subset of the real operational profile data, or derived from real data.

In all cases, differences between the testing operational profile and the real operational profile should be noted, if possible. To realize a highly representative set of inputs for the RPS application, the researchers had two choices: collect data from existing plant operations as was done in [Smidts 2011], or use high fidelity simulation-based plant data from TRACE [Commission 2011].

Context is important in fault injection. For a fault injection assessment methodology, the operational profiles must represent the input conditions and system interactions that can occur not only during nominal operations, but also in off-nominal operations and, more importantly, during “accident” event scenarios. Gathering real plant profile data across all of these domains of operations is a challenging task. Not all plants in operation have experienced accident events. Also, data may be limited due to proprietary sensitivities.

The use of high fidelity NPP simulator tools to generate nominal, off-nominal, and accident event profiles is a promising means to provide a diverse and representative set of operational profiles for the benchmark systems. Again, the use of NPP simulator tools for OP generation should be gauged on the fidelity of the data the tools produce, and if possible, how the data

from the tools compare with existing plant data. The challenges in this approach are (1) determining how to integrate thermo-hydraulic modeling tools like TRACE [Commission 2011] into the fault injection environment to act as the operational profile generator for the target system; and (2) how to coordinate the selection of the operational profiles with the fault injection process. At present, the methodology developed in the research provides guidance on how to use an operational profile for fault injection, but does not provide detailed guidance on the various means to realize an operational profile. The next sections describe the development and implementation of the TRACE NPP simulator as an operational profile generator for fault injection studies.

6.4. TRACE Modeling Tool

TRACE is a high-fidelity simulator developed for the NRC that is capable of solving complex fluid dynamics and heat transfer problems in components typical of a nuclear power plant (e.g., pipes, valves, boilers, and pumps). TRACE models are developed to represent reactor systems and thus are able to capture important interactions between the various systems within a plant. It is generally used by the NRC to assess plant designs and investigate possible accident scenarios such as Loss of Coolant Accidents (LOCAs) in pressurized light-water reactors (PWRs) and boiling light-water reactors (BWRs).

Models used in TRACE include multidimensional two-phase flow, non-equilibrium thermo-dynamics, generalized heat transfer, re-flood, level tracking, and reactor kinetics. Automatic steady-state and dump/restart capabilities are also provided. TRACE takes a component-based approach to modeling a reactor system. Each physical piece of equipment in a flow loop can be represented as some type of component, and each component can be further nodalized into some number of physical volumes (also called cells) over which the energy, momentum, heat conduction, and kinetics equations are averaged.

There are three major phases in a full TRACE calculation – input processing, initialization, and the solution itself. Input processing is the first stage of a calculation. At this point, TRACE reads in the input model and checks to make sure that the data is properly formatted and that all the information required for the calculation is present.

Once the model has passed input processing, it is initialized to ready it for the solution procedure. During initialization, the code performs the necessary bookkeeping functions to ensure that data is managed properly during the actual solution. Once all the input data has been processed, and the calculation has been initialized, the code proceeds to the actual solution procedure.

The solution is advanced forward in time in small increments (called time-steps). The time-steps are variable depending on the steady state or fast transient nature of the dynamics of the simulation. That is, the time-steps are unchanging if the simulation is a relatively steady state operation. If a transient or any other plant model event occurs, then the time-steps resolve down to lower time scales to capture the fast dynamics of the plant.

The calculation ends when any one of the following three conditions are met — the user-specified transient end time is reached, a steady-state is declared (only during steady-state runs), or some fatal error in the calculation takes place.

6.5. Big Picture View of TOP Modeling Tool

The TOP modeling tool is co-resident with the UNIFI fault injection environment. TOP normally operates as a separate set of programs from LabView and passes its operational profile data

sets to UNIFI/LabView environment. Figure 6–1 shows a schematic view of how TOP generates open loop operational profiles for the UNIFI fault injection environment.

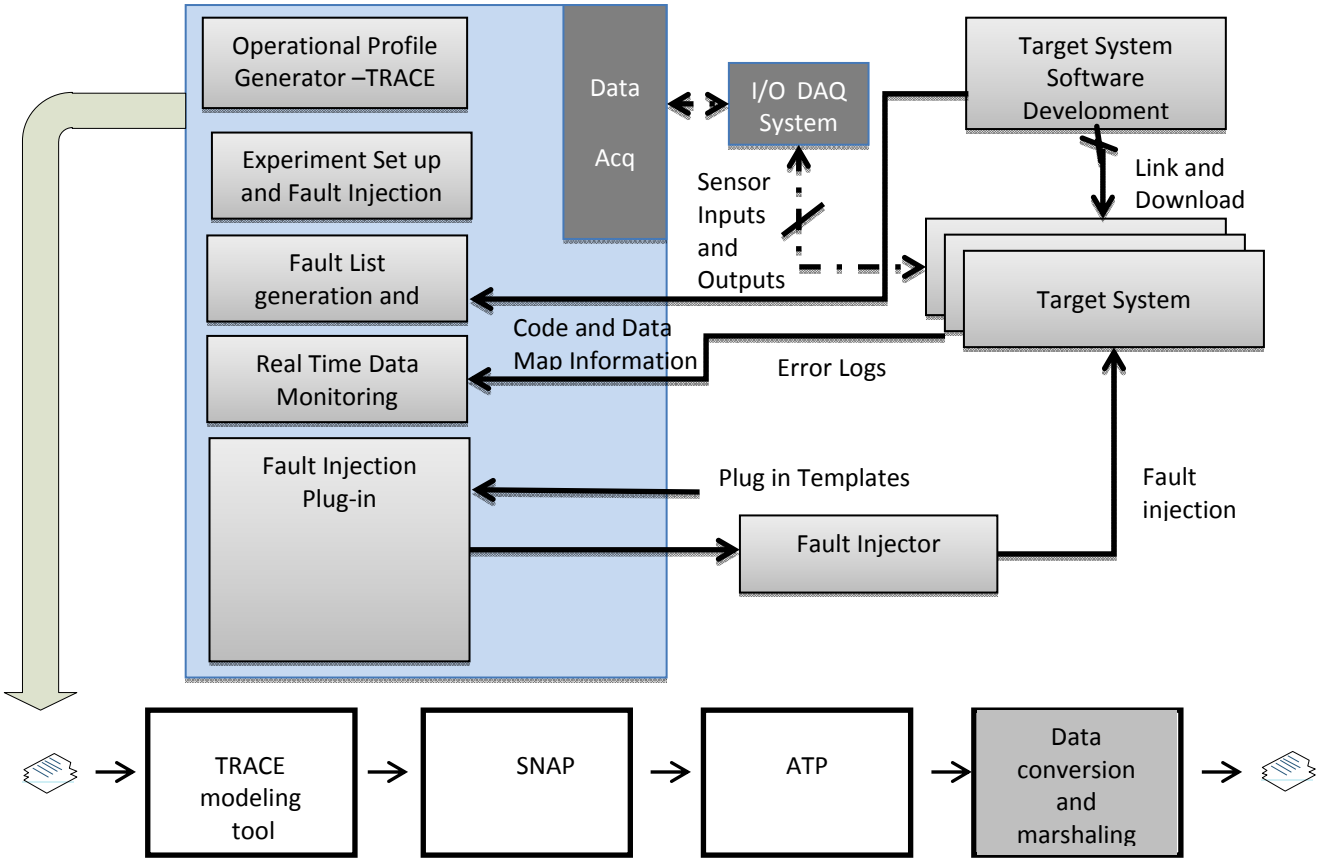


Figure 6-1 TRACE-based operational profile generation tool

There are two modes of operation for the TOP tool. The first is open loop mode where operational profiles are generated for the target system for each type of operational profile or test case of interest. The operational profile data for a specific test case or basis event is then repeatedly used for a set of fault injection campaigns. Changing the operational profile or test case or obtaining a new set of process variables only entails rerunning the TRACE simulation to collect a new set of data. For an actuation system like the RPS, the open loop mode of operation is usually preferred. In this mode of operation, the primary concern is with the trip/no-trip response of the system.

The second mode of operation is closed loop. In the closed loop mode of operation the response of the target digital I&C system is fed back to the TRACE simulation model to see how the failure response of the digital I&C system affects the overall plant response. In this way, the plant dynamics and interactions of plant systems become part of the fault injection testing process. Closed loop operation is favored for digital I&C systems where continual process control or interactions occur such as feedwater control or turbine control. The closed loop mode of operation is considerably more complex than the open loop mode. For one, the TRACE simulation tool must be able to run in real time with the digital I&C system which is typically 50ms to 100ms per control cycle. Second, the integration of digital I&C system responses back to the simulated plant model requires close synchronization and coordination of processes. Presently, the closed loop mode of operation is under development and testing. The concept

has been demonstrated in the lab, however, it has not been used beyond proof of concept for this research effort.

Referring to Figure 6–1, in the first step the user inputs plant model information into TRACE. TRACE is then run as described above to produce a set of output files. The data sets generated by a TRACE simulation include pressures, flow rates, temperatures, etc., for the various modeled components. Not all the data generated by the TRACE simulation will be needed by the benchmark system under test.

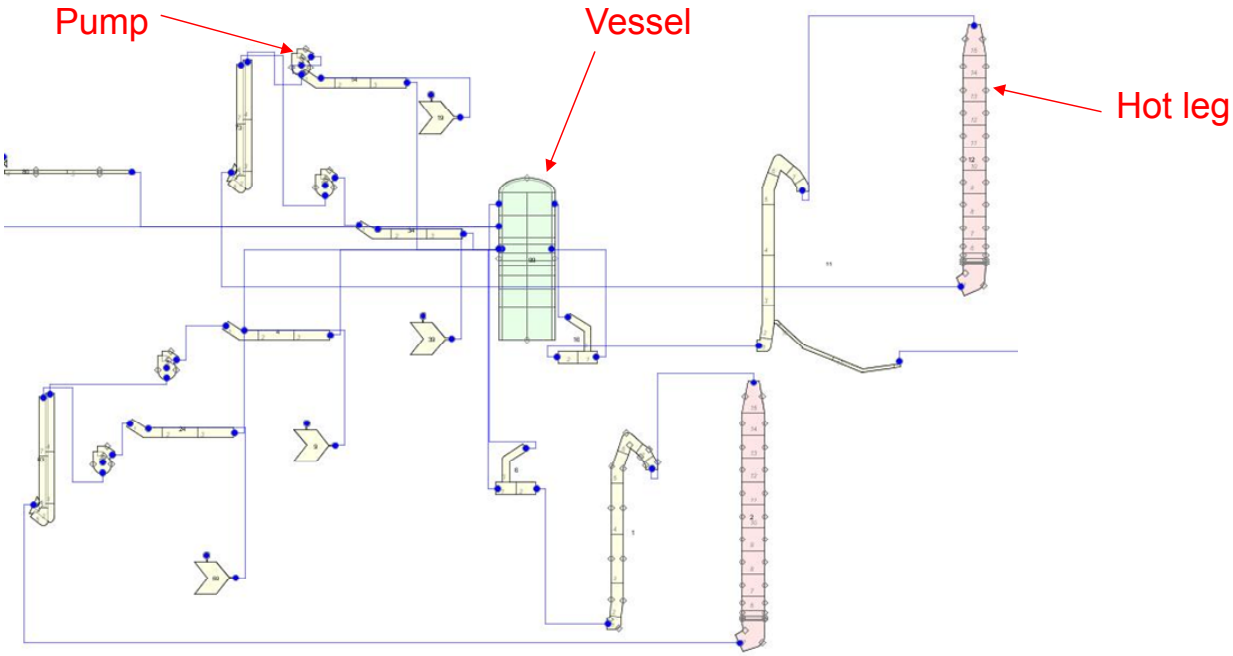


Figure 6-2 SNAP plant representation

The next step is data identification. During data identification the data sets are located using a TRACE GUI model editor (SNAP) developed by the NRC that is able to graphically represent and edit a TRACE input model. This allows one to visually identify a particular plant component and its data set. Figure 6–2 shows the SNAP representation of the plant model. Note the various components of the plant are shown in the SNAP graphical representation.

During data identification, the plant model used during data collection is loaded into SNAP. Importing the TRACE model generates a visual representation of the model structures. This provides those not familiar with the TRACE structure naming conventions to identify the components of the model that correspond to the sensor data that will be fed into the digital I&C system. The names of the structures, identified by unique identifiers, are recorded for use during data extraction.

The third step involves extracting the data sets from the TRACE/SNAP data sets. The unique IDs identified in the previous step are used to extract the data sets from the raw data file using a tool called AptPlot. The AptPlot tool is a freely available data manipulation tool. Figure 6–3 shows a screen shot of AptPlot. AptPlot includes functionality that allows raw TRACE data files to be imported. While AptPlot includes functionality to manipulate the imported data, such as generating data plots and various statistical functions, for this purpose it is merely used to extract the data sets of interest out of the raw data file using the names gathered during data identification. The extracted data is exported to an ASCII text file.

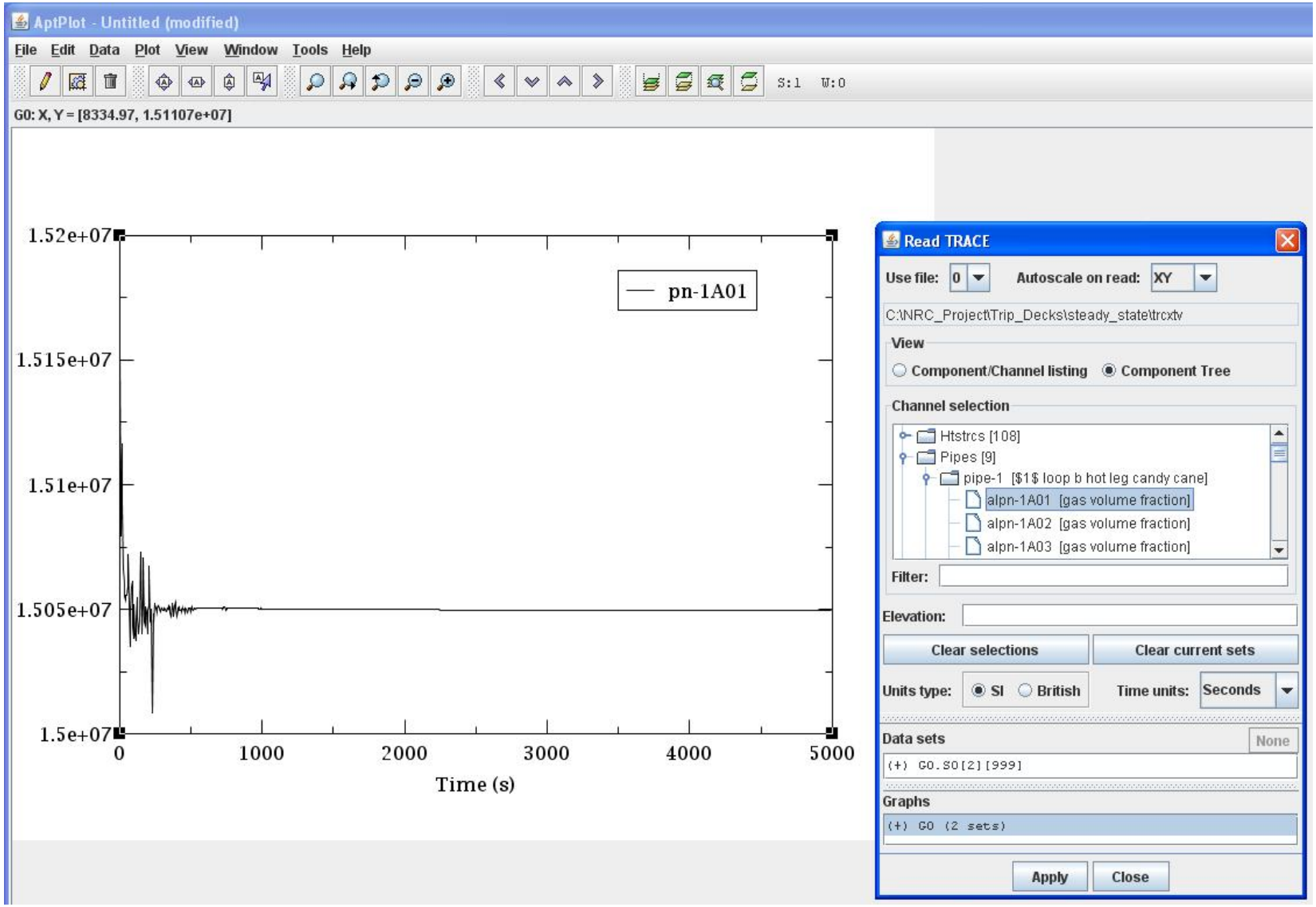


Figure 6-3 AptPlot tool

The file is reformatted (by transposing) to represent a vector of sensor readings as shown in Figure 6–4. Each row represents a collection of sensor readings for one time instance.

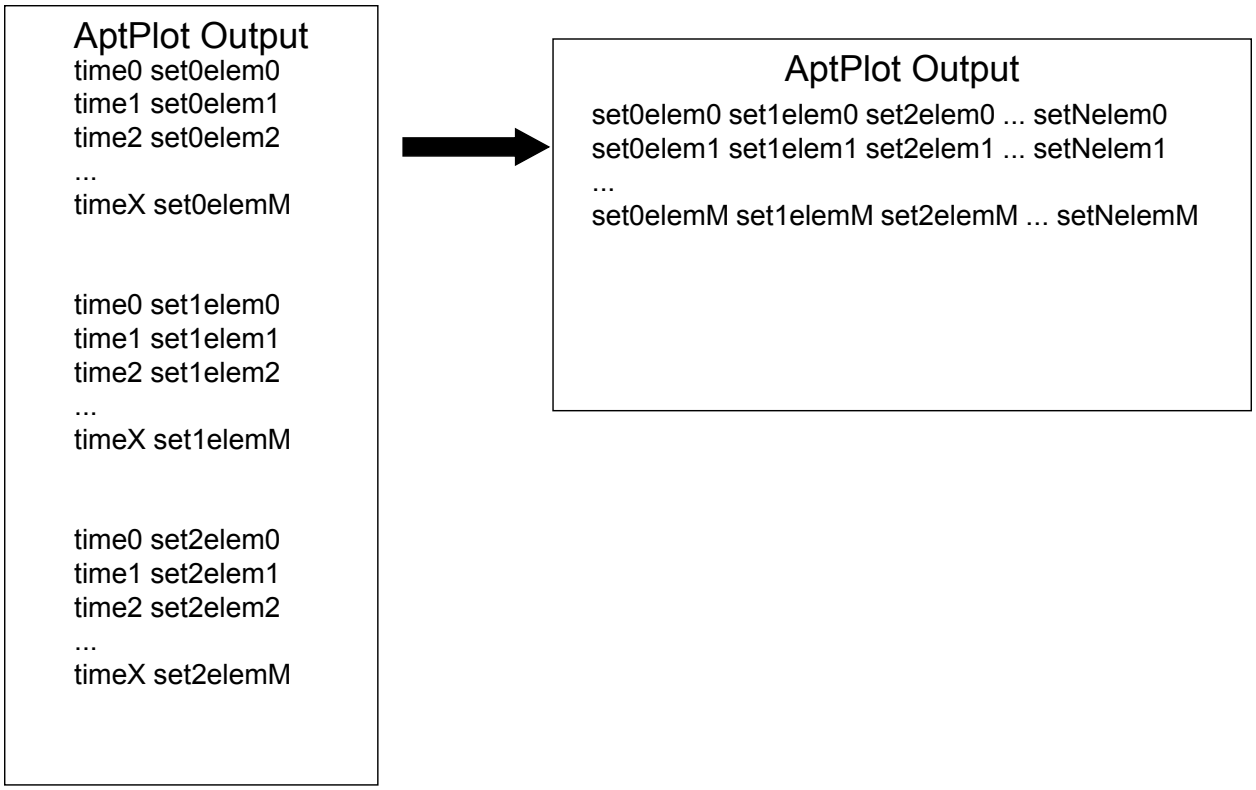


Figure 6-4 Configuring the Trace AptPlot output file

The last step in the operational profile generation process is configuring the data so that it can be executed in the UNIFI fault injection environment, combining different TRACE runs to compose a operational profile, and converting the process variables from TRACE into voltage representations for the digital I&C system. These steps are carried out by a series of Excel workbooks and programs. These steps are described in the next section.

6.5.1. Data Marshaling for Operational Profile Generation

Thus far, existing software tools have been used in the process of profile generation. However, the data files created by AptPlot are not ready for use in fault injection. The final step in the operational profile generation process is to convert the files for use in UNIFI. This step is performed by custom software tool using Excel spreadsheets and data conversion programs written in C. The user opens the Excel spreadsheets and is presented with a series of workbooks to convert the data to operational profile format for the target digital I&C system. This process performs the following functions:

(1) Read data from AtpPlot
(2) Combine steady-state with transient data
(3) Interpolate data over a constant time step
(4) Convert data from real-world units to sensor readings
(5) Instrument the data
(6) Format and output the data to a file

During step 1 the software reads all of the data into a file. For the experiments in this research both steady-state plant data and transient plant data were combined based on the reactor event. In this case the reactor event was a LOCA. For this type of profile, TRACE executions are run in two stages. First, the plant is simulated until it reaches a steady-state. A large set of steady state process variables are then recorded from TRACE. Second, the simulation is resumed with a transient scenario that represents a possible accident or event. For the fault injection experiments, the operational profile must contain portions of data from both executions.

Step 2 combines the steady state and the transient data. This step is illustrated in Figure 6–5.

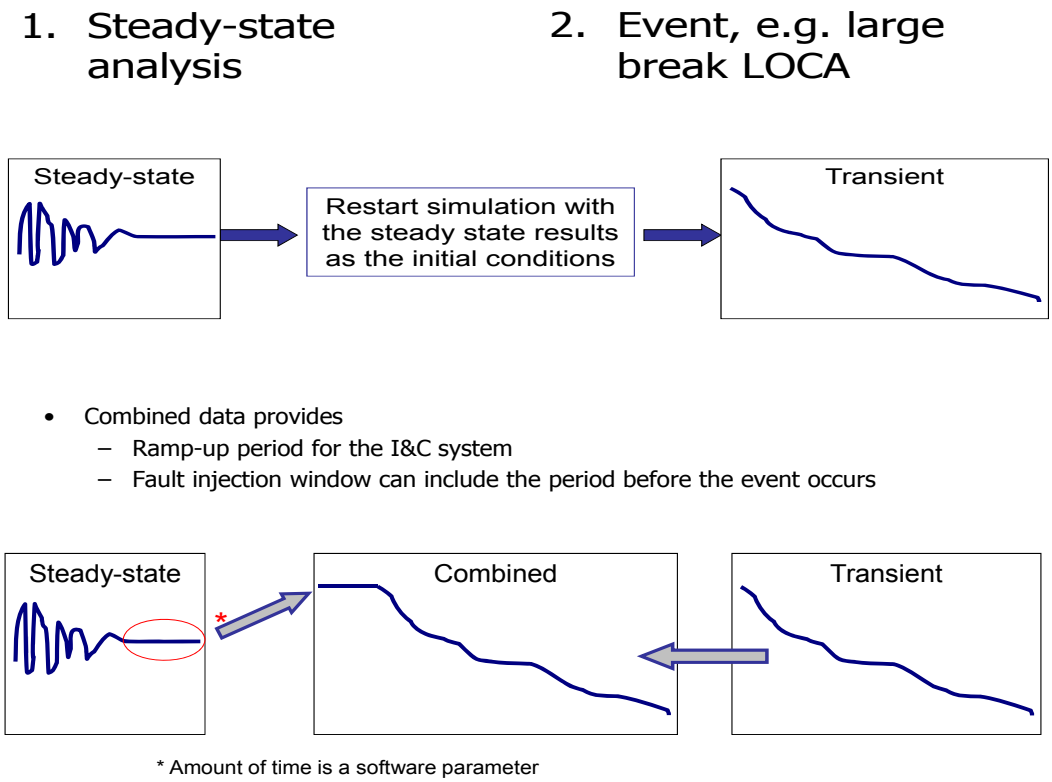


Figure 6-5 Combining the steady and transient output runs for a complete profile

Step 3 interpolates the data over a constant time step. TRACE time steps are not constant; the time step size varies depending on the dynamics of the plant model and the initial conditions. The LabView data I/O modules must send sensor data to the digital I&C system at regular time step intervals as would occur in a real plant environment. In order to provide regular sensor sampling intervals the TOP software interpolates the data over a constant time step, typically 50ms, to provide a constant delta time between data points and a resolution that is accurate of sampled sensor data. A program called *time-step.c* takes the data file from AptPlot and interpolates the time-steps to produce a constant set of sample points. The output of this process is a text ASCII file where each row is a constant time step set of sensor values.

