

THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

A Software Profiling Methodology for Design and Assessment of Dependable Software

Martin Hiller

Department of Computer Engineering
School of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, SWEDEN, 2002

A Software Profiling Methodology for Design and Assessment of Dependable Software

Martin Hiller

ISBN 91-7291-215-4

Copyright © 2002 Martin Hiller, All Rights Reserved

Doktorsavhandlingar vid Chalmers tekniska högskola

Ny serie 1897

ISSN 0346-718X

School of Computer Science and Engineering

Chalmers University of Technology

Technical Report 3D

Department of Computer Engineering

School of Computer Science and Engineering

Chalmers University of Technology

SE-412 96 Göteborg

Tel. +46(0)31-772 10 00

www.ce.chalmers.se

Author email: hiller@ce.chalmers.se

Chalmers Reproservice

Göteborg, Sweden, 2002

A Software Profiling Methodology for Design and Assessment of Dependable Software

Martin Hiller

Department of Computer Engineering, Chalmers University of Technology

Abstract

The advent of computerized consumer products, such as for example automobiles, mobile systems, etc., has produced a large increase in the need for dependable (or robust) systems. As cost is a relevant issue for such systems, the cost of dependability has to be kept low. Furthermore, as the replication of software is virtually free compared to the replication of hardware, the trend is to implement more and more functions in software. This motivates the search for methodologies for cost efficient design and assessment of dependable software.

An established approach for designing dependable software entails addition of error detection mechanisms (EDM's) and error recovery mechanisms (ERM's). The effectiveness of these mechanisms, however, is achieved only if their composition is matched with their placement in locations where they are actually effective. It is the development of a systematic methodology to profile software in order to compose and locate EDM's and ERM's, that this thesis endeavors to achieve.

Presented in this thesis is a set of approaches for profiling software such that the most vulnerable and/or critical modules and signals can be identified in a quantifiable way. The profiling methodology relies on the analysis of error propagation and error effect in modular software. The results obtainable with these profiles indicate where in a given software system, errors tend to propagate and where they tend to cause the most damage as experienced by the environment.

The main contribution of this thesis is a software profiling methodology that encompasses development of the fault injection tool suite PROPANE (Propagation Analysis Environment) and the analysis framework EPIC (Exposure, Permeability, Impact, Criticality—the four main metrics introduced in the framework). The vision is that this contribution can aid software developers in the design and assessment of dependable software in the early stages of development.

Keywords: software profiling, error propagation analysis, error effect analysis, fault injection, embedded software, dependability, fault tolerance

List of Papers

This thesis is based on and extends the work and results presented in the following papers and publications:

- [A] Martin Hiller, *Error Recovery Using Forced Validity Assisted by Executable Assertions for Error Detection: An Experimental Evaluation*, Proceedings of the 25th EUROMICRO Conference, Vol. II, pp. 105–112, 1999
- [B] Martin Hiller, *Executable Assertions for Detecting Data Errors in Embedded Control Systems*, Proceedings of the International Conference on Dependable Systems and Networks (DSN), pp. 24–33, 2000
- [C] Martin Hiller, Arshad Jhumka, and Neeraj Suri, *An Approach for Analysing the Propagation of Data Errors in Software*, Proceedings of the International Conference on Dependable Systems and Networks (DSN), pp. 161–170, 2001
Recipient of the William C. Carter Award
- [D] Martin Hiller, Arshad Jhumka, and Neeraj Suri, *On the Placement of Software Mechanisms for Detection of Data Errors*, Proceedings of the International Conference on Dependable Systems and Networks (DSN), pp. 135–144, 2002
- [E] Martin Hiller, Arshad Jhumka, and Neeraj Suri, *PROPANE: An Environment for Examining the Propagation of Errors in Software*, Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), ACM Software Engineering Notes (SEN), Vol. 27, No. 4, pp. 81–85, 2002
- [F] Martin Hiller, Arshad Jhumka, and Neeraj Suri, *EPIC: A Framework for Error Propagation and Effect Analysis in Software*, submitted for publication to IEEE Transactions on Computers, 2002
- [G] Martin Hiller, Arshad Jhumka, and Neeraj Suri, *PROPANE: Analyzing the Propagation of Errors in Software*, submitted for publication to International Journal on Automated Software Engineering, Kluwer Academic Publishers, 2002

The following publications are related but not covered in this thesis:

- [H] Jörgen Christmansson, Martin Hiller, and Marcus Rimén, *An Experimental Comparison of Fault and Error Injection*, Proceedings of the 9th International Symposium on Software Reliability Engineering (ISSRE), pp. 369–378, 1998
Recipient of the Best Paper Award
- [I] Arshad Jhumka, Martin Hiller, and Neeraj Suri, *Assessing Inter-Modular Error Propagation in Distributed Software*, Proceedings of the 20th Symposium on Reliable Distributed Systems (SRDS), pp. 152–161, 2001
- [J] Arshad Jhumka, Martin Hiller, and Neeraj Suri, *On Systematic Design of Consistent Executable Assertions for Distributed Embedded Software*, Proceedings of the ACM Joint Conference Languages Compilers and Tools for Embedded Systems/Software and Compilers for Embedded Systems (LCTES/SCOPES), pp. 74–83, 2002
- [K] Arshad Jhumka, Martin Hiller, and Neeraj Suri, *Component-Based Synthesis of Dependable Embedded Software*, Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT), Lecture notes on Computer Science (LNCS) 2469, pp. 111–128, 2002
- [L] Arshad Jhumka, Martin Hiller, and Neeraj Suri, *An Approach to Specify and Test Component-Based Dependable Software*, to appear in Proceedings of the 7th International Symposium on High Assurance Systems Engineering (HASE), 2002
- [M] Örjan Askerdal, Magnus Gäfvert, Martin Hiller, and Neeraj Suri, *A Control Theory Approach for Analyzing the Effects of Data Errors in Safety-Critical Control Systems*, to appear in Proceedings of the Pacific Rim International Symposium on Dependable Computing (PRDC), 2002

Acknowledgments

Well, now the time has finally come when I can look back at my years as a graduate student and exclaim “Yes! It’s over! I did it!”. But, to be honest, that would not be entirely true. My graduate studies are over, that much is true. But I cannot really say that “I did it!”, partially because, when I write this, I have not yet had my defense. Mostly, however, because I had a lot of help on the way. I should really say “Yes, *we* did it! Soon...”

I would probably not have made it this far without the help and support of my advisor Professor Neeraj Suri. He showed me that research can actually be fun. Neeraj has not only provided professional support and guidance, but has also succeeded in creating a great social environment among his students in the DEEDS group, of which I see myself as a proud member.

I am also deeply indebted to my colleague Arshad Jhumka, who not only is a smart fellow, but also a very nice one, and a close friend. Arshad has been of invaluable help ever since we started our collaboration a few years back. Sail on, my friend, you will find your own mooring soon!

The rest of the DEEDS group also deserves many thanks for creating a great working atmosphere. Here we have the soccer and chess playing Örjan Askerdal. I hope I have not ruined any more of your research topics. We also have Vilgot Claesson/Klasson, the gadget maniac who always knows what’s hot and what’s not in the realm of electronic stuff. The two most recent additions to the group (who have been around for quite a while now) are the moose-hunting Robert Lindström, who is something as exotic as a northerner who talks a lot, and the cap-wearing Andréas Johansson, who is the one in our group who can actually rival Vilgot in gadget mania. Thanks a lot, lads!

Before joining the DEEDS group, I started out working for my former advisor Professor Jan Torin. This was in the Software Fault Tolerance group headed by Håkan Edler. Seniority was provided by Dr. Jörgen Christmansson who, together with Dr. Marcus Rimén, formed a co-advisor team for me through my first two-and-a-half years as a graduate student. I am very grateful for all the help and support you two gave me during this time! Also in this group was my colleague and former room-mate Robert Feldt, who I brutally forced to read my early papers. Thank you,

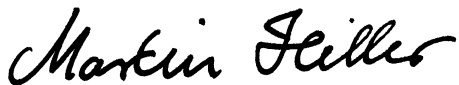
Robert, for all advice, stimulating discussions, comments and much more over the many years we were room-mates!

I would also like to mention the many students at the department that all add their own distinctiveness to the great atmosphere: Counter-strikers, computer architects, tele-tubbies, software dudes, hardware dudes, other dudes...you are too many to name here!

My parents, Karin and Siegfried, are of course ultimately responsible for putting me in this position. I guess they did not really know what they started when they gave me my first computer, a Commodore C64, back in 1985. From then on it was all down-hill. Also, in a supporting role, we have Mattias, the best little brother (well, maybe not so little anymore...) anyone can possibly ask for!

Finally, my absolute favorite, Angela. You are the best! I love you!

The work presented in this thesis would of course not have been performed without financial support. The main funder of this work was Volvo, via the DCN (*Dependable Computer Nodes*) project in the research framework *Volvo-Chalmers Fordonsteknisk Forskning*. The work was also partially supported by NUTEK (1P21-97-4745), by Saab Endowment, and by EC DBench (IST-2000-25425).

A handwritten signature in black ink, reading "Martin Hiller". The script is cursive and fluid, with the first letters of "Martin" and "Hiller" being capitalized and prominent.

Martin Hiller

Göteborg, September 2002

Contents

1	Introduction	1
1.1	The Fundamentals of Dependability	2
1.1.1	Dependability Attributes	2
1.1.2	Dependability Impairments	3
1.1.3	Dependability Means	4
1.1.4	Fault Tolerance	5
1.1.5	Fault Removal and Fault Forecasting	6
1.2	Goals, Activities and Problem Statements	7
1.2.1	Goals and Activities	7
1.2.2	Problem Statements	8
1.3	Main Contributions	10
1.4	Thesis Structure	12
2	Related Work	13
2.1	Fault Injection	14
2.1.1	Simulation Based Fault Injection	14
2.1.2	Physical Fault Injection	16
2.1.3	Software Implemented Fault Injection	16
2.2	Software Implemented Fault Tolerance	18
2.3	Error Propagation Analysis	20
3	Assumed Models and Example Target System	23
3.1	Models and Assumptions	24
3.1.1	System and Software Model	24
3.1.2	Fault and Error Model	25
3.2	Target: Aircraft Arresting System	25
3.2.1	System Overview	26
3.2.2	Software Structure	27
3.2.3	Failure Classification	28
3.2.4	Adjustments for Example Studies	29

4	Software Mechanisms for Handling Data Errors	33
4.1	Introduction	34
4.2	Executable Assertions in Modular Software	36
4.3	Signal Classification: Taking an Abstract View on Data	37
4.3.1	Continuous Signals	37
4.3.2	Discrete Signals	41
4.3.3	Signal Modes	42
4.4	Mechanisms for Error Detection and Recovery	42
4.5	Finding Locations and Defining Parameters	46
4.6	Evaluating the Capabilities of the Mechanisms	47
4.6.1	Software Instrumentation	48
4.6.2	Fault Injection Environment	50
4.7	Evaluation 1: Error Tolerance	51
4.7.1	Setup of Evaluation 1	51
4.7.2	Results of Evaluation 1	53
4.7.3	Discussion of Evaluation 1	54
4.8	Error Detection Coverage and Error Propagation	55
4.9	Evaluation 2: Error Detection	56
4.9.1	Setup of Evaluation 2	56
4.9.2	Results of Evaluation 2	57
4.9.3	Discussion of Evaluation 2	60
4.10	Summary and Conclusions	63
5	PROPANE - The Propagation Analysis Environment	67
5.1	Introduction	68
5.2	Target System Model	69
5.3	Overview of the Tool	70
5.3.1	Basic System Structure	70
5.3.2	Work Process for Using PROPANE	73
5.4	Setup: Experiment Design and Target Instrumentation	75
5.4.1	Selecting Which Faults and Errors to Inject	75
5.4.2	Faults and Fault Triggers	76
5.4.3	Error Types and Injection Locations	77
5.4.4	Triggering the Error Injections	78
5.4.5	Logging Variables, Memory Areas, Events	79
5.4.6	Environment Simulators and Test Cases	80
5.4.7	Target System Instrumentation	80
5.4.8	Setup Using Description Files	83

5.5	Injection: Running Experiments	83
5.6	Analysis: Obtaining Propagation Characteristics	85
5.6.1	Golden Run Comparisons	85
5.6.2	Channel Logs	88
5.6.3	Injection Information	88
5.6.4	Propagation Information	88
5.7	Example Results Generated by PROPANE	89
5.8	PROPANE's Attributes and Comparison with Other FI-tools	95
5.8.1	Main Characteristics of PROPANE	95
5.8.2	Comparison Details	96
5.9	Summary and Conclusions	100
6	Error Propagation and Effect Analysis	103
6.1	Introduction	104
6.2	Software and System Model	105
6.3	EPIC: Generating Software Profiles	105
6.3.1	Error Permeability - Letting Errors Pass	106
6.3.2	Ascertaining Propagation Paths	108
6.3.3	Assessing the Error Exposure of Modules and Signals	112
6.3.4	Analyzing the Effect of Errors on System Output	114
6.3.5	Identifying Candidate Locations for ERM's and EDM's	119
6.4	Obtaining Numerical Estimates of Error Permeability	121
6.5	Experimental Analysis: An Example System	123
6.5.1	Target Software System	123
6.5.2	System Analysis	124
6.5.3	Experimental Setup	127
6.5.4	Experimental Results and Obtained Profiles	129
6.6	Selecting Locations for EDM's	132
6.6.1	Propagation-Based Selection of Locations	132
6.6.2	Adding the Effect Profile to the Selection Process	133
6.7	Comparing the Two Location Selections	134
6.7.1	Memory and Execution Time Requirements	135
6.7.2	Error Detection Coverage	135
6.8	Discussion on Framework Limitations and Caveats	139
6.9	Summary and Conclusions	140

7	Summary and Conclusions	143
7.1	Summary of Research Contributions	144
7.1.1	Error Detection and Recovery Mechanisms	144
7.1.2	Evaluation of Mechanisms	144
7.1.3	Error Propagation and Effect Analysis	145
7.1.4	Evaluation of Analysis Framework	146
7.1.5	Tool for Analyzing Error Propagation	147
7.2	Conclusions	148
8	Outlook on Future Work	151
8.1	The Future and Executable Assertions	152
8.2	The Future and Software Analysis	152
8.3	The Future and PROPANE	153
8.4	The Future and The Rest	153
	Bibliography	155
	Appendix A. PROPANE – Details	165
A.1	Instrumentation of Target Systems	165
A.1.1	Instrumenting for Probes	166
A.1.2	Instrumenting for Fault Injection	174
A.1.3	Instrumenting for Error Injection	177
A.2	Fully Automated Instrumentation of Target Systems	182
A.3	Interfacing with Environment Simulators	184
A.4	Adding Error Types and Error Triggers	185
A.5	Description Files for PCD and PL	188
A.5.1	Database Descriptions	188
A.5.2	Campaign Descriptions	189
A.5.3	Experiment Descriptions	190
A.6	Analysis Scripts for PDE	194
A.6.1	Error Margins for Golden Run Comparisons	196
A.7	Setup Scripts for PSC	196
A.8	PROPANE Architecture	199
A.8.1	The PROPANE Campaign Driver	199
A.8.2	The PROPANE Library	201

CHAPTER 1

Introduction

Basic research is what I'm doing when I don't know what I'm doing.

— Wernher von Braun (1912–1977)

Since the invention of electrical computers in the middle of the 20th century, they have been put to use in a number of areas. Computers started out as extremely expensive resources used for computations of mathematical problems in research, defense and industrial applications. From that time, computers have undergone a dramatic change. They have become smaller, cheaper and easier to use. This has made computers attractive not only for pure computational purposes, but also as integral components in systems that are traditionally mechanical in nature. First, they made their entrance in high-end systems such as spacecraft, aircraft and nuclear power-plants. Now, they are steadily gaining acceptance in more consumer-oriented areas, such as automobiles. This chapter briefly describes the area of dependability and introduces the fundamentals in general and the topics covered in this thesis in particular.

1.1 The Fundamentals of Dependability

The fundamental concepts of dependability used throughout this thesis are adopted directly from the compilation of concepts by made Laprie [Laprie (ed.), 1992]. This section contains a short overview of the main terms and definitions used here.

The term *dependability* is defined as “the trustworthiness of a system such that reliance can justifiably be placed on the service it provides”. What this means is that a dependable system is one upon which the user (either human or non-human) can place its trust in that the services provided by the system are correct. Dependability of a system is characterized by a set of *attributes*, compromised by a set of *impairments*, and achieved and analyzed by a set of *means*. The fundamental terms and concepts of dependability described in the subsequent sections can be organized in a taxonomy tree as shown in Fig. 1.1.

1.1.1 Dependability Attributes

The dependability *attributes* characterize, and profile, the dependability of a given system. These attributes are the following:

Availability is a measurement of how available the system is, i.e. the probability that the system is operational and providing its service at any given time. The higher the availability, the higher the probability that the system provides its service at the time that service is requested.

Reliability is used to measure the probability that a system provides the service it was originally set to provide during a finite period of time. That is, the higher the reliability, the higher the probability that the response given by a system is correct.

Safety is the extent to which a system provides a service which is safe to its environment, i.e., it does not endanger its user. Note that even though the system may provide a service which was originally not intended, this service may still be safe for the users. Therefore, the safety measure may be higher than the reliability measure.

Confidentiality, Integrity, and Maintainability are attributes which are not addressed in this thesis and, thus, will not be described further at this point.

A system would have no trouble fulfilling all these attributes perfectly if it were not for disturbing factors as described in the next section.

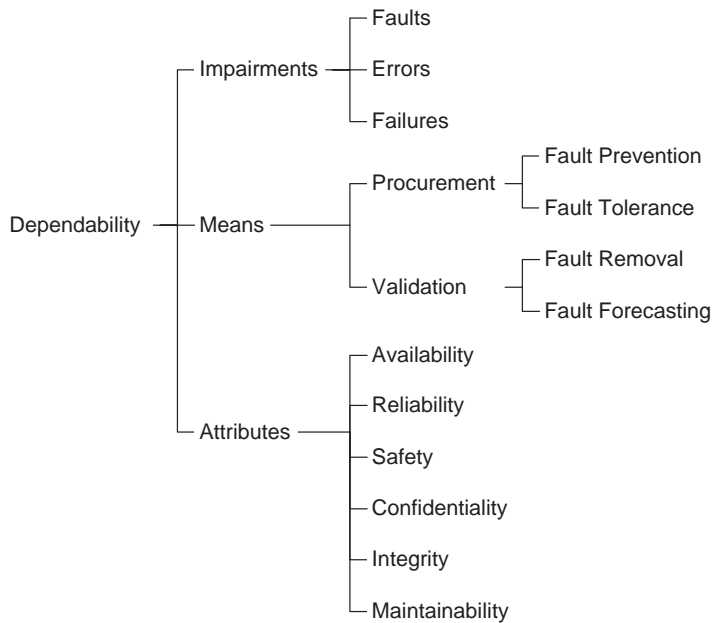


Figure 1.1: The taxonomy of dependability

1.1.2 Dependability Impairments

During the construction and the operation of a dependable system, events may occur which reduce the trustworthiness of the system by introducing *faults* into the system. For example, the developers may (inadvertently) design the system in such a way that for certain conditions the system cannot provide its specified service, i.e., the system contains defects (which can be hardware and/or software defects). During system operation, external disturbances or aging of components may introduce faults that, again, prevent the system from providing its intended service. The events that may reduce the dependability of a system are referred to as the *impairments* of dependability.