In Step 4 real-world units are converted to electrical units. TRACE and SNAP provide results in physical units such as Pa, m^3/s, etc. The digital I&C system expects data values to be coming from plant sensors where voltage and current represent sensor outputs. The scaling is accomplished using linear scaling.

During step 5, additional information is added to the plant data that is later used to time the fault injections. Because the point in the data events at which the accident events occur is known, it is possible to time the injection of faults to coincide with those events. A countdown index is added to the data file so that UNIFI/LabView can read that countdown. When the countdown reaches zero, UNIFI is notified of this event by a file write operation. UNIFI then commands the fault injector to inject the fault. The countdown index file can be varied to inject the fault at the start of the event, before the event, after the event, or during steady state operation.

In Step 6, the marshaled data is written to a text file in a format that is easily read by LabView. Figure 6–6 is a screenshot of the Excel workbook tool used for generating the operational profiles. Starting in the leftmost column is the sample time of the measured process value. This value represents how often the digital I&C system acquires the plant data for the RPS I&C function. In this case, it is every 50ms. The next column represents the countdown index to inform the fault injection process of when to inject the fault relative to the plant dynamics. As stated previously, this value can be adjusted to start the fault injection point at various places relative to the dynamics of the plant. The remaining columns are values of the process variables from the TRACE plant model generated by AptPlot. In this figure, the values are still in physical units and have not been converted to voltage representation. The graph figures on the right show the transient dynamics of a LOCA event starting a time “0”. The plant dynamics previous to time zero are steady state, but are not shown in the graphs.

6.6. Conclusions

The wide range of uses of digital I&C systems in NPP operations illustrates that digital I&C systems are not just characterized by their internal form and function, but also by their interaction context with the environment in which they operate.

The high-fidelity data created by TRACE provides several benefits for operational profile generation. TRACE models are capable of modeling different types of nuclear power plants. Therefore, it is possible to conduct fault injection tests on a wide range of power plant configurations and RPS combinations. Using a simulator to generate plant data provides the possibility to simulate accident scenarios that, for obvious reasons, would be difficult to test in a fully hardware-in-the-loop plant test. Generating Operational Profiles from the TOP tool provides highly representative plant data that is needed for fault injection based testing. The type and variety of operational profiles that could be generated by TOP are significant. The user can choose from many types of plant or component failures in the TRACE library that are expected to trigger a response from the digital RPS system. By integrating real digital I&C systems into exiting plant modeling tools, a significant step forward toward the integrated assessment of plant and digital I&C interactions has been demonstrated.

While all of the fault injection work completed to date used TOP in open-loop mode, the engineering and interface details of using TRACE in closed loop mode where the actual digital I&C system under fault injection test sends its response data back to the TRACE plant model has been conceptualized. Completion of this effort will allow actual digital system failure responses to propagate back to the plant model where one could see how the failure mode affects the operation or mitigation response of the plant. UVA intends to continue developing TOP to this end to be used in a “full system/plant context”.

tstep	countdown	core_power	coolant_flow	pzr_pressure	pzr_level	hot_leg_pres	sg_pressure
0	-40	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.05	-39	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.1	-38	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.15	-37	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.2	-36	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.25	-35	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.3	-34	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.35	-33	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.4	-32	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.45	-31	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.5	-30	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.55	-29	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.6	-28	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.65	-27	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.7	-26	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.75	-25	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.8	-24	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.85	-23	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.9	-22	2568000000	15.084053	15008485	0.3577	15049520	6350807
0.95	-21	2568000000	15.084053	15008485	0.3577	15049520	6350807
1	-20	2568000000	15.084053	15008485	0.3577	15049520	6350807
1.05	-19	2568000000	15.084053	15008485	0.3577	15049520	6350807
1.1	-18	2568000000	15.084053	15008485	0.3577	15049520	6350807
1.15	-17	2568000000	15.084053	15008485	0.3577	15049520	6350807
1.2	-16	2568000000	15.084053	15008485	0.3577	15049520	6350807
1.25	-15	2568000000	15.084053	15008485	0.3577	15049520	6350807
1.3	-14	2568000000	15.084053	15008485	0.3577	15049520	6350807
1.35	-13	2568000000	15.084053	15008485	0.3577	15049520	6350807
1.4	-12	2568000000	15.084053	15008485	0.3577	15049520	6350807
1.45	-11	2568000000	15.084053	15008485	0.3577	15049520	6350807
1.5	-10	2568000000	15.084053	15008485	0.3577	15049520	6350807

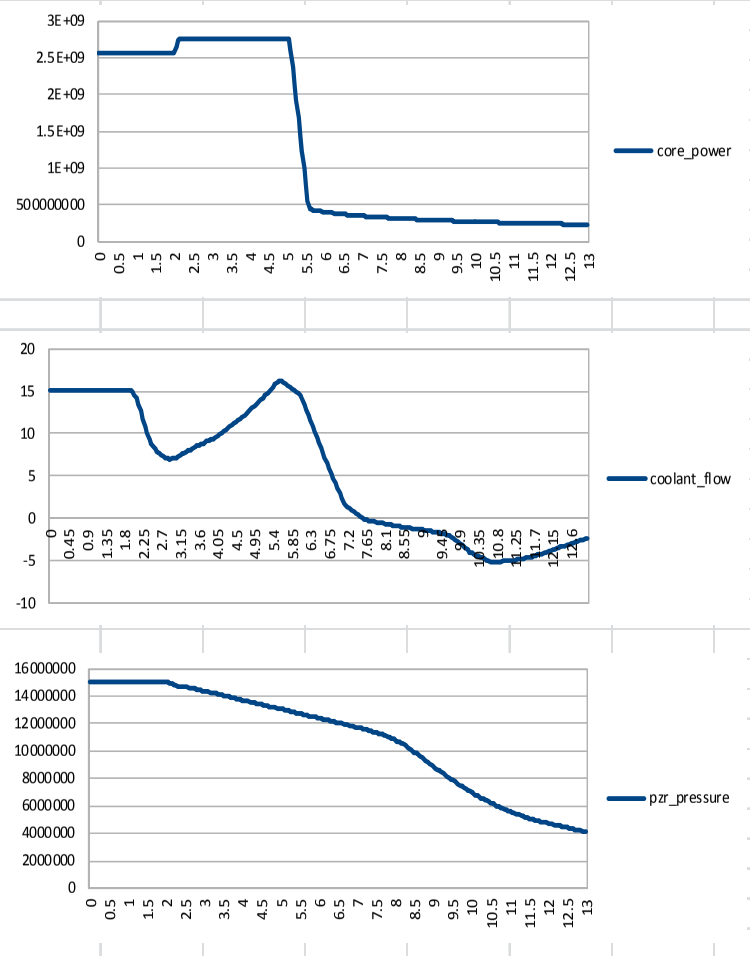


Figure 6-6 Excel screenshot used to generate the operational profile

6.7. References

[Smidts 2011] C. Smidts, Y. Shi, M. Li, W. Kong, J. Dai. *A Large Scale Validation of a Methodology for Assessing Software Reliability.* NUREG/CR-7042, U.S. NRC, 2011.

[Commission 2011] Commission, U.S. Nuclear Regulatory. *Computer Codes.* April 2011. http://www.nrc.gov/about-nrc/regulatory/research/comp-codes.html (accessed 2011).

[Folkesson 1999] Folkesson, P, and J Karlsson. "Considering Workload Input Variations in Error Coverage Estimation." 1999. 171-188.

[Musa 1998] Musa, J. *Software Reliability Engineering.* McGraw Hill, 1998.

[Shukla 2004] R. Shukla, D. Carrington, P. Strooper. "Systematic Operational Profile Development for Software Components." *Software Engineering Conference, 11th Asia-Pacific.* 2004. 528-537.

[Wieringa 2003] Wieringa, R.J. *Design Methods for Reactive Systems, 1st ed.* Morgan Kaufman, 2003.

7. PRE-FAULT INJECTION ANALYSIS AND FAULT LIST GENERATION METHODS

7.1. Introduction

The purpose of this task was to improve on existing techniques of pre-fault injection analysis by performing dynamic and static code analysis of the application prior to fault injection. Fault selection processes were refined through exploring and demonstrating methods designed to improve the efficiency of fault injection experimentation on physical digital I&C systems. This was achieved by performing analyses of program execution behavior at different levels.

Dynamic execution analysis of code was performed to select a window of opportunity to maximize error propagation of injected faults. Static code analysis was used to determine regions of code that could be deemed critical towards correct program execution. This research was carried out in two parts.

The first part of this work (which was the majority of the work) was developing and proving the efficacy of the pre-fault injection analysis methods in a system simulation environment before transitioning to the benchmark system. The effectiveness of the methods was demonstrated in a simulation environment by conducting fault injection experiments on simple applications in the simulation environment. The second part of this effort was transitioning the pre-injection analysis methods to the physical fault injection process so they could be applied to the benchmark system.

7.2. Pre-fault Injection Analysis

7.2.1. Motivation

Taken together, the methods developed in this research allow a user unprecedented capability to conduct efficient campaigns for different assessment purposes. For example, by generating fault lists with respect to functions and function blocks the user can trace fault effects that are specific to the failure of a specific function or function block. This section briefly discusses the principle fault list generation capabilities.

Being a statistical-based experiment or testing process, fault injection testing may require a large number of experiments to be conducted in order to warrant statistically significant results. Thus, efficiency of the fault injection testing is important. As discussed in Section 5, the coordination of a number of resources involved to effectively automate a fault injection campaign take some amount of time. In addition to the automated fault injection setup, the time required to perform a large number of experiments is non-trivial.

Each experiment involves initialization of the system (which includes reset and initialization of the target I&C system), followed by application of an operational profile, the actual fault injection, and then the monitoring of the system for a period of time after the fault injection. The amount of time required to initialize the digital I&C hardware can be several minutes because of the systematic nature of the diagnostics and self-tests that the system initiates at startup. The total time required to perform all steps for one fault injection experiment is typically one to four minutes on contemporary digital I&C systems. This limits the number of fault injection experiments to ~300 – 1400 experiments per day. Therefore, ensuring that a large percentage of fault injection outcomes result in producing a response from a system is very important for estimating PRA model parameters and dependability metrics.

A typical I&C system will have significant memory space (tens to hundreds of megabytes are not uncommon), and (relatively) long control cycle times (50ms to 200ms). With random fault injection experiments (i.e., experiments with no regard to when and where a fault is injected), a large fraction (up to 90%) of fault injection experiments may have no-response outcomes [Sekhar 2009]. A large percentage of these no-response outcomes resulting from fault injections are due to non-use of the corrupted data by the executing program. For example, a randomly generated fault could be injected into a memory location that is not used by an application, or could be injected into a processor register that is not in use by the application. These instances in which the injected system would not respond to an injected fault do not convey meaningful information about the fault tolerance capabilities of the system under test. Since time has an associated cost value, if the efficiency of the fault injection campaign is low, then the cost of the fault injection campaign is increased.

Therefore, it is important to minimize the number of no-response experiments so that fault injection resources are maximally utilized towards more accurate estimation of the parameters being evaluated. This is the motivation behind performing pre-injection analysis. Pre-injection analysis is a method of analyzing a fault list to ensure it is efficient. That is, to determine the locations and times for fault injection so that no-response fault injection experiments are minimized and meaningful system responses are maximized. Pre-injection analysis essentially means to determine the space-time dimension of the executing program with respect to its hardware/software interactions. A state in an executing program will be active at a specific location at a certain time. Here *location* means the use of the CPU and memory resources like data and instruction registers, code and data memory segments, and I/O registers. *Time* means the discrete clock time of the CPU. Knowing these dimensions of space-time ensures that the injected fault will corrupt the data execution flow and thus have a high probability of inducing an error.

Additionally, pre-injection analysis aids post analysis activities. For example, by ensuring that a corrupt value is used in the execution of a function on one channel of a redundant multi-channel system, the user knows that data used in the affected redundant channel is different from the data stream in the separate, redundant channels. If the affected channel or other redundant channels detect and mitigate the error, this indicates the error would not produce a significant deviation in the program behavior. Thus, the user can deduce the target I&C system is not sensitive to the error. By varying the value of the fault corruption it is possible to determine both the sensitivity of the system to a particular fault value and the propagation thresholds of the error. Secondly, injected errors that result in no responses even after pre-injection analysis may suggest a long latency period for the fault. Since a user would expect faults to be detected, the presence of no response would stimulate users to identify these faults for farther investigation.

The next section describes two pre-injection methods that were investigated, implemented and evaluated during the course the course of this research.

7.2.2. Toward Efficient Fault Injection

In order to conduct efficient fault injection experiments, the system assessor must be provided with adequate information to make informed decisions on fault injection experiments. This availability of information is classified into the following three categories: 1) The tester is provided with no information regarding the internal behavior of the system; 2) the tester is provided with some information about the internal behavior of the system; and 3) the tester is provided with all the information required to test the system.

When no information is available and fault injection testing must be performed, random fault injection is the most feasible method of fault injection. When some information such as execution traces or system logs is available, fault locations can be selected based on the

information from the execution traces and system logs. When all the information about a system is available, such as source code or architectural information, fault injection experiments can be carried out at the program information level (i.e., the symbolic content of the program).

The analysis described in this Section is focused on category (1) and (2) situations. The reason category (1) and category (2) are significant is that very often source level code is not available to the tester for proprietary concerns. More importantly, assembly level or binary level representation is what actually executes on the processor, it is not an abstraction of machine behavior – it *is* the machine behavior. For this reason, extracting fault lists at the hardware/software interaction level is the appropriate and most representative place to extract fault lists.

When a conservative approach is adopted to estimate the coverage of a fault tolerant system, the estimate is based on those fault injection experiments that cause a system response. When the efficiency of fault injection is low, as in random fault injection, more than the required number of fault injection experiments must be conducted to estimate the coverage at a desired confidence level. For example, if the efficiency of fault injection experimentation is 50%, only half the fault injection experiments lead to a system response. Further, if the desired confidence level in the coverage estimate requires 2000 fault injections with system responses, the actual number of fault injection experiments that need to be conducted is 4000. Given that the time taken to conduct each fault injection experiment can be significant, fault injection experimentation can thus be very expensive and lead to wasting resources (funds, labor etc.) if the faults do not cause a system response.

When pre-fault injection analysis is conducted, fault activation is ensured. Thus, the possibility of obtaining system responses to the fault injection experiments is also higher. This improves the efficiency of the fault injection campaign and thus fewer fault injection experiments need to be conducted without compromising the statistical confidence intervals in the coverage estimate.

7.3. Related Work on Pre-fault Injection Analysis

The INERTE® tool for fault injection is a NEXUS®-based tool for embedded systems [Yuste 2003]. In this technique, information from execution traces is used to determine memory locations used by the application. Fault injection experiments are conducted on these resources but at random instants of time [Yuste 2003]. This technique achieved only 12% fault activation in the cited reference.

The work presented in this NUREG is closest to the fault list generation method used in the GOOFI® tool [Vinter 2005; Barbosa 2005]. In the GOOFI technique, registers and memory locations used by the application are obtained from the execution traces. The control loop where the fault injection experiment is to be performed is also chosen. Fault injection experiments are performed just before the resources (registers or memory locations) are accessed. The results obtained by [Vinter 2005; Barbosa 2005] are very similar to the results observed in the experimentation underlying this NUREG, that is, about 95% occurrence of no-response faults when no pre-fault injection analysis is used, and a reduction of no-response by about 45% when pre-fault injection analysis is used. Given that the application used in [Vinter 2005; Barbosa 2005] was very different than the application described in this report, using a different processor instruction set, and development environment tends to validate observations and findings that non-optimized fault injections in modern processors produce highly inefficient fault injection experiment campaigns.

The dynamic analysis method described in this report is differentiated from other similar schemes by the use of pre-determined times for injection of faults from execution traces. This more accurate determination of time for injection of faults provides the analyst a much larger set of faults that will be activated. This enables a higher degree of confidence in the estimates of parameters being evaluated. Also, instead of injecting faults just before a read access, the window allows the analyst to inject a fault over a period of time allowing for flexibility in fault injection. This differentiates the UVA methods in this report from the GOOFI method described above.

7.4. Pre- Fault Injection Analysis to Improve Fault Injection Efficiency

When faults are injected into randomly selected locations, they mostly result in no-responses from the system. There are four possible reasons for this:

(1) The location where the fault is injected may not be used by the application.

(2) If a memory location is overwritten after a fault was injected, but before it is accessed, the faulty value gets overwritten and the fault never propagates.

(3) The error value may propagate but is not detected by the system error detection mechanisms and the output of the system does not deviate from expected service.

(4) The fault has a very long latency period.

Reason 3 implies that even though an injected fault may be absorbed by the system state, it could lead to a no response fault injection experiment. This is illustrated in Figure 7–1. Set **A** represents the set of all faults that are absorbed by the system state. These are *activated* faults. Set **B** represents the set of all no response faults. The intersection of the two sets represents the set of all faults that are activated, but do not cause a system response.

A = Set of all activated faults
B = Set of all no-response faults

A – (A∩B) = Set of all activated faults that result in a response

B – (A∩B) = Set of all faults that are not activated

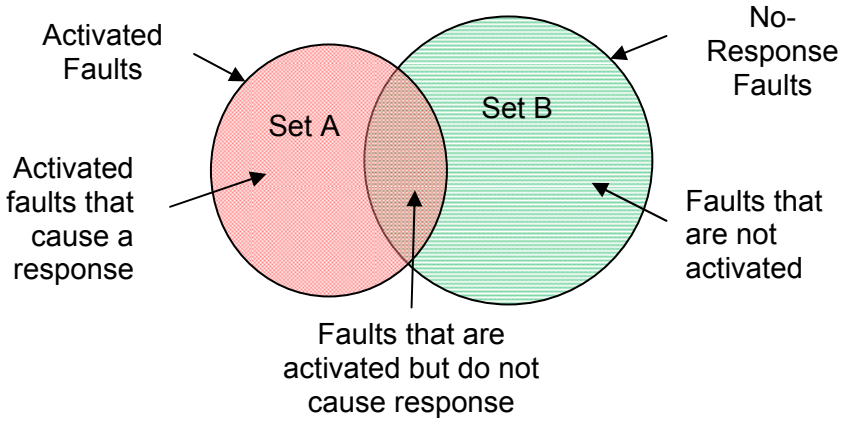


Figure 7-1 Venn diagram representation of fault space

If a fault injected into the system should produce a response, the fault should first be absorbed by the system state, that is, it should be activated. Fault activation refers to the activation or access of injected faults [Tsai 1999]. For a given application, the locations that are accessed during execution for a specified input sequence can be represented as a finite subset of the fault space. This subset may vary along all or any of the dimensions of the fault space depending upon the application workload. This is because different executions of the application would access different locations based on the applied inputs/workload. This is represented in Figure 7–2.

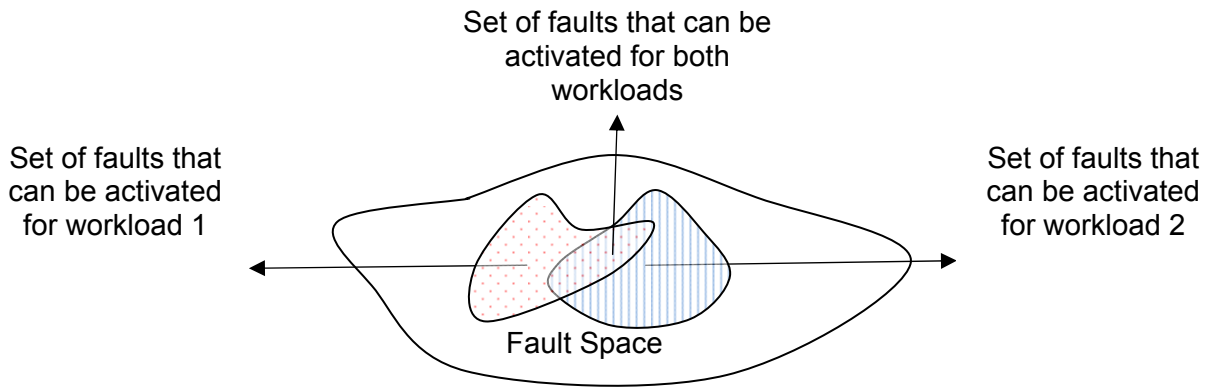


Figure 7-2 Fault activation for different workloads

Fault activation cannot be guaranteed when faults are injected into random locations. In order to increase the possibility of an injected fault affecting the system response, faults should be injected into locations that have a high probability of being activated. The choice of such locations requires careful analysis of the application from a static and dynamic point of view prior to the experimentation.

Figure 7–3 illustrates two levels of pre-fault injection analysis. The first level determines the faults that will be activated upon execution of the application, thereby increasing the chances of obtaining a system response to the fault injection. These faults are a subset of the fault space. The second level further analyzes these activated faults to obtain a list of faults that will cause a failure of the system. This level represents the set of faults that are not covered by the system. These faults are a subset of the activated faults apart from being a subset of the fault space.

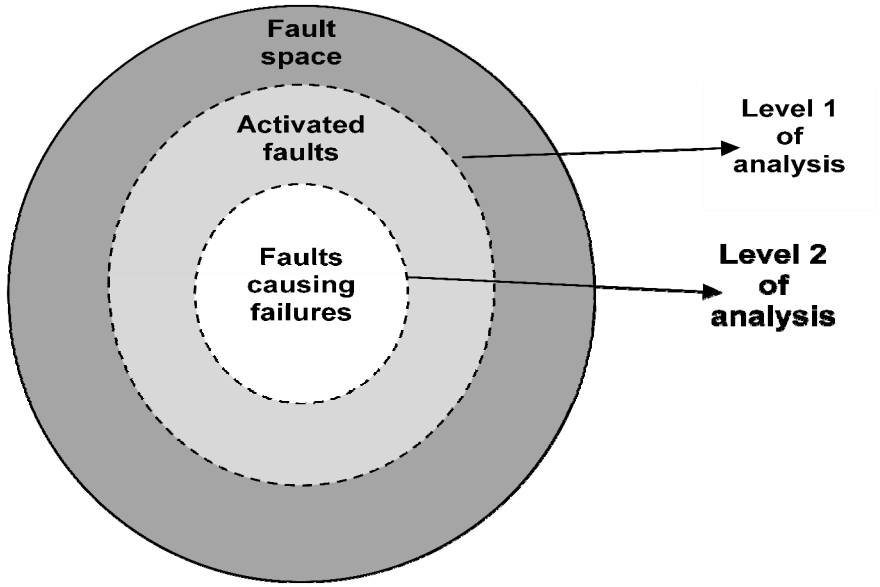


Figure 7-3 Levels of analysis

The scope of pre-fault injection analysis presented in this Section is limited to improving the number of activated faults that can be used by a fault injection process to find the covered and uncovered faults in the system. Fault list generation by pre-fault injection analysis plays a crucial role in the fault injection process. When random fault injection experiments are performed, the number of experiments that need to be conducted to thoroughly test the system can be enormous [Barbosa 2005]. However, by injecting faults into specific locations based on the probability of activation, the faults that remain latent, or those that get overwritten, in the fault list can be significantly reduced. Thus, the number of fault injection experiments that need to be conducted to exhaustively test the system will be fewer.