The mere presence of faults is, however, not enough to reduce the dependability of a system. A fault must be activated, i.e., the part of the system in which the fault is located must be exercised in some way during system operation (e.g., faulty code must be executed, defective memory locations must be referenced, etc.). If this is happen, the result may be an *error*. If a fault is viewed as a disease, an error can be said to be a symptom of that disease. An error is defined as an erroneous state in the system, i.e., the state is different from the state the system would have had if the fault had not been present. An error which is activated may cause other errors to occur in the system. This process is called *error propagation*. If errors propagate beyond the system barrier, i.e., if they are visible to the environment of the system,

a *failure* has occurred, as this prevents the system from providing proper services.

The causality chain, $\text{fault} \rightarrow \text{error} \rightarrow \text{failure}$, is also recursive in nature. Thus, what can be seen as a failure at one level of the system can be seen as a fault on the next higher level. Therefore, we get the following sequence:

$$...\text{failure} \rightarrow [\text{fault} \rightarrow \text{error} \rightarrow \text{failure}] \rightarrow \text{fault}...$$

With this definition we can say that a dependable system is a system which is able to break this chain at some point before *Something Very Bad* happens, i.e., before the loss of equipment, investments or perhaps even human lives.

For the development of dependable systems, a set of means have been identified. These are described in the next section.

1.1.3 Dependability Means

When developing dependable systems, there are a number of means by which dependability can be achieved and analyzed, namely:

Fault Prevention is the process of preventing faults from occurring in the first place. Examples of fault prevention activities may be the use of certain development processes and methodologies.

Fault Tolerance is to actively handle the occurrence of faults and errors and reverting an erroneous system state into a state which is either correct (preferably) or at least safe.

Fault Removal is the process of reducing the number and seriousness of faults (a more popular term for this is *debugging*). This phase often involves verification and validation, diagnosis and correction.

Fault Forecasting is performed in order to get an estimate of the consequence faults would have if they should occur.

The focus of this thesis is mainly on *fault tolerance* with a little spice of *fault removal* and *fault forecasting*. The following sections will describe these means more in detail.

1.1.4 Fault Tolerance

If a system is able to function properly even in the presence of faults and errors, it is considered to be fault tolerant. However, fault tolerance is not a binary property, i.e., a system may be able to tolerate certain types of faults, whereas other types still disrupt system operation. The basic model for the fault tolerance process is divided into four phases (compiled from [Randell *et al.*, 1978], [Anderson and Lee, 1982] and [Lee and Anderson (ed.), 1990]):

1. Error Detection
2. Damage Assessment
3. Error Processing, and
4. Fault Treatment.

Error detection is the action of detecting that an erroneous system state is actually present. After an error has been detected, *damage assessment* is necessary to see to what extent that error caused damage to the system and where that damage is located. With the information gathered during damage assessment, the system can then initiate *error processing*, where an erroneous system state is transformed into a good state in which no (detectable) errors are present. The combined actions in damage assessment and error processing are called *error recovery*. The last phase, *fault treatment*, has the goal of preventing the same faults from being activated again. In this phase faults are diagnosed and treatments are devised to passivate the identified faults. Fault treatment is generally performed off-line and often involves several parties. For instance, fault diagnosis may be performed by a Board of Inquiry and fault passivation by the system designers.

The main focus of this thesis is on those parts of the fault tolerance process that are incorporated into the system itself, namely, error detection and error recovery. Illustrated in Fig. 1.2 are the phases error detection and error recovery in the context of the *fault* \rightarrow *error* \rightarrow *failure* causality chain described in Section 1.1.2. Illustrated here is that a system starts out without any active faults (however, design faults are of course always present). Faults occur from external disturbances and may disappear again. An activated fault may an error which potentially is detected by the error detection mechanisms (EDM's) of the system. After an error is detected, the error recovery mechanisms (ERM's) of the system try to convert the state of the system into one which is error free. If recovery is only partially successful, errors may still reside in the system state. These errors may subsequently lead to system failure. Also, an unsuccessful error recovery will result in system failure. If recovery is successful, the system can continue with normal operation.

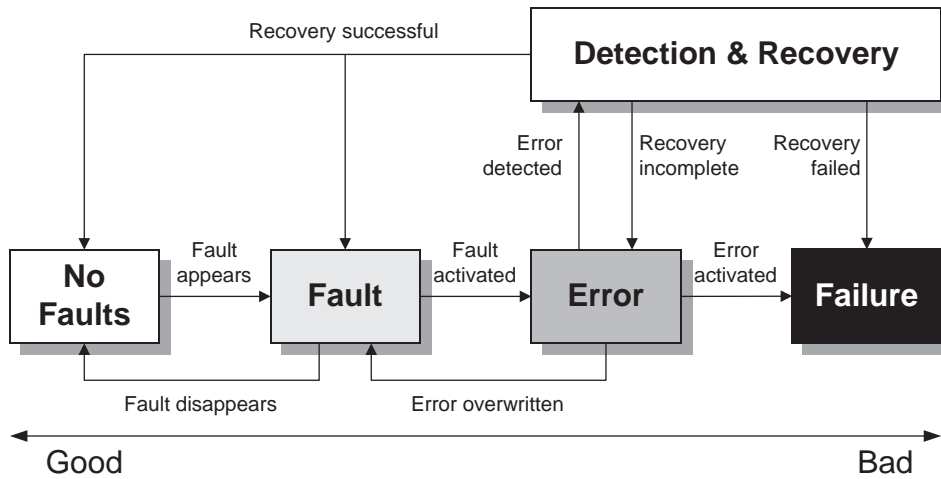


Figure 1.2: Error detection and recovery mechanisms in the context of the *fault* \rightarrow *error* \rightarrow *failure* causality chain

The focus of this thesis is on error detection and error recovery with regard to data errors, i.e., erroneous values in system variables (as opposed to, for example, control flow errors which alter the execution trajectory of a system).

1.1.5 Fault Removal and Fault Forecasting

Once a dependable system has been constructed, it is time to assess how dependable the system really is. For this, fault forecasting is used. Fault forecasting is the process of estimating and ascertaining system performance when subjected to faults. This activity is often combined with fault removal, i.e., the process of actively removing faults (mainly design faults) from the system. The combination of the two is referred to as *validation* of dependability. Traditionally, validating the dependability of systems can be done either analytically or experimentally.

In analytical validation, a formal (often mathematical) representation of a system is analyzed instead of the actual system itself. One way of doing this is by applying *formal methods* (see, e.g., [Clarke and Wing, 1996]), where techniques based purely on mathematics and logic are used for describing system properties and actually proving these to be correct (or at least, without internal contradictions). As a research area, formal methods is an active area which is perhaps still “growing up”. In the dependability area, formal methods have been used, for example, in the verification of protocols (see [Echtle and Masum, 1996] and [Sinha and Suri, 1999]).

Experimental validation is based on testing the actual implementation of a system. In order to experimentally validate for example the fault tolerance properties

of a system, a common way to go about it is to artificially insert faults and/or errors into the system in order to create conditions where the deployed mechanisms for fault tolerance are activated. This is referred to as *fault injection* and has been a popular method for testing and assessing dependability of systems. For more information on *fault injection* see, e.g., [Chillarege and Bowen, 1989], [Arlat *et al.*, 1990], [Iyer, 1995], [Powell *et al.*, 1995], and [Iyer and Tang, 1996].

Of course, *testing* is also a very important part of experimental validation. Here, a system is subjected to a number of test cases (vectors of input data) in order to detect any defects/faults that may be present. If the behavior of the system differs from the expected behavior, then the reason may be that a fault changes the system properties such that it fails to provide its intended service.

A very important, and hard, problem in experimental validation is identifying which faults and errors to inject (if fault injection is used) and which test cases to run (if testing is used).

The (probably) best, and most expensive, way to identify which faults and errors to inject into a system when validating dependability is to analyse faults and error that occur in reality, i.e., those that are experienced by and reported from systems which actually operate on the field. Even though numerous such studies have been performed (for an overview, see [Lyu (ed.), 1995], pp. 303–358 and 439–487), only a few have had a focus on injection of software faults (see, e.g., [Iyer *et al.*, 1990], [Sullivan and Chillarege, 1991] and [Christmansson and Chillarege, 1996]).

In the identification of test cases for testing activities, formal methods can sometimes be helpful. In [Suri and Sinha, 1998], for example, formal methods were used to help find test cases for validating dependable protocols. Also, in this example, dependability validation was performed as a combination of testing and fault injection.

1.2 Goals, Activities and Problem Statements

1.2.1 Goals and Activities

The main goal of the work presented in this thesis has been to find and evaluate new construction methods for dependable computer nodes with little amount of redundancy in embedded (and possibly distributed) control systems. In this respect, the focus has been on software-based methods for fault tolerance and software robustness as well as analysis methods for software. These methods apply to control systems in automobiles and other systems in which low production cost is crucial. The research presented here has mainly contained the following two activities:

Activity 1 Finding and evaluating techniques and mechanisms for fault tolerance suitable for low-cost (in the sense of both development cost and production cost) embedded control systems, and

Activity 2 Devising techniques for software analysis with regard to error propagation and error effect such that vulnerable and critical parts of software systems can be identified such that suitable locations for placing error detection and recovery mechanisms can be selected.

A key driver for the research presented in this thesis has been that the results should be applicable to systems with demands on low cost levels for development as well as for production. This has led to the efforts in **Activity 1** being concentrated on techniques and mechanisms implemented in software. Even though software potentially requires large amounts of resources for development, production is virtually free of charge once development is complete. Therefore, software methods are very attractive for development of dependable systems that are produced in very high volumes (hundreds of thousands, or even millions) such as, e.g., the embedded control systems in automobiles. Another reason for choosing software implemented fault tolerance over hardware implemented fault tolerance is that software can be adapted more easily to the application and to the environment. However, this property does also bring with it some less attractive properties such as the fact that software mechanisms to a higher extent are application specific.

Another consequence of the low-cost driver is that it is not sufficient to describe mechanisms for error detection and recovery alone. Deciding where in the software system these mechanisms should be placed also becomes an important problem. As cost is an issue, the amount of resources required in a system should be kept as low as possible. Thus, being able to analyze software such that cost can be weighed against obtained benefit (i.e., a cost-benefit analysis) would be very valuable. Therefore, the efforts in **Activity 2** have been geared towards the analysis of error propagation and error effect. If one were able to identify those locations in software which attract most propagating errors and also how these errors affect the service of the system, designing and placing mechanisms to tolerate those errors would be helped.

In the next section is a more detailed account of the problem statements that guided the work presented in this thesis.

1.2.2 Problem Statements

The work started out with an effort to identify available software implemented mechanisms and techniques for fault tolerance in embedded control systems and to evalu-

ate their weaknesses and strengths. Thus, we can state the following problem statement:

PS1 What techniques and mechanisms exist for software implemented fault tolerance? What are their advantages and disadvantages when seen from the point-of-view of an embedded control system? Especially when considering production cost, overhead in hardware and software, and so on?

This problem statement was addressed in the early stages of the work that resulted in this thesis, and these efforts are summarized in [Hiller, 1998], which contains an overview of various software techniques for fault tolerance. Note that this overview is not part of the contents of this thesis. Once this statement has been addressed, candidate mechanisms for further analysis and evaluation can be chosen. When these candidates have been identified the following problem statement must be considered:

PS2 How can developers be aided in developing the mechanisms and incorporating them into the software of an embedded control system? To what extent is dependability improved by using the proposed mechanisms? That is, what is the combined value of the error detection coverage and the error recovery coverage? How well do the mechanisms detect errors? That is, how high is the probability of detecting errors (the error detection coverage) using the mechanisms and how long is the error detection latency?

Having efficient software mechanisms for error detection and error recovery is of course desirable. However, knowing where they would do the (in some sense) most good is likely to prevent costly resources being spent on inefficient use of the mechanisms. Thus, the final problem statement to be considered is:

PS3 How can a system designer identify the most suitable locations in software systems for placing error detection and recovery mechanisms? Which parts of a software system are most vulnerable to faults and errors that might be present in the system? How do errors propagate through a software system? Where do occurring errors cause the most damage? What role does the error model play when analyzing software with regard to error propagation and error effect? Can an analysis framework be developed to handle any negative impact?

These problem statements have guided the work that is now presented in this thesis, and, hopefully, some light can be shed upon these problems. The main contributions and results are briefly described in the next section.

1.3 Main Contributions

The hope and vision is that the results presented in this thesis may in some way help software developers to construct dependable systems by showing how a software system can be equipped with mechanisms for error detection and error recovery. The necessary prerequisites for putting EDM's and ERM's in software is illustrated in Fig. 1.3. This figure illustrates that the system developer requires knowledge about three things:

1. The type of errors that the system is supposed to be able to handle; their type, how often they occur, etc. If the software designer has no knowledge of what kind of threats the system is subject to, it is very hard to know how to obtain any dependability. This would make both the development as well as the assessment/analysis of the system difficult (if not impossible).
2. The mechanisms available for error detection and error recovery. When equipping the software system with EDM's and ERM's it is of course important to know the characteristics and properties of the mechanisms at ones disposal, including their strengths and weaknesses. It is likely that the overall architecture of the software is affected by the properties of the mechanisms.
3. The characteristics of the software with regard to vulnerabilities and hot-spots. In order to place the mechanism where they are the most effective, it is important to know where errors tend to propagate and where errors tend to do the most damage. This will aid in using the available resource in such a way that the benefit is optimized (at least in the compiler-sense of the word).

The contributions put forward in this thesis address the latter two items, and they are the following:

Error Detection and Recovery Mechanisms. Software mechanisms for error detection and error recovery are developed. These mechanisms are based on the concept of *executable assertions*, operating at the signal/variable level and are implemented as generalized software mechanisms that are instantiated with parameters. The sets of parameters required for each signal are predefined according to a certain *signal classification* and the values of the parameters are set by the system designer.

Evaluation of Mechanisms. The presented error detection and recovery mechanisms are evaluated with regard to coverage and latency using fault injection experiments.

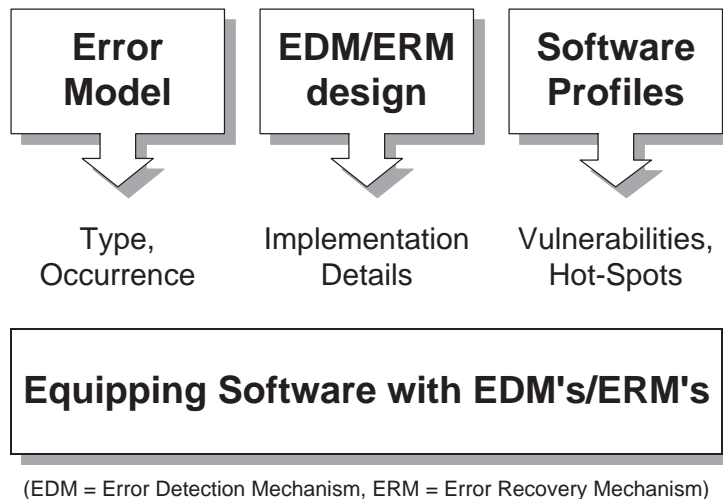


Figure 1.3: The process of equipping software with mechanisms for error detection and recovery

Error Propagation and Effect Analysis. A framework is presented which enables system designers to profile software systems such that vulnerable and critical modules and signals/variables can be identified. This framework is able to produce two distinct quantitative profiles, namely: (i) a propagation profile, and (ii) an effect profile. The propagation profile shows how errors propagate through the software system, and the effect analysis shows to what extent errors in the various signals/variables affect system output. The framework introduces four basic measures: (i) *Exposure*, (ii) *Permeability*, (iii) *Impact*, and (iv) *Criticality*, and thus is called EPIC.

Evaluation of Analysis Framework. The presented analysis framework is evaluated on real software in order to illustrate its applicability.

Tool for Analyzing Error Propagation. A tool-suite called PROPANE, the Propagation Analysis Environment, has been developed in order to demonstrate the applicability of the presented analysis framework, EPIC. PROPANE can perform fault and error injection and is capable of tracing the values of variables in software such that error propagation and error effect can be analyzed. PROPANE can also log events which enables the evaluation of error detection and recovery mechanisms. There are built-in extension possibilities, enabling users to construct their own injectors and logging probes, making PROPANE a versatile tool for software and dependability analysis.

1.4 Thesis Structure

The remainder of this thesis is structured as follows:

Chapter 2 reviews related work and results in the areas of *fault injection*, *software implemented fault tolerance*, and *error propagation analysis*, and tries to show where the work presented in this thesis fits in.

Chapter 3 introduces the software and system model and associated assumptions used throughout this thesis. This chapter also describes the target system used in the various evaluations performed in the thesis. The chosen target system is an embedded system utilized for arresting aircraft, i.e., helping landing aircraft to stop on short runways and aircraft carriers.

Chapter 4 contains a description of the executable assertions framework developed for error detection and recovery. This chapter also contains an evaluation of the mechanisms where the example aircraft arresting system is equipped with the described mechanisms and then subjected to fault injection in order to estimate detection and recovery coverage as well as detection latency.

Chapter 5 presents PROPANE, the tool developed for error propagation and effect analysis and for evaluation of error detection and recovery mechanisms. PROPANE is purely software based tool designed for running on desktop systems. This chapter also contains a comparison of PROPANE against other contemporary tools.

Chapter 6 introduces EPIC, the developed framework for error propagation and effect analysis. The framework is also demonstrated on the aircraft arresting system and its applicability illustrated. As the presented approach for estimating the measures of EPIC is based on fault injection, the effect of varying the underlying error model after analysis is examined.

Chapter 7 summarizes the work presented in this thesis and lists the main conclusions to be drawn from the obtained results.

Chapter 8 provides an outlook on future work, open issues and questions yet to be answered.

CHAPTER 2

Related Work

If I have seen further, it is by standing on the shoulders of giants.

— Sir Isaac Newton (1642–1727)

This chapter contains an overview of previous work and results in the area of dependable software for embedded systems. More specifically, the areas that most closely relate to the work presented in this thesis, i.e., the areas of *fault injection*, *software implemented fault tolerance*, and *error propagation analysis*. This chapter also tries to show where in these areas the work presented in this thesis fits in.

2.1 Fault Injection

As mentioned in the introduction to this thesis, Chapter 1, *fault injection* is a popular method of testing and assessing dependability of constructed systems. The basic approach in fault injection is to artificially insert faults and/or errors into the system and then analyse its behavior. More information on fault injection in general can be found in, e.g., [Chillarege and Bowen, 1989], [Arlat *et al.*, 1990], [Iyer, 1995], and [Iyer and Tang, 1996]. Fault injection allows detailed studies of the complex interaction between faults and fault handling mechanisms. For example, fault injection can be used to estimate the coverage of error detection mechanisms, i.e., the success rate of the mechanisms. In [Powell *et al.*, 1995], the authors show that if n_{inj} faults or errors are injected and n_{det} of these are detected, an unbiased estimate of the detection coverage, c_{det} , can be obtained with $c_{det} = \frac{n_{inj}}{n_{det}}$. This, however, assumes that the fault/error models used for selecting the faults and/or errors to inject are representative of the “real” faults and errors the system under consideration is subject to. The issue of representativeness has been, and still is, a fairly open research question, especially when considering software faults and their resulting errors.