7.4.1. Pre-fault Injection Analysis for Improving the Efficiency of Coverage Estimation

Fault Coverage ***C*** is the conditional probability that a system detects and recovers; given the existence of a fault. It is a measure of a systems' ability to detect and recover from faults and maintain operational status, or reach a fail-safe state bounded in time [Johnson 1989].

$$C = P\,[(\text{proper handling of fault})|(\text{occurrence of a fault } \varepsilon\ \tau)] \tag{7.1}$$

The random event described by the predicate 'proper handling of fault – "occurrence of a fault **ε τ**" can be associated to a binary random variable **Y**, which assumes the value 1 when the predicate is true and 0 when it is false. The variable γ is then distributed like a Bernoulli distribution with parameter **C**. From the definition of the Bernoulli variable, the parameter of the distribution equals the mean of the variable. Thus,

$$C = E[\gamma] = 1.\ P(\gamma = 1) + 0.\ P(\gamma = 0) = P(\gamma = 1)$$

$$C = \sum_{f \in \tau} P(\gamma = 1 \mid F = f)\ P(F = f) \tag{7.2}$$

The last expression is obtained by applying the theorem of total probability. **F** is a random variable whose probability function is as given below, when a uniform fault distribution is assumed.

$$P(F = f) = \frac{1}{|\tau|} \text{ for every } (f \in \tau) \tag{7.3}$$

According to this assumption, every fault in the fault space is assigned equal relative probabilities of occurrence. **|τ|** represents the cardinality of the fault space.

As mentioned earlier, pre-fault injection analysis can be used to improve the efficiency of fault injection by reducing the number of fault injection experiments that must be conducted for estimating the coverage at a desired confidence level. A conservative estimation of coverage is based on those fault injection experiments that cause a system response. Based on single sided confidence intervals, coverage is given by,

$$C_L = (1 - \gamma)^{1/N} \tag{7.4}$$

where ,

C_L = Lower limit of coverage

γ = Confidence or significance coefficient

N = Number of experiments

This equation can be re-written as,

$$N = \frac{\ln(1-\gamma)}{\ln(C_L)} \tag{7.5}$$

From equation (7.5) it can be seen that in order to achieve 90% confidence in a coverage estimate of 0.99, 230 fault injection experiments that cause a system response must be conducted. The efficiency of a fault injection campaign (Δ) can be given by,

$$\Delta = \frac{Number\ of\ fault\ injection\ experiments\ that\ cause\ a\ system\ response}{Total\ number\ of\ fault\ injection\ experiments} \tag{7.6}$$

If 50% of the fault injection experiments result in no responses, then a total of 460 fault injection experiments must be conducted to obtain 230 experiments with responses. This is

twice the number of fault injection experiments that need to be conducted. Such low levels of efficiency are common when fault injection campaigns are conducted randomly. Table 7–1 lists the number of fault injection experiments that must be performed with the coverage estimated conservatively at 80% confidence when only 50% efficiency is achieved through fault injection.

Table 7-1 Number of fault injection experiments.

N Number of experiments with responses	C Coverage	γ Confidence	Δ Efficiency of fault injection	N_{Δ} Number of experiments required
100	0.983	0.80	0.5	200
1000	0.9983	0.8	0.5	2000
10,000	0.99983	0.8	0.5	20,000
100,000	0.999983	0.8	0.5	200,000
10^k	$0.9_{(k-1)}83$	0.8	0.5	$2 * 10^k$

It can thus be seen that when fault injection efficiency is low, significantly more than the required number of fault injection experiments must be performed. For example, if the fault injection efficiency is 50% and each fault injection experiment requires three minutes to complete (see Volume I Section 7), and a target coverage of 0.99983 is to be estimated at 80% confidence is desired, then the entire fault injection effort would take 60,000 minutes which is approximately 42 days. With pre-fault injection analysis efficiency, a Δ approaching 95% to 100%, the time is effectively reduced to 21 days to 24 days.

When pre-fault injection analysis is applied, faults are injected such that fault activation is ensured, thereby increasing the possibility of obtaining system responses. This could improve the efficiency of fault injection experiments (Δ) significantly, thereby reducing the number of fault injection experiments may be conducted to achieve the desired confidence in the coverage estimated.

7.4.2. Dynamic Analysis-based Pre-fault injection analysis

The analysis conducted by observing running code is called dynamic analysis [Sekhar 2009]. Dynamic analysis is usually conducted with the help of execution information such as execution traces. The inferences from this analysis are pertinent to the specific input sequence for which the application execution is studied [Sekhar 2009]. The technique of dynamic analysis was used to determine resources (registers and memory locations) that are used by the application for the purpose of conducting efficient fault injection experiments. Fault lists were generated based on this analysis and fault injection experiments were conducted to prove the effectiveness of this method. This method is described schematically in the flow chart shown in Figure 7–4.

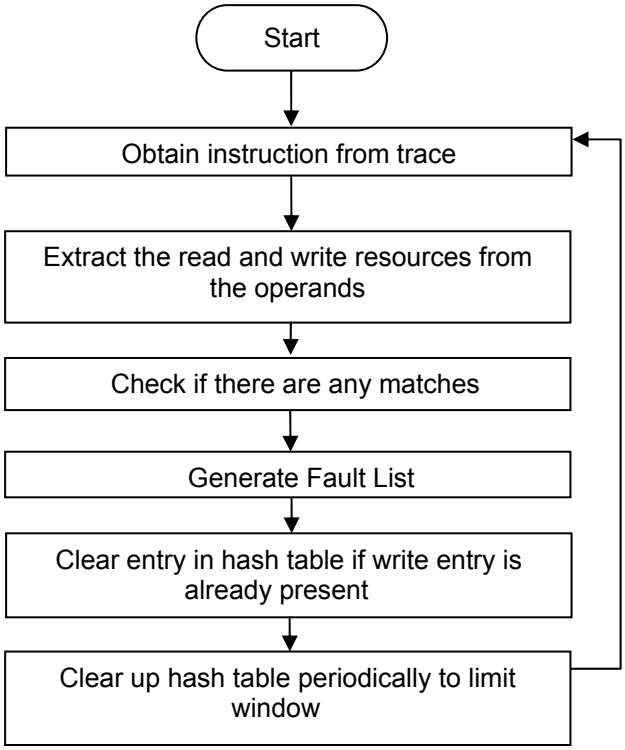


Figure 7-4 Flow chart representing fault list generation using dynamic analysis

The first step in this process is the procurement of a fault free execution trace of the application for the specific input sequence. The execution trace is a record of all instructions executed by the application. Thus, it contains information on the registers and memory locations used during program execution. Every instruction in the trace is parsed to extract the operands and classify them as source or destination operands for that particular instruction. The source operands are entered into a read array and the destination operands are entered into a write array.

Another array is maintained for each of the read and write arrays. This array stores the instruction number when the corresponding resource in the read/write array is accessed. The instruction number is a count of all the instructions that have been executed, with the first instruction having an instruction number 0.

Further, a write_hash hash table is maintained that contains a record of all registers and memory locations to which data has been written. The key to the hash table is the resource itself and the value is the instruction number at which this resource was written. The read and write arrays are cleared for every instruction that is parsed. The size of the write_hash table is controlled to limit the size of the resulting fault list. The extraction of resources into the arrays is shown in the Figure 7-5.

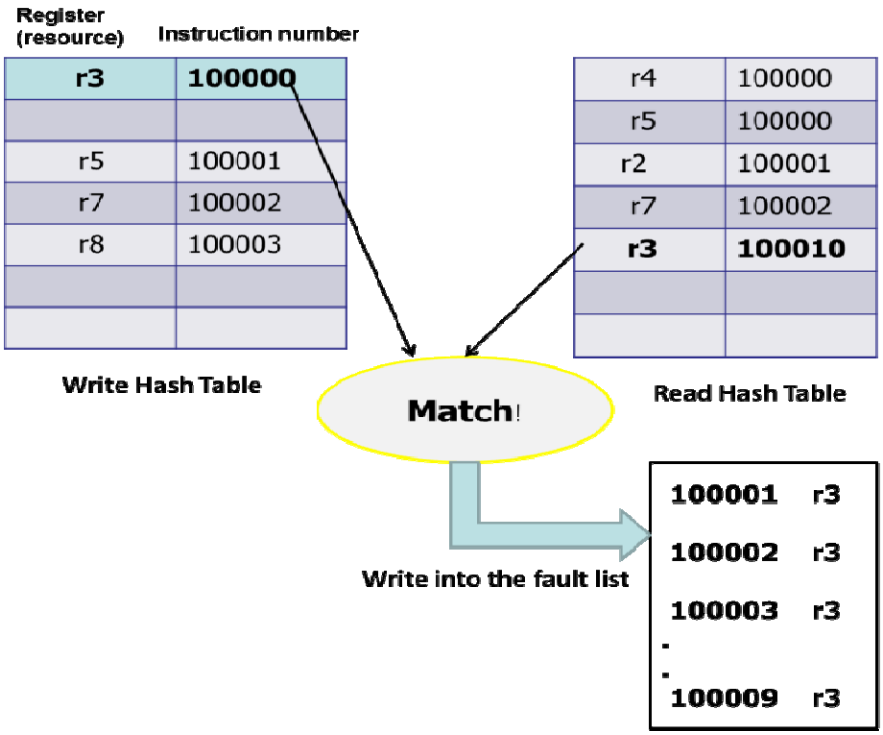


Figure 7-5 Populating the data structures with source and destination operands

Every resource in the read array is compared with the write_hash hash table to determine a write to the resource. Provided the write access took place before the read access, this time interval between the write access to a resource and the read access to the same resource is referred to as a window of opportunity. The concept of window of opportunity is illustrated in Figure 7–6.

Every instruction number within the window of opportunity is regarded as a "time" when a fault can be injected into the resource. A fault injected into the resource when the program execution reaches any of these instruction numbers, will be accessed at the read access instruction number, which would be ahead in time. Thus, fault activation is ensured. Each of the time instants in this window is translated into an individual fault injection experiment. The fault list thus contains resources and the instruction numbers in the windows of opportunity.

Once the fault list has been generated for a particular resource, the value of the resource in the write_hash table is replaced with the instruction number of the read access of the resource. This is to prevent duplicate fault injection experiments on the fault list, when a second read access is performed on the same resource. Also, if the current instruction number and least value in the write_hash table are more than 20 instructions apart, the corresponding entry in the hash table is eliminated. This will result in read accesses that do not have matches in the write_hash table when the window is greater than 20 instructions. For such accesses, a window of opportunity of 5 instructions was used. The fault list generation process is shown in Figure 7–7.

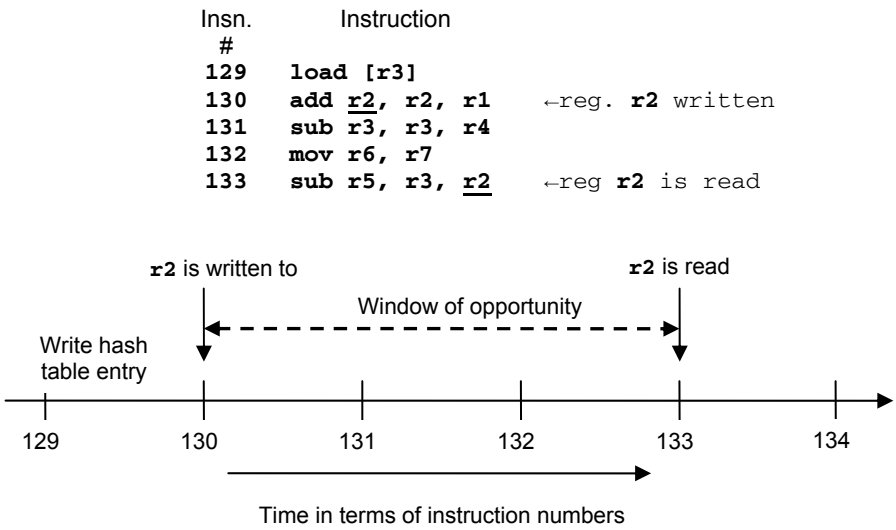


Figure 7-6 Illustration of the window of opportunity

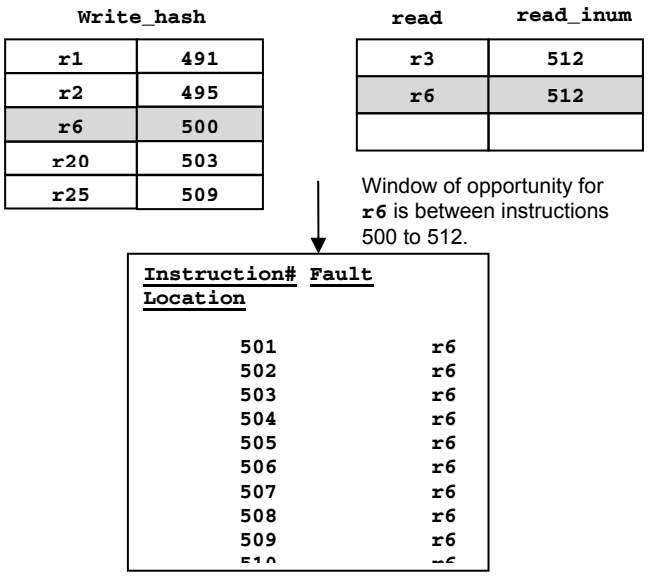


Figure 7-7 Generating the fault list

The presence of a window of opportunity allows the tester to inject faults at any time within this interval. This allows variation and control of the fault injection process. A fault list generated by this method contains faults that are ensured to be activated during fault injection.

Another important benefit of finding a “window of opportunity” is that every fault injected in the window is equivalent. This allows the use of fault expansion techniques to be applied to the data to increase the “virtual” number of fault injections thereby increasing the efficiency even more. The concept of statistical fault equivalence is discussed in Appendix A of Volume 1. The concept is briefly described here to show how pre-injection analysis supports statistical fault equivalence estimation.

The error propagation window of opportunity specifies the window of time in which a specific fault can be applied to a specific memory location to produce the same erroneous system behavior. Figure 7–8 illustrates this concept. A fault injected into register r12 at the start of the window will produce the same errors as a fault injected into r12 at the end of the window.

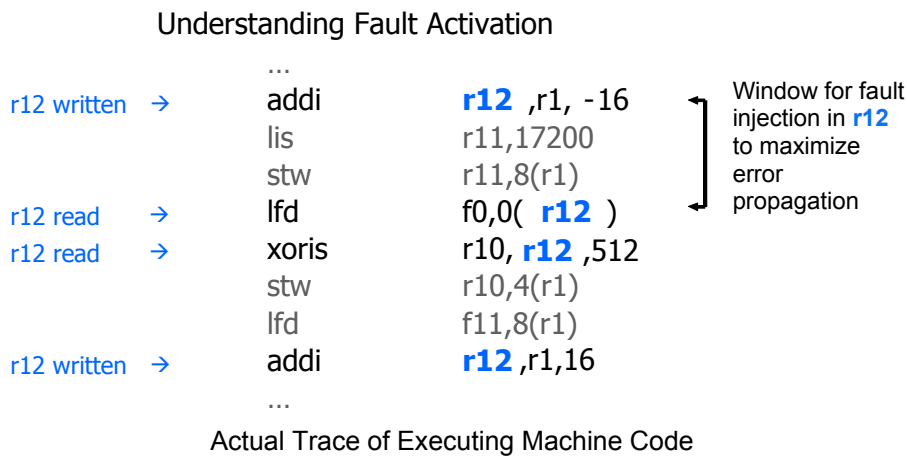


Figure 7-8 Error propagation window of opportunity

The start time, referred to as t_b , for this window of opportunity specifies how early in time that a fault, if injected, would either remain latent and not produce an error, or the earliest point in time that the fault will produce an error but not be used by the system. The end time, referred to as t_f, is the last point in time at which the injected fault will produce the same type of erroneous system behavior. If the fault is injected after t_f or before t_b then there is a possibility that the fault will produce a different erroneous behavior.

The number of faults contained in the window of opportunity is infinite if one considers time as a continuous variable. Digital systems, however, are designed based on the concept of discrete time units. Thus, the number of faults contained in the window of opportunity is measured based on this fundamental discrete unit of time. This discrete unit of time is derived from the minimum time required for the system to reach the next system state. This fundamental time unit is referred to as a system instruction clock cycle. In most cases, this discrete unit of time is the inverse of the processor clock frequency. For example, most microprocessors have some measure of pipelining and superscalar instruction issue that allows at least one instruction to be executed per clock cycle. Under this assumption, the number of system clock cycles contained in the window of opportunity is the number of equivalent faults for this particular fault injection experiment. This quantification assumes that the fault occurrence is an independent event that can occur at any time independent of the system clock. The effects of a fault occurrence event, however, are observed and propagate through the system based on the system clock. Stating this concept in mathematical terms yields:

$$C_s = \left\lfloor \frac{t_f - t_b}{T_c} \right\rfloor \tag{7.7}$$

where C_s is the number of equivalent faults, and T_c is the time period associated with one system clock cycle for the system under test.

The payoff for determining the equivalent fault set in a window of opportunity is the *fault expansion factor* for a given window of opportunity. The fault expansion factor is simply C_s. From a fault injection experiment perspective, only one fault from each window of opportunity must be sampled and injected into the target system. The response to this injected fault is noted as covered or uncovered. By knowing that all faults are equivalent in a given window, the faults can be grouped together as equivalent faults without having to inject all of them – only one representative fault sample from a window of opportunity is needed. This has the effect of "virtually" increasing the number of faults injected into the system, provided the fault expansion factor C_s is greater than 1.

Several methods for pre-analyzing fault lists so that maximum error propagation and equivalent fault sets can be achieved are presented in [Smith 1995]. Using the algorithms UVA developed and reported in [Smith 1995], the instruction stream is analyzed to uncover potential "windows" where faults will induce and propagate errors. As this increases the possibility of obtaining a system response, the no-response fault injection experiments are reduced. With fewer no response faults, efficiency of fault injection is thus increased.

7.4.3. Static Analysis for Pre-Fault Injection Analysis

Static analysis is the analysis of code prior to execution. While dynamic analysis is used to determine the program behavior for a specific input sequence, static analysis can be used to study program behaviors for different input sequences. Static analysis can be performed with control flow and data flow graphs. Static analysis can be conducted on the disassembled binary code or on the source code, if it is available. Typically, static analysis is performed on the disassembled binary code as source code information is usually not available when fault injection experimentation is performed by a third party tester.

The extent of static analysis that can be performed depends on the amount of information about the program that is available. This is classified into the following two groups: i) binary code of applications compiled without debugging information; and ii) binary code compiled with debugging information. The research explored the use of a commercially available tool, IDA Pro®, to determine areas in the code that can be identified as suitable candidates for fault injection purposes.

IDA Pro® is a binary analysis/disassembler tool that supports many instruction set architectures (Intel, Motorola, ARM, etc.) [Eagle 2008]. IDA Pro® is considered one of the best binary analysis tools on the market. For that reason it is widely used in the security arena for security vulnerability analysis of embedded systems. The tool disassembles binary code and allows the binary code to be loaded at a desired offset in the memory. IDA Pro® can identify symbolic information if the binary code has been compiled for debugging purposes. This allows IDA Pro® to identify function names and other strings present in the source code from the binary. The analysis conducted in this research is based on the assumption that symbolic information is available for the disassembled binary code, which often is the case.

An important feature in IDA Pro® that is of interest to fault injection is the graphing utilities. These utilities provide control flow graphs of the disassembled binary code as well as graphs of cross references to the functions. A cross reference to/from a function of interest refers to a function that is directly invoked by or invokes the function of interest, as illustrated in Figure 7–9.

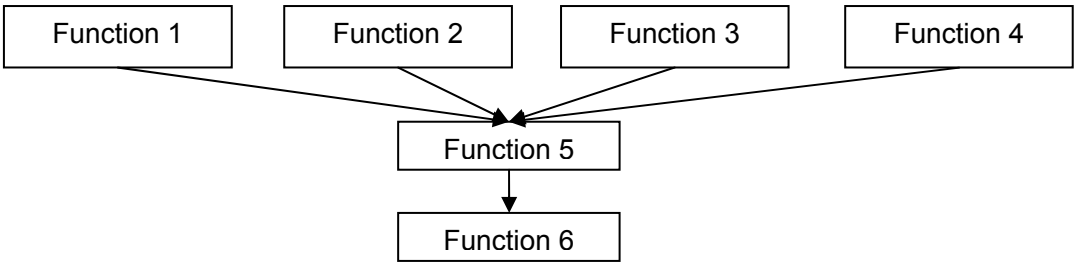


Figure 7-9 Cross referencing in IDA Pro®

In Figure 7–9 there are four cross-references *to* Function 5 and one cross reference *from* Function 5. For the purpose of this analysis, the cross reference count of a function is the greater of the two types of cross references (*to* and *from*). Based on the cross referencing feature, the functions with a large number of cross references are likely to be in the execution path. Additionally, the functions with a large number of cross references are likely to be called many times during the program execution. No assumptions are made about functions with a small number of cross references. These assumptions can be verified through experimentation. As the functions are likely to be invoked during program execution, the resources accessed during the function call can be identified and faults can be injected accordingly.

7.5. Development and Implementation

7.5.1. Dynamic Analysis

The sim-safe simulator in SimpleScalar [Burger 1999] was chosen as the fault injection system. The SimpleScaler family of simulators provides system developers with a virtual version of their target hardware. The virtual target hardware operates completely within a virtualized environment running on a standard laptop or desktop computer. The virtual hardware runs the same binary software as the physical target system, including firmware, device drivers, operating system, middleware stacks, and the application software. Software for the target machine runs unmodified on the virtual hardware. The configuration for a SimpleScalar execution is specified through a command line sequence of options that are to be applied to the target application.

Being an open-source tool, the entire source code of SimpleScalar was available for this research. Modifications were performed on the source code to enable a fault injection capability in SimpleScalar. Along with the default options provided by the toolset, the source code was modified to accept a few more arguments that specified fault and fault properties. When the instruction number specified at the command line was reached during the program execution, the fault value was injected into the resource, which was also specified in the command line.

The application to be analyzed was compiled for SimpleScalar and the fault free execution trace was obtained. Dynamic analysis was performed on this trace and the fault list was generated. This fault list was then parsed to a script generator that converted every entry in the fault list to an individual fault injection experiment to be performed on SimpleScalar. During the script generation process, the fault value to be injected was chosen. Fault injection experiments were conducted with two fault values, i) a value of 0 was injected into the fault location, and ii) a random bit in the fault location was flipped. Scripts were generated and approximately 28,000 fault injection experiments were conducted to determine the effectiveness of pre-fault injection analysis towards improving the efficiency of fault injection experiments.

7.5.2. Static Analysis

IDA Pro® was used to analyze the application to verify the postulates. The cross reference count of each function was determined and a few functions were selected based on the cross reference count. Functions with more than 5 cross-references were considered to have a high number of cross references whereas those with less than 5 cross-references were considered to have a low number of cross references. These functions were then checked for invocation and frequency of invocation by executing the functions on a debugger such as GDB (GNU Debugger).

GDB is an open source debugging tool that was used to prove the research assumptions. When debugging information is available in the specified binary code, GDB can be used to observe the variables in the stack frame at any instant of time. The target binary should be present in the search path of GDB. The application can be executed from GDB after the symbol file of the application is loaded into GDB.

7.6. Experimentation and Results

7.6.1. Dynamic Analysis

The application considered for analysis was *basicmath,* an arithmetic application that is used to solve cubic functions and also perform conversion from degrees to radians. This application can be obtained from the Mibench® benchmark suite [Guthaus 2001].

Fault lists generated from the dynamic analysis of the fault free execution traces were converted into scripts for fault injection on SimpleScalar. In order to compare the results of the fault injection experiments, random fault injection experiments were also performed for each fault value chosen. The results of the fault injection experiments are tabulated in Table 7-2.