The various fault injection techniques and tools that have been introduced over the years can generally be divided into three categories:

1. Simulation Based Fault Injection
2. Physical Fault Injection
3. Software Implemented Fault Injection

The technique used in this thesis primarily fits into the Software Implemented Fault Injection (SWIFI) category. Some of the studies performed, however, have a whiff of Simulation Based Fault Injection, as the target is real software running interacting with a simulated hardware environment (sensors and actuators). Therefore, the focus here is mostly on SWIFI and a little on Simulation Based Fault Injection, whereas Physical Fault Injection is covered very briefly.

Note that the term *fault injection* is not limited to injecting faults according to the definition described in Chapter 1, where faults are seen as the root cause of erroneous system states. This term is generalized to also cover *error injection*, i.e., instead of injecting defects, one injects erroneous system states directly.

2.1.1 Simulation Based Fault Injection

In early design stages of dependable systems, fault injection is often more easily utilized on a model of the system, rather than the actual system itself. This presents

several advantages over injecting faults into physical systems. The most important of those advantages is probably the fact that it can be used very early in system design. Thus, simulation based fault injection can facilitate the detection of design faults in fault tolerance mechanisms, thereby reducing the cost of developing such mechanisms (as costly prototype phases may be reduced or eliminated). Other advantages are that controllability of experiments and observability are very high, as one has total control over injection time and location as well as data acquisition.

Among the drawbacks of this techniques is the fast increase of time-complexity when running simulations. The more details are simulated, the more time is required for running the simulations. This in turn may have a limiting effect on what can actually be included in a system simulation, thus limiting the accuracy of the simulation.

Simulation based fault injection has mainly been used for evaluating the hardware of dependable systems. The simulations are often divided into three levels of abstraction (see [Iyer, 1995]): i) electrical, ii)logical and iii) functional.

At the electrical level, circuits are simulated with currents and voltages and faults are usually emulated by changing these physical properties. Fault injection is here primarily used for analyzing the impact of physical causes to faults and errors.

At the logical level, systems are simulated as gates, i.e., at abstraction levels above those of currents and voltages. At this level, larger systems, such as VLSI circuits and microprocessors, can be simulated and their dependability assessed. Fault models used here are typically stuck-at-0, stuck-at-1 and inversion. Tools for injecting faults at this level are, e.g., MEFISTO (see [Jenn *et al.*, 1994]) and VERIFY (see [Sieh *et al.*, 1997]).

Simulations at the functional level are primarily used for evaluating architectures and policies for large systems such as networks or large computer systems. The fault models at this level are often based on the fault models of lower levels, such as bit-manipulations in memory and registers, component failures, etc. As the simulations are on an arbitrarily high level of abstraction, the fault model must be chosen accordingly. At this level, we have tools like DEPEND (see [Goswami and Iyer, 1991] and [Goswami, 1997]) and Loki (see [Cukier *et al.*, 1999] and [Chandra *et al.*, 2000]).

The work on fault injection presented in this thesis is on the functional level of simulation based fault injection. The tool presented in Chapter 5 can be used for evaluation of simulated architectures, similar to the approach used by DEPEND. The target systems used in the studies performed in this work also have parts which are simulated, such as, sensors and actuators, and the environment.

2.1.2 Physical Fault Injection

When prototypes or real implementations of a system exist, the dependability of these can be evaluated and analyzed by injecting actual physical faults. This approach is important as it tests the actual implementations of fault handling mechanism and, thus, implementation faults can potentially be caught here. The drawback of this approach, though, is reduced controllability and observability, compared to simulation based fault injection. Faults are commonly injected at pin-level, i.e., by inserting faults at the pins of the circuits of the system, or by radiation, i.e., subjecting the target system to a radioactive source which creates upsets in the electronic parts of the system.

Examples of studies where this approach to fault injection has been used are [Arlat *et al.*, 1990], [Walter, 1990], and [Madeira *et al.*, 1994] for pin-level fault injection, and [Koga and Kolasinski, 1984] and [Gunneflo *et al.*, 1989] for radiation based fault injection. A comparison between various techniques for physical fault injection can be found in [Folkesson, 1999].

Physical fault injection does not directly relate to the work presented in this thesis and is mentioned just to give an overview of various fault injection techniques.

2.1.3 Software Implemented Fault Injection

By far the most versatile and now probably also the most widely used approach for fault injection is Software Implemented Fault Injection, or SWIFI. This approach uses software, rather than hardware, to inject faults into physical, and sometimes also simulated, systems. Thus, the advantages of this approach are, among other things, cost-effectiveness and flexibility, as no (or little) additional hardware is required. The disadvantages include the fact that when using SWIFI for injecting faults into hardware system, controllability and observability are sometimes reduced compared to simulation based fault injection, and the effects of physical faults may not always be properly emulated on account of reduced reachability (SWIFI can only reach those parts of the system which software can reach).

One of the first attempts of using SWIFI is reported in [Segall *et al.*, 1988] and [Barton *et al.*, 1990] where a tool called FIAT is presented. This tool is able to inject faults into user code and data by flipping bits (that is, setting a bit from 1 to 0 or from 0 to 1) in the task image of a process, and the main driver in these studies was to evaluate system architectures. These studies evaluated a real-time checkpointing workload and showed that the obtained results are dependent on the underlying fault classes used, i.e., changing the types of faults injected also changes the results (error detection coverage in this case).

The aim of FIAT is to be able to evaluate tolerance against both hardware faults and software faults. Examples of other tools in this category are DEFINE (see [Kao and Iyer, 1995]) and FTAPE (see [Tsai and Iyer, 1996]). In DEFINE, instruction level faults are injected by switching op-codes in the text segment of the target system. Faults are also injected by manipulating bits in memory and on the address bus of the system. The tool was designed to evaluate UNIX-networks and their dependability. FTAPE was designed to evaluate entire systems (as opposed to low-level mechanisms)

There are SWIFI tools which focus on evaluating fault tolerance against hardware faults only. For example, FERRARI (see [Kanawati *et al.*, 1995]) which injects faults using the UNIX process handling system by spawning target processes in a special trace mode enabling manipulation of process images, thereby facilitating injection of transient faults and permanent faults. DOCTOR (see [Han *et al.*, 1995]) is another tool which injects faults by mutation (i.e., by changing the actual code that is executed) and errors by bit manipulations and by disturbances in communication between system components.

Another tool which has gained a lot of publicity in this category is Xception (see, e.g., [Carreira *et al.*, 1995] and [Carreira *et al.*, 1998]). This tool uses the debug port present on many modern microprocessors to inject faults and errors. Xception is able to perform op-code switches and bit-manipulations in the sub-parts of the system (such as sub-units of the processor, address bus, memory and register banks, etc.).

The work presented in this thesis has a focus on error propagation analysis (as well as the classical error effect analysis) which requires very high observability. The tools mentioned so far do not display an observability high enough to analyse the propagation of (data) errors in software. There are tools, however, which do that. For example, MAFALDA (see [Fabre *et al.*, 1999]) is a tool for analyzing the effects of software faults and to some extent the propagation of errors in real-time micro-kernels. This tool requires some hardware support in order to function and performs its injection at the OS-level. From the available information, it is unclear if MAFALDA has the logging functions required for detailed analysis of error propagation. Another tool, NFTAPE (see [Stott *et al.*, 2000]), on the other hand provides a wide range of injection as well as logging facilities and can be used for detailed analysis of error propagation. However, NFTAPE is designed to run on a LAN and is designed with a separate control host and a target node.

The tool described and used primarily for propagation and effect analysis in this thesis, PROPANE (see Chapter 5), is most closely related (in functional terms) to MAFALDA and NFTAPE and other tools which offer high observability and controllability. PROPANE also has functionality which resembles that of DEPEND or

Loki in the sense that it can be used to inject faults and errors in simulations of systems and architectures (as PROPANE uses standard executable code as its target). A more comprehensive comparison between PROPANE and some of the SWIFI-tools mentioned here is found in Chapter 5.

2.2 Software Implemented Fault Tolerance

With the increased demand for computer control in consumer products and inexpensive dependable systems, it is only natural to want to use software for implementing dependability mechanisms.

A common way of coping with faults and errors using software is to deploy multiple, diverse versions of the software. These versions may be organized in a variety of structures such as, for example, N-Version-Programming (NVP, described in [Avizienis, 1985]) or Recovery Blocks (RB, described in [Randell *et al.*, 1978] and [Randell and Xu, 1995]). In addition, a number of combinations and enhancements of these two basic structures have been suggested, such as Consensus Recovery Blocks (described in [Scott *et al.*, 1983]) and Distributed Recovery Blocks (described in [Kim, 1989]).

Other structures have also been introduced and presented. For instance, N-Self-Checking-Programming (a unifying term of several real life implementations described in [Laprie *et al.*, 1987]), where software components are associated with built-in tests checking the produced outputs, either individually (using acceptance tests) or in pairs (using comparison tests). We also have N-Copy-Programming or Retry Blocks (described in [Ammann and Knight, 1988]), where the main approach is data diversity as opposed to code diversity.

However, systems using such structures will, generally, be high-cost systems as multiple, perhaps functionally equivalent, versions may have to be developed. Furthermore, more powerful hardware is often needed, further increasing the cost of these systems. Therefore, such structures are almost exclusively found in systems that can carry such a high cost level, e.g., systems controlling aircraft, spacecraft or nuclear power plants. Also, in [Randell and Xu, 1995] the authors state that “the overall success of recovery block schemes rests to a great extent on the effectiveness of the error detection mechanism used—especially on the acceptance tests”. This makes the search for inexpensive error detection techniques valid also for structures like recovery blocks.

Error detection may be provided using on-line tests of internal data in the form of executable assertions. As stated in [Leveson *et al.*, 1990], error detection in the form of executable assertions can potentially detect any error in internal data caused

by either software faults or hardware faults. Some of the first appearances of this technique are found in [Hecht, 1976] and [Saib, 1978]. In [Saib, 1978], the programming languages PASCAL and FORTRAN were extended to include an **assert** instruction. Executable assertions test the validity of the value of an individual variable or a set of variables using predefined rules and can be used both during software development to aid developers in finding faults in the system, as described in [Mahmood *et al.*, 1984], and when the system is operational as part of fault-tolerance mechanisms, as described in [Rabéjac *et al.*, 1996]. In addition to on-line error detection, executable assertions may be used during the development of a system for testing purposes, as for instance in [Andrews, 1979], and to assess the vulnerability of the system.

In [Rosenblum, 1995], a tool called the Annotation PreProcessor (APP) is described. This is a processing tool for assertions addressing ease-of-use and effectiveness issues when dealing with assertions in C programs developed for UNIX. A similar approach was presented in [Yin and Bieman, 1994]. A method for reducing the number of executables assertion using static analysis of source code was presented in [Gough and Klaeren, 1997]. Here, the authors also argue that using inter-modular analysis can even further reduce the number of assertions required. Here, preconditions are specified in interface descriptions of encapsulated objects (software modules).

The main drawback of executable assertions is that they are highly application specific, meaning that in order to construct effective assertions, developers must have extensive knowledge of the target system. Studies in [Leveson *et al.*, 1990] have shown that the ability to develop effective assertions is highly individual among software developers. Making the development of executable assertions a part of the normal system design process rather than a task that is performed when the system enters a test phase or after the system has been made operational, may decrease the effect of differences between individuals.

Rabéjac presented in [Rabéjac *et al.*, 1996] a development methodology for executable assertions. Unfortunately, no in-depth description of this method was provided. Stroph and Clarke presented in [Stroph and Clarke, 1998] dynamic acceptance tests, which are executable assertions with dynamic constraints. However, their proposed scheme applies only to linear, causal, time-invariant systems (which implies that the systems may not have any state).

The work regarding error detection and recovery presented in Chapter 4 of this thesis attempts to make design and incorporation of executable assertions more rigorous by proposing mechanisms that are generic test routines which only have to be instantiated with parameters. Also, the data to be tested is categorized accord-

ing to a signal classification scheme which in turn dictates which mechanisms to use. Another motivating factor behind this approach is to make the assertions less application specific.

2.3 Error Propagation Analysis

Error propagation analysis for logic circuits has been in use for many decades. Numerous algorithms and techniques have been proposed, such as, the D-algorithm in [Roth, 1980], the PODEM-algorithm in [Goel, 1981] and the FAN-algorithm in [Fujiwara and Shimono, 1983] (which improves on the PODEM-algorithm).

Error propagation in hardware is also addressed in [Shin and Lin, 1988], where a stochastic propagation model based on error propagation times is described. However, the authors do not cover locations for EDM's/ERM's as is done in this thesis. Also, the model is defined at the module level, i.e., if there are several signals linking two modules together, these will not be considered individually, but as a group.

An approach for dependability analysis, including error propagation, based on data flow analysis in HW-SW co-design is presented in [Csertán *et al.*, 1995]. Here, a data flow model of the system (including only functional requirements) is extended with information regarding fault occurrence, fault latency and detection probabilities such that a dependability analysis can be performed. This approach works on a high-level model of a system which is not yet divided into hardware and software.

In [Voas and Morell, 1990], propagation analysis in software was used for debugging purposes. Here the propagation analysis aimed at finding probabilities of source level locations propagating data-state errors if they were executed with erroneous initial data-states. The framework was further extended in [Voas, 1992] and [Morell *et al.*, 1997] for analyzing source code under test in order to determine test cases that would reveal the largest amount of defects. In [Voas *et al.*, 1998], the same framework was used for determining locations for placing assertions during software testing, i.e., aiming to place simple assertions where normal testing would have difficulties finding defects.

Analysis based on control flow is described in [Geoghegan and Avresky, 1996]. Here, a software system is analyzed with regard to control flow and, based on the results of this analysis, flow checks are placed in order to detect errors dynamically. As this approach only deals with control flow errors, it is very different from ours as we deal with data errors. The control flow approach will not handle detection of data errors unless these change the control flow such that it can be detected by the obtained EDM's.

An investigation in [Michael and Jones, 1997] reported that there was evidence of uniform propagation of data errors. That is, a data error occurring at a location l in a program would, to a high degree, exhibit uniform propagation, meaning that for location l either all data errors would propagate to the system output or none of them would.

Finding optimal combinations of hardware EDM's based on experimental results was described in [Steininger and Scherrer, 1997]. They used coverage and latency estimates for a given set of EDM's to form subsets which minimized overlapping between different EDM's, thereby giving the best cost-performance ratio.

In [Leveson *et al.*, 1990], a study on the use of self-checks and voting for software error detection concludes, among other things, that placement of self-checks seemed to cause problems, i.e., self-checks that might have been effective failed on account of being badly placed.

The work presented in Chapter 6 of this thesis contains an approach for analyzing how data errors propagate in modular software and how they affect the output of the system. This way of software analysis, or software profiling, allows system designers to perform a cost-benefit analysis to select locations suitable for error detection and recovery mechanisms. That is, the placement process is going away from *ad hoc* and becomes more rigorous as propagation and effect of errors can be quantified.

CHAPTER 3

Assumed Models and Example Target System

Things should be made as simple as possible, but not any simpler.

— Albert Einstein (1879–1955)

In order to be able to produce general results in the area of dependable software, one must first specify some model of the systems and software considered. This chapter describes the model of the view on modular software in embedded system used in the work presented in this thesis along with underlying assumptions on communication between the software modules. This chapter also introduces an example system which is used as a target system to evaluate and illustrate the techniques presented in this thesis.

3.1 Models and Assumptions

3.1.1 System and Software Model

The work in this thesis is based on the assumption of modular software, i.e., discrete software functions interacting to deliver the requisite functionality. A module is considered to be a generalized black-box module having multiple inputs and outputs (as illustrated in Fig 3.1).



Figure 3.1: A generalized black-box software module **M** with m inputs and n outputs.

The fact that black-box knowledge is assumed means that the internals of the modules is unknown and, thus, unchangeable. However, the techniques and methods for design and analysis of dependable software presented in this thesis are not limited to black-box software. In fact, the techniques and methods are just as applicable to white-box software.

A software module performs computations using the provided inputs to generate the outputs. At the lowest level, such a black-box module may be a procedure or a function but could also conceptually be a basic block or particular code fragment within a procedure or function (at a finer level of software abstraction).

Modules communicate with each other in some specified way using varied forms of signaling, e.g., shared memory, messaging, parameter passing, etc., as pertinent to the chosen communication model. All communication pathways are here called signals, irrespective of the actual underlying communication paradigm. That is, a signal is just an abstraction indicating some form of information and data exchange between modules.

A number of black-box modules constitute a system and they are inter-linked via signals, much like hardware components on a circuit board. Of course, this system may be seen as a larger component or module in an even larger system. Signals can originate internally from a module, e.g., as a calculation result, or externally from the hardware itself, e.g., a sensor reading from a register. The destination of a signal may also be internal, being part of the input set of a module, or external, for example the value placed in a hardware register. A system with multiple inter-linked modules is illustrated in Fig 3.2.

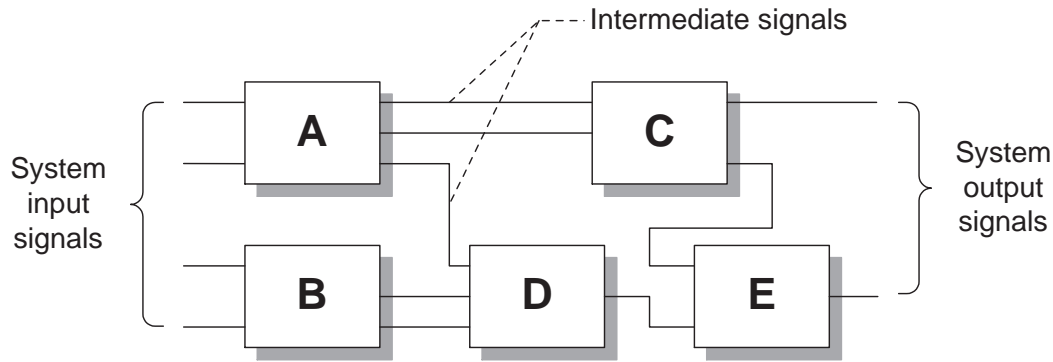


Figure 3.2: A system consisting of 5 black-box inter-linked modules. Indicated are *system input signals*, *intermediate signals*, and *system output signals*.

Software constructed as described above is found in numerous embedded systems. For example, most applications controlling physical events, such as in automotive systems, are traditionally built up as such. The studies described and presented in this thesis mainly focus on software developed for embedded systems in consumer products (high-volume and low-production-cost systems).

3.1.2 Fault and Error Model

The work in this thesis concentrates on data errors. Thus, how these errors have occurred is not a main concern. The underlying faults may be software faults (defects), hardware design faults, external disturbances, etc. In this respect, the error models used in the experiments described in this thesis are all such that erroneous values are created by manipulating the bits of the target variables or memory locations. It can be argued that random bit-flips can mimic the effects of transient hardware faults, as shown in [Rimén *et al.*, 1994]. More information regarding error models can be found for each experiment as they are described in the thesis.

3.2 Target: Aircraft Arresting System

In order to evaluate and illustrate the mechanisms and techniques presented in this thesis, a system following the model described in Section 3.1 is used as an example system. The target is a system is used for arresting aircraft on short runways, as found on, for instance, aircraft carriers or small airfields, and is designed according to specifications found in [USAF, 1986].

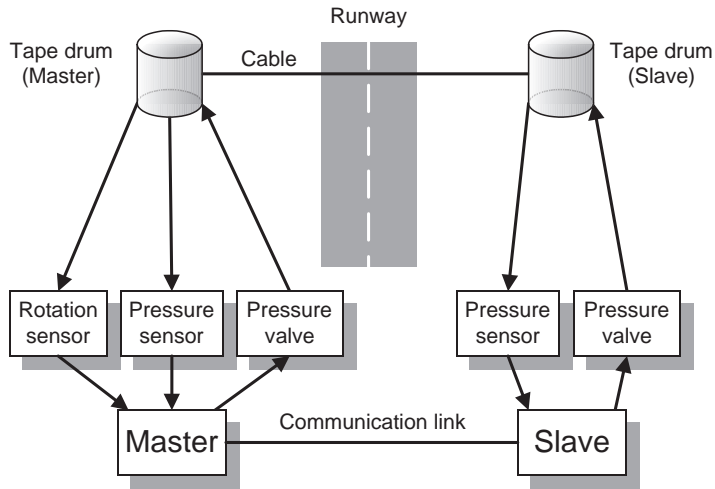


Figure 3.3: Overview of aircraft arrestment system used in example studies in this thesis.

3.2.1 System Overview

The system consists of a cable strapped across the runway and is attached to two tape drums, one on each side of the runway (see Fig 3.3). Two computer nodes control the drums: i) a master node and ii) a slave node, one node for each drum. These two computer nodes are connected via a serial link on which the master node transmits commands to the slave node and the slave node transmits result information to the master node. The processors used in these computers are Motorola 68HC11 running at a clock speed of 7 MHz.

An incoming aircraft grabs hold of the cable with a hook attached to the fuselage, and the cable immediately begins to rotate the tape drums as the aircraft travels along the runway. Attached to the rotating drums are tooth wheels and optic sensors measuring the number of teeth passing by the sensor. The sensors are periodically read by the master node which can then, using the number of pulses generated by the tooth wheels, calculate the length of the pulled out cable, the rotational speed of the tape drums, as well as the current speed of the aircraft. The master node calculates the set point pressure that is to be applied to the drums by means of hydraulic pressure valves in order to slow the rotation, eventually bringing the aircraft to a complete stop. The slave node receives its set point pressure value from the master node and applies this to its drum. Pressure sensors on the valves give feedback to their respective nodes about the pressure that is actually being applied so that a software-implemented PID-controller can keep the actual pressure as close to the set point value as possible.

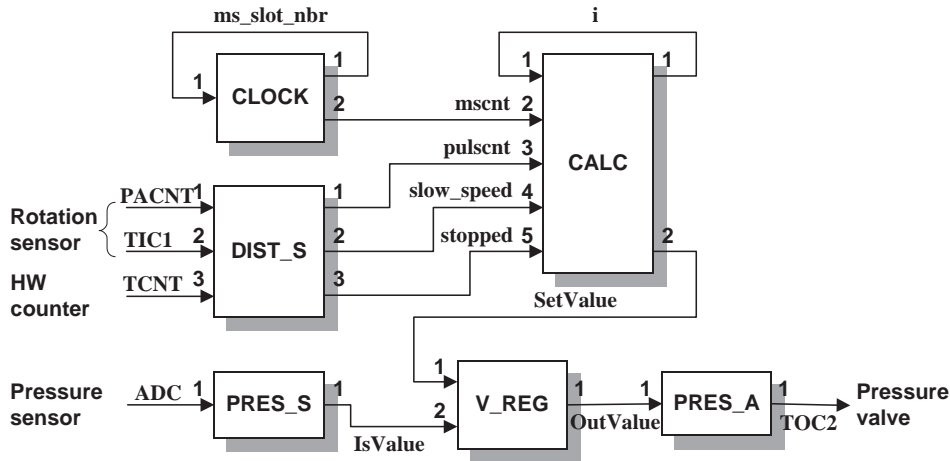


Figure 3.4: Software structure of the master node of the aircraft arrestment system.

3.2.2 Software Structure

In the studies performed for this thesis, focus was on the master node of the system. The software of the master node is mainly composed of six modules of varying size and input/output signal count (see Fig 3.4). The numbers shown at the inputs and outputs in Fig 3.4 are used for numbering the signals. For instance, *PACNT* is input #1 of *DIST_S*, and *SetValue* is output #2 of *CALC*. The software is composed of six modules of varying size and input/output signal count. System input is received from a number of sensors at *PRES_S* and *DIST_S*, and system output is provided to an actuator at *PRES_A*. The remaining modules (*CALC*, *V_REG* and *CLOCK*) provide internal/intermediate signals.

The scheduling of the system is slotted, meaning that a main loop is running indefinitely and this loop is divided into a number of slots. The period of the iterations is 1 millisecond and in each iteration of the loop the functions/modules belonging to a certain slot will be executed and a slot-counter is then incremented. The module specifics are:

CLOCK provides a millisecond-clock, *msCnt*. The system operates in seven 1-ms-slots. In each slot, one or more modules (except for *CALC*) are invoked. The signal *ms_slot_nbr* tells the module scheduler the current execution slot. This module has a period of 1 ms.

DIST_S receives *PACNT* and *TIC1* from the rotation sensor attached to the tape drum, and *TCNT* from the hardware counter modules. The rotation sensor reads the number of pulses generated by a tooth wheel on the drum. *DIST_S*

provides a total count of the pulses, *pulscnt*, generated during the entire arrestment, as well as two boolean values, *slow_speed* and *stopped*, indicating if the velocity is below a certain threshold (*slow_speed* == TRUE) or if the aircraft has stopped altogether (*stopped* == TRUE). This module has a period of 1 ms.

CALC uses *mscnt*, *pulscnt*, *slow_speed* and *stopped* to calculate a set point value for the pressure valves, *SetValue*, at six predefined checkpoints along the runway. The distance between these checkpoints is constant, and they are detected by comparing the current *pulscnt* with pre-defined, internally stored *pulscnt*-values corresponding to the different checkpoints. The current checkpoint is stored in *i*. This module is the background task, i.e., it runs when other modules are dormant. Thus, it has no period.

PRES_S reads the pressure sensor measuring the pressure that is actually being applied by the pressure valves, using *ADC* from the internal A/D-converter, and provides this pressure value in *IsValue*. This module has a period of 7 ms.

V_REG uses *SetValue* and *IsValue* to control *OutValue*, the output value to the pressure valve. *OutValue* is based on *SetValue* and then modified to compensate for the difference between *SetValue* and *IsValue*, by means of a software-implemented PID-controller. This module has a period of 7 ms.

PRES_A uses *OutValue* to set the pressure valve via the hardware register *TOC2*. This module has a period of 7 ms.

The modules described above are the modules used for implementing the actual service of the system. There are some more modules in the actual software handling the communication on the serial link between the master node and the slave node. However, these have been left out as they are not considered in the example studies performed in this thesis.

The software of the slave node is in essence the same as that of the master node. However, as the slave node does not itself calculate any set point pressure for its tape drum (i.e., the *SetValue* signal), and instead receives this from the master node, the software of the slave node only contains the modules **CLOCK**, **PRES_S**, **V_REG**, and **PRES_A**.

3.2.3 Failure Classification

The specifications according to which the system is implemented [USAF, 1986] dictate certain physical constraints, which the system must honor. These constraints are

that the retardation must not exceed a certain limit in order to not affect either the plane or the pilot in a negative way, and that the force applied to the aircraft by the cable must not exceed certain limits in order to not endangering the aircraft. Also, the length of the runway is limited. However, this constraint may vary from installment to installment. The constraints are as follows:

1. Retardation (r). The retardation of the aircraft shall not have a negative effect on the pilot. Constraint: $r < 2.8g$
2. Retardation force (F_{ret}). The retarding force shall not exceed the structural limitations of the aircraft. Constraint: $F_{ret} < F_{max}$.
The maximum allowed forces (F_{max}) are defined for several aircraft masses and engaging velocities in [USAF, 1986]. Force constraints for combinations of masses and velocities other than those given in [USAF, 1986] are obtained using interpolation and extrapolation.
3. Stopping distance (d). The braking distance of the aircraft shall not exceed the length of the runway. Constraint: $d < 335m$

A violation of one or more of these constraints is defined as a failure. This is a pessimistic failure classification, in the sense that not all runs which according to this classification were failures would have turned out to be critical in reality. For instance, in most cases a retardation of up to $3g$ will not significantly damage the aircraft or injure the pilot. The duration of a typical, failure-free, arrestment ranges from about 5 seconds (low kinetic energy) up to about 15 seconds (high kinetic energy).

3.2.4 Adjustments for Example Studies

When performing experiments, real aircraft could obviously not be used. Instead, an environment simulator was constructed which simulates the sensors and actuators, and the incoming aircraft (see Fig. 3.5). One simplification made in the environment simulator is that the slave node is removed. The simulator only considers the master node. Thus, the set point pressure calculated for the pressure valve on the tape drum of the master node is also used as the set point pressure for the other tape drum (formerly handled by the slave node). Moreover, it is assumed that both tape drums exhibit the same behavior. Thus, only the pressure sensor of the master tape drum is simulated.

The environment simulator also simulates the incoming aircraft and takes the aircraft mass (in kg) and incoming velocity (in m/s) as simulation parameters. During

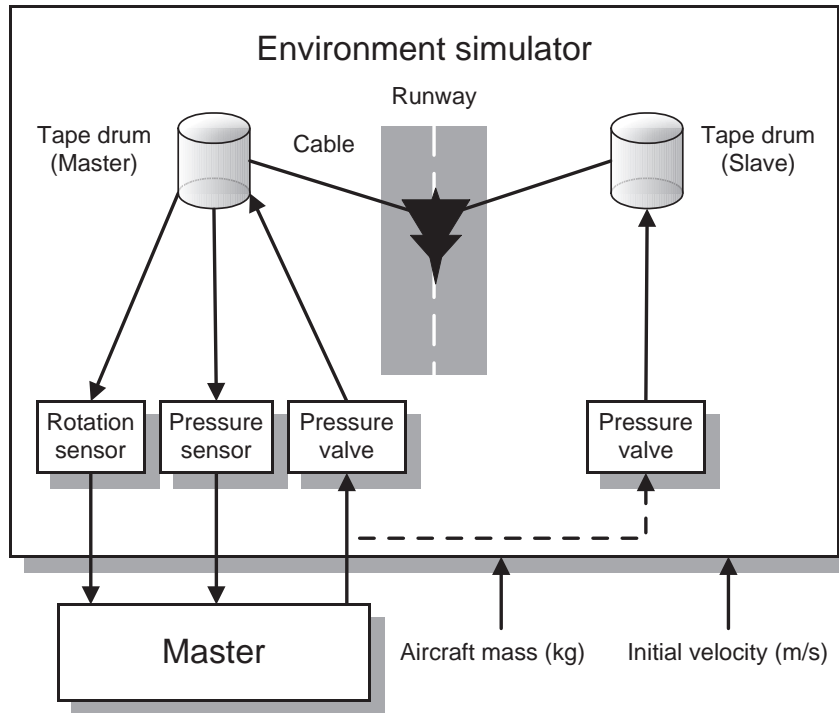


Figure 3.5: The aircraft arrestment system and environment simulator

execution, the simulator feeds input data, in the form of sensor data, to the master node and uses the output of the node, in the form of actuator data, to drive the simulation.

As the slave node was not used by the environment simulator and the communication between the master node and the slave node was only informational as far as the master node was concerned, it was decided to take out that part of the master node software which handled the communication. This was done to gain more CPU resource on the original hardware setup which could be used for evaluating error detection and recovery mechanisms instead.

The setup described above was used for experiments conducted using the FIC³ tool (see [Christmansson and Rimén, 1997] and [Christmansson *et al.*, 1998]) which was the main tool for evaluating the error detection and recovery mechanisms proposed in Chapter 4. When using the target system with PROPANE (Propagation Analysis Environment, described in Chapter 5), further adjustments had to be made as PROPANE is a tool intended for use on desktop computers, as opposed to FIC³ which operated on an actual implementation of the target system. The actual software of the master node was ported to run on a Windows-based computer. As the scheduling is slot-based and non-preemptive there is, from the software viewpoint, no difference in running on the actual hardware or running on a desktop computer.

Some glue software was developed to simulate registers for A/D-conversion, timers, counter registers, etc., accessed by the application. The environment simulator was also ported, so the environment experienced by the system in the first setup and the desktop setup was identical.

CHAPTER 4

Software Mechanisms for Handling Data Errors

Failure is not an option!

— Gene Krantz, NASA Flight Director, Apollo 13

Consumer products, such as automobiles, are safety-critical systems that traditionally require low-cost solutions to engineering problems. Thus, with increased use of computer control in such systems, low-cost solutions for dependability are required. To this end, this chapter describes software implemented approach for error detection and error recovery focusing on handling data errors, i.e., erroneous values in system state (signals and variables). The approach consist of a set of parameterized software mechanisms for detection and recovery of data errors. The detection part of the mechanisms is based on the established concept of *executable assertions*, and the recovery part is a “best-effort” approach called *forced validity*. In order to facilitate easy and rigorous insertion of the mechanisms into embedded modular software, a signal classification scheme is introduced. This scheme lets a system designer classify the signals of a software system and then choose appropriate mechanisms to protect them. The chapter also contains evaluations of the proposed mechanisms.

4.1 Introduction

Fault-tolerance is no longer required only in high-end systems such as aircraft, nuclear power plants or spacecraft. Consumer products, such as automobiles, are increasingly dependent on electronics and software and require low-cost techniques for achieving fault-tolerance. Low-cost in this sense means that these techniques are inexpensive to develop and that the product is (relatively) inexpensive to produce.

The first step in tolerating the effects of faults is to detect the symptoms of faults, i.e. the errors. Several techniques and methods have been proposed for error detection. An N-Version-Programming (NVP) style approach to error detection is achieved by running several versions or variants of the system in parallel and then compare their results [Avizienis, 1985]. If the results differ, an error must have occurred in at least one of the versions. This approach is very effective but tends to be also very expensive. A more inexpensive way of error detection is to explicitly check for errors in the system-state. Several techniques for such self-tests have been proposed (e.g., [Mahmood *et al.*, 1984], [Rabéjac *et al.*, 1996], and [Stroph and Clarke, 1998]), but often little is known about their effectiveness.

Most self-tests are based on the concept of executable assertions (see, e.g., [Hecht, 1976] and [Saib, 1978]). Executable assertions are usually statements, that can be made about the variables in a program and can potentially detect any error in internal data caused by either software faults or hardware faults. These statements are executed in on-line tests to see if they hold true. If they do not, an error has occurred and processes for assessment and recovery may be invoked. In addition to on-line error detection, executable assertions may be used during the development of a system for testing purposes [Andrews, 1979], and to assess system vulnerability.

Self-tests, as for instance executable assertions, also play major roles in software fault tolerance structures such as Recovery Blocks (RB) [Randell and Xu, 1995] and its variants (e.g., Consensus Recovery Blocks in [Scott *et al.*, 1983] and Distributed Recovery Blocks in [Kim, 1989]), and other structures (see, e.g., N-Self-Checking-Components in [Laprie *et al.*, 1987], and N-Copy-Programming or Retry Blocks in [Ammann and Knight, 1988]).

The effectiveness of executable assertions is highly application dependent. In order to develop tests with high error detection coverage, the developers require extensive knowledge of the system. Introducing rigorous ways of defining the statements used for executable assertions, or even better, providing generic mechanisms that can be instantiated by parameters alone, reduce the importance of this drawback.

Once an error has been detected, attempts to recover the system can be initiated. In the NVP-system mentioned above, errors are tolerated by masking them rather

than actively changing an erroneous system state. However, as already established, NVP-systems are very expensive and thus are not considered here. Instead, the focus will be on recovery where an erroneous state is corrected, or at least made acceptable (according to some specification).

In this chapter, parameterized mechanisms for detection and recovery of data errors are introduced and evaluated with regard to their detection and recovery capabilities. The mechanisms operate at the signal level, meaning that only one signal/variable is tested in each individual test routine. The detection part of the mechanisms is based on the executable assertion concept. For each signal a set of validity constraints are set up and if a signal does not comply to those constraints, this is considered to be an error. The proposed recovery part of the mechanisms is called *forced validity* and attempts to remove an erroneous signal value by forcing it into the valid domain of the signal. A basic process using an experience/heuristic approach for selecting where to incorporate the mechanisms into a software system is also discussed.

In order to evaluate the error detection and recovery capabilities of the proposed mechanisms a case study using error injection experiments on the target system described in Chapter 3 was performed. In this study, two separate evaluations were conducted: i) an evaluation of the error tolerance capabilities (i.e., the combination of error detection and error recovery) of the proposed mechanisms, and ii) the error detection capabilities (coverage and latency) of the mechanisms.

Even though the mechanisms may handle errors induced by software faults as well as hardware faults, the case study concentrates on errors induced by hardware faults.

The first evaluation shows that the failure rate for errors injected into the monitored signals was reduced by 32.56%. However, for errors injected into random locations in memory (including system stack and CPU registers) the reduction in failure rate was only 4.69%.

The results of the second evaluation show that given that an error is present in a monitored signal, and that this error leads to system failure, the detection probability is over 99%. For errors injected into random locations in the memory areas of the target system, the errors that caused system failure were detected with a probability of over 81%. The presented technique is therefore a viable candidate for error detection with reasonably high detection coverage if costs have to be kept low.

The remainder of this chapter is organized as follows: Section 4.2 describes the adopted model for modular software. In Section 4.3 is the signal classification from which the proposed error detection and recovery scheme in Section 4.4 is derived. A basic approach for selecting locations and parameters is discussed in Section 4.5.

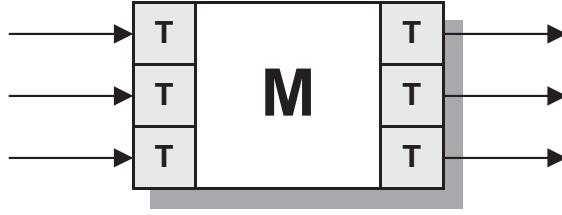


Figure 4.1: A software module *M* with executable assertion at its inputs and outputs

Section 4.6 introduces the case study performed for evaluating the mechanisms. The first evaluation is covered in Section 4.7. A discussion on error detection coverage and error propagation in Section 4.8 precedes the second evaluation in Section 4.9. Finally, Section 4.10 summarizes this chapter and draws conclusions.

4.2 Executable Assertions in Modular Software

In this thesis, the adopted system model is that of modular software where the different modules are interconnected using some kind of signaling (as described in Chapter 3). Executable assertions operating at the signal level fit very well into this system model. To illustrate this, consider the software module, *M*, in Fig. 4.1. This module is equipped with tests at the input signals and at the output signals. These tests are executable assertions, i.e., small snippets of code checking the validity of the signals.

When input arrives, it is subjected to executable assertions determining whether they are acceptable or not. Output from calculations may also be tested to see if the results seem acceptable. Should an error be detected, measures can be taken to recover from the error, and the signal can be returned to a valid state (although it may still have a value different from what it would have had if the system had not suffered from errors).

Error detection in the form of executable assertions can potentially detect any error in internal data caused by either software faults or hardware faults (as stated in [Leveson *et al.*, 1990]). However, one of the main drawbacks of executable assertions, and indeed of all kinds of acceptance tests, is that they are very application specific. One way of lessening the impact of this specificity is to devise a rigorous way of classifying the signals that are to be tested. Parameterized mechanisms can then be devised for entire classes of signals and need not be tailor-fitted for each individual signal. In the next section, we introduce an approach for dividing signals in software into different classes.