Table 7-2 Results of fault injection experiments in SimpleScaler.

Fault Value	Fault Injection type	No. of faults resulting in faulty output	No. of faults resulting in correct output	Total No. of Fault Injection Experiments
Fault value 0	Random Fault Injection	289	4710	4999
	Fault injection with analysis	6326	7439	13765
Bit flip	Random Fault Injection	1138	3862	5000
	Fault Injection with analysis	2549	2451	5000

Table 7–2 shows the results of the fault injection experiments obtained by comparing the outputs of the fault injection experiments to the fault free output. It is observed that the number of activated faults for random fault injection is at least 5.78% of all the fault injection experiments

(289/4,999) while that of directed fault injection is at least 45.96% (6,326/13,765) when a fault value 0 is injected into all the locations. Fault injection experiments were also conducted by flipping the bits of registers in the register files. When these faults were injected randomly, only 22.76% (1,138/5,000) of the fault injection experiments resulted in faulty outputs. However, when the locations were obtained from analysis of the trace, nearly 50.98% (2,549/5,000) of the fault injection experiments resulted in erroneous outputs.

The number of faults that resulted in faulty outputs does not represent the number of activated faults because not all activated faults result in erroneous outputs. Sometimes, a faulty value can be propagated but may not affect the program behavior or output. Thus, the number of activated faults could be more than the number of faults that resulted in faulty outputs. These results are further illustrated by means of the bar graph in Figure 7–10 to compare results of fault injection experiments.

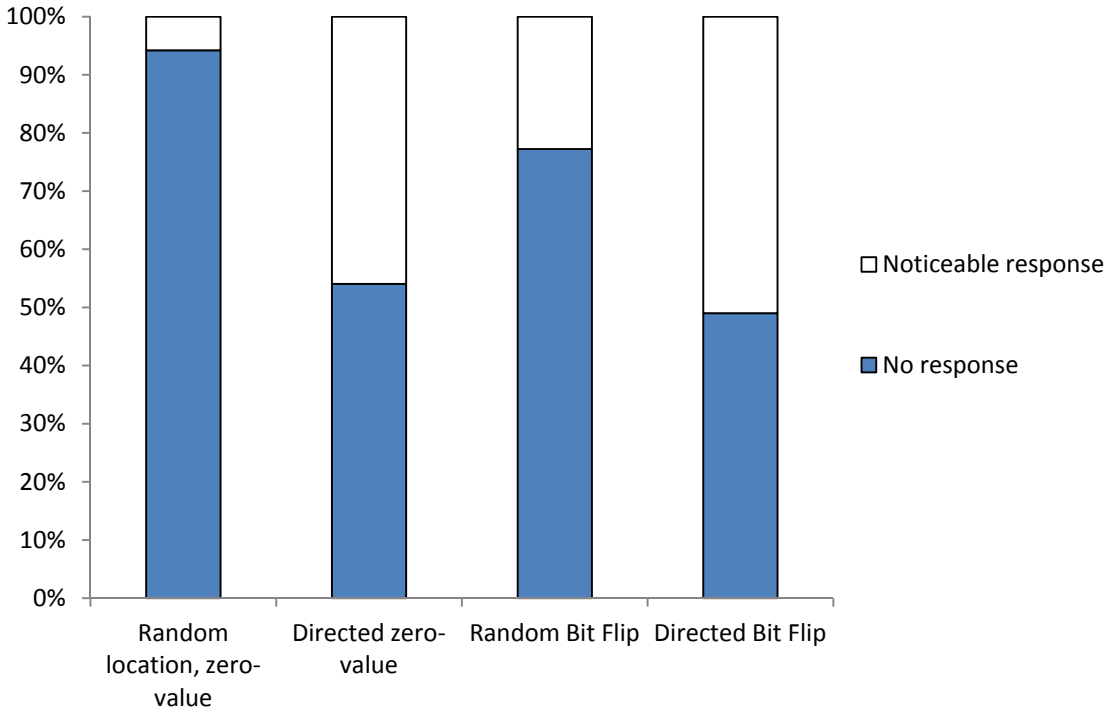


Figure 7-10 Results obtained from fault injection experiments

As shown by the results of the fault injection experiments in Table 7–2, a set of 5,000 random fault injection experiments conducted by flipping a random bit in a random location at a random time activated a fault into architecturally correct execution (ACE) bits in the register file in only 22.76 % of the experiments. However, when fault injection experiments were conducted with the help of trace analysis, 50.98% of the experiments injected faults into ACE bits. It was known that these are ACE bits in the register file because the output was erroneous. Thus, pre-fault injection analysis can give a better estimate of the ACE bits in the micro-architectural structure. Since, the experiments were not performed on a fault tolerant machine, there are no DUE bits. All the ACE bits thus become SDC bits. The results are shown in Figure 7-11.

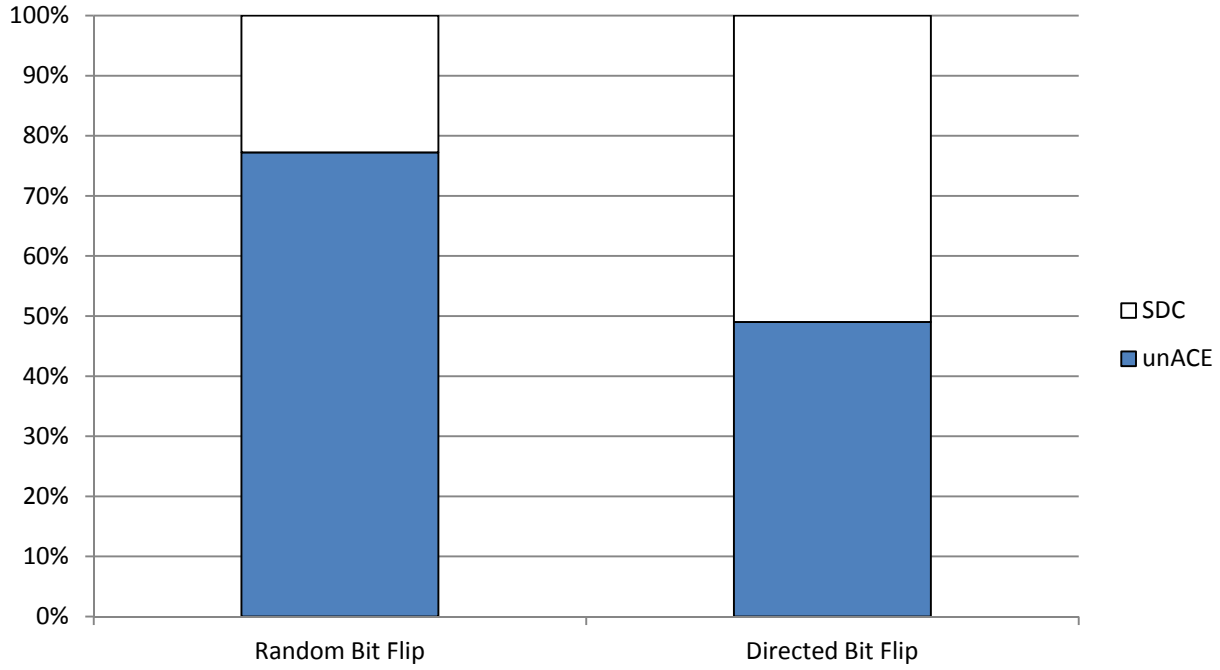


Figure 7-11 Comparison of ACE Bits obtained between random and directed fault injection experiments

7.6.2. Static Analysis

The target application is the ***ls*** command. The ***ls*** command is used to list the contents of the current working directory. There are many command line arguments that can be specified in the input to the *ls* command. The different inputs obtained for the experiment were based on the command line arguments. The target application was loaded in the debugger and executed. The static analysis verified the postulates, thus enabling justification of the choice of code regions that could be suitable candidates for fault injection.

The ***ls*** application consists of 256 functions. Out of these 256 functions, a large majority had only 1 cross reference. A total of 26 functions were identified that had more than 1 cross reference. Functions with more than 5 cross-references were considered functions with a high number of cross-references, and those with less than 5 cross-references were regarded as functions with a low number of cross references. The application was executed for a set of 8 different inputs. For each function identified, the number of times the function was invoked for each input was determined and plotted. The results are shown in Figure 7–12.

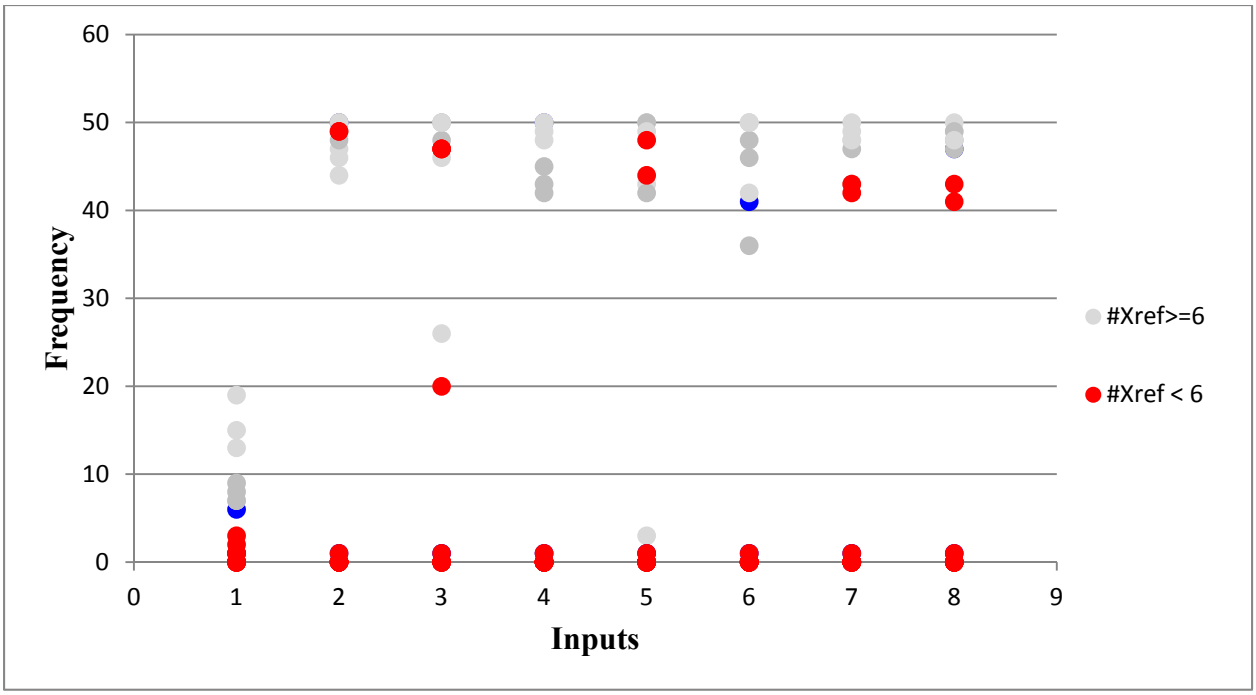


Figure 7-12 Frequency of invoked functions for each of eight inputs

It was also observed that functions with a higher number of cross references were likely to be present on multiple execution paths. The results are shown in Table 7–3. A tick (√) indicates that the particular function was invoked when the program was executed with the input specified in the column. An 'x' indicates otherwise.

It can be seen that the functions with a high number of cross-references are likely to be invoked. Thus, cross referencing provides a reliable means to identify functions or code regions that can occur in the program execution and thus represent suitable candidates for fault injection.

Table 7-3 Figure 7-12 function invocation and cross reference count.

Function name/Input	-lnRsa	-nprRsa	-Ra	-n	-pa	-ln	-rs	-nR	#Xref
decode_switches	√	√	√	√	√	√	√	√	>10
print_long_format	√	√	×	√	×	√	×	√	>10
human_readable	√	√	√	√	√	√	√	√	>10
Xmalloc	√	√	√	√	√	√	√	√	>10
gobble_file	√	√	√	√	√	√	√	√	>10
Nstrftime	√	√	×	√	×	√	×	√	8
Xstrdup	√	√	√	√	√	√	√	√	7
quote_name	√	√	√	√	√	√	√	√	7
close_stdout_status	√	√	√	√	√	√	√	√	6
Xstrcoll	√	√	√	√	√	√	√	√	6
hash_insert	√	√	√	×	×	×	×	√	5
hash_rehash	×	×	×	×	×	×	×	×	5
Usage	×	×	×	×	×	×	×	×	4
quotearg_colon	×	×	×	×	×	×	×	×	4
getuidbyname	×	×	×	×	×	×	×	×	4
quotearg_n_options	×	×	×	×	×	×	×	×	4
put_indicator	×	×	×	×	×	×	×	×	3

Table 7-3 Figure 7-12 function invocation and cross reference count.

Function name/Input	-lnRsa	-nprRsa	-Ra	-n	-pa	-ln	-rs	-nR	#Xref
length_of_file_frills	×	×	√	×	√	×	√	×	3
check_tuning	√	√	√	×	×	×	×	√	3
quote_n	×	×	×	×	×	×	×	×	3
quotearg_n_style	√	×	×	×	×	×	×	×	3
Mbsnwidth	×	×	×	×	×	×	×	×	3
queue_directory	√	√	√	√	√	√	√	√	2
print_name_and_frills	×	×	√	×	√	×	√	×	4

7.7. Applying Dynamic Pre-Fault Injection Analysis to the Benchmark System

The final step for this research task was to transition the pre-fault injection analysis algorithms to the CPU instruction set architecture of the benchmark systems. The process of this adaption is shown in Figure 7–13. To begin, the algorithms adapted to recognize the Instructions of the benchmark microprocessor, which was based on an x86 instruction set architecture (ISA). The x86 ISA mnemonics were loaded into the parsing table so each instruction could be recognized. Instructions that read and write to memory and register locations were tagged so that the algorithm could identify when these instructions occur in the execution trace.

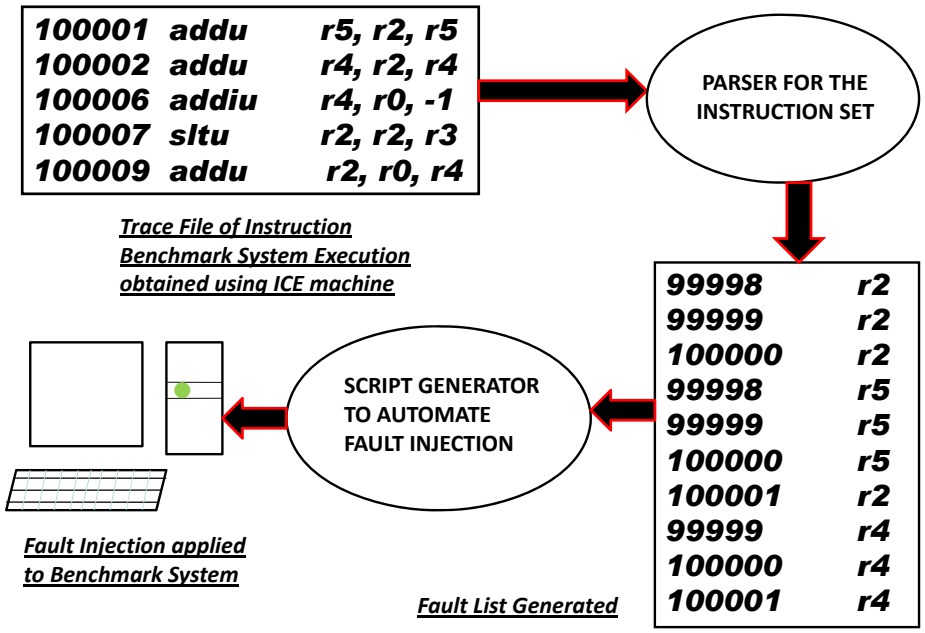


Figure 7-13 Integration of pre-fault injection analysis algorithms into Benchmark System I and execution trace files generated

Once the algorithms were converted to recognize the x86 instructions from the benchmark system, partial real time instruction traces were collected from the benchmark system using the HiTex DProbe Incircuit Emulator. There were limitations on how long the execution traces could be collected due to buffer size. Most of the traces collected were less than 10ms in duration, less than a full execution of the control cycle. Referring to Figure 7–13, the execution trace file from the benchmark system was input into the parsers to produce a fault injection list with windows of fault injection noted. These fault lists were then fed into another parser to generate a fault list in suitable format for the ICE based fault injector. Finally, the fault list was processed by UNIFI to inject faults into the benchmark system.

7.8. Results of Applying Pre-Injection Analysis to Benchmark System I

Applying the pre-analyzed fault injection list to the benchmark system required that a breakpoint be set in the fault injector for a specific memory or register reference at a specific time. This was easily accomplished using the HiTOP breakpoint commands.

The type of fault injected using the pre-injection analysis was a transient fault using the breakpoint-halt-modify-write-resume command sequence of the fault injector. When the fault injection campaigns were initiated it was noticed that this sequence of commands for fault injection was taking too long to execute and many of the fault injections caused the benchmark system watchdog timer to trip thus invalidating the fault injection result. A modification was attempted to change the sequence with breakpoint-halt-write. This was somewhat more productive; however, many of the fault injection trails caused the watchdog to trip. However, it was noticed that for almost every successful fault injection trial the outcome produced an error which was detected by the system self-tests. In a sense, this validated the simulation results and method from an observational point of view.

The ineffectiveness of the fault injections on the benchmark systems using the pre-fault injection analysis was not due to the pre-injection methodology, but due to the inability of the ICE based fault injector to support timely fault injections using sophisticated breakpoint control. This problem could be easily resolved with faster fault injection techniques such as OCD based fault injection.

7.9. Other Techniques: Map File-based Fault List Generation

Traditionally, physical-based fault injection has been performed by corrupting memory locations, register values, and memory mapped I/O with knowledge of data structures and values resident in these resources. The problem with this approach is that when something “interesting” occurred, it was difficult to trace the cause and effect back to the specific program variable or data structure that was responsible for the “interesting” result. This characteristic was not supportive of the assessment process. To address this challenge, a method of fault list generation was developed directed at determining all locations in the memory that corresponded to locations used by the application.

From the user perceptive, the safety application program contains variables, data structures, and functions used to implement the application. The user usually has some familiarity with the nature of these data structures as either the tester or the coder of the application. To develop a fault list, the map file for the application is obtained from the application compilation to determine all locations in the memory that correspond to locations used by the application and their associated symbols. The map file contains a listing of all variables and functions used by the application (see Figure 7–14).

UVA has developed a number of tools to extract all possible memory locations specified by their address and symbolic information from the map files of digital I&C systems. The process of fault list generation is now completely automated. The list of faults is then converted into fault injection scripts ready for the fault injector.

```
...
9000H   3DF9H  PUB  M_RECHNERNAME
9000H   3BC8H  PUB  REMPU1
9000H   0010H  PUB  STACK00
9000H   0610H  PUB  STACK03
9000H   3720H  PUB  STACK0F
```

Figure 7-14 Snip of a map file

7.10. Conclusions

A comparison with the conventional method of random fault injection using execution traces shows an increase in fault activation from ~6% to almost ~45% when a fault value of 0 is injected into selected locations. An improvement from ~23% to ~51% was observed when the injected fault is a flipped bit at the location selected. The faults used for fault injection were transient in nature.

Less than 100% fault activation is primarly due to the fact that some bit flip fault injections did not cause a significant error to be observed in the calculation of the output result. First, all errors induced do not produce observable failures. Approximately 50% of the errors induced into the system had negligble or no impact on the output of the system, even though the system state was corrupted. This implies that approximately 50% of the error corruptions for the given fault induced were of the un-ACE type.

Secondly, experiments were conducted on simulations of systems that had error detection features. The experimental “error” detection process was to observe the erroneous nature of the results of calculation at the output of the suimulated system. It is expected that system and processor level error detection mechanisms would detect many more errors, thus the number of un-ACE errors would go down depending on the effectiveness of the error detection mechanisms.

An interesting attribute of pre-fault injection analysis is its ablity to collapse the fault space of digital systems to those faults that are active with respect to the application domain. presented this aspect of the work in this report was briefly discussed, but it is clearly an important consideration for fault injection given that large numbers of fault injections may be required for ultra reliable systems. The prre-injection methods from a fault list collapsing perspective are being analyzed with the expectation of preliminary results in the future.

Static analysis conducted on disassembled binary code helps determine code regions that are suitable candidates for fault injection based on the structure of the control flow graph. The cross referencing utility in IDA Pro® was used to identify regions that were likely to occur on multiple execution paths and also functions that would be called many times during a single execution run of the program based on cross references as well for the ***ls*** application. Static analysis is intended to provide a starting point for fault selection when execution information may not be available.

For a bit to be declared ACE, the corresponding resource (memory location or register) must be first accessed. The resources that are accessed and the time duration during which fault activation in such resources can be achieved are obtained by analysis of the execution traces. A large number of ACE bits are contained in these resources. It was observed that fault injection experiments conducted on the bits of these resources resulted in a large number of erroneous outputs indicating that a large number of ACE bits had been uncovered in the process. Random fault injection experiments resulted in ~23% of 5,000 injected faults causing erroneous outputs thereby, revealing ACE bits. However, fault injection performed in selected resources at times obtained from analysis of execution information resulted in ~51% of 5,000 faults causing erroneous outputs. This means that of the 5,000 faults injected into different bits, approximately 50% of the bits were found to be ACE bits.

Thus, it has been shown that pre-fault injection analysis is a means of improving the effectiveness of fault injection experiments. It is also a method that can be used to determine a significant number of ACE bits in the micro-architectural structure for the purpose of evaluating the Architectural Vulnerability Factor (AVF).

7.11. References

[Barbosa 2005] Barbosa, R., Vintern, J., Fokesson, P., Karlsson, J. "Assembly-Level Pre-Injection Analysis for Improving Fault Injection Efficiency." *Lecture Notes in Computer Science, vol. 3463*, 2005: 246-262.

[Burger 2008] D. Burger, T. Austin. "The SimpleScalar Tool Set, Version 2.0." Technical Report, 1999.

[Eagle 2001] Eagle, C. *The IDA Pro Book.* San Francisco, CA: William Pollock, 2008.

[Guthaus 2001] Guthaus, M. "MiBench: A Free, Commercially Representative Embedded Benchmark Suite." Technical Report, 2001.

[Vinter 2005] J. Vinter, J. Aidemark, D. Skarin, R. Barbarosa, P. Folkesson, J. Karlsson. *An Overview of GOOFI a Generick Object-Oriented Fault Injection Framework.* Technical Report 05-07, Goteborg, Sweden: Chalmers University of Technology, 2005.

[Johnson 1989] Johnson, Barry W. "Design and Analysis of Fault Tolerant Digital Systems." *Design and Analysis of Fault Tolerant Digital Systems.* 1989.

[Sekhar 2009] M. Sekhar, C. Elks, R. Williams, B. Johnson. "Generating Fault Lists for Efficient Fault Injection into Processor Based I&C Systems." *6th International Topical Meeting on Nuclear Plant Instrumentation Control and Human Machine Interface Technology.* Knoxville, TN: NPIC&HMIT, 2009.

[Yuste 2003] P. Yuste, D. deAndres, L. Lemus, J. Serrano, P. Gil. "Inerte: Integrated Nexus-Based Real-Time Fault Injection Tool for Embedded Systems." *International Conference on Dependable Systems and Networks.* Sanf Francisco, CA, 2003. 669-669.

[smith 1995] Smith, D Todd, B Johnson, and J Profeta D. "A Fault-List Generation Algorithm for the Evaluation." 1995. 425-432.

[Tsai 1996] T.K. Tsai, R.K. Iyer, D. Jewitt. "An Approach Towards Benchmarking of Fault-Tolerant Commercial Systems." *26th International Symposium on Fault Tolerant Computing.* 1996. 314-323.