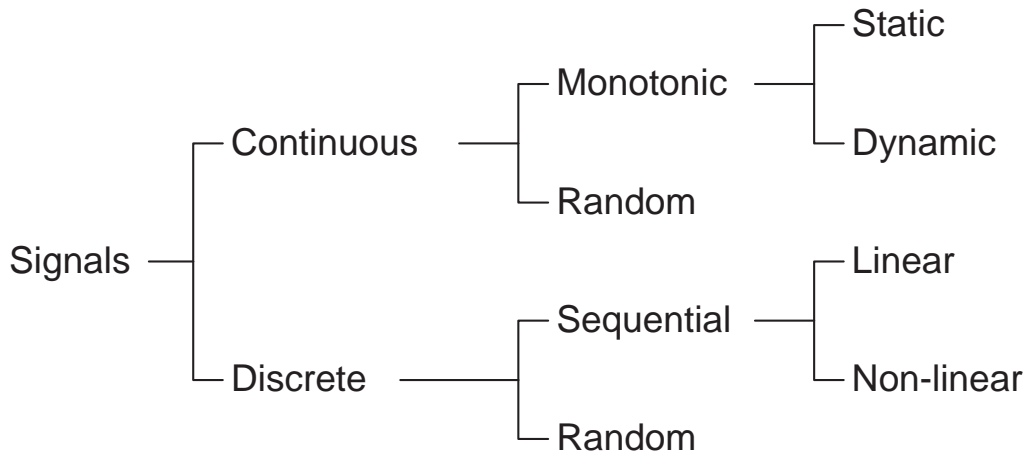


Figure 4.2: The signal classification scheme

4.3 Signal Classification: Taking an Abstract View on Data

In this section we will present a scheme for classifying signals in software which will help when determining the valid domain for the signals. For each class of signals we will then devise a mechanism for error detection and error recovery. The classification scheme used in this investigation is shown in Fig. 4.2.

The two main categories in the classification scheme are *continuous* and *discrete* signals. These categories have sub-categories that further classify the signal. For every signal class we can set up a specific set of constraints, such as boundary values and rate limitations, which are then used in the executable assertions. In order to enable a signal to have different behaviors during different modes of operation in the system, a signal may have one set of constraints for each such mode. Which set of constraints are to be used is defined by the current mode of the signal/system. During operation, signal values are repeatedly tested to check whether they violate the defined constraints or not. If they do, an error has been detected and error recovery can be initiated.

Next, we will discuss the two main signal classes in detail and set up the parameters that are used for constraining the behavior of the signals.

4.3.1 Continuous Signals

The continuous signals are often used to model signals in the environment that are of continuous nature. Such signals are typically representations of physical signals such as temperatures, pressures or velocities.

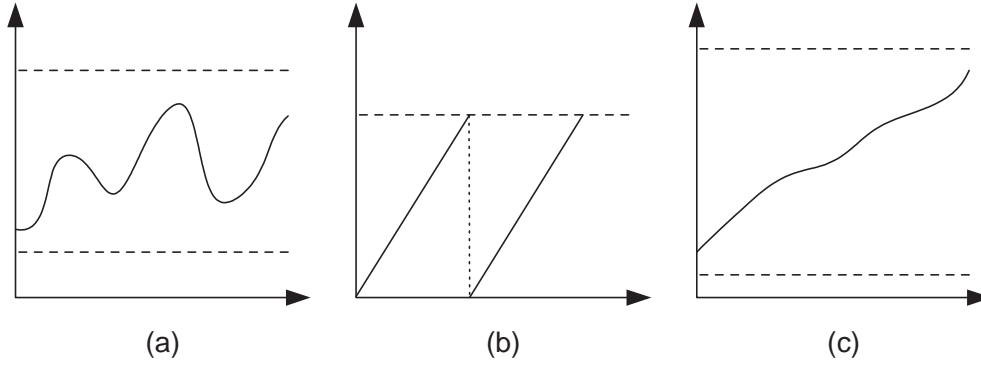


Figure 4.3: Continuous signals: (a) random, (b) static monotonic (with wrap-around), (c) dynamic monotonic

The continuous signals can be divided into monotonic and random continuous signals. Monotonic signals must either increase or decrease their value monotonically and cannot, for example, increase between the first and the second test and then decrease between the second and the third test. However, they may be allowed to remain unchanged between tests. The monotonic signals can have either a static rate or a dynamic rate. A signal with static rate must either increase or decrease its value with a given constant rate. A signal with dynamic rate, however, can change at any rate that is within the specified range. The random continuous signals may decrease or increase (or remain unchanged) between tests (that is, they may randomly increase or decrease between tests).

Also, a signal may be allowed to wrap around, i.e., when it has reached its maximum or minimum value, it may continue “on the other side”. This is visualized in Fig. 4.3, which shows examples of the three types of continuous signals.

For the proposed error detection and recovery mechanisms, we assign to each continuous signal a set P_{cont} containing seven different parameters: s_{max} (maximum value), s_{min} (minimum value), $r_{min,incr}$ (minimum increase rate), $r_{max,incr}$ (maximum increase rate), $r_{min,decr}$ (minimum decrease rate), $r_{max,decr}$ (maximum decrease rate), and w (wrap-around allowed/not allowed). The parameters are illustrated in Fig. 4.4(a). Here we have, at time t' , a signal value of s' . At time t , i.e., the next sample tick of the signal, we have the signal value s . As illustrated in Fig. 4.4(a), s is valid if $s' + r_{min,incr} \leq s \leq s' + r_{max,incr}$ or $s' - r_{max,decr} \leq s \leq s' + r_{min,incr}$. At this point we have only considered static parameters, but dynamic parameters, as in [Clegg and Marzullo, 1997] or [Stroph and Clarke, 1998], may also be considered.

Each signal class imposes certain constraints on these parameters. For both stat-

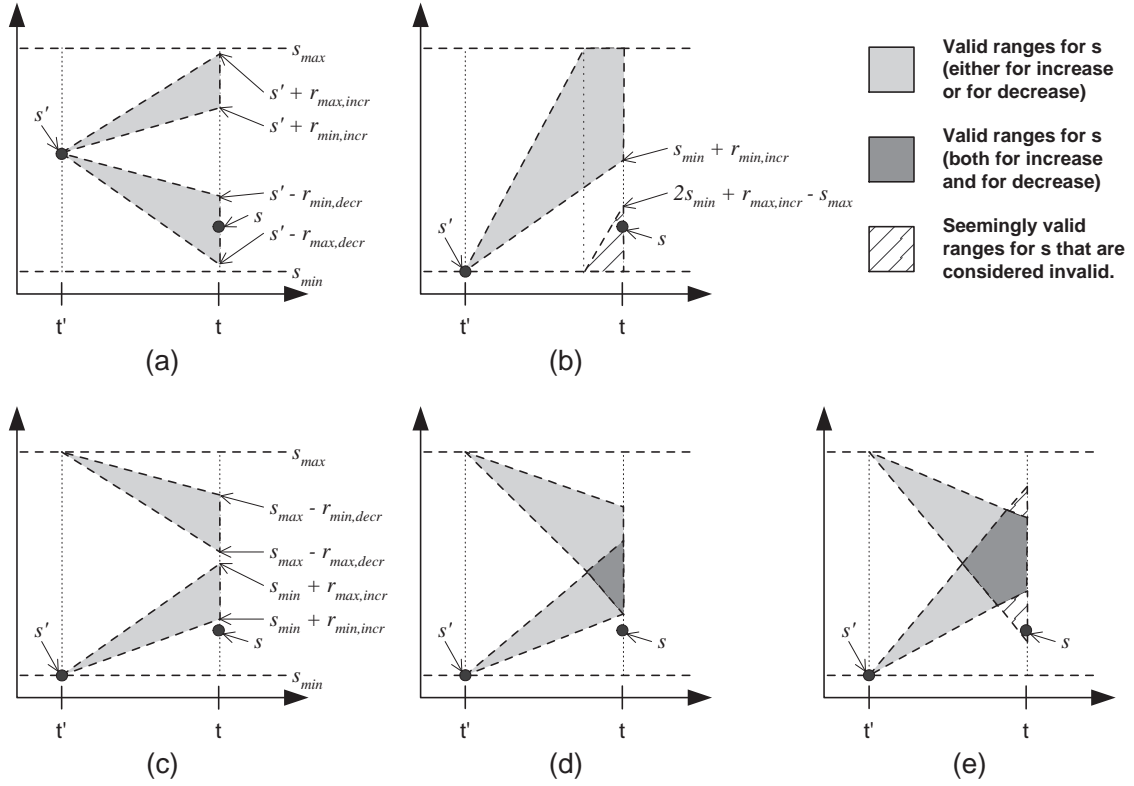


Figure 4.4: The parameters of continuous signals and reasons for constraining them

ically and dynamically increasing monotonic signals the change rate limits for decrease are set to zero (i.e., $r_{max,decr} = r_{min,decr} = 0$). However, for statically increasing monotonic signals, the change rate limits for increase are set to be identical (i.e., $r_{max,incr} = r_{min,incr} > 0$), whereas for dynamically increasing monotonic signals the change rate limits for increase differ (i.e., $r_{max,incr} > r_{min,incr} > 0$). For a monotonically decreasing signals, the opposite is set analogously. For random continuous signals, i.e., signals that can either increase or decrease between consecutive samples, we have non-zero values for all change rate limits (i.e., $0 < r_{min,incr} \leq r_{max,incr}$ and $0 < r_{min,decr} \leq r_{max,decr}$).

In addition to these constraints there are several other constraints we put on the parameters in order to deal with situations where validity of a signal value is undecidable. Consider the case depicted in Fig. 4.4(b). Here we have a signal where wrap-around is allowed and we have signal value $s' = s_{min}$ at time t' (we have chosen to show this example with $s' = s_{min}$ as this makes the illustrations easier to understand—the principle is, however, the same for any value of s') and signal value $s > s_{min}$ at time t . We can see that the signal value has increased but the

change is less than the minimum increase rate, i.e., $s < s' + r_{min,incr}$. This alone would make us consider the signal value invalid. However, if we take into account the fact that wrap-around is allowed, we can in this case see that s could be considered valid if the increase is so high that the value has actually wrapped around. This situation could occur if $r_{max,incr} > (s_{max} - s_{min})$. Therefore, in order to avoid this situation, we set a constraint on the maximum increase rate such that $r_{max,incr} \leq (s_{max} - s_{min})$. This means that the maximum allowed increase is the entire range from the minimum value to the maximum value. If wrap-around is not allowed, this constraint is actually implied. The same line of argumentation can be made for decrease rates, thus we set a constraint on the maximum decrease rate such that $r_{max,decr} \leq (s_{max} - s_{min})$.

Allowing wrap-around generates more situations where ambiguities may arise. Consider a case shown where we have a continuous random signal, i.e., it can increase or decrease randomly between two consecutive samples of the signal. Assume that the signal has the value $s' = s_{min}$ at time t' (again, this value is chosen as the illustrating figures then are easier to understand—the principle is the same, however, for any value of s') and the value s such that $s' < s < s' + r_{min,incr}$ at time t . As long as the situation is as depicted in Fig. 4.4(c), where the valid ranges at time t do not overlap at all, there are no ambiguities when checking the validity of the value—it is always considered erroneous. In the situation depicted in Fig. 4.4(d) the valid ranges at time t do overlap, but s can still safely be flagged as erroneous.

In the situation depicted in Fig. 4.4(e), we have valid ranges at time t that overlap and also “spill over”, i.e., the parameters of the signal are such that in this situation, where $s' < s < s' + r_{min,incr}$, the value s could still be considered valid as it falls into the valid range if one assumes that the signal was decreased instead of increased. Thus, there are ambiguities in deciding whether the signal's value s here is valid or not. In order to avoid this situation, we put constraints on the parameters. The ambiguities arise if we have random continuous signals where wrap-around is allowed and $s_{max} - r_{max,decr} < s_{min} + r_{min,incr}$, i.e., when $r_{max,decr} + r_{min,incr} > s_{max} - s_{min}$. Therefore, we set a constraint such that $r_{max,decr} + r_{min,incr} \leq s_{max} - s_{min}$. A similar line of arguments can be derived when $s_{min} + r_{max,incr} > s_{max} - r_{min,decr}$. This situation instead yields the constraint $r_{max,incr} + r_{min,decr} \leq s_{max} - s_{min}$.

Now we have derived several constraints for the various classes of continuous signals. In Table 4.1 is a summary of these constraints. It should be noted that the situations that are labeled as ambiguous here can, by the designer of the executable assertions, instead be considered valid (this is perhaps more of a policy issue rather than a correctness issue).

Table 4.1: Parameter constraints for continuous signal classes

Signal class	Parameter constraints
All	$s_{min} \leq s_{max} \wedge$ $w = \text{allowed/not allowed} \wedge$ $0 \leq r_{min,incr} \leq r_{max,incr} \leq s_{max} - s_{min} \wedge$ $0 \leq r_{min,decr} \leq r_{max,decr} \leq s_{max} - s_{min}$
Static monotonic	Decreasing signals: $0 = r_{min,incr} = r_{max,incr} \wedge$ $0 < r_{min,decr} = r_{max,decr}$ Increasing signals: $0 = r_{min,decr} = r_{max,decr} \wedge$ $0 < r_{min,incr} = r_{max,incr}$
Dynamic monotonic	Decreasing signals: $0 = r_{min,incr} = r_{max,incr} \wedge$ $r_{min,decr} < r_{max,decr}$ Increasing signals: $0 = r_{min,decr} = r_{max,decr} \wedge$ $r_{min,incr} < r_{max,incr}$
Random	$r_{max,decr} + r_{min,incr} \leq s_{max} - s_{min} \wedge$ $r_{max,incr} + r_{min,decr} \leq s_{max} - s_{min}$

4.3.2 Discrete Signals

Discrete signals are allowed to take on a set of discrete values. They often contain information on the settings of an operator panel or the operation mode of the system. Actually, all signals containing some kind of state information, internal or external to the system, may be classified as discrete signals. For instance, execution sequences that must be followed in a certain order, or state machines with a number of states and a number of transitions between the states, may be modeled as discrete signals. The discrete signals are divided into sequential and random signals.

A sequential signal has constraints on how it may change its value from any given other value, i.e., the order of change is restricted. Sequential signals are divided into linear and non-linear signals. Linear signals must traverse their valid domain in a fixed predefined order, one value after another. For instance, the execution sequence mentioned above could be modeled as a linear signal. Non-linear signals traverse their valid domain in predefined ways. Random signals are allowed to make any transition from one value to another within the valid domain of the signal.

For the proposed error detection mechanisms we assign to each signal a set P_{disc} containing the following parameters: D (the set of valid values) and $T(d)$ (the set of valid transitions from element d in D ; there is one such set for each element in D).

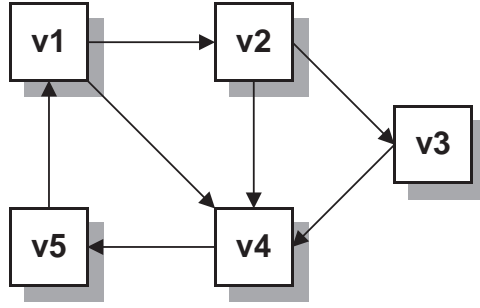


Figure 4.5: Example state diagram for a non-linear sequential discrete signal

A typical example of a discrete signal is the state variable of a state machine. From any given state, the machine may transit to a (fixed) number of other states (maybe including the current state). Consider for example the state machine shown in Fig. 4.5. There are five states ($v1$ through $v5$) and a number of transitions between these states. The valid domain is therefore $D = \{v1, v2, v3, v4, v5\}$ and the transition sets are $T(v1) = \{v2, v4\}$, $T(v2) = \{v3, v4\}$, $T(v3) = \{v4\}$, $T(v4) = \{v5\}$, and $T(v5) = \{v1\}$.

4.3.3 Signal Modes

The behavior of a signal may differ between different phases of operation of the system. Therefore, a signal can have different modes. A specific set of constraints is generated for each such mode, i.e. a signal with several modes has one parameter set P_{cont} or P_{disc} for each mode. The set used in a certain mode m is $P_{cont}(m)$ or $P_{disc}(m)$. Mode variables (m in this case) can be classified as discrete signals in themselves, so that error detection may be implemented for them as well. Modes may also be used to model certain dependencies between signals. That is, if the behavior of signal A is limited due to the operational mode of signal B , these two signals can be grouped by means of signal modes representing this dependency. Furthermore, using different modes may increase the possibility of detecting errors.

4.4 Mechanisms for Error Detection and Recovery

Error detection is performed using the configuration parameters of the signals to build executable assertions. An error in a signal is detected as soon as the signal violates the constraints given by the configuration parameters. For error recovery, we have introduced an approach called *forced validity*, which assigns a recovery value within the valid domain of the signal that is either as close to the actual, but

erroneous, value (used for continuous signals) or a default value (used for discrete values).

The executable assertions used for continuous values are described in Tables 4.2 and 4.3. The executable assertions for discrete values are described in Table 4.4. In these tables, s is the current signal value, s' is the previous signal value and s_r is the recovered signal value.

For continuous signals, the tests for detecting an erroneous value are divided into three levels with increasing granularity (columns *Test Level 1*, *Test Level 2* and *Test Level 3*). If a condition at one level is evaluated to *true*, the test continues with the conditions at the next lower level. If a condition at one level is evaluated to *false*, the test continues with the next test at the same level. If there are no more tests at one level, the signal value is considered to be correct (according to the parameters set for that particular signal). If a condition at level 3 is evaluated to *true* then an error has been detected and the corresponding expression for error recovery is used.

The first two tests for continuous signals have a special status, they are always performed before any of the other tests are performed and should an error be detected at this point, the remaining tests will be performed using the recovered signal as the test value. That is, if a signal value above s_{max} is being tested it will first be set to s_{max} before the remaining tests are performed.

The tests requiring a previous value, s' , are only performed from the second sample point and on. This means that the first time a signal is checked it is only made sure that it is within its maximum and minimum values.

Please note here that the last set of tests performed in case a signal has not changed its value, i.e., when $s = s'$, covers the case when a signal is not allowed to remain the same between two consecutive samples (unless, of course, it already is at its maximum or minimum value and is not allowed to wrap around). If a signal is forced to either increase or decrease, there is choice to be made regarding the recovery mechanisms for this error. The two options for recovery here are to either increase or decrease the value. In our studies we have chosen to always increase the value if required. One may also consider more elaborate recovery schemes where the decision of whether to increase or decrease the value depends on the direction the signals had between the previous sample and the sample before that. However, for simplicity, a history of only one sample is considered here.

The executable assertions for discrete signals are less complicated than those for the continuous signals. Every sub-class of discrete signals has its own set of error detection and recovery mechanisms, as shown in Table 4.4. The tests basically check whether the signal value is in the valid domain, D , of the signal and that the transition made from the previous signal value is valid. The first time a discrete

Table 4.2: Expressions for error detection and recovery for continuous signals

Error Codes: E_MaxV/E_MinV = Signal above maximum/below minimum value, E_MinIR/E_MinDR = Signal change below minimum increase/decrease rate, E_MaxIR/E_MaxDR = Signal change above maximum increase/decrease rate				
Test Level 1	Test Level 2	Test Level 3	Error Code	Recovery (Forced Validity)
The following two tests are always performed. If an error is detected, the remaining tests are done with the recovered signal value instead of the original signal value. The first time a signals is tested, these two tests are the only ones performed.				
$s > s_{max}$			E_MaxV	$s_r \leftarrow s_{max}$
$s < s_{min}$			E_MinV	$s_r \leftarrow s_{min}$
If a condition is true at one level, the test continues at the next level. If the condition is false, the test continues with the next test at the same level. If there are no more tests, the signal is considered to be valid. These tests are performed only if there is a previous signal value, i.e., from the second sample and on.				
$s > s'$	$s - s' > r_{max,incr}$	$s' - s_{min} > r_{max,decr} \vee$ $w = \text{ not allowed}$	E_MaxIR	$s_r \leftarrow s' + r_{max,incr}$
		$(s' - s_{min}) + (s_{max} - s) > r_{max,decr}$	E_MaxDR	$s_r \leftarrow (s_{max} - r_{max,decr}) + (s' - s_{min})$
		$(s' - s_{min}) + (s_{max} - s) < r_{min,decr}$	E_MinDR	$s_r \leftarrow (s_{max} - r_{min,decr}) + (s' - s_{min})$
	$s - s' < r_{min,incr}$	$s < s_{max} \vee$ $w = \text{ allowed}$	E_MinIR	if $s_{max} - s' \geq r_{min,incr}$ $s_r \leftarrow s' + r_{min,incr}$ else if $w = \text{ allowed}$ $s_r \leftarrow (s_{min} + r_{min,incr}) - (s_{max} - s')$ else $s_r \leftarrow s_{max}$
$s < s'$	$s' - s > r_{max,decr}$	$s_{max} - s' > r_{max,incr} \vee$ $w = \text{ not allowed}$	E_MaxDR	$s_r \leftarrow s' - r_{max,decr}$
		$(s_{max} - s') + (s - s_{min}) > r_{max,incr}$	E_MaxIR	$s_r \leftarrow (s_{min} + r_{max,incr}) - (s_{max} - s')$
		$(s_{max} - s') + (s - s_{min}) < r_{min,incr}$	E_MinIR	$s_r \leftarrow (s_{min} + r_{min,incr}) - (s_{max} - s')$
	$s' - s < r_{min,decr}$	$s > s_{min} \vee$ $w = \text{ allowed}$	E_MinDR	if $s' - s_{min} \geq r_{min,decr}$ $s_r \leftarrow s' - r_{min,decr}$ else if $w = \text{ allowed}$ $s_r \leftarrow (s_{max} - r_{min,decr}) + (s' - s_{min})$ else $s_r \leftarrow s_{min}$
This list of tests is continued in Table 4.3.				

Table 4.3: Expressions for error detection and recovery for continuous signals, continued

Error Codes: E_MaxV/E_MinV = Signal above maximum/below minimum value, E_MinIR/E_MinDR = Signal change below minimum increase/decrease rate, E_MaxIR/E_MaxDR = Signal change above maximum increase/decrease rate				
Test Level 1	Test Level 2	Test Level 3	Error Code	Recovery (Forced Validity)
This is a continuation of the list of tests in Table 4.2.				
$s = s'$	$r_{min,incr} = 0 \wedge$ $r_{max,incr} = 0 \wedge$ $r_{min,decr} > 0$	$s > s_{min} \vee$ $w = \text{allowed}$	E_MinDR	if $s' - s_{min} \geq r_{min,decr}$ $s_r \leftarrow s' - r_{min,decr}$ else if $w = \text{allowed}$ $s_r \leftarrow (s_{max} - r_{min,decr}) + (s' - s_{min})$ else $s_r \leftarrow s_{min}$
	$r_{min,decr} = 0 \wedge$ $r_{max,decr} = 0 \wedge$ $r_{min,incr} > 0$	$s < s_{max} \vee$ $w = \text{allowed}$	E_MinIR	if $s_{max} - s' \geq r_{min,incr}$ $s_r \leftarrow s' + r_{min,incr}$ else if $w = \text{allowed}$ $s_r \leftarrow (s_{min} + r_{min,incr}) - (s_{max} - s')$ else $s_r \leftarrow s_{max}$
	$r_{min,decr} > 0 \wedge$ $r_{min,incr} > 0$	$s < s_{max} \vee$ $w = \text{allowed}$	E_MinIR	if $s_{max} - s' \geq r_{min,incr}$ $s_r \leftarrow s' + r_{min,incr}$ else if $w = \text{allowed}$ $s_r \leftarrow (s_{min} + r_{min,incr}) - (s_{max} - s')$ else $s_r \leftarrow s_{max}$
	If the signal is forced to either increase or decrease, recovery may be to always increase or always decrease if the signal has not changed. The policy chosen here is to always increase, as shown above. If the policy to always decrease would be chosen instead, the recovery would be performed as shown below.			
		$s > s_{min} \vee$ $w = \text{allowed}$	E_MinDR	if $s' - s_{min} \geq r_{min,decr}$ $s_r \leftarrow s' - r_{min,decr}$ else if $w = \text{allowed}$ $s_r \leftarrow (s_{max} - r_{min,decr}) + (s' - s_{min})$ else $s_r \leftarrow s_{min}$

Table 4.4: Error detection and recovery for discrete signals

Error Codes: E_InvV = Invalid value, E_InvT = Invalid transition			
Signal Class	Test	Error Code	Recovery (Forced Validity)
Random	$s \notin D$	E_InvV	$s_r \leftarrow d_{def} \in D$
Linear	$s \notin D$	E_InvV	$s_r \leftarrow T(s')$
Sequential	$s \neq T(s')$	E_InvT	
Non-linear	$s \notin D$	E_InvV	$s_r \leftarrow d_{def} \in T(s')$
Sequential	$s \notin T(s')$	E_InvT	

signal is tested, only the first test for each class is performed (i.e., only membership in D is checked). Recovery of discrete signals is here made by setting the signal to a default value either in the valid domain, D , or in the transition set of the previous signal value, $T(s')$. Note that for linear sequential discrete signals, each transition set, T , only contains one element. For discrete signals, the assertions are always executed.

4.5 Finding Locations and Defining Parameters

A number of different methods may be used to determine which signals should be monitored and where the executable assertions should be placed. From system design, the software should already be divided into functional blocks. In safety-critical systems, FMECA (Failure Mode Effect and Criticality Analysis) is widely used as a method for identifying the safety critical parts of the system and assessing the consequences of failures in these parts.

Parameter information may be obtained by the characteristics of the system itself. For instance, sensors naturally have a time constant dictating the maximum rate of change for the data provided by that sensor. Properties of the physical surroundings of the systems are also a source of parameter values. For discrete signals, typical sources of information are allowed settings on user panels, or internal state machines.

The process of gathering information for parameter values for executable assertions forces developers to review the system they have developed. This may assist in identifying contradicting specifications and/or parts that have not yet been properly analyzed. The following is the process used in the case study presented here for equipping a system with the error detection and recovery mechanisms described in this chapter:

1. Identify the input and output signals of the system.
2. Identify the signal pathways from each input signal through the system and to one or more output signals.
3. Identify internally generated signals that have a direct influence on intermediate and output signals.
4. Determine, e.g., by using FMECA, which of the identified signals are the most crucial for flawless operation of the system and should therefore be guarded by error detection and recovery mechanisms.
5. Classify each signal found in (4) according to the scheme described in Section 4.3.
6. Determine values for the characterizing parameters of the signals. Remember that a signal may behave differently for different modes of operation in the system.
7. Decide on locations for the mechanisms.
8. Incorporate the mechanisms in the system.

This process is a very simple approach relying on the experience of the system designer. In Chapter 6 of this thesis a more rigorous approach based on the analysis of error propagation and effect is introduced. However, for now we will use the process presented here.

4.6 Evaluating the Capabilities of the Mechanisms

As an assessment of the effectiveness of the error detection and recovery mechanisms when employed in an embedded control system, we conducted a series of evaluations using error injection. The first evaluation (referred to as *Evaluation 1*) focused on the obtained error tolerance if the proposed mechanisms are incorporated into a system, and the second evaluation (referred to as *Evaluation 2*) focused on the error detection capabilities of the mechanisms. The target system used is described in Chapter 3. This section describes the special fittings that had to be made to the target system for using it with the experiment tool used in this case study.

The software of the slave node is slightly different from that of the master node. No calculations of set point values for the applied pressure are performed. The slave node simply receives a set point value from the master node, which it then applies

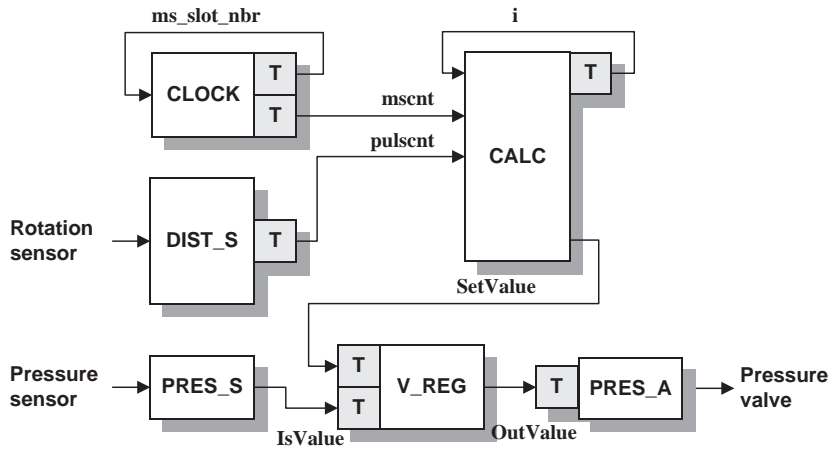


Figure 4.6: The selected signals and the locations of the corresponding executable assertions. In *Evaluation 1*, the signal *ms_slot_nbr* was not included in the setup.

to its tape drum. The modules existing also in the slave node are PRES_S, V_REG, CLOCK, and PRES_A. The modules DIST_S and CALC are not present.

The system specifications in [USAF, 1986] set a number of physical constraints within which the system must operate. These constraints are described in Chapter 3. In the experiments described in this chapter, this failure classification has been used when obtaining coverage estimates of error detection mechanisms.

4.6.1 Software Instrumentation

Using the process described in Section 4.5, we identified 7 signals (of a total of 24 signals) in the target system that are service critical, i.e., essential for providing proper service (see Fig. 4.6). The selected signals were classified according to our classification scheme (see Table 4.5).

Table 4.5: Classification of signals in the target system

Signal	Producer	Consumer	Test Location	Class
SetValue	CALC	V_REG	V_REG	Co/Ra
IsValue	PRES_S	V_REG	V_REG	Co/Ra
i	CALC	CALC	CALC	Co/Mo/Dy
pulsent	DIST_S	CALC	DIST_S	Co/Mo/Dy
ms_slot_nbr	CLOCK	CLOCK	CLOCK	Di/Se/Li
mscnt	CLOCK	CALC	CLOCK	Co/Mo/St
OutValue	V_REG	PRES_A	PRES_A	Co/Ra

In Table 4.5, the *Producer* is the originating module of a signal, the *Consumer* is

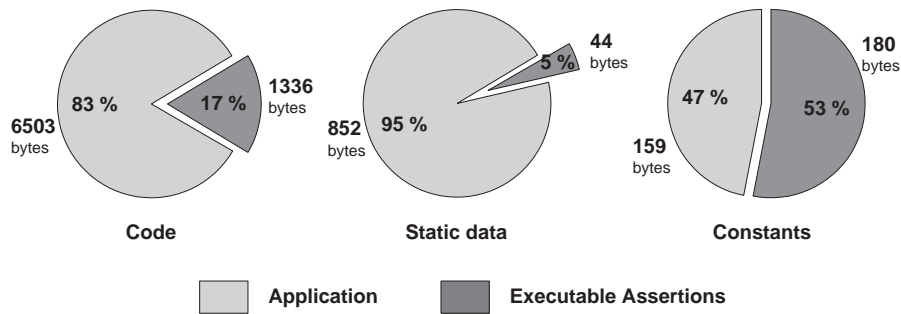


Figure 4.7: The overhead in code size, static data (RAM) and constants (ROM) incurred by the mechanisms.

the receiving module, and the *Test Location* is where the executable assertions were placed. Whether a mechanism was placed at the producer or consumer side was just a matter of finding the place where implementation would be the least complicated. The *Class* is how the signal was classified (Co = continuous, Ra = random, Mo = monotonic, St = static rate, Dy = dynamic rate, Di = discrete, Se = sequential, Li = linear). In Fig. 4.6 are the selected signals and their corresponding mechanisms.

The overhead in memory resources of the instrumentation is shown in Fig. 4.7. Here we can see that the code of the mechanisms gives an overhead of 17% or 1336 bytes. The mechanisms are implemented as generic code in a static library. Thus, the relative overhead in code very much depends on the size of the actual application as the absolute library size in bytes will not change. On the other hand, the two overhead categories *static data* and *constants* are dependent on the number of mechanisms incorporated into the system and thus, the relative overhead will not increase or decrease as much with the (inverse of) application size. In this example, the static data gave an overhead of 5% or 44 bytes, compared to the static data used by the application, and the constants gave an overhead of 53% or 180 bytes, roughly doubling the amount of ROM needed.

At this point it is important to mention that when instrumenting the target system with mechanisms for *Evaluation 1*, it was found that there was too little space in the on-chip FLASH memory in order to add all mechanisms, so one signal had to be left unprotected. Signal *ms_slot_nbr* was selected as the victim as this was considered the least critical of the seven signals. Between *Evaluation 1* and *Evaluation 2*, the target system was modified so that all communication to the slave node was removed. This could be done without changing the characteristics of the system as the slave node was passive, i.e., it only received commands from the master node and never sent commands back to the master node. Thus, from the master node's point-of-view, there was no difference in functionality. Also, the environment simulator used only

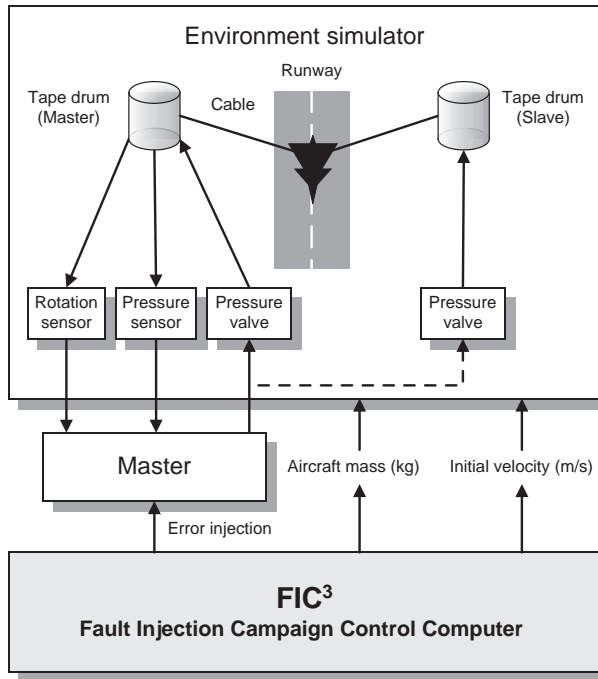


Figure 4.8: The *FIC*³ and the target system - the tool provides input to the environment simulator and injects errors in the master node of the target

considered input from the master node and replicated that for the slave node drum. Thus, the slave node did not really take part in arresting aircraft in the setup used in the evaluations. Removing the communication between the master and the slave nodes freed resources in sufficient FLASH memory (and of course lowered the total CPU load) such that the signal *ms_slot_nbr* could be equipped with a mechanism.

4.6.2 Fault Injection Environment

As seen in Fig. 4.8, the target system was hooked up to the fault injection tool *FIC*³ (the Fault Injection Campaign Control Computer, for details the reader is referred to [Christmansson and Rimén, 1997] and [Christmansson *et al.*, 1998]).

The *FIC*³ is capable of injecting errors into the target system by means of SWIFI (SoftWare Implemented Fault Injection). Specifically, before initiating an experiment run, the *FIC*³ downloads error parameters to an injection interrupt routine in the target system, which is then, during the experiment run, triggered by the *FIC*³ when the actual injection is to be performed. The error detection mechanisms report detection by setting a digital output pin on the target processor high. This is detected by the *FIC*³, which records and time-stamps the event. The injected errors consist of modifications of the memory areas where variables and signal values are stored. Pre-

vious studies have shown that injecting bit-flips into a system using SWIFI closely resembles the behavior of hardware failures (see [Rimén *et al.*, 1994]). The downloaded injection parameters for this type of error are the address and bit position.

An environment simulator acts as the barrier (i.e. cable and tape drums) and as the incoming aircraft. This simulator is initialized using test case data (mass and incoming velocity). The FIC³ triggers the simulator to start simulating an incoming aircraft. The simulator then feeds the system with sensory data (rotation sensor and pressure sensor) and receives actuator data (pressure value) from the system used for calculating new sensory data. All input to and output from the environment simulator is stored as experiment readouts and is subsequently analyzed for system failure.

4.7 Evaluation 1: Error Tolerance

In *Evaluation 1*, the goal was to assess the total error tolerance obtained by incorporating the proposed mechanisms into an embedded system. Error tolerance here is defined as the combination of error detection and error recovery, thus, if an error has been tolerated, it was first detected and then successfully recovered. In this evaluation, tolerating an error was defined as being able to avoid system failure even if an error was present in the system. It should be mentioned that a system without specific error tolerance mechanisms often exhibits a certain level of error tolerance (or robustness) due to the fact that errors can be overwritten or that the system has a built-in resiliency against errors.

4.7.1 Setup of Evaluation 1

In order to evaluate the obtained error tolerance, i.e., the combined effect of error detection and error recovery, two error sets, **ET**₁ and **ET**₂, were set up. Error set **ET**₁ contains 160 errors configured as random bit-flips in the stack area, the memory areas of the various modules or the internal registers of the processor. Most of the errors were injected in the memory areas, as the mechanisms are geared towards data errors. Errors in the stack will with a higher probability cause control flow errors (i.e., errors that change the path of execution of a program). The errors injected into memory were distributed according to size of the memory areas of the modules. Thus, a module with a larger memory area was subjected to more error injections than a module with a smaller memory area. The errors in error set **ET**₁ are meant to model transient hardware faults. Results in [Rimén *et al.*, 1994] indicate that random bit flips in memory can mimic the effect of such faults. The distribution of

errors in \mathbf{ET}_1 is shown in Table 4.6.

Table 4.6: The distribution of errors in the error set \mathbf{ET}_1

Memory Area	Size (bytes)	# Errors n_e	# Injections $n_e \cdot 25$
Stack	1008	25	625
CALC	164	55	1375
CLOCK	12	15	375
PRES_A	12	10	250
PRES_S	2	5	125
DIST_S	13	15	375
V_REG	36	15	375
Registers	3	20	500
Total	-	160	4000

The other error set, \mathbf{ET}_2 , contains errors targeting the individual signals monitored by the executable assertions. For each signal, five errors were selected giving a total of $5 \cdot 6 = 30$ errors in \mathbf{ET}_2 . These errors are bit flips in the signals, modeling data errors. No particular assumption has been made as to how these data errors occur as the goal of the evaluation is to examine the error tolerance capabilities of the mechanisms, i.e., how well the mechanisms performed, given that there was an error which they potentially could detect and correct.

All errors were injected in the master node. For each error in the error set, the system was subjected to 25 test cases, i.e., incoming aircraft, with velocity ranging uniformly from 40 m/s to 70 m/s , and mass ranging uniformly from 8000 kg to 20000 kg . For \mathbf{ET}_1 , we have $160 \cdot 25 = 4000$ different combinations $[m, v, e]$ of mass, velocity and error and for \mathbf{ET}_2 we have $30 \cdot 25 = 750$ combinations. All test cases are such that if they are run on the target system without error injection, none of the error detection mechanisms report detection. The two error sets were used on two versions of the target system: one which did not include the error tolerance mechanisms and one which did include them. Thus, a total of $(4000 + 750) \cdot 2 = 9500$ error injections were performed for the evaluation of error tolerance.