8. APPLICATION OF FAULT INJECTION TO BENCHMARK SYSTEM I: RESULTS

8.1. Introduction

After the UNIFI fault injection environment, operational profile generator, and RPS code activities were completed fault injection campaigns were conducted intermittently over several months on the benchmark system. These campaigns were conducted near the end of phase 2 of the project, and therefore the researchers were limited in the time they could conduct fault injection campaigns. Nonetheless, a significant number of fault injections were performed on the benchmark system. The campaigns were run in groups of experiments to access applicability factors of fault injection with respect to the capability of the digital I&C system to produce information that would support PRA assessment activities and produce information to support claims of system operation.

Overarching these objectives is the "process of discovery", that is the lessons learned way continuing up to the culmination of results. This Section presents the types of quantitative and qualitative lessons obtained by applying the fault injection-based dependability assessment methodology to the benchmark system.

It is important to note that the data and results presented in this Section are meant to be interpreted as the types of information that can be acquired from a fault injection-based methodology, and not an assessment of the capabilities of the benchmark system. The benchmark systems used as test platforms in this work were scaled representations of digital-based Reactor Protection Systems and thus do not encompass all of the features that are found in a typical digital-based RPS.

8.2. System Test Configuration

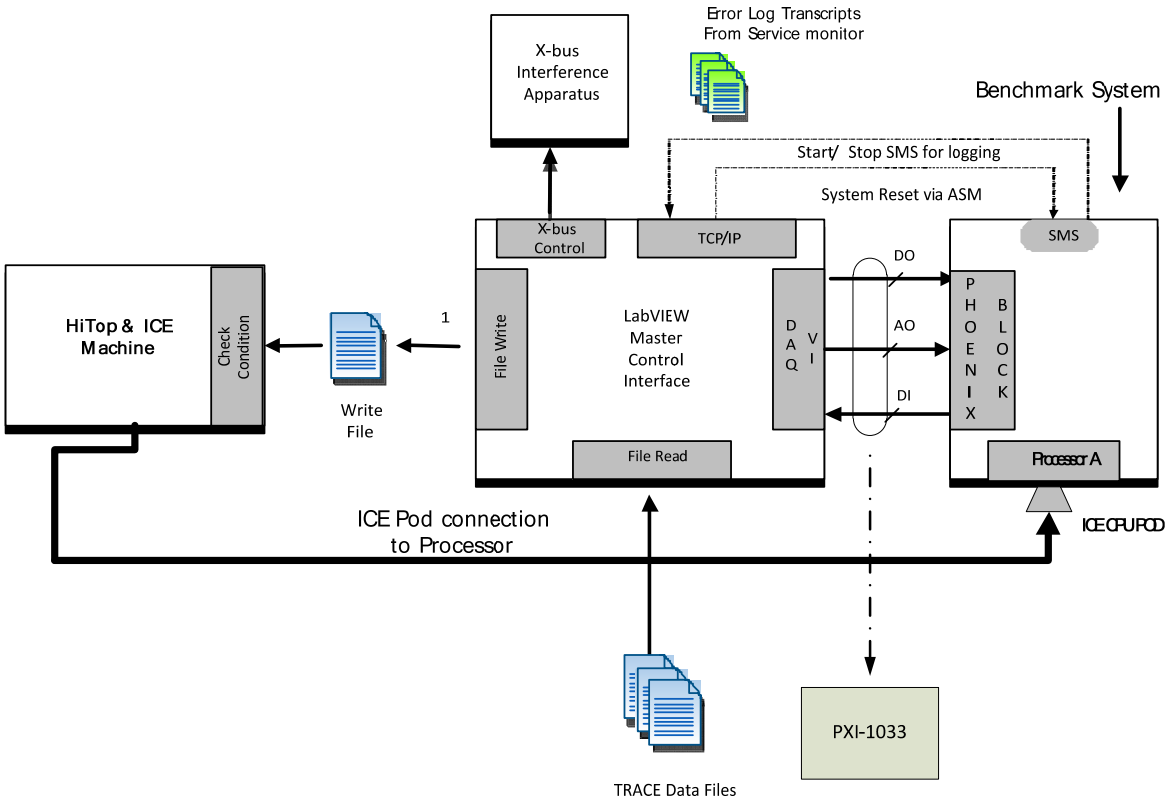
Referring to Figure 8–1, the benchmark system was configured from two perspectives. The first perspective involved configuring UNIFI and the associated support equipment to properly interface into the system for fault injection testing. Specifically, all of the benchmark system digital and analog sensor I/O were routed through the PXI-1033 data acquisition control module and controlled by the UNIFI/LabView fault injection environment. The analog input sensors included coolant flow, hot leg pressure, and steam generator pressure. The digital output signals included trip signals for each monitored sensor value; coolant flow trip, hot leg pressure trip, and steam generator trip.

The trip signals coming from the benchmark system were sampled every 10ms and recorded by UNIFI/LabView to indicate when they changed state. The ICE-based fault injector and the X-bus fault injector were interfaced into the system as previously described in Section 5.10. Recording system responses was accomplished by invoking the SMS service monitor unit through the TCP/IP connection socket from UNIFI. These responses were error log files for each fault injection trial. Each error log file contained the system response to the fault injection.

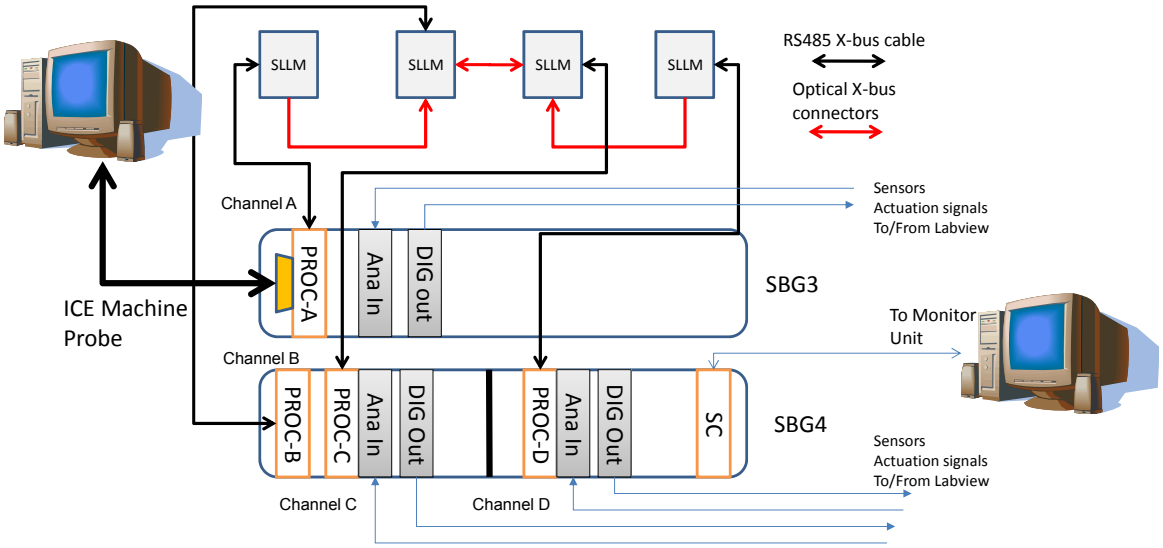
The TOP generator tool was run prior to an extensive fault injection campaign to provide an operational profile for the fault injection experiments. The same operational profile was used uniformly throughout all the fault injection experiments.

For a majority of the fault injection experiments, the benchmark system was configured as shown in Figure 8–1. The RPS I&C function was distributed over all 4 processing channels as

discussed in Section 3. Processor-based fault injection was applied to channel A of the RPS I&C function. X-bus fault injection was applied to the RS-485 electrical cable of channel A by way of the interposer adapter to the DIN connector of the X-bus SLLM unit.



(a)



(b)

Figure 8-1 Benchmark system configurations

8.3. Typical Fault Injection Sequence For Benchmark System I

After the UNIFI is properly configured and interfaced to a target digital I&C system, a fault injection campaign can proceed using the Master GUI. Figure 8–2, shows the typical automated operations that are performed by UNIFI master GUI for a single fault injection trial. Typically an experimenter will run hundreds of fault injection trials per campaign.

The sequence begins with resetting the target processor prior to a fault injection trial to ensure the system is in an error free state. Diagnostic routines and self-tests that are executed during system startup are the primary means to determine that error effects from previous fault injection trials have been purged from the system. The benchmark system has the capability to externally signal its health status to SMS service monitor. If the target computer signals to UNIFI by the way of the SMS service monitor that it is operational, and then UNIFI will proceed with the fault injection sequence; otherwise, UNIFI will abort the sequence with a NO-GO message to the user. From this point forward UNIFI will automatically run the fault injection sequence to completion.

Referring to Figure 8–2, the first few steps in the process are associated with initializing the measurement systems, and providing operational inputs to the target system. These operational inputs include enabling count down timers, setting the length of the censor time, initializing measurement instruments in UNIFI, sending operational profile sensor data to the target system through the I/O interface module, and establishing a connection to the benchmark system SMS service monitor.

The SMS service monitor observes the target system during operation and stores health and error messages from the benchmark system. These error messages form an error log of events during a fault injection trial. During this phase of the fault injection sequence the system is provided with steady state inputs from the TRACE operational profile generator. The next few steps in the process take care of the fault injection initialization processes.

The fault list is initialized and incremented to the current active fault to be injected and the fault injection parameters such fault type and fault mask are indexed and loaded into the fault list. One of the timers used in UNIFI is an operational profile countdown timer. When this timer reaches zero, the TRACE operational profile file initiates a plant event, in this case a LOCA.

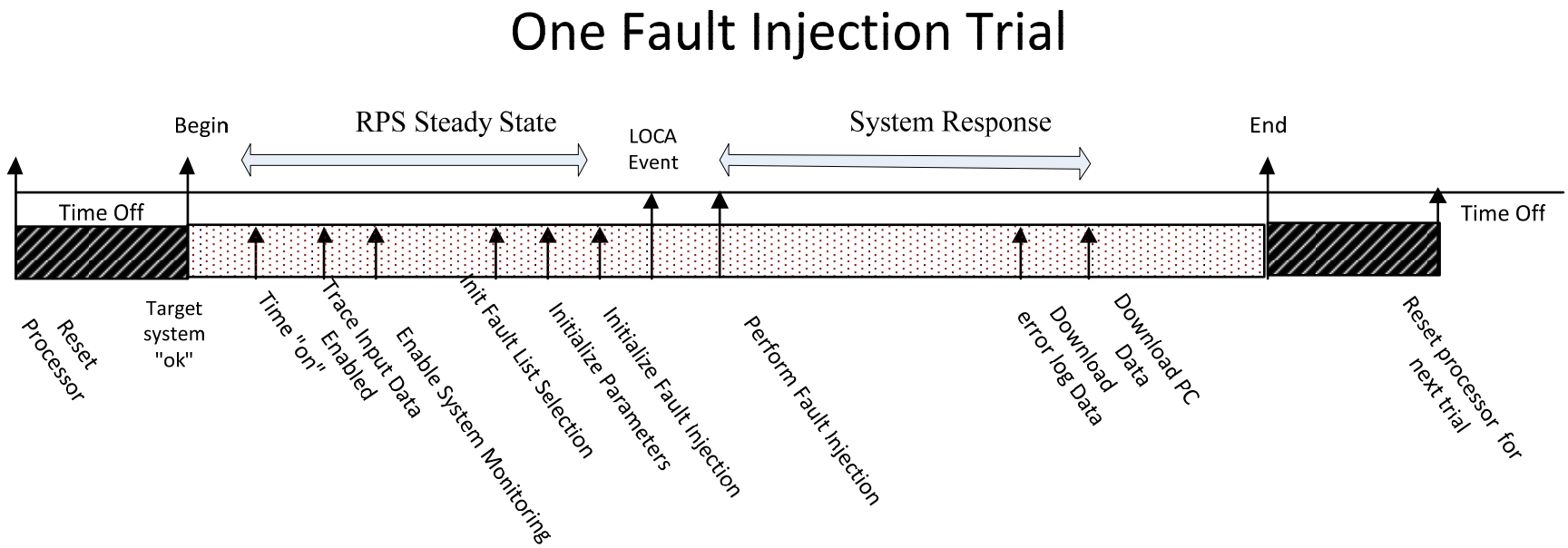


Figure 8-2 Fault Injection sequence for Benchmark System I

In addition to the operational profile timer, there are fault injection timers. These timers establish when the fault injection is to take place. A timer can be absolute (X seconds from the

start of the sequence), or relative (inject the fault after some event) or random. In the above timeline, the fault injection is triggered after the LOCA event is initiated. This scenario was used uniformly throughout the campaigns.

The benchmark system is allowed to run until the censor time of the experiment expires. This is set by the user during campaign set up. The censor time for the experiments was 12 seconds.

The last steps in the sequence download and store all of the recorded system response and error information in the database. Once all of the data is collected, UNIFI sends a reset to the benchmark system, and the fault injection process for the trial is completed. After the sequence is completed, UNIFI initializes all fault injection parameters to begin the next fault injection trial sequence.

8.4. Experiment Definition

A number of fault injection experiments that represented the types of fault injection tests that typically would be conducted by an assessment organization or the digital I&C equipment vendor during the course of a V&V activity were executed on the benchmark system. These experiments were chosen to provide a basis for determining the utility of the methodology to support system safety assessment activities (e.g., license reviews, failure modes and effects analysis (FMEA), and PRA activities). The quantitative results only reflect the ability to obtain such information. The experiments run on the benchmark system are summarized in Table 8-1. The experiment details and results are discussed in subsequent sections.

Table 8-1 Summary of experiments run on Benchmark System I.

Experiment	Purpose	Type of fault Injection	Fault model used	Fault Injection reference time point	Applied to Module “ ”	In support of verifying
CPU register Fault coverage	Provide data to quantify the error detection coverage of register faults	ICE based fault injection	Transient Single and multi-bit flips	After LOCA	Channel A main processing module	Self-tests and error detection mechanisms for processor based faults
Memory Fault coverage	Provide data to quantify the error detection coverage of memory faults	ICE based fault injection	Transient Single and multi-bit flips	After LOCA	Channel A main processing module	Self-tests and error detection mechanisms for processor based faults
Dual Port memory Fault Coverage	Provide data to quantify the error detection coverage of DP memory faults	ICE based fault injection	Transient Single and multi-bit flips	After LOCA	Channel A main processing module	Self-tests and error detection mechanisms for processor based faults
Pre-injection analysis verification	Provide data to verify the efficiency gains of pre-injection analysis	ICE based fault Injection	Transient – single bit memory	After LOCA	Channel A main processing module	Self-tests and error detection mechanisms for processor based faults

Table 8-1 Summary of experiments run on Benchmark System I.

Experiment	Purpose	Type of fault Injection	Fault model used	Fault Injection reference time point	Applied to Module " "	In support of verifying
Output Disable	Provide data that the digital output signals are disabled when a fault is detected	ICE based fault Injection	Transient – single bit memory	After LOCA	Channel A main processing module	Self-tests and error detection mechanisms for processor based faults
Digital Output Trip function timing response	Provide data that the digital output signals are actuated in a timely manner	ICE based fault Injection	Transient – single bit memory	After LOCA	Channel C and D main processing module	Actuation timing
Fault and error Latency analysis	Provide data to quantify the error detection latency of register and memory faults	ICE based fault Injection	Transient – single bit memory	After LOCA	Channel A main processing module	Self-tests and error detection mechanisms for processor based faults
X-bus token fault injection	Determine the X-bus controller token re-insertion times for X-bus	X-bus fault injection	Permanent and transient	No LOCA	X-bus network	Self-tests and error detection mechanisms for X-bus based faults
X-bus data message corruption	Provide data to quantify the error detection capability of corrupted X-bus messages	X-bus	Permanent	No LOCA	X-bus network	Self-tests and error detection mechanisms for X-bus based faults

8.5. Processor-based Fault Injection Experiments

8.5.1. CPU register corruptions

Fault Injection using the ICE-based fault injector was applied to various registers of the system Pentium I processor. These fault injections were transient single bit and multiple bit corruptions injected into the 16 bit and 32 bit registers of the register file. The first five registers are 32-bit general purpose registers. The second four registers are general purpose 16-bit registers. There are additional registers used in the Pentium architecture, however the fault injections were limited to this sub-set mainly for demonstration purposes because these registers are used predominantly by both the application and system software. Table 8–2 shows the details of the registers used in the fault injection experiment.

The fault injection experiments were conducted in the following manner. The selection of the register to be corrupted was randomly picked. This accounts for some of the variation seen in the fault injection totals for each register. The location of the fault in the register was selected at random as well. After these random selections were made, they were written to a file so the results of the fault injection could be traced.

All CPU register corruptions that were not of the “no-response” class were detected properly. Pre-fault injection analysis was not used for these experiments, thus accounting for the high number of no-response experiments (approximately 30% of the experiments produced no meaningful results). The criterion for “error detected or not detected” was that a valid error log was produced for the fault injection experiment. No-response experiments were those experiments that did not produce an error log at all and did not produce any other observable errors such as output deviations.

Table 8-2 Details of registers used in fault injection experiment.

Reg. Type	Single Bit fault transient	Multi-bit fault transient	Bit location In Register	No Response	Effectively Detected	Not detected	% Total detected
EAX	610	67	Random	230	677	0	100
EBX	478	80	Random	197	558	0	100
ECX	497	34	Random	163	531	0	100
EDX	340	0	Random	130	340	0	100
EBP	672	0	Random	272	672	0	100
AX	497	206	Random	238	708	0	100
BX	707	109	Random	271	816	0	100
CX	284	310	Random	125	594	0	100
DX	469	383	Random	104	879	0	100
Total	4554	1189	Random	1730	5743	0	100

8.5.2. Memory Based Fault Injection

Experiments were performed to determine whether the target I&C systems with one faulty channel would fail to initiate a reactor trip during a LOCA. In Benchmark System I, approximately 500 different locations in memory space (which included application and DP) were targeted for fault corruptions. These 500 fault locations were corrupted with single random bit flips and multiple random bit flips. Each campaign was conducted with a fault list of 500 faults. The campaigns were repeated and run at slightly different times to expose the target system to different fault activation intervals. Approximately 7,500 faults were injected for this main experiment in Benchmark System I.

No uncovered faults were revealed during the course of this experiment. The target system correctly actuated a reactor trip while handling a single channel fault in the target I&C system. The summary results of this experiment are shown in Table 8–3.

Since no uncovered faults were found, the method of variance reduction was applied for coverage estimation to provide a conservative bound on the coverage [Smith 1997]. This statistical method removes one of the covered faults from the campaign and marks it as uncovered for the purposes of computing a non-unity coverage bound.

Table 8-3 Summary of results from memory based fault injection experiments.

Location	Point Estimate of C	Variance of C	Confidence Bounds of C	Number of fault Injections	No response faults
Memory	0.999697	9.188×10^{-8}	0.9991<C<1.0	5274	1975
DP memory	0.9998	1.39×10^{-6}	0.9965<C<1.0	1470	622

8.6. Pre-fault Injection Analysis Verification

As stated in Section 7.7, the successful application of pre-fault injection analysis methods to the benchmark system was impeded by the long time delay associated with the *breakpoint-halt-read-modify-write* fault injection operation of the ICE-based fault injector. Nonetheless, a very small set of data was available for comparative analysis to non-pre fault injection results. In this case, a memory location that stores the variable for the "coolant flow" set-point was corrupted with both methods.

There were 22 successful pre-fault injection fault injections, meaning the benchmark system watchdog timer did not trip due to the lengthy time of the fault injection process. Of these 22 fault injection experiments, 21 fault injections resulted in a detectable error (i.e., a valid error log file was produced). This is a 95% effectiveness rate. By comparison, the same location in memory was fault injected 12 times with no regard to the window of opportunity for when to inject the fault to propagate an error. Out of these 12 fault injections, 7 of the fault injections resulted in no response faults. This is a 42% effectiveness rate. While these results are meager, they tend to confirm the value of using pre-fault injection analysis.

8.7. Digital Output Response and Output Disabled Experiments

Experiments were performed to test the capability of the methodology and the UNIFI measurement systems to gather critical timing information about the actuation and disengagement capabilities of the system. The purpose of the experiments was to detect whether the outputs were disabled after a fault injection that was detected by the benchmark system. The faults injected into the processing module were register based faults in which 250 transient single bit faults were applied to various registers of the CPU.

There are several metrics of interest with the experiments. The first is output disengagement coverage. That is, after detecting a fault in the processor, does the benchmark system disengage the outputs? The second metric of interest is the time delay or transport lag associated with the disengagement signal.

The disengagement measurement was made with the LabView sample and record instrument. The time of the fault injection was known by way of a time-stamp from UNIFI, which was accurate to a resolution of 1 ms. The fault injection triggered the measurement counter to begin counting in 10 ms increments. The outputs of the benchmark system were sampled every 10 ms. State changes from 1 to 0 indicated disengagement. These types of experiments allow the real-time output response of the I&C function to be measured in a no-fault or faulted case.

Another experiment measured the time from when the set point of a signal goes high until a reactor trip signal goes high at the output. This time measurement was facilitated by feeding a set-point signal to the digital output of the benchmark system and comparing it with the reactor trip signal going high. This was conducted in a no-fault scenario. The outputs of the benchmark system were sampled at 1 ms intervals.

The results of the experiments are summarized in Table 8-4.

Table 8-4 Results from UNIFI experiments to detect CPU register faults

Experiment	Single Bit Fault Transients	Bit Location in Register	Error Detected	Error Not Detected	No Response	Output Disengage-ment	Mean Time of Disengage-ment	Output Actuation Time
Output Disengage-ments Faulted	250	Random	156	0	94	156	17ms +/- 10ms	1 to 2 ms

8.8. Fault and Error Latency Analysis

Another important metric often used in dependability analysis is error and fault latency. Figure 8-3 shows the concept of fault and error latency. Upon the occurrence of a fault the fault may remain latent until it is activated by use in the program or hardware. Once it becomes active it may produce an error that propagates until it is detected by the error detection mechanisms or self-test functions of the system. Fault latency is important because the longer an error or fault remains undetected in the system, the higher the probability the undetected fault will collude with another fault in the system. This could result in two errors manifesting in the same time interval thereby requiring the system to handle a double fault situation.

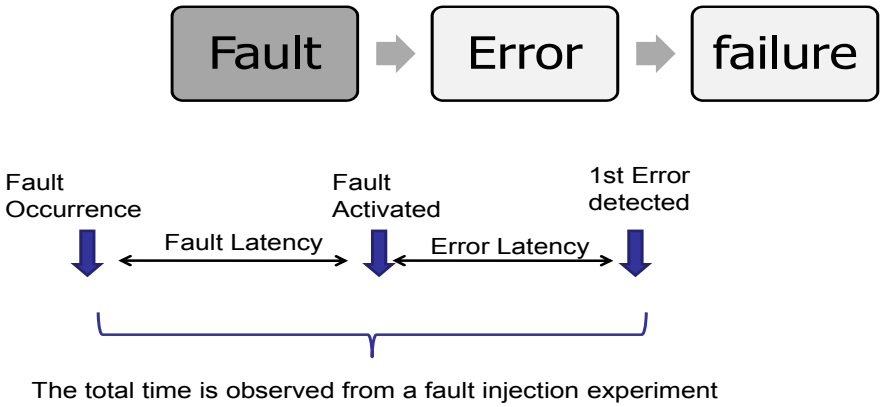


Figure 8-3 Example of fault and error latency

The fault injection experiments allow the total time to be measured with assistance from the time-stamps on the error messages recorded in the error logs. Error logs from the benchmark system are essential for determining when the system detects a fault and when the system responds to the fault. In Benchmark System I the SMS service unit collected the error messages from the benchmark system error detection mechanisms and self-tests. The SMS time-stamped these error messages as it received them. It was assumed in the latency analysis

that the transport time of the error messages from the benchmark system to the SMS server unit was minimal and fixed. In a more detailed measurement set-up, the time delay transport could be accounted for by measuring this delay. Figure 8–4 shows an error log transcript from the SMS server after a fault injection experiment.

```
Service Monitor Server- Version 1.12 / 2003-01-22
Benchmark System,© 1993-2002,
sms (info): Loading hardware diagrams...
sms (info): Loading software specification...
sms (info 65003): 20 messages have been loaded from database 'rpsver2'.
sms (info): SMS started. Date:2009-01-07 19:17:40
2009-01-07 19:17:43.177  state (up / not available):
  CPU 111    state: up since 2009-01-07 19:17:17.427 [25 s]
```

Beginning of SMS logging

Example RTE error

Timestamp

```
...
2009-01-07 19:26:30.979 CPU 124
RTE error 605, incoming, location RECV_DIRECT, cycle 3939483 7323
Outdated RTE message (message ID 4).
(SIGNALLING-)RTE msg 4 leads from CPU 111 to CPU 124.
...
```

Figure 8-4 Error log transcript from the SMS server

For each fault injection that results in a detected fault, the time of the first occurrence of an error message response is noted. A few important points with respect to the measurement of the fault latency are discussed in this section.

The time measured for latency includes the transport time of the error message from the benchmark system to the service unit server where it is time-stamped. Therefore, the true fault/error latency is less than what is shown in the graphs. This transport time from benchmark system to service unit server is less than 10 ms given the communication is over a direct Ethernet link. Another consideration that comes into play is that self-tests and error messages are processed at the lowest priority of the RTE, which is level 3. This feature of the benchmark system can further delay the error messaging of the injected fault. That is, an error may have been detected and mitigated, but the message indicating the error could be pre-empted by the system cyclic processing until there is idle time to transport the error message. Thus, the end-to-end latency measurement is given as:

$$End\ to\ end\ Latency = T_{fault} + T_{error_{detect}} + T_{error_{messgage}} + T_{Transport} \quad (8.1)$$

where

T_{fault} is the latency of the fault activation,
$T_{error\ detect}$ is the error detection latency,
$T_{error\ message}$ is the latency of the error message, and
$T_{transport}$ is the time to transport the error message to the service unit.

The variables T_{fault}, $T_{error\ detect}$, and $T_{error\ message}$ are variable or random in nature, and the constant $T_{transport}$ is a fixed but unknown time.

The variable nature of T_{fault} is due to the memory location containing the corrupted process variable may be accessed at slightly different times depending on when the fault was injected with respect to the program execution flow. Thus the fault activation is function of when the program accesses the memory location.

The value of $T_{error\ detect}$ can be variable due to a number of factors. One factor depends on the error detection mechanism (EDM) that was activated by the fault. If the EDM was part of the cyclic processing (highest priority) then the error would be detected quickly. Cyclic processing EDMs would be functions like 2-out-of-4 voting with error signaling, a set point trip with error signaling, etc. Secondly, if the error was detected by background self-tests, then the response may be delayed due to the priority of the cyclic processing I&C functions. Lastly, if the error was detected by self-tests running in background mode, there is a delay associated with when the self-test is executed in the background task scheduler. Background tasks are queued in a task list for the background scheduler to execute. There are many self-tests that are executed by the background task scheduler; therefore,, it may take some time for a particular self-test to become active and that self-test to execute its functions to detect the fault.

The value of $T_{error\ message}$ can be variable if the EDM that signaled an error is a self-test running in background mode. In this case, the error message may be delayed until cyclic processing finishes and all service commands with a higher priority are processed.

Given the nature of the latency measurements, for all measurements the time of the fault injection is differenced from the first time of error detection as noted on the error message timestamp to yield the end-to-end fault/error latency of the response. Figure 8-5 shows the sorted scatter plot of the fault latency responses. The x-axis is the fault injection experiment conducted with respect to the program and system variables that were corrupted. That is, each fault injection experiment performed one corruption on a single variable in memory space. For the 500+ variables selected, multiple independent fault injection trials were repeated for each variable on each response. The Y-axis is the end-to-end latency time.

Referring to Figure 8-5, the scatter plot reveals two distinct “bands” of time responses. One band is located below the 2,000 ms fault latency line, the other is centered around the 10,000 ms fault latency line. At first glance, the unusual shape of the scatter plot would suggest that the latency may be associated with a particular program variable.

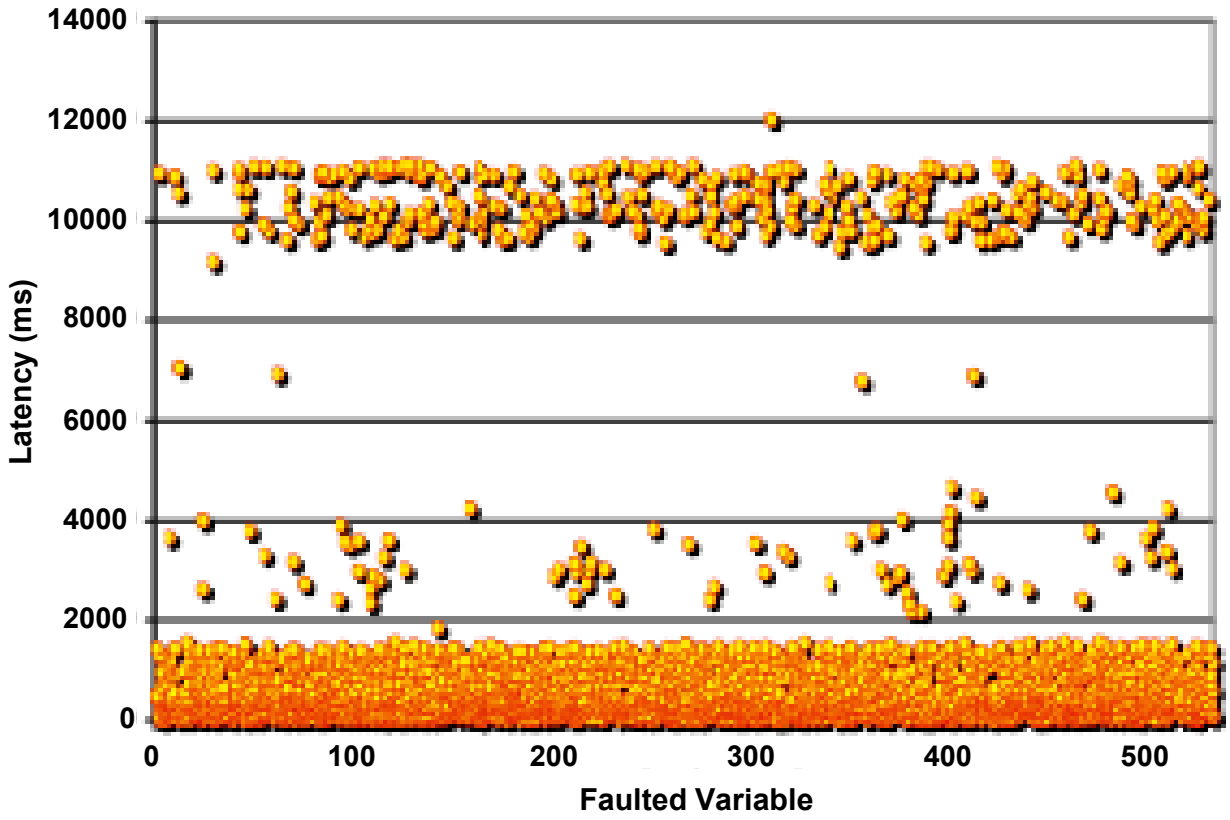


Figure 8-5 Fault latency of memory fault injections

However, in the data shown in Figure 8-6 for the first 50 fault injections, the same program variable is in both response bands. In one fault injection experiment it was detected rather quickly (less than 1,000 ms), in another it took almost 7,000 ms to detect. This feature was noted in most of the bimodal data sets. From this observation it was concluded that the bimodal nature of the latency is not entirely due to the characteristics of a particular set of program variables and their use by the RPS application.

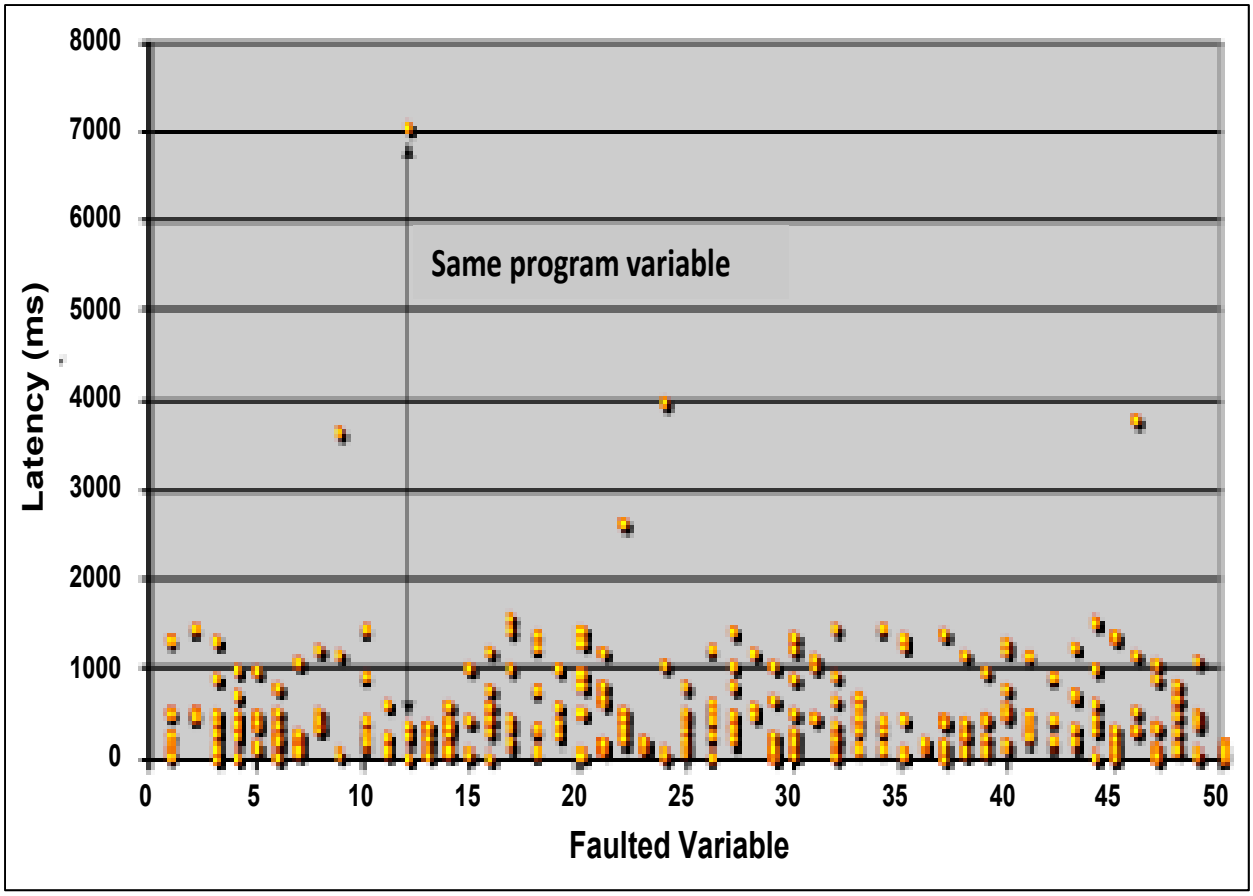


Figure 8-6 First 50 variable locations of the memory fault injection campaign

Figure 8-7 shows the empirical distribution of the fault latency. Here, the bimodal nature of the empirical distribution is more clearly visible. For most of the injected faults, the error detection mechanisms and/or self-tests detected these faults in 1,500 ms or less. However, for a small subset (less than 10%) the error detection mechanisms/self-test required on average about 10,000 ms to detect the faults; hence, the shape of the distribution.

Figure 8-8 shows the cumulative distribution of the fault latencies plotted in increasing order. As shown in Figure 8-8, 88 per cent of the fault responses occur in less than 1,600 ms.

Without further detailed experiments to obtain measured data sets on $T_{error\ detect}$ and $T_{error\ message}$ characterize the nature of bimodal distribution cannot be definitively characterized. Obtaining $T_{error\ detect}$ would entail instrumenting the system so that when a self-test or error detection mechanism is triggered by a fault the output of the EDM (e.g. the fault detection predicate) is time-stamped. Presently, the researchers do not have access to the outputs of the self-test functions (their fault detection predicates). This type of instrumentation is possible with vendor support. The same type of set-up is needed for the error message activation and sending.

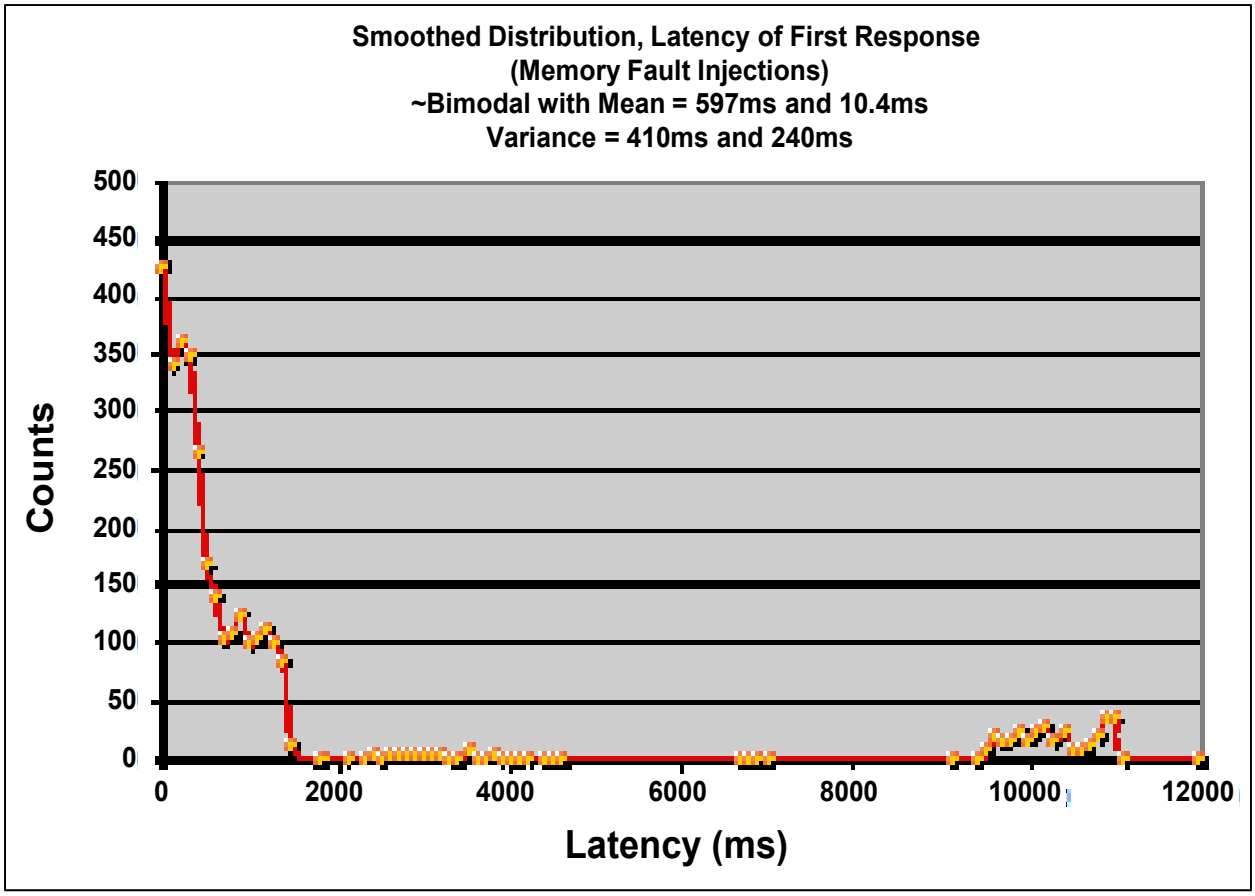


Figure 8-7 Distribution of memory-based fault latency

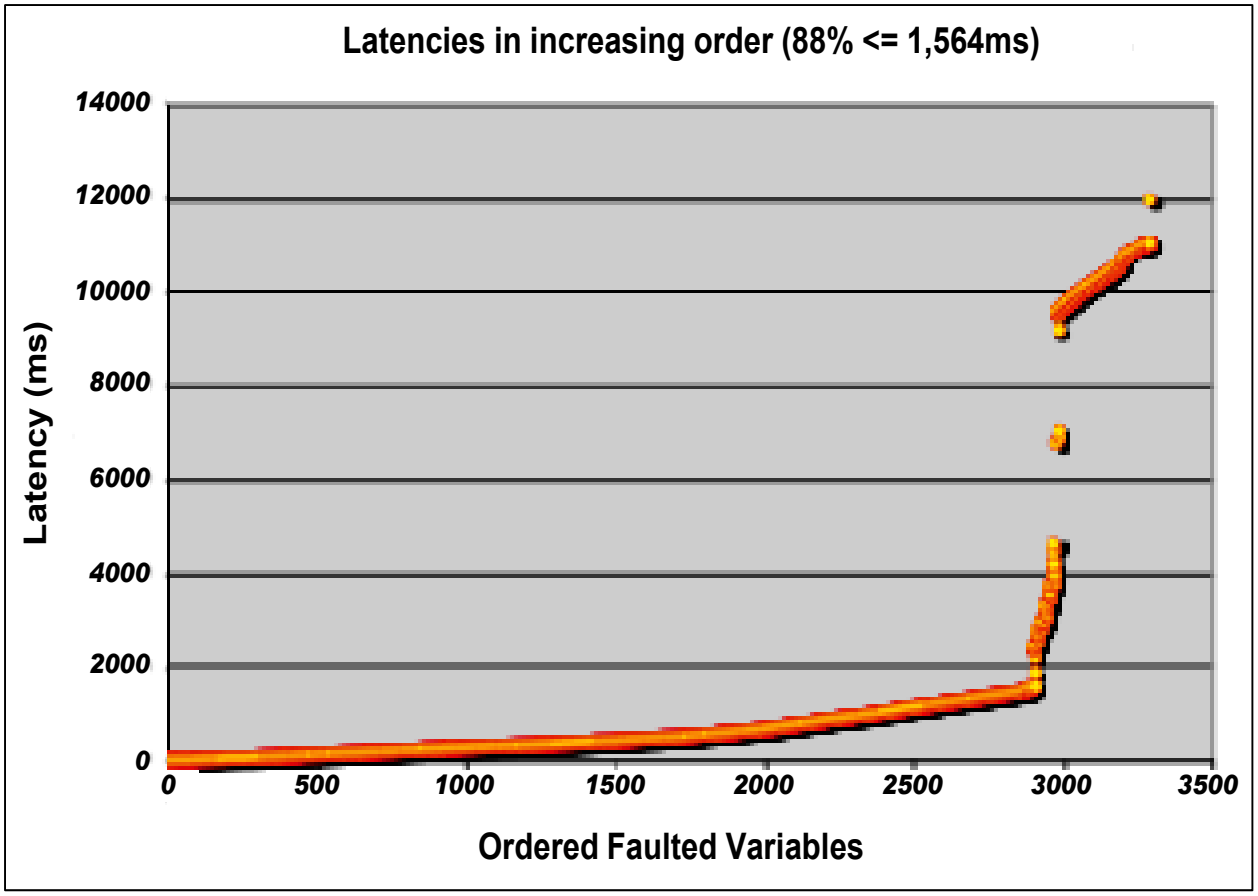


Figure 8-8 Cumulative fault latency distribution

Another set of data that would have been very useful for this experiment is the the output disengagement data. Specifically, for the long latency faults, an analysis using this data could determine whether the outputs of the benchmark system disengaged. Unfortunately, this data set was not taken in the experiments due to an oversight of the experiment set up. By the time the oversight was discovered, the benchmark system had been disassembled for subsequent use by another organization. The output disengage functionality is easily measured on the benchmark system, so there is no technical barrier to such measurements in a fault injection experiment.

A reasonable suggestion for the bi-modal shape of the latency data shown in Figure 8-5 thru Figure 8-7 is that self-test phasing or scheduling along with the workload dependent detection are the principle causes. This type of behavior is not atypical for distributed real-time systems. The principle value of bimodal latency analysis to the vendor and/or regulator is to verify that self-tests are effective in detecting the faults they are expected to detect and the detection times of those faults can be empirically measured to see how they compare with the analytic fault detection time bounds. In addition, the fault coverage estimates of the self-testing functions can be deduced from the data sets to determine error detection coverage and real-time error response coverage. Error response coverage refers to detecting an error within a specific time bound; if the system does not detect the error within a specific period, then error detection response is said to be defective or non-compliant.

Fault/error latency in the processor register space was more typical of what might be expected. The distribution shown in Figure 8-9 is a scatter plot of the fault latency data obtained from the

CPU register fault injection experiments. The fault latency appears to be nearly uniform from 500 ms to 1,500 ms.

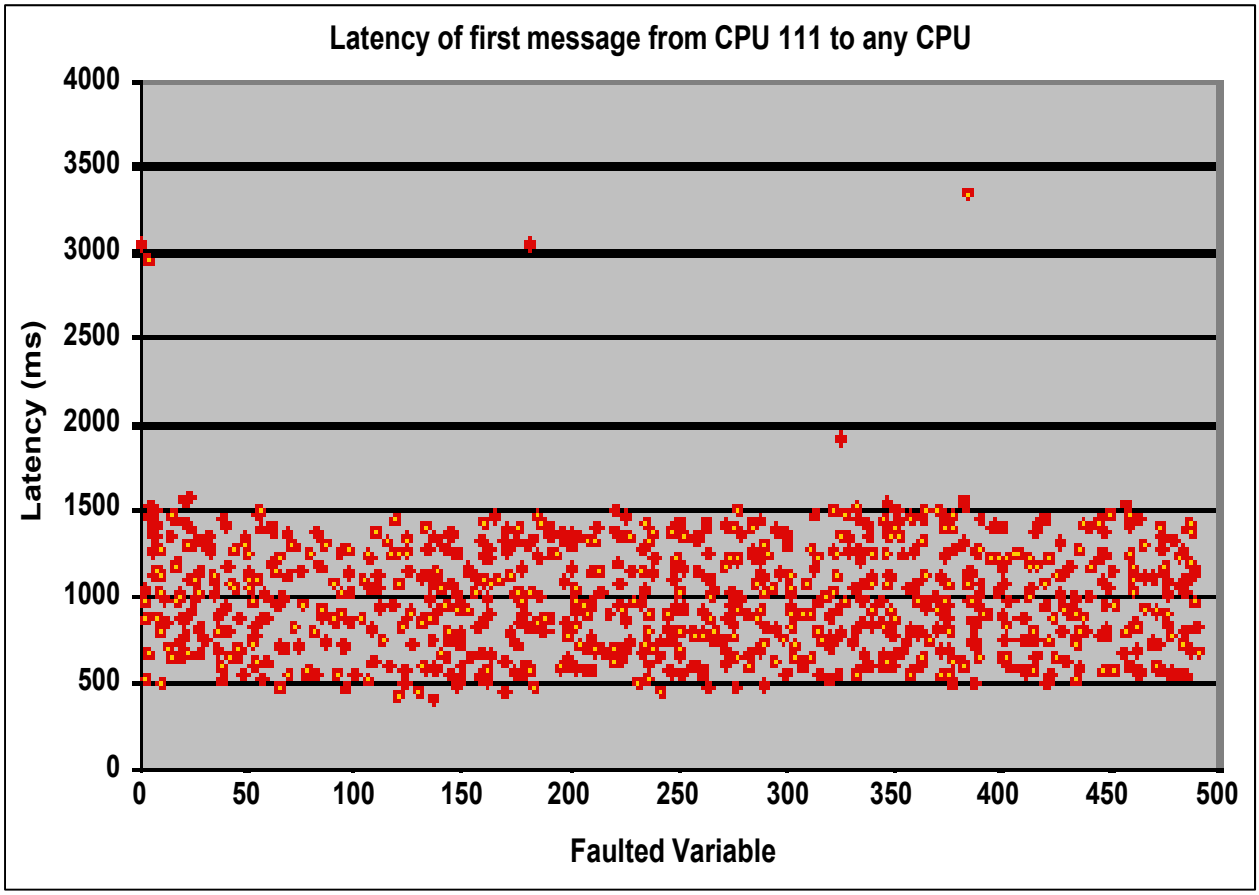


Figure 8-9 Fault latency of register-based fault injections

The empirical distribution of the fault latency data shown in Figure 8-9 is shown in Figure 8-10. The distribution is approximately Gaussian in shape, with a mean of ~1,000 ms. Faults that occur in registers of the CPU are generally expected to be detected quickly since the probability of using the register with the corrupted information is relatively high. This is especially true of the Pentium I processor, which is known to have a very small register set compared to its instruction set. The distribution of the fault latency data appears to confirm this assumption.