The error sets were generated by random assignment of errors to the various memory areas and signals. However, the same errors were used for both versions of the system. The error injections were time triggered and were injected with a period of 10 milliseconds (recall that most modules in the target system have a period of 7 milliseconds). Thus, errors may have been injected during the execution of the executable assertions or the execution of the recovery mechanisms.

Table 4.7: The results from the injection experiments with error set \mathbf{ET}_1

Memory Area	# Inj.	# Fails, F_o (orig.)	Success Rate (orig.)	# Fails, F_i (instr.)	Success Rate (instr.)	Reduction
Stack	625	52	0.9168	49	0.9216	5.77%
CALC	1375	57	0.9585	46	0.9665	19.30%
CLOCK	375	10	0.9733	21	0.9440	-110.10%
PRES_A	250	0	1.0000	0	1.0000	N/A
PRES_S	125	3	0.9760	10	0.9200	-233.33%
DIST_S	375	65	0.8267	43	0.8853	33.85%
V_REG	375	5	0.9867	7	0.9813	-40.00%
Registers	500	277	0.4460	271	0.4580	2.17%
Total	4000	469	.8828	447	.8883	4.69%

Table 4.8: The results from the injection experiments with error set \mathbf{ET}_2

Signal	# Inj.	# Fails, F_o (orig.)	Success Rate (orig.)	# Fails, F_i (instr.)	Success Rate (instr.)	Reduction
i	125	48	0.6160	36	0.7120	25.00%
SetValue	125	4	0.9680	3	0.9760	25.00%
OutValue	125	1	0.9920	0	1.0000	100.00%
mscnt	125	9	0.9280	7	0.9440	22.22%
pulscnt	125	63	0.4960	39	0.6880	38.10%
IsValue	125	4	0.9680	2	0.9840	50.00%
Total	750	129	.8280	87	.8883	32.56%

4.7.2 Results of Evaluation 1

In *Evaluation 1*, errors were injected into two versions of the target system: the original version, without error tolerance mechanisms, and one version instrumented with error tolerance mechanisms incorporated. Specifically, two error sets, \mathbf{ET}_1 (see Table 4.6) and \mathbf{ET}_2 , containing 160 errors and 30 errors respectively, were injected. Each individual error was injected for 25 test cases in each system. Table 4.7 shows a summary of the results for the errors in \mathbf{ET}_1 and Table 4.8 shows a summary of the results for the errors in \mathbf{ET}_2 . For each area or signal, the tables show the number of injections and the number of resulting failures. The success rate is the normalized fraction of successful runs performed by the system. The number of failures and the success rate are shown for both systems. The reduction is a measure of how well the instrumented system handled failures as compared to the original system, and is calculated as $100\% \cdot \frac{(F_o - F_i)}{F_o}$. A 100%-reduction means that all failures in the original system were handled in the instrumented system. A negative reduction means that the number of failures increased in the instrumented system.

4.7.3 Discussion of Evaluation 1

The results obtained in this evaluation are specific for the target system, the error model and the test cases we have chosen. For other systems, error models, and/or test cases the results may vary. Having said that, we can now start our discussion of the results shown in the previous section.

We injected two sets of errors into our target system: one modeling random hardware faults (\mathbf{ET}_1), where each error was a bit-flip in random areas of the system memory and CPU registers, and one modeling data errors in specific signals (\mathbf{ET}_2), targeting the signals which were fitted with error tolerance mechanisms.

The errors were injected periodically with a period of 10 milliseconds. The target system has a main period of 7 milliseconds. Therefore, the errors are likely to affect the system in a manner that cannot be said to model software faults, since such faults would most likely induce data errors with a period matching that of the system.

The results from error set \mathbf{ET}_1 show that errors injected into the stack and the registers caused approximately the same amount of failures in both systems. It may be argued that these errors are more severe than those injected into the memory area of specific software modules and would very likely lead to control flow errors. The proposed mechanisms are not aimed at detecting or recovering from such errors.

For the different software modules, the reduction in failures induced by random hardware faults varies (error set \mathbf{ET}_1 , see Table 4.7). The low overall reduction for errors injected randomly in the memory areas of the modules is mainly due to the following reasons:

1. Errors occurred with a period not matching that of the system. This increased the probability of errors occurring between the test of a signal and the usage of that signal, thereby nullifying the effect of any recovery that may have been performed.
2. Errors were injected into variables not covered by assertions and recovery mechanisms. These errors are likely to affect the system in a way that the executable assertions cannot detect.
3. Errors were injected into variables belonging to the executable assertions and recovery mechanisms. Since these mechanisms were inactive in the original system, those errors did not cause any failures, whereas in the instrumented system, where the executable assertions and recovery mechanisms were active, they caused failures.

The first point highlights a fundamental difference between software-based and hardware-based fault-tolerance techniques. Whereas the hardware-based techniques are always active and ready to handle errors, the software-based techniques are active only at certain points in time. If a data error occurs between the execution of a test on the data and the usage that data, the software-based techniques cannot detect the error, much less recover from it.

The second point shows that error detection and recovery mechanisms aimed at specific signals and data areas are not effective against errors occurring in signals not directly monitored by the mechanisms.

The last point shows the importance of separating the memory areas of error detection and recovery mechanisms from the memory areas of the application. Preferably, the mechanisms should also be located in other, more reliable, memory circuits.

The results from error set \mathbf{ET}_2 (Table 4.8) show that when the errors directly affect variables monitored by the mechanisms, the reduction in the number of resulting failures is greater than if the errors affect randomly chosen variables. These findings suggest that the locations of the detection and recovery mechanisms largely influence the degree of their success, which is consistent with findings reported in [Leveson *et al.*, 1990]. Our results indicate that error detection and recovery mechanisms should be located as close to the receiver of a signal as possible or be performed when the receiver of a signal accesses the value.

Error tolerance is a combination of first detecting an error and then recovering from it. The evaluation performed at this point is of this combination. As the results show, the error tolerance does have room for improvement. In an attempt to identify whether it the detection capabilities of the proposed mechanisms are insufficient and thus the reason behind the somewhat low reduction in system failure, another evaluation was performed, focusing on error detection. This evaluation, *Evaluation 2* is presented in the Section 4.9. First, however, a brief discussion on the process of error propagation and its implications on error detection coverage.

4.8 Error Detection Coverage and Error Propagation

The detection coverage that may be obtained with these mechanisms is very dependent on the characteristics of the errors that may occur. If we, given that an error has occurred, define the probabilities $P_{em} = Pr\{\text{error location is in a monitored signal}\}$, $P_{en} = Pr\{\text{error location is not in a monitored signal}\} = 1 - P_{em}$, $P_{prop} = Pr\{\text{error propagates to a monitored signal}\}$, and $P_{ds} = Pr\{\text{an error is detected given that the error is located in a monitored signal}\}$. The total probability of detecting an error that is present can then be written as $P_{detect} =$

$(P_{en}P_{prop} + P_{em})P_{ds}$. For a given system, the probability P_{ds} can be assessed separately from the other probabilities and is independent of the probability distributions for error occurrence and error location. A common way of performing such an assessment is by conducting error injection experiments. The second evaluation of the presented mechanisms is an attempt to assess P_{detect} and P_{ds} for a given target system (see the following sections).

4.9 Evaluation 2: Error Detection

In *Evaluation 2*, the focus is the first part of the process of tolerating errors, i.e., error detection. If an error cannot be detected, then error tolerance will not be obtained, no matter how sophisticated the error recovery mechanisms. In this evaluation, error detection was defined as successful if an injected error (or a propagated effect of it) was detected during the active operation of the target system.

4.9.1 Setup of Evaluation 2

The experimental set-up for evaluating error detection capabilities calls for two error sets for evaluation purposes. In order to assess the probability P_{ds} , as defined in Section 4.8, an error set **ED₁** containing 112 errors was created. Each error in **ED₁** is configured as a bit-flip in the monitored signals. Bit-flips can be used to model intermittent hardware faults, and it may be argued that using bit-flips in variables only may also model other faults inducing data errors in variables. Since single-bit errors are uniformly probable in all bit positions we chose to inject errors in each bit position of each signal in order to get a good estimate of the detection probability. Each signal is 16 bits long, hence, we have $7 \cdot 16 = 112$ errors in the error set (each individual error was used as an error location).

The other error set, **ED₂**, contains 200 errors configured as bit-flips in random bit positions in random locations (addresses) in application RAM (417 bytes) and stack (1008 bytes) areas, and is used to assess the total detection probability P_{detect} as described in Section 4.8. These errors were selected from a uniform distribution (both location and bit-position), and the sampling was performed with replacement. Of the 200 errors, 150 were located in application RAM areas (i.e., no unused RAM areas) and 50 in the stack area.

All errors were injected in the master node. For each error in the error set, the system was subjected to 25 test cases, i.e., incoming aircraft, with velocity ranging uniformly from 40 m/s to 70 m/s , and mass ranging uniformly from 8000 kg to 20000 kg . For **ED₁** we have for each signal $16 \cdot 25 = 400$ different combinations

$[m, v, e]$ of mass, velocity and error. For 7 signals we thus have a total of $7 \cdot 400 = 2800$ different combinations of $[m, v, e]$. And for **ED**₂ we have $200 \cdot 25 = 5000$ combinations. All test cases are such that if they are run on the target system without error injection, none of the error detection mechanisms report detection.

For **ED**₁, eight different versions of the system were tested - one for each of the seven individual executable assertions and one in which all seven executable assertions were active simultaneously. For each system every combination of mass, velocity and error was exercised, giving us a total of $2800 \cdot 8 = 22400$ experiment runs with error injections for **ED**₁. The error set **ED**₂ was used only on the version containing all seven executable assertions. Therefore we have 5000 experiment runs with error injections for **ED**₂. The error injections were time triggered and were injected with a period of 20 milliseconds (recall that most modules in the target system have a period of 7 milliseconds). Thus, errors may have been injected during the execution of the executable assertions. We say that we have successful error detection if an error is detected at least once during the entire observation period (40 seconds). The detection probability is then the probability of detecting an error at least once during the observation period. The detection latency is the time from the first injection of an error to the first reported detection.

4.9.2 Results of Evaluation 2

In Table 4.9, we can see the estimates of the detection probabilities per signal, per executable assertion, and totals, as obtained using error set **ED**₁. The measures are calculated according to the formulas for coverage estimation in [Powell *et al.*, 1995]. The measure $P(d) = \frac{n_d}{n_e}$ (where n_d is the number of runs in which errors were detected and n_e is the number of runs in which errors were injected) is an estimate of the probability that the error is detected during the observation time, $P(d/f) = \frac{n_{d,fail}}{n_{e,fail}}$ (where we only take into account those runs in which the system failed) is an estimate of the probability that the error is detected given that a failure occurred, and $P(d/nf) = \frac{n_{d,nofail}}{n_{e,nofail}}$ (where we only take into account those runs in which the system did not fail) is an estimate of the detection probability given that no failure occurred. The relation $n = n_{fail} + n_{nofail}$ is true for both errors and detections. For the individual signals we have $n_e = 400$ and for the totals we have $n_e = 2800$. The All column contains the results obtained when using the version of the software, which had all seven executable assertions activated simultaneously. The table also contains the 95% confidence intervals for the estimates of the detection probabilities. We can use the measure $P(d)$ as an estimate of P_{ds} in the expression of the total detection probability for the entire system (see Section 4.8). If a cell is empty in the

Table 4.9: Error detection probabilities (%) with confidence intervals at 95% for error set \mathbf{ED}_1 . No confidence intervals can be estimated for measured detection probabilities of 100.0%.

Signal	Measure	EA1	EA2	EA3	EA4	EA5	EA6	EA7	All
SetValue	P(d)	55.5±4.1	31.3±3.8	4.0±1.6				44.3±4.1	59.5±4.0
	P(d/f)	92.6±3.7	72.4±6.4	1.5±1.7				87.9±4.7	97.1±2.4
	P(d/nf)	36.6±4.9	10.5±3.1	5.3±2.3				22.8±4.2	39.7±5.0
IsValue	P(d)		52.5±4.1					47.0±4.1	54.4±4.1
	P(d/f)		89.6±7.3					93.3±6.2	100.0
	P(d/nf)		47.4±4.4					41.1±4.3	47.2±4.4
i	P(d)	26.8±3.6	29.8±3.8	100.0	1.0±0.8	1.0±0.8	0.5±0.6	47.8±4.1	100.0
	P(d/f)	33.7±7.8	55.4±8.2	100.0	0.1±1.5	2.3±2.1	1.1±1.8	78.0±6.8	100.0
	P(d/nf)	24.4±4.1	21.1±3.9	100.0	1.0±1.0	0.4±0.6	0.3±0.5	37.7±4.6	100.0
pulsent	P(d)	50.3±4.1	42.8±4.1	0.3±0.4	12.5±2.7			0.3±0.4	100.0
	P(d/f)	38.1±5.3	34.5±4.8	0.3±0.5	0.0			0.7±1.2	100.0
	P(d/nf)	66.9±6.0	58.3±6.9	0.0	16.5±3.5			0.0	100.0
ms_slot_nbr	P(d)		20.0±3.3			100.0		6.8±2.1	100.0
	P(d/f)		34.6±5.7			100.0		11.6±3.9	100.0
	P(d/nf)		7.1±2.9			100.0		2.7±1.8	100.0
mscnt	P(d)	8.3±2.3	12.3±2.7				100.0	17.5±3.1	100.0
	P(d/f)	20.0±13.4	18.2±13.8				100.0	13.0±11.8	100.0
	P(d/nf)	7.5±2.2	11.9±2.7				100.0	17.8±3.2	100.0
OutValue	P(d)		1.0±0.8					11.3±2.6	4.0±1.6
	P(d/f)		33.3±34.7					85.7±23.5	100.0
	P(d/nf)		0.5±0.6					9.9±2.5	3.3±1.5
Total	P(d)	20.1±1.2	27.1±1.4	14.9±1.1	1.9±0.4	14.4±1.1	14.4±1.1	25.0±1.3	74.0±1.4
	P(d/f)	35.0±2.9	47.0±3.0	12.2±1.9	0.2±0.3	21.7±2.3	3.2±1.0	42.7±3.3	99.6±0.3
	P(d/nf)	14.9±1.3	19.7±1.4	16.0±1.4	2.4±0.5	11.1±1.2	19.0±1.5	19.9±1.4	60.6±1.9

Table 4.10: Error detection latency for all errors in error set **ED₁** (all times are in milliseconds).

Signal	Latency	EA1	EA2	EA3	EA4	EA5	EA6	EA7	All
SetValue	Min	160	570	50				20	20
	Avg.	690	2445	1241				842	496
	Max	6259	5588	6099				5297	6490
IsValue	Min		10					10	20
	Avg.		612					654	980
	Max		8142					4466	6630
i	Min	311	270	80	3254	4686	3495	151	91
	Avg.	2125	2100	210	4111	5538	3891	1900	205
	Max	11397	8272	401	4807	7601	4286	6499	421
pulsent	Min	390	1182	1563	20			230	20
	Avg.	1371	1379	1563	273			230	293
	Max	2284	2283	1563	1202			230	4246
ms_slot_nbr	Min		1172			20		1703	20
	Avg.		3654			32		3462	31
	Max		8912			140		5738	101
mscnt	Min	1112	1352				10	1091	20
	Avg.	2050	1741				25	1673	23
	Max	4196	3525				60	3415	61
OutValue	Min		440					20	2413
	Avg.		1344					1604	3375
	Max		2704					6179	7781
Total	Min	160	10	50	20	20	10	10	20
	Avg.	1286	1725	248	593	126	163	1314	394
	Max	11379	8912	6099	4807	7601	4286	6499	7781

table, this means that no detection was registered for that combination of signal and executable assertion.

The values shown in boldface are those that correspond to the "correct" signal-mechanism pair. For instance, the signal *SetValue* is directly monitored by mechanism *EA1*, and the signal *IsValue* is directly monitored by *EA2*.

In Tables 4.10 and 4.11 are the detection latencies measured during our experiments. The value is the time from the first injection of an error until the first registered detection, and it is measured in milliseconds. The tables show the minimum, average and maximum values for the detection latencies. Again, the boldface values correspond to the primary signal-mechanism pairs. In Table 4.10 all detected errors are considered, those leading to system failure as well as those not leading to system failure, whereas in Table 4.11 only those errors that ultimately lead to system failure are considered.

The results from the experiments with error set **ED₂** are shown in Table 4.12. The table contains detection coverage with 95% confidence intervals and detection

Table 4.11: Error detection latency for those errors in error set \mathbf{ED}_1 which caused the target system to fail (all times are in milliseconds).

Signal	Latency	EA1	EA2	EA3	EA4	EA5	EA6	EA7	All
SetValue	Min	170	570	982				20	20
	Avg.	412	2240	982				782	351
	Max	1652	5588	982				3184	5178
IsValue	Min		40					10	20
	Avg.		461					708	782
	Max		2193					4466	4506
i	Min	311	310	80	3254	4897	3495	151	91
	Avg.	2397	2263	190	3254	5822	3495	1767	178
	Max	6840	7992	401	3254	7601	3495	6499	360
pulsent	Min	390	1182	1563				230	20
	Avg.	1367	1380	1563				230	297
	Max	2284	2283	1563				230	4246
ms_slot_nbr	Min		1172			20		2213	20
	Avg.		3503			32		3533	31
	Max		5748			110		5738	101
mscnt	Min	1232	1382				20	1111	20
	Avg.	1753	1850				24	1278	20
	Max	2464	2604				40	1482	21
OutValue	Min		851					20	2654
	Avg.		1778					1143	3358
	Max		2704					2394	4276
Total	Min	170	40	80	3254	20	20	10	20
	Avg.	1040	2035	212	3254	186	603	1288	289
	Max	6840	7992	1563	3254	7601	3495	6499	5178

latencies in milliseconds. As with the measures for error set \mathbf{ED}_1 , we used the formulas in [Powell *et al.*, 1995] to derive the probabilities shown in the table.

The probabilities shown in Table 4.12 are estimates of P_{detect} , whereas the probabilities shown in Table 4.9 are estimates of P_{ds} (for more information on the definition of these probabilities, see Section 4.8). In the following section, the results presented here are discussed.

4.9.3 Discussion of Evaluation 2

The results obtained in this evaluation are specific for the target system, the error model and the test cases we have chosen. For other systems, error models, and/or test cases the results may vary. Having said that, we can now start our discussion of the results shown in the previous section.

Table 4.12: Results from experiments with error set **ED₂**.

Area	Detection Probability (%, 95% conf. int.)		Detection Latency (ms, totals)		Detection Latency (ms, failures)	
RAM	P(d)	12.8±0.9	Min	20	Min	20
	P(d/f)	81.1±6.8	Avg.	1359	Avg.	1203
	P(d/nf)	11.1±0.9	Max	5608	Max	5608
Stack	P(d)	4.2±0.9	Min	20	Min	20
	P(d/f)	13.7±4.7	Avg.	250	Avg.	2077
	P(d/nf)	2.9±0.8	Max	2684	Max	6449
Total	P(d)	10.6±0.7	Min	20	Min	20
	P(d/f)	39.4±5.2	Avg.	1086	Avg.	1298
	P(d/nf)	9.2±0.7	Max	5608	Max	6449

Error Detection Probability, P_{ds}

This section discusses the results regarding error detection coverage obtained with error set **ED₁** (see Table 4.9). The results are the estimated values for the probability P_{ds} , i.e., the probability that an error is detected given that an error is present in one of the monitored signals and therefore can be detected by the mechanisms.