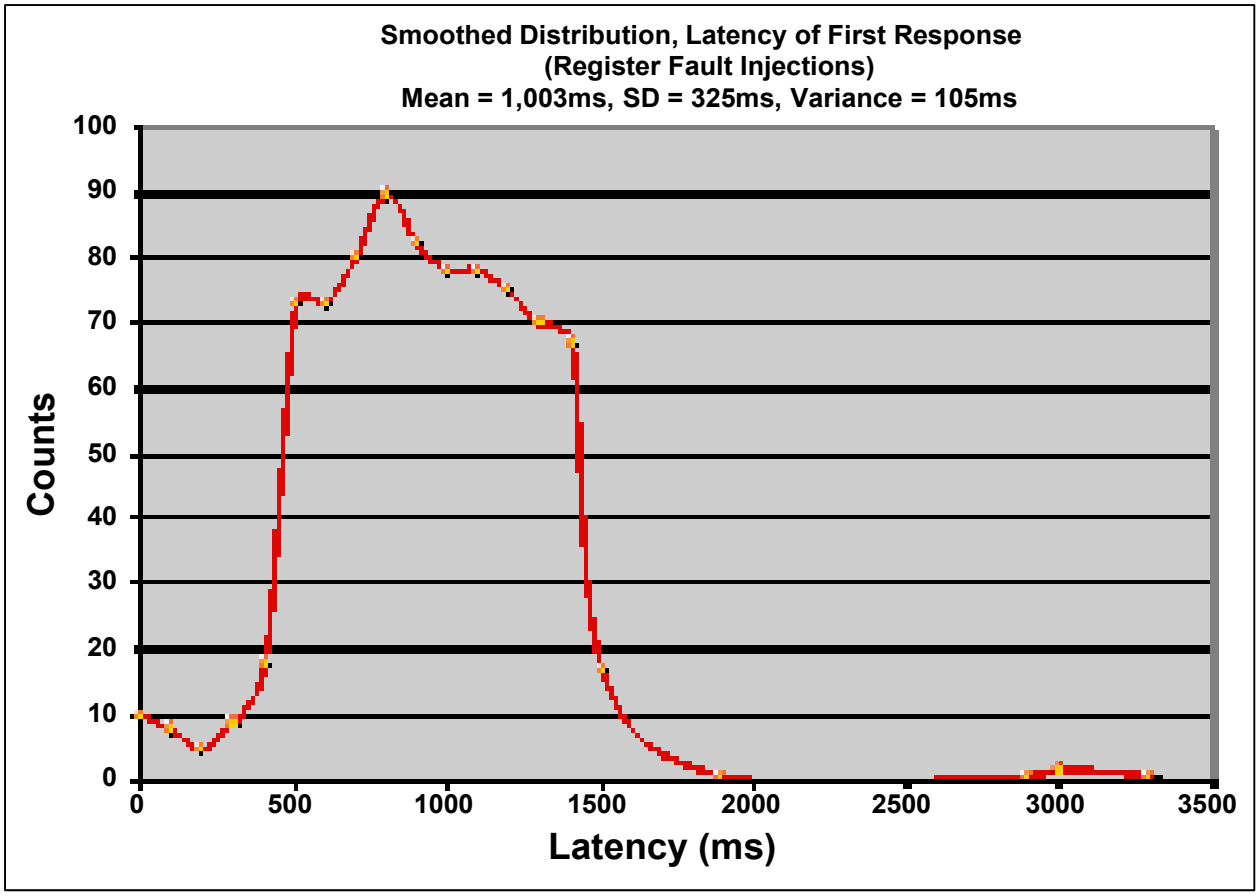


Figure 8-10 Distribution of register-based fault latency

Another important set of information is the “crash” (i.e., halt) behavior and latency of the fault injected processor. Figure 8-11 shows the fault/error latency of fault injections that resulted in microprocessor halts. There were very few instances of halts observed (less than 50). However, when a processor did halt it was usually due to the sensitive location of where the fault was injected (e.g., in an operating system variable or process stack space).

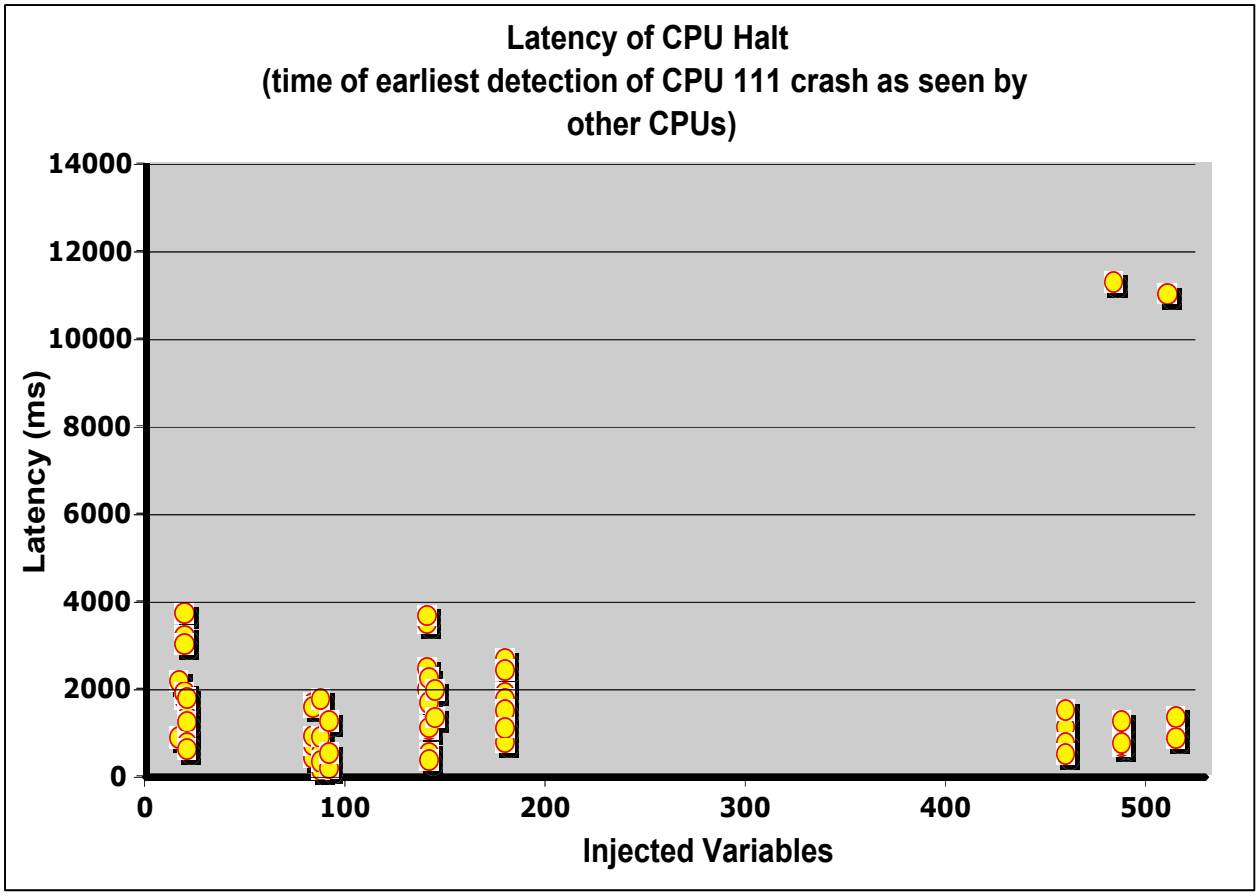


Figure 8-11 Halt latency of injected processor

8.9. Addressing No-response Faults

In the fault injection experimentation the difference between no-response faults and long latency faults could not be uniquely identified due to the time constraints of the research. However, in the real world and to adhere to NRC single failure criteria testability, no-response faults need to be truly distinguished from long latency faults or hidden faults. The pre-injection analysis presented in Section 7 is one means to identify no-response faults and distinguish them from long latency faults. Below is a suggested methodology to effectively deal with no-response faults.

(1) Classify faults that result in a "no-response" as "further experiments needed".

(2) Group no-response faults into classes according to their function - Safety block functions, registers, special purpose IC registers, OS, etc.

(3) Conduct pre-injection analysis to ensure the fault is in a part of the system that has executable code and data.

(4) Re-perform each no-response fault injection experiment according to the following process:

    (1) Repeat the fault injection with same fault injection parameters, but extend the observation time by increments of 5X.

(2) For all faults detected thru extended monitoring, identify how long the latency was. Determine if these long latencies have impact on reliability by entering them nto the PRA models. If so, then those faults with reliability impact should be reassessed.

(3) If no response is observed after three successive FI experiments with increasing monitoring (e.g., the last FI monitoring period is 30 minutes) then classify fault as "very long latency".

(4) Classify the "very long latency" faults as potentially dangerous faults.

(5) Alter the FI parameters of the very long latency faults - time, duration, and value – parametrically to determine sensitivity to these parameters.

(6) Execute different input profiles (trace events and data) to determine sensitivity to the input and event space.

If faults still exist that produce no response, the analyst should try to identify why the fault is latent/non-responsive. This may entail the use of dynamic and static analysis of the RPS code to determine why the fault is not detectable. If the fault is in an area of the system code/data, parameter space then, by definition, it is a latent fault. Very long latency, undetected faults require complete analysis to determine why they are not detected, a complete analysis of the faults, and a corrective plan of action. The severity of all fault Injections should be classified by their observed responses. The classification should include latency, detection effectiveness, and the system time response.

8.10. X-Bus Fault Injections

8.10.1. Introduction

After completing the processor and memory based fault injection, attention was turned to injecting faults on the X-bus inter-processor communication network using the X-bus fault injector developed and described in Section 4. Two types of fault injections were performed: token corruptions and data message corruptions. Both types of corruptions are discussed in the following sections.

8.10.2. Token Fault Injection

The idea behind token-based fault injection was to corrupt the Token Header PTP in order to make the message appear to be unreadable to the destination station. To accomplish this, it was essential to understand the structure of a Token message (see Figure 8-12), in which SD is the Starting Delimiter, DA is the Destination Address, SA is the Start Address, and ED is the Ending Delimiter.

Figure 8-12 Structure of a token message

It was determined that the simplest route to disable the readability of the Token message was to alter the SD PTP to modify the bit pattern that the destination station expected. This ensures

that the destination station discards the message, unknowing that the source station actually transmitted the Token DLPDU correctly and expects the destination station to take ownership of the X-bus logical ring.

The data represented by each PTP is known after reading the first nine bits of the PTP, the start bit followed by the eight data bits. Therefore, the FPGA is capable of recognizing the message type based on the SD PTP well ahead of the transmission of the last bit of the current PTP. This provided reasonable time to execute the correction in real-time without unwillingly modifying any additional bits. The data value of the X-bus Token SD is 0xDC, which is 11011100 in binary. However, the most significant bit in the X-bus data ordering system is the rightmost bit, hence the traffic data bit stream was observed by the fault injector as 00111011. Including the first start bit, an even parity bit, and the stop bit, the resulting PTP is described as the following bit sequence: 00011101111.

Based on the physical limitations of interfering with the X-bus signals, this sequence presents the possibility to alter the last two bits, modifying them from high to low voltage, thus making the SD PTP unreadable for the other stations on the X-bus network because of an incorrect parity and stop bit. As mentioned earlier, if the SD of an X-bus DLPDU is corrupted, the receiving stations are unable to decode it and must discard it. However, the transmitting station has no information regarding this problem and it expects a normal continuation of traffic (i.e., the receiving station taking ownership of the logical ring by transmitting its Data Messages or passing the Token to the next station). Because of the corruption of the Token SD, this scenario cannot occur and the station that is attempting to pass the Token must wait for a significant period of time (T_{sl}) before it can resend the Token. Therefore, it is crucial to quantify these delays and observe the responses of the target real-time safety-critical system under test.

8.10.3. Data Message Fault Injection

The foundation of executing a correct Data Message corruption is similar to corrupting a Token; it is necessary to correctly recognize the type of message and to supply an interference signal that affects one or more specific bits of the message, hence creating an altered message for the destination station. However, this fault injection is not supposed to corrupt the Data Message but rather to modify it. When using this approach, the destination station is unable to detect any irregularities in any of the PTPs of the message. By looking at the structure of the Data Message illustrated in Figure 8-13, it is obvious that its structure is very similar to the structure of a Token message. In addition to the fields present in a Token message, a Data Message includes PTPs containing the actual data information, an LE (Message Length) PTP, an FC (Function Code) PTP, and an FC (Frame Check Sequence) PTP.

Figure 8-13 Structure of a variable length data message

To create the desired effect described earlier, the main focus of the Data Message corruption is on the two repeating, consecutive LE PTPs. The strategy to correctly deceive the destination station is to modify both PTPs containing information regarding the length of the Data Message in the same exact way. Therefore, the receiving station would not be able to recognize any inaccuracy of the corrupted Data Message. To accomplish this sophisticated corruption, two data bits of each PTP must be corrupted in order to preserve the correct parity of the LE PTP. If the fault injection is performed in this manner, both of the length fields will contain the same value and the parity will remain correct.

The specific implementation of X-bus on the tested system (Benchmark System I) only utilized data messages of length 249 (11111001 in binary) and 237 (11101101 in binary). Translating this in a complete PTP block with the correct bit ordering, the actual bit streams observed by the fault injector were the following:

Length 237 bits – 01011011101
Length 249 bits – 01001111101

The simplest means of corrupting two bits within the same data field is to provide interference to two consecutive bits, essentially just by holding the corruption signal high for a longer period of time. By analyzing the two different bit streams, it is clear that the best opportunity for performing this complicated fault injection is to corrupt the 7th and 8th bit of the PTP. This would change the data fields of the two different lengths to 10110001 (141) vs. 10110111 (237) and 10011001 (153) vs. 10011111 (249), respectively. If executed correctly, this should deceive the receiving station and make it recognize the received Data Message to be shorter than it is in reality. This situation could result in an abnormal behavior of the destination station because the Data Message must be passed to upper layers of the X-bus functionality for verification of its correctness because the bit level representation is correct without any illegitimate and unexpected bit patterns.

After devising the described fault injection approach, the algorithm for performing a Data Message corruption is the following:

(1) Read the SD of each message on the X-bus network.

(2) If the SD matches the VLDD (0x68, bit stream 00001011011), the fault injector observes the next five bits.

(3) If the detected bits correspond to either one of the LE PTPs of a Data Message (01011 or 01001), the fault injector outputs the corruption signal, targeting the 7th and 8th bit.

(4) The same action is performed on the second LE PTP.

(5) The receiving station perceives the Data message to be shorter than its actual length, and should detect it.

8.10.4. X-Bus Fault Injection Campaigns and Results

The designed X-bus fault injector was used to conduct fault injection campaigns targeting token and data messages. The fault injection campaign was based on altering the duration of the corruption applied to the Token and Data Messages. During the specified time, each occurrence of Token or Data Message (based on the setting) was corrupted and the response of Benchmark System I was observed. The response of the system was obtained by reading error messages from the SMS service unit, which logs errors and warning messages.

8.10.4.1. Faulted Token Corruption Times

The correct functionality of the X-bus fault injection module was verified by observing the X-bus traffic in the presence and absence of injected faults. This was achieved by using the logic analyzer to capture and subsequently analyze X-bus signals.

Figure 8-14 and Figure 8-15 display the behavior of X-bus signals with and without the corruption interference, respectively. By observing the response of the system under test, it

was confirmed that the X-bus fault injections were successful and executed at the correct time. The X-bus network traffic is analyzed by the fault injection module in real-time and the corruption signal interferes with the network signal at a specific moment. From Figure 8-15 it can be seen that the corruption is performed instantaneously without any incurred overhead.

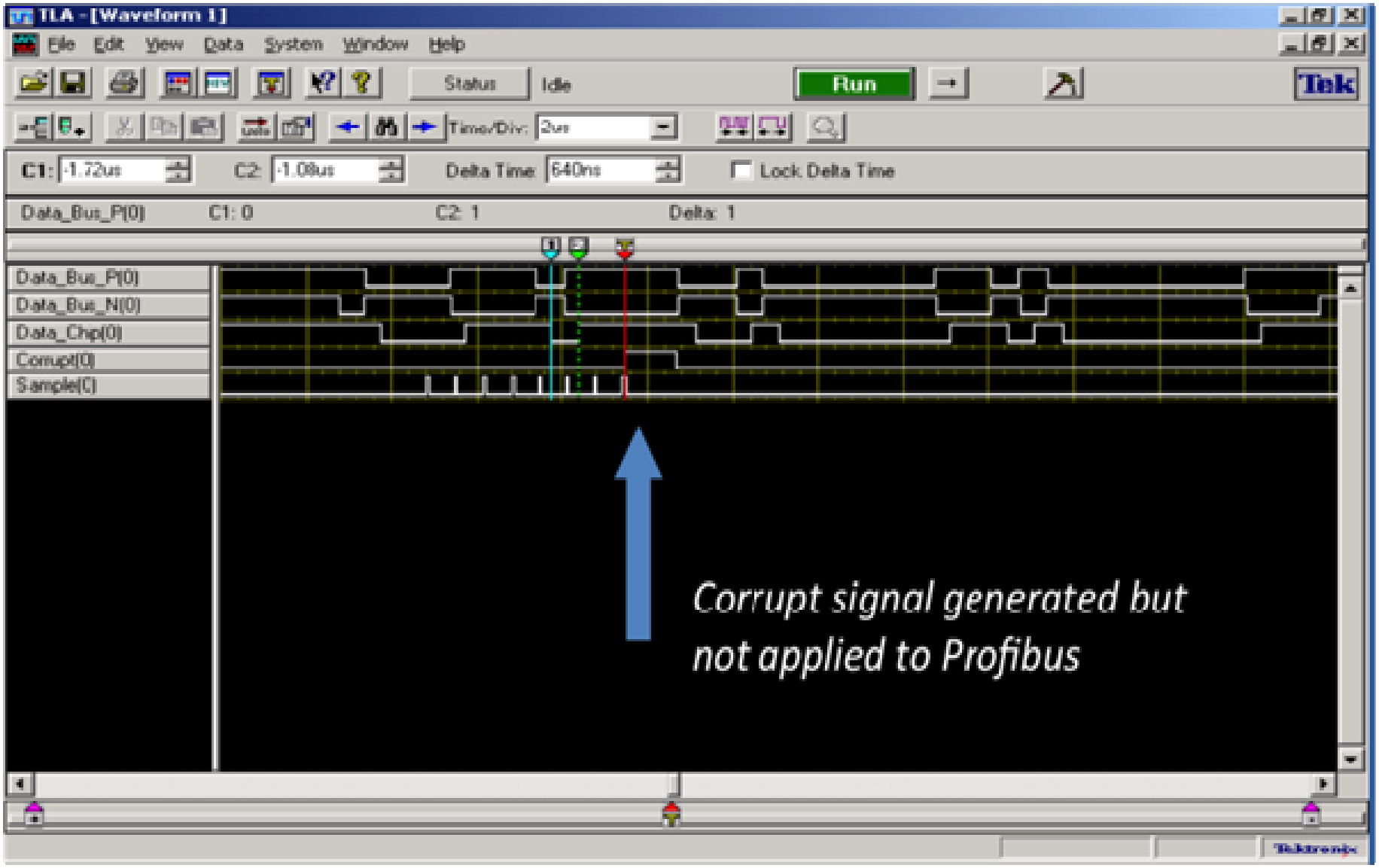


Figure 8-14 Jamming signal output correctly, but not applied to the X-bus circuit

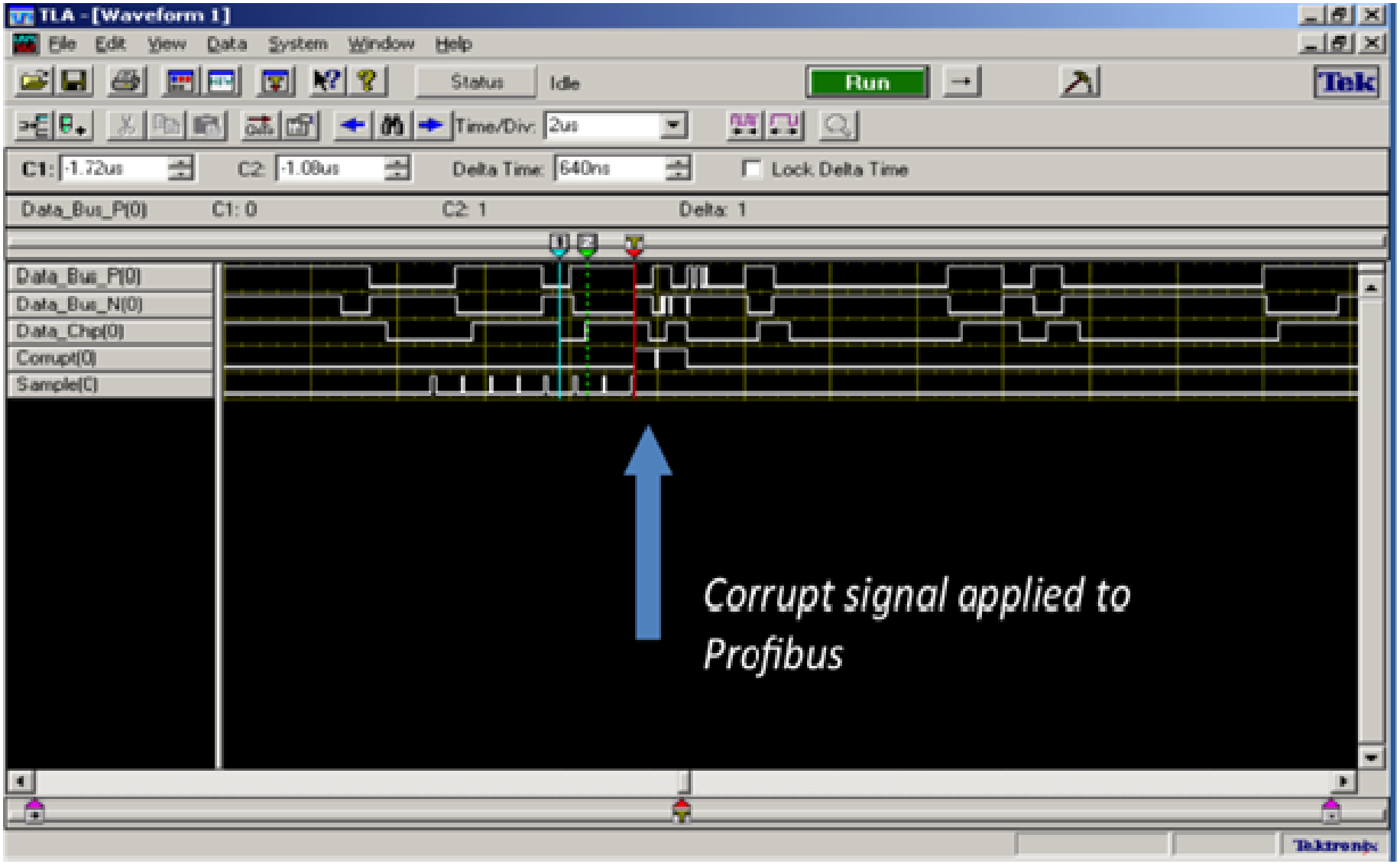


Figure 8-15 Jamming signal applied to X-bus thereby corrupting transmission

The effectiveness of the fault injection was confirmed by measuring the synchronization time after a corrupted Token had not been received by the destination Master station. This synchronization time was equal to the Slot Time (T_{sl}, see description in Section 4.1). The time represented by T_{sl} is significantly longer than the usual synchronization time during uninterrupted transmissions [Miklo 2009]. As shown in Table 8-5, this time is almost a magnitude larger than the average synchronization times observed during normal traffic. This X-bus time response allowed researchers to determine when the fault injection resulted in a successful token corruption.

Table 8-5 Synchronization times following uncorrupted and corrupted token in X-bus.

Transmission Type	Synchronization Time
Uncorrupted Token followed by a Token	261.7 μs
Uncorrupted Token followed by non- Token	327.4 μs
Corrupted Token	2,236.3 μs

Experiments with multiple consecutive corrupted Tokens were executed in order to determine the time required for reinsertion of a Master station after it was removed from the logical X-bus ring. This time was measured to be 15,278 μs, measuring from the corruption of the first Token to the time when the removed Master station was reinserted into the X-bus logical ring and acquired ownership of the logical ring by executing a transmission [Miklo 2009]. This time was determined when the target system was configured with two Master X-bus stations; the time could be longer in a four X-bus Master configuration. In situations where the benchmark system is heavily loaded with X-bus traffic, the reinsertion time can further increase because a Discovery Message is only sent from a Master station in case it has spare time during the rotation. The consequences of this problem could create a significant problem in a real-time safety-critical system with a hard deadline, specifically in the case when a Master station is removed from the Logical ring for a time longer than the system computational deadline.

For this reason, the Benchmark System I X-bus network is often configured to have additional physical redundancy to cope with faulty X-bus Master stations and corrupted Tokens. The additional redundancy mitigates cases of detectable faults in an X-bus communication module.

8.10.4.2. Token and Data Message Fault Injection Experiments

The detection of an unrecoverable corrupted message or Token was defined to be recognized if an X-bus network communication error appeared in the log. When there was no error message in the service log regarding the performed fault injection, then one of two possibilities can exist. Either the Token was successful on its second retry and thus no error message was generated, or the Token corruption was undetectable. Since the token synchronization time was monitored, it could be verified that Tokens were being corrupted in each fault injection.

The duration of the X-bus fault injection for Tokens ranged from 1 ms to 50 ms, in increments of 1 ms. To obtain a significant amount of data, fault injections were repeated for each time increment, 18 times in each Token fault injection campaign and 14 times in each Data Message corruption campaign, totaling 900 Token and 700 Data Message fault injections, respectively. For the data message corruption experiments the corruptions were varied from 1 ms to 100 ms. This was to ensure that the data messages were corrupted beyond the control cycle of the benchmark system (50 ms).

The response graph for the Token fault injection campaign is illustrated in Figure 8-16; and the Data Message fault injection campaign in Figure 8-17. The Token response graph is interpreted

as follows. When the fault injection duration time for corrupting a Token was in the range of 10 ms to 24 ms, the system reported the Token corruption accurately about 80% of the time. When the fault injection duration time for corrupting a Token was greater than 24 ms, the system reported the Token corruption 100% of the time. However, it cannot be inferred that the benchmark system did not detect the lower duration time fault injections. The most reasonable explanation for this non-reporting is that the processor X-bus controller re-sent the Token and the data on a second try, and the transmission was successful. From the graph it can be seen that sometimes the transmission was successful on the second try and at other times it was not. However, as the fault injections grew longer in duration, retransmissions were unsuccessful and the system reported a communication error that it was unsuccessful in communicating with the destination processor. So, from this it can be concluded that the Token corruption error detection mechanisms of the benchmark system perform as described in the system documentation.

The graph for the Data Message fault injection can be interpreted similarly to the Token message graph. For short time durations of fault injection on the Data Message length field, the system was able to detect the corruption most of the time and send a retry message successfully. When the system could not send a retry message, the system reported an error message. In addition, with Data Message corruptions there is the possibility of a no-response. For this reason, the duration of the fault injection corruptions was extended to 100 ms, thus ensuring that the Data Message would be corrupted over at least one control cycle. When the duration of the fault injection corruptions was extended to longer intervals, the system correctly reported that it had a communication error. However, in two instances when the duration of the fault injection corruptions was extended the system did not report a communication error, one instance at a 60 ms duration, and one instance at a 75 ms duration. The “yes” in the table indicates a proper detection and reporting. The “x” indicates no fault injection was done in this field.

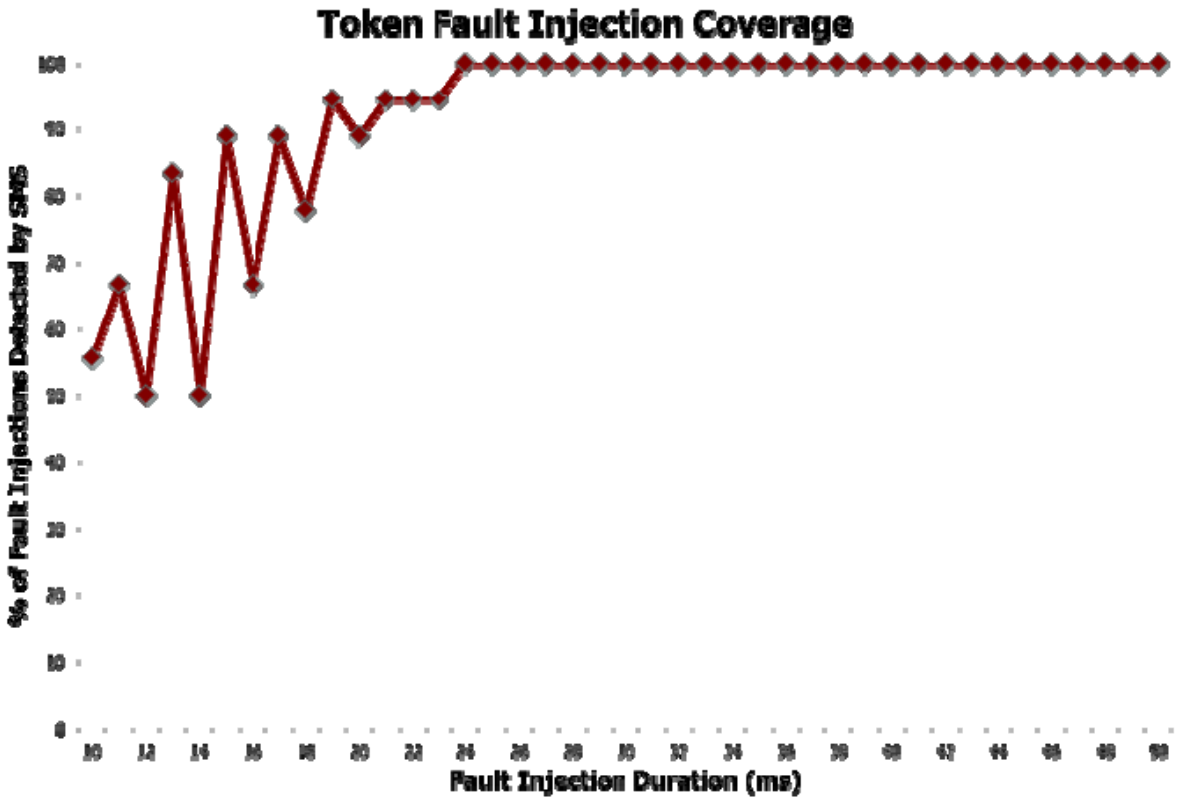


Figure 8-16 Token message fault response graph

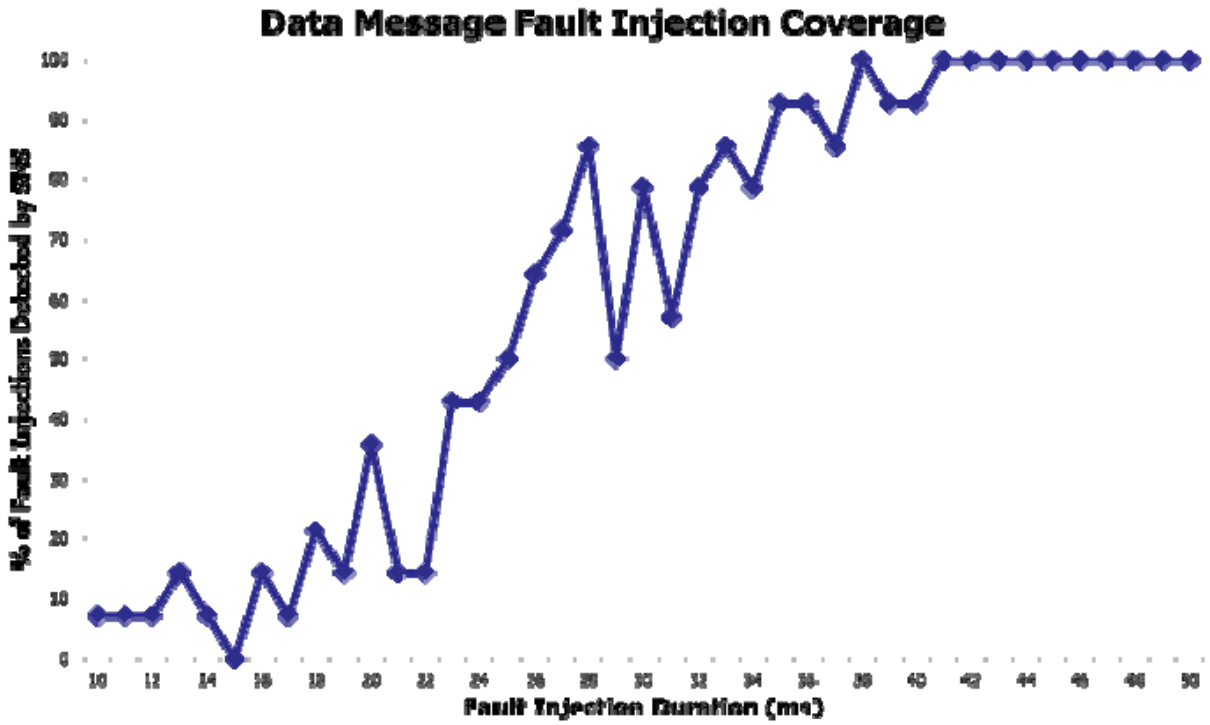


Figure 8-17 Data message fault response graph

Table 8-6 Data message response for long duration fault injections.

Fl Length	5ms_mult	10ms_mult_1	10ms_mult_2	10ms_mult_3	15ms_mult	20ms_mult	25ms_mult	30ms_mult	35ms_mult
50	Yes	Yes	Yes	NO	x	x	NO	x	x
55	Yes	x	x	x	x	x	x	x	x
60	Yes	Yes	Yes	Yes	NO	Yes	x	Yes	x
65	Yes	x	x	x	x	x	x	x	x
70	Yes	Yes	Yes	Yes	x	x	x	x	Yes
75	Yes	x	x	x	NO	x	Yes	x	x
80	Yes	Yes	Yes	Yes	x	Yes	x	x	x
85	Yes	x	x	x	x	x	x	x	x
90	Yes	Yes	Yes	Yes	Yes	x	x	Yes	x
95	Yes	x	x	x	x	x	x	x	x
100	Yes	Yes	Yes	Yes	x	Yes	Yes	x	x

8.11. References

[Smith 1997] D.T. Smith, B.W. Johnson, N. Andrianos, J.A. Profeta III. "A Variance Reduction Technique Using Fault Expansion for Fault Coverage Estimation." *IEEE Transactions on Reliability*, September 1997: 366-374.

[Miklo 2009] M. Miklo, R.D. Williams, C.R. Elks. “Token fault time in Profibus DP”. *In 6th American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control and Human Machine Interface Technologies,* April 2009.

9. SUMMARY, FINDINGS, AND CONCLUSIONS

This Section summarizes the activities described in this of this report (Section 9.1), and lists the principal study findings (Section 9.2) derived from those activities. This section provides preliminary conclusions drawn by the authors by applying and assessing the fault injection-based assessment methodology to Benchmark System I. In closing, some final observations and recommendations are made in order to better refine the fault injection-based assessment methodology toward the application to digital I&C systems.

9.1. Summary of Key Activities and Results

The work described in this report presents in a detailed manner;

(1) The development of the fault injection methods and techniques that were applied to the benchmark system,

(2) The development of fault injection environment for digital I&C systems

(3) Development of pre-injection analysis methods for automatically generating fault lists for digital I&C systems,

(4) Results of the application of fault injection to the benchmark system,

(5) The challenges to applying fault injection to contemporary digital I&C systems, and

(6) The findings for addressing these challenges and establishing a basis for implementing fault injection to digital I&C platforms.

The requirements and challenges of realizing fault injection on digital I&C systems are summarized in the following discussion.

9.1.1. Identification and Selection of Appropriate Fault Injection Techniques

In Section 4 appropriate physical fault injection techniques were identified for the benchmark system based on:

- The types of faults that could affect end-to-end system processing and thus impact I&C functionality

- The sub-systems or modules where fault injection should be applied to represent faults realistically

Based on these criteria, a fault injection technique matrix was developed that indicated appropriate fault injections for each sub-system in the benchmark system. These recommendations are presented from the view of the vendor's I&C technical staff or an independent assessor who has technical expertise and knowledge equivalent to the vendor. Thus, the matrix provides information on what is possible if complete system level documentation is available to the assessor.

Due to the proprietary nature of some of the JTAG test ports on the benchmark system and the inability to acquire system level source code, the research was limited to implementing two fault injection techniques. These were ICE-based fault injection and X-bus fault injection. Both of these techniques were developed to provide a capability to inject processor level faults and

protocol level faults into X-Bus. The important point is that several of the fault injection techniques surveyed and reviewed were not feasible or applicable to the benchmark system. For example, SWIFI-based fault injection assumes (without stating it) that the source code of the system software is available. While this may be the case for open operating systems such as POSIX or Linux, most digital I&C systems have proprietary operating systems that make it difficult to implement this popular type of fault injection on digital I&C systems by independent assessors.

9.1.2. Development of a Platform Independent Fault Injection Environment

Most fault injection tools have been developed with a specific fault injection technique in mind, targeting a specific system, and using a custom designed user interface. Extending such tools with new fault injection techniques, or porting the tool to new target systems is usually a cumbersome and time-consuming process. Since one of the objectives in this work was to apply fault injection to digital I&C systems typical of the type found in NPPs, a flexible and portable fault injection environment is required for efficient application of the UVA fault injection based dependability assessment methodology.

The work presented in Section 5 toward developing appropriate fault injection techniques and environments for digital I&C systems produces a body of work that the NRC and the nuclear industry could use to establish a basis for the development and standardization of fault injection processes.

The work toward developing the UNIFI serves a larger purpose in that it provides a detailed understanding of the complexities and processes involved in implementing physical fault injection effectively and efficiently in contemporary digital I&C systems. The successful application of the fault injection methodology using UNIFI shows that it has the capability to allow fault injections on complex digital I&C systems. The benchmark system used in this study was not designed or developed with fault injection in mind; therefore, the system presents the same challenge an independent assessor would encounter if employing a fault injection methodology on a comparable digital I&C system.

9.1.3. Tools for automated operational profile generation

Context is important in fault injection. Operational profiles must be representative of different system configurations and workloads that would be experienced in actual field operations. For a fault injection-based assessment methodology, the operational profiles must represent the input conditions and system interactions that can occur not only during nominal operations, but also in off-nominal operations and more importantly during “accident” event scenarios. Gathering profile real plant data across all of these domains of operations is a challenging task. As a research part of this effort, an innovative approach to providing high fidelity operational profiles of NPP digital I&C systems was developed. Before this phase of the research project, the methodology provided guidance on how to use an operational profile for fault injection, but provided little guidance on the various means to realize an operational profile. To address this need for the digital I&C systems, a TRACE-based Operational Profile model generation tool (TOP) was developed.

The TOP modeling tool is co-resident with the UNIFI fault injection environment. TOP normally operates as a separate set of programs from LabView and passes its operational profile data sets to the UNIFI/LabView environment. Operational profile data files are generated for the target system for each type of operational profile or test case of interest. The operational profile data for a specific test case or basis event is then repeatedly used for a set of fault injection campaigns. Changing the operational profile or test case or obtaining a new set of process

variables only entails rerunning the TRACE simulation to collect a new set of data. The important contribution of this work is that the set of tools that were developed allows for the profile data to be seamlessly integrated into the digital I&C fault injection processes for digital I&C testing in general.

9.1.4. Methods to improve the efficiency and effectiveness of fault injection (pre-fault injection analysis)

Pre-injection analysis is a means to reduce or eliminate the "no-response" and the long fault latency problem associated with typical fault injection campaigns. Being a statistical experiment, fault injection testing may require a large number of experiments to be conducted in order to guarantee statistically significant results. Thus, efficiency of the fault injection testing is important.

A typical digital I&C system will have a significant memory space (hundreds of megabytes is not uncommon), a large number of processor register files, special purpose configuration registers, and (relatively) long control cycle times (50 ms to 200 ms). With random fault injection experiments (i.e., with no regard to when and where a fault is injected), a large fraction (up to 90%) of fault injection experiments may have no-response outcomes [Sekhar 2008; Barbosa 2005].

A large percentage of these "no-response" outcomes resulting from fault injections are due to non-use of the corrupted data by the executing program. For example, a randomly generated fault could be injected into a memory location or a processor register that is not used by an application. These instances in which the tested system would not respond to an injected fault do not convey meaningful information about the fault tolerance capabilities of the system. Since time has an associated cost value, if the efficiency of the fault injection campaign is low, then the cost of the fault injection campaign is increased.

The pre-fault injection analysis techniques developed in this research and demonstrated by way of simulation have the potential to significantly improve the effectiveness and efficiency of physical-based fault injection. Preliminary results show at least a 50% improvement over random-based fault injection. Another important benefit of pre-fault injection analysis is being able to deduce the fault equivalence from a space-time perspective once the window of opportunity is known. Knowing the fault equivalence sets of a window of opportunity allows for fault expansion in the window. Fault expansion provides a means to increase the number of equivalent fault injections without having to actually perform each fault injection.

9.1.5. Application of the Fault Injection Methodology to Benchmark System I

The culmination of this research effort was the application of the fault injection-based dependability assessment methodology to the benchmark system. These experiments represent the types of fault injection tests that would be typically conducted by an assessment organization or the digital I&C equipment vendor during the course of a V&V activity. The experiments were chosen to stress the methodology and the supporting tools (UNIFI) in order to provide a basis for determining the effectiveness of the methodology to support system safety assessment activities (e.g., license reviews and FMEA) and PRA activities. All of the experiments were conducted successfully, providing a rich set of information on the fault handling behavior of the benchmark system that would be very supportive of PRA assessment activities.

9.2. Conclusions

This research effort lays a foundation for vendors and regulators to consider fault injection as a method to help inform the assessment of digital I&C systems in nuclear energy applications. Several findings were significant with respect to applying fault injection to the benchmark system digital I&C system, namely:

- Establishing the baseline elements and functionality of a fault injection environment for digital I&C systems;
- Developing new methods and tools for generating high-fidelity operational profiles from NPP simulation tools and establishing a basis for integrated digital I&C and plant analysis and testing.
- Developing new methods to improve the efficiency and effectiveness and to guide fault list generation for digital I&C systems;
- Creating new methods for applying fault injection testing to digital I&C systems.

The fault injection methodology applied to the benchmark system successfully obtained independent information about the benchmark system that corroborated vender and regulator information, and in some cases produced information that would have been very difficult to deduce from vendor information alone. The experience of conducting fault injection often yields more information than just quantifying fault tolerance aspects of the system; it also is a means to comprehend the behavior of complex fault tolerant I&C systems to support overall assessment activities for both the regulator and the developer.

9.3. References

[Barbosa 2005] Barbosa, R., Vintern, J., Fokesson, P., Karlsson, J. "Assembly-Level Pre-Injection Analysis for Improving Fault Injection Efficiency." *Lecture Notes in Computer Science, vol. 3463*, 2005: 246-262.

[Sekhar 2008] Sekhar, M. *Generating Fault Lists for Efficient Fault Injection into Processor Based I&C Systems.* Charlottesville, VA: University of Virginia, 2008.

NRC FORM 335
(12-2010)
NRCMD 3.7

U.S. NUCLEAR REGULATORY COMMISSION

BIBLIOGRAPHIC DATA SHEET

(See instructions on the reverse)

1. REPORT NUMBER
(Assigned by NRC, Add Vol., Supp., Rev., and Addendum Numbers, if any.)
NUREG/CR-7151
Volume 2

2. TITLE AND SUBTITLE
Development of a Fault Injection-Based Dependability Assessment Methodology for Digital I&C Systems: Volume 2

3. DATE REPORT PUBLISHED

MONTH	YEAR
December	2012

4. FIN OR GRANT NUMBER
N6124

5. AUTHOR(S)
C.R. Elks, N.J. George, M.A. Reynolds, M. Miklo, C. Berger, S. Bingham, M. Sekhar, B.W. Johnson

6. TYPE OF REPORT
Technical

7. PERIOD COVERED (Inclusive Dates)

8. PERFORMING ORGANIZATION - NAME AND ADDRESS (If NRC, provide Division, Office or Region, U. S. Nuclear Regulatory Commission, and mailing address; if contractor, provide name and mailing address.)
The Charles L. Brown Department of Electrical and Computer Engineering
The University of Virginia
Charlottesville, Virginia

9. SPONSORING ORGANIZATION - NAME AND ADDRESS (If NRC, type "Same as above", if contractor, provide NRC Division, Office or Region, U. S. Nuclear Regulatory Commission, and mailing address.)
Division of Engineering
Office of Nuclear Regulatory Research
U.S. Nuclear Regulatory Commission
Washington, DC 20555-0001

10. SUPPLEMENTARY NOTES
S.A. Arndt, J.A. Dion, R.A. Shaffer, M.E. Waterman, Project Managers

11. ABSTRACT (200 words or less)
This report is volume 2 of a multi-volume set of reports that present the cumulative efforts, findings, and results of NRC contract JCN N6124 – "Digital System Dependability Performance"

Volume 2 presents the findings of developing a fault injection based quantitative assessment methodology with respect to processor based digital I&C systems for the purpose of evaluating the capabilities of the method to support NRC probabilistic risk assessment (PRA) and review processes of digital I&C systems. The purpose of this work was to (1) determine the effectiveness of fault injection (as applied to a digital I&C system) for providing critical safety model parameters (e.g., coverage factor) and system response information required by the PRA and reliability assessment processes, (2) develop and refine the methodology to improve applicability to digital I&C systems, and (3) establish a basis for using fault injection applied to a diverse set digital I&C platforms.

12. KEY WORDS/DESCRIPTORS (List words or phrases that will assist researchers in locating the report.)
Fault injection, dependability, PRA, digital instrumentation and control systems, I&C

13. AVAILABILITY STATEMENT
unlimited

14. SECURITY CLASSIFICATION
(This Page)
unclassified
(This Report)
unclassified

15. NUMBER OF PAGES

16. PRICE

Printed
on recycled
paper

Federal Recycling Program

UNITED STATES
NUCLEAR REGULATORY COMMISSION
WASHINGTON, DC 20555-0001

OFFICIAL BUSINESS

NUREG/CR-7151
Volume 2

Development of a Fault Injection-Based Dependability Assessment Methodology for Digital I&C Systems

December 2012