The overall detection probability was 74%, and if we consider the errors that lead to failure, as defined in Chapter 3, the detection probability was over 99%. Roughly, 60% of the errors that did not lead to failure were detected. If we examine the individual executable assertions, we have detection probabilities ranging from just over 11% up to 100%. The assertions that achieved a 100% detection probability monitored signals that were all essentially counters by nature; they were periodically incremented by some limited (small) amount. This makes errors easy to detect since the freedom of change was very small in these signals. We must remember that it is possible, even probable, that we do not achieve a 100% detection probability for other error models or test cases. However, the results suggest that these mechanisms can be very effective in detecting errors.

The assertions monitoring signals representing continuous values in the environment have a lower detection probability. This can be explained by the fact that these signals have more liberal constraints than the counter signals mentioned above. The liberal constraints let those errors pass which in the value domain constitute a small change in the signal, i.e., the errors most likely to remain undetected are those affecting the least significant bits of the signal. In fact, for continuous signals, errors in the least significant bits may be indistinguishable from noise in the sampling process.

The mechanism *EA4*, which is set to monitor the signal *pulscent*, was not at all effective in detecting errors in that signal that lead to failure as it did not detect any

of those errors. Of the errors that did *not* lead to failure, only 16% were detected. The overall detection capability of this mechanism was very low, only about 12%. Errors in *pulscnt* were more easily detected by indirect mechanisms such as *EA1* and *EA2* (over 50% and over 42% respectively) that monitor signals which are affected by the value of *pulscnt*. This indicates that either are errors very hard to detect in this signal, or the parameters of the mechanism are not optimal (or both). We can therefore identify *pulscnt/EA4* as an area where more work is needed in regard to error detection.

The detection probability for *EA7* in the signal *OutValue* was roughly 11%, whereas for all mechanisms it was 4%. This is mainly due to the fact that the behavior of the target system is not entirely deterministic.

The results of *Evaluation 2* shows that by using a number of error detection mechanisms covering different parts of the system, a fairly high total coverage may be obtained.

Total Error Detection Probability, P_{detect}

As shown in Section 4.8, the probability of detection given that an error is present in a monitored signal is part of a larger expression for total error detection probability for the entire system: $P_{detect} = (P_{en}P_{prop} + P_{em})P_{ds}$. The value obtained for P_{ds} for the target system in our evaluation was 74% (see the *Total* row in Table 4.9). To obtain $P_{detect} = 74\%$ would mean that all the occurring errors, directly or after propagation, are uniformly distributed over the monitored signals. This is most likely not the case since there probably are some signals that are more dependent on other parts of the system than the remaining signals. If, for example, errors in our target system with a high probability propagate to the *SetValue* signal, P_{detect} would be closer to the detection probability for that signal, which here is roughly 59%.

From the experiments performed with error set **ED₂** (in Table 4.12), we can see that the overall detection coverage for all errors is about 10%. For errors that lead to failures, we obtained detection coverage of 39%. The values differ a lot for the two areas in which we injected errors. Generally, errors injected into the RAM area of the application were detected with a higher probability than were those injected into the stack area. An explanation for this may be that errors in the stack area more often lead to control flow errors. The evaluated mechanisms are not aimed at detecting such errors.

For the errors injected into the RAM area that eventually caused the system to fail, the detection coverage was over 81%, whereas the total detection coverage was just under 13%. We can see that if an error were of such nature that it would cause

system failure we can detect it with a fairly high probability using the presented mechanisms.

Error Detection Latency

We can see in the results for **ED₁** that the assertions which monitor signals that are essentially counters in nature have the shortest average detection latency (see Tables 4.10 and 4.11). The three mechanisms that showed a 100% detection probability were also the top three mechanisms when examining the error detection latency.

Looking at the individual mechanisms in Table 4.10 shows us that the detection latencies are rather short. Most of the mechanisms had average latencies of well below one second, only mechanism *EA7* had an average exceeding one second (1.604 seconds). The average detection latency for all mechanisms was 394 milliseconds.

If we take a closer look at the latencies for the errors that lead to failure (see Table 4.10), we can see that most mechanisms have an average below 500 microseconds. Again, only mechanism *EA7* has an average detection latency greater than one second (1.143 seconds). We also clearly see that *EA4* did not detect any of the errors that eventually lead to failure. When all mechanisms are activated, the average detection latency for errors leading to failures is 289 milliseconds.

The latencies for errors in **ED₂** (see Table 4.12) are longer than the latencies for errors in **ED₁**. This, however, is not very surprising since most of the errors in **ED₂** were not located in the monitored signals and therefore had to propagate to the monitored signals before the mechanisms could have a chance of detecting them. This propagation process increases the total time from injection to detection.

4.10 Summary and Conclusions

In this chapter we investigate the properties of error detection and recovery mechanisms derived from a proposed classification scheme for signals in modular software. The mechanisms are parameterized test algorithms based on the concept of executable assertions, and are instantiated individually for each signal that is to be monitored. Two evaluations were performed using error injection experiments. In the first evaluation bit-flips were injected in all bit positions of the monitored signals and in the second experiment bit-flip errors were injected in random bit positions in memory, registers, and stack locations. The first experiment investigated the error tolerance obtained from incorporating the proposed mechanism into an embedded system. The second experiment investigated the error detection coverage and detection latencies obtained with the mechanisms.

The results from the first evaluation show that the location of executable assertions and recovery mechanisms is of the utmost importance to the effectiveness of the mechanisms. For errors injected into the monitored signals, the failure rate was reduced with 32.56% in the used target system. However, errors that occur in data areas not monitored by the error detection and recovery mechanisms are poorly handled. The reduction in failure rate was only 4.69%. From the results one can also conclude that the memory areas of the mechanisms should be separated from the application memory or else the mechanisms will be as vulnerable to errors as the signals they monitor.

The detection probability was defined to be the probability of an error being detected at least once during the observation period. The detection latency was defined to be the latency between the first injected error and the first reported detection.

In the second evaluation, we achieved an overall detection probability for errors in the monitored signals of 74%, and if we only take into account those errors that lead to system failure we had a detection probability of over 99%. The average error detection latency when all mechanisms were activated simultaneously was 511 milliseconds. For errors that caused the system to fail, the average detection latency was 289 milliseconds.

The second evaluation also showed that for errors in the memory areas of the application, over 81% of all errors that caused system failure were detected. Errors in the stack that caused system failure were detected with a probability of 13%. The low detection probability for stack errors is likely due to the fact that errors in the stack often cause control-flow errors, and the evaluated mechanisms are not aimed at detecting such errors. The detection latencies were longer than those obtained in the first experiment. This, however, is not surprising since most injected errors must propagate to the monitored signals in order to be detected. This propagation process increases the detection latency.

From the results shown in the evaluations of the proposed mechanisms, one can conclude that the mechanisms are good candidates for software-implemented error detection in low-cost embedded systems. They are based on an intuitive concept and easy to implement and have the potential of providing high detection coverage for data errors in software signals. Regarding the error recovery capabilities of the *forced validity* concept, this should be studied further. In the evaluations, the error models used were quite aggressive (in that the errors were injected periodically, and with a period not matching the period of the system), and other error models (with error occurrence at only one point in time) should be used for further evaluations.

Also, the results indicate that the efficiency of the mechanisms, and indeed any kind of error tolerance mechanisms, depends on where they are located. Thus, know-

ing how errors propagate through a software system can improve the possibility of a system designer to obtain a high error detection probability, and thereby also a high error recovery probability, in a given system.

CHAPTER 5

PROPANE - The Propagation Analysis Environment

I didn't think; I experimented.

— Wilhelm Röntgen (1845–1923)

In order to produce reliable software, it is important to have knowledge on how faults and errors may affect the software. In particular, designing and placing efficient error detection mechanisms requires not only knowledge on which types of errors to detect but also the effect these errors may have on the software, as well as how they propagate through the software. This chapter presents the Propagation Analysis Environment (PROPANE) which is a tool for profiling of and fault injection in software running on desktop computers. PROPANE supports the injection of both software faults (by mutation of source code) and data errors (by manipulating variable and memory contents). Various error types are provided out-of-the-box and user-defined error types are supported. For logging, PROPANE can automatically instrument a target system with probes for charting the values of variables and memory areas as well as for registering events during execution of the system under test. A comparative evaluation of PROPANE with other contemporary tools demonstrate the prominent advantages of PROPANE for its combination of portability, flexibility and observability at a low cost.

5.1 Introduction

In order to develop software that functions in a non-harmful manner in the presence of faults and errors (as defined in [Laprie (ed.), 1992]), one requires knowledge of the behavior of the software under these exceptional conditions. In particular, one needs to know how faults and errors propagate to affect the execution of software. Knowing propagation pathways may, for instance, be of great help when deciding where to place error detection and recovery mechanisms.

Learning about error propagation characteristics of a software system requires not only that one should be able to inject errors and monitor the effect these have on system output, but also that one is able to monitor how these errors are transported through the system. Thus, high observability is required for these activities. Ideally, one should be able to observe every individual variable and data structure in the software.

This chapter gives a detailed presentation of PROPANE, the Propagation Analysis Environment. which is a tool suite enabling the injection of faults and errors into software running on a desktop computer (currently for Windows 2000/XP). PROPANE supports varied ways of probing a system, i.e., tracing internal variables and events during system operation, as well as injection of software faults and data errors.

Fault injections are performed by instrumenting the source code with both the correct code and the defect that is to be injected. With every fault, there is a fault-trigger that decides which of the correct code or faulty code to execute. All inserted faults are inactive by default—the experiment descriptions used for setting up PROPANE specify which faults are to be activated.

Error injections are performed by using predefined error types (defined individually for each target system), predefined locations in the software (equivalent to software traps) and then in the experiment descriptions specifying combinations of locations and error types. When the specified locations are reached during system operation, the specified errors are injected based on error triggers specified in the description files.

PROPANE can be useful in a number of situations. For instance, in Component-Based Software Development (CBSD) generic configurable software components are manufactured and assembled to form an entire system (inspired by the use of generic hardware components for building hardware systems). These components are often ported to several different hardware platforms. This limits generalized verification and validation use of tools that focus on specific hardware configurations. PROPANE on the other hand has no such limitations as it does not require any spe-

cial hardware assistance. Thus, software components may be verified and validated with PROPANE before porting them to various target hardware. This argument will of course also be valid for testing embedded software which in many cases may exist before the hardware platform has been finalized.

In a comparison with other FI-tools (as detailed in Section 5.8), we comment that very few other tools can provide the detailed data necessary for software propagation analysis at a level comparable to PROPANE. Also, its portability and flexibility uniquely distinguish PROPANE as a simple, low-cost error propagation analysis tool. We emphasize that PROPANE, through its depiction of error propagation paths, is primarily designed as a software design aid with complementary capability of being used in the evaluation of effectiveness of error handling mechanisms.

The remaining chapter is structured as follows: In Section 5.2 we describe the target system model for which PROPANE is aimed, and Section 5.3 will give an overview of the tool. Section 5.4 details the concepts and structure of PROPANE and also shows how experiments are set up. Experiment execution and data analysis are covered in Sections 5.5 and 5.6, respectively. An example of actual PROPANE usage is shown in Section 5.7. Once we have described the details of PROPANE we can compare it to other FI-tools. Section 5.8 contains this comparison. Finally, Section 5.9 summarizes this chapter and states the conclusions.

5.2 Target System Model

PROPANE aims at modular software, i.e., discrete software functions interacting to deliver the requisite functionality. A module in this context is a generalized software block having possibly multiple inputs and outputs. Modules communicate with each other in some specified way using varied forms of signaling, e.g., shared memory, messaging, parameter passing, etc., as pertinent to the chosen communication model. A detailed description of this model is described in Chapter 3.

Software constructed in such a modular way is found in numerous systems, such as desktop systems as well as embedded systems. For example, most applications controlling physical events such as in automotive systems are traditionally built up as such. Our studies mainly focus on software developed for embedded systems in consumer products (high-volume and low-production-cost systems).

PROPANE is designed with a focus on software for single-process user applications on desktop systems. However, this single process may be multi-threaded. The PROPANE injection and logging mechanisms are generic and are provided in a static C-library, thus allowing for a vast range of applications. For example, it has been used in experimentally analyzing the propagation of data errors in the soft-

ware of an embedded control system simulated on a Windows-based desktop computer [Hiller *et al.*, 2001, Hiller *et al.*, 2002(a), Jhumka *et al.*, 2001]. The requirement for using PROPANE is that the language used for the source code is able to interface with libraries implemented in the C programming language.

Although the system model described in Chapter 3 is a black box model, and is adopted in this thesis, PROPANE may be used in a white box approach as well, as there is nothing in PROPANE that limits the kind of data that is logged, i.e., one may log input and output signals of modules as well as their internal static and temporary data areas. However, in this thesis, PROPANE has been primarily used in a black box setting.

5.3 Overview of the Tool

This section provides an overview of how PROPANE is structured, and also its proposed usage.

5.3.1 Basic System Structure

PROPANE is designed to run on a desktop system (such as Windows 2000/XP or UNIX) and consists of a suite of tools, namely: the PROPANE System Instrumentor (PSI), the PROPANE Setup Creator (PSC), the PROPANE Campaign Driver (PCD), the PROPANE Library (PL), and the PROPANE Data Extractor (PDE). An overview is shown in Fig. 5.1.

PL is used by the target system to gain access to the probing and injection functionality of PROPANE and is written in the C programming language. PCD is responsible for handling the actual execution of experiments and is in a sense the main administrator of PROPANE. It has a user interface through which the user can control and follow the experiments. In order to be able to log variables and event and to inject faults and errors the target source code must be instrumented. Given a description of the modular composition of the system including I/O and internal characteristics of the various modules, and the original source code, PSI will automatically generate instrumented versions of the source files and intermediate file used by PSC to create experiment setups. PDE may be used during analysis to extract specific data from the experiment readout files. PCD and PL are integrated with each other, whereas PSI, PSC and PDE are stand-alone components of PROPANE. The environment simulator, the target software and any user injectors and user triggers are provided by the user. The environment simulator will act as a stimuli generator for the target software and may be partially controlled by the output generated by the

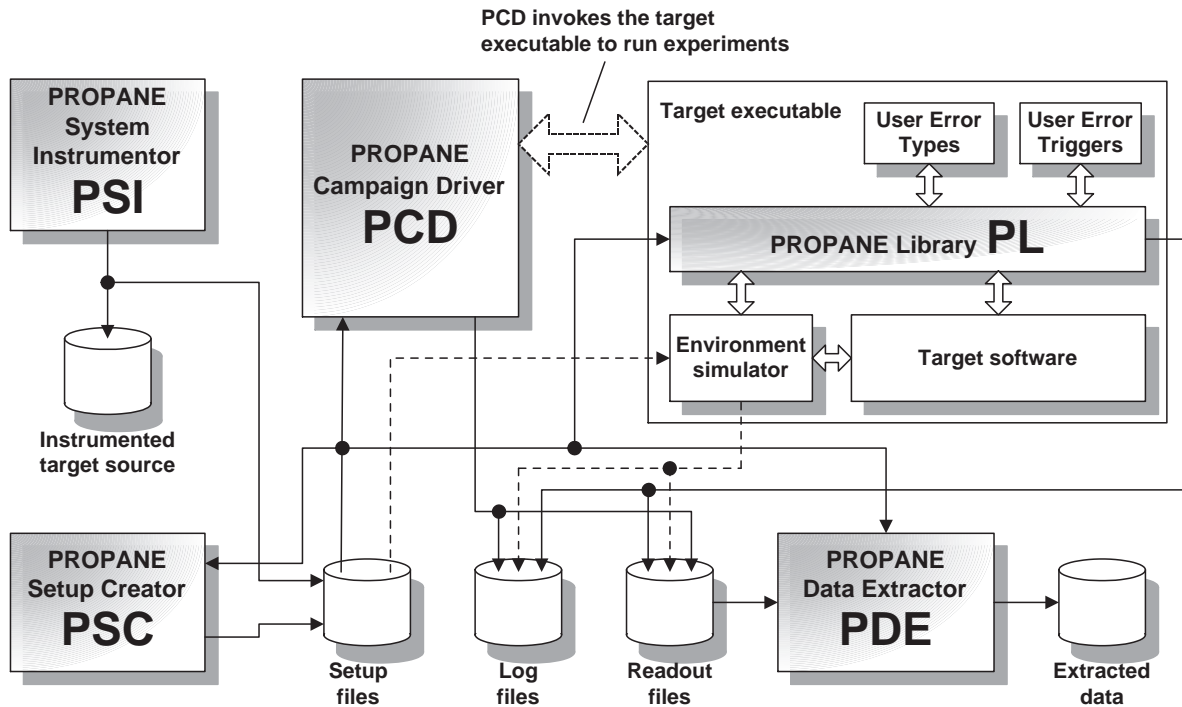


Figure 5.1: An overview of PROPANE together with target software and environment simulator

target software (e.g., as in a control loop). The interactions between these two sub parts of the target executable are user-defined.

An FI-experiment is set up with a number of description files containing details on which faults and errors to inject, which variables and events to trace and so on. There are three types of description files: database description files, campaign description files, and experiment description files. A *database* is a set of *campaigns*, which in turn are sets of *experiments*. Each *experiment* contains the set up of one execution of the target system (one combination of input data and error injection). The PSC aids in the creation of these files needed for controlling PROPANE. Given information regarding errors and faults, probes, injection locations, etc., it will generate the requisite description files. The PSC will also generate description files used by the PDE during analysis. Details on the description files are presented in Section 5.4.

For each experiment specified in the description files, the PCD spawns a new process running an executable file containing a complete specification for conducting one experiment. This executable contains the PL which performs the actual injection of errors and logging of variables. The executable also has to contain everything necessary to run the target system and the environment simulator.

The reason for running experiments in individual processes is twofold:

- A) Parallel execution may shorten the total time for executing sets of experiments. This is especially true for systems that have excess computing capacity so that there is ample idle time during the execution of a single experiment, primarily multi-processor systems.
- B) Each experiment begins execution from identically specified initial conditions. This ensures that there are no residuals that make one experiment affect another experiment. This is given that during an experiment the different processes do not compete over a (limited) shared resource, such as file handles.

During the execution of the experiments, log files and readout files are created. The log files contain information regarding the execution of the experiments, i.e., PROPANE performance and behavior information, and does not contain any readout data gathered from the target software. If the experiment could not be executed successfully for some reason, the log files provide hints to potential problems. The readout files contain the data obtained by the inserted probes and the performed injections and are the basis for subsequent error propagation analysis. Details about log files and readout files are provided in Section 5.5. The environment simulator is designed by the user of the PROPANE tool, hence it follows the user-specifications on use and generation of description, log and readout files. The format of the files read and/or written by the environment simulator is also user-definable.

PL requires interfacing to the environment simulator. However, if an environment simulator exists which does not comply with the interface specifications, a wrapper layer is warranted which has the PROPANE interface on one side and the environment simulator interface on the other, acting as a translator between the two components. Thus, the environment simulator need not necessarily be an integrated part of the target executable.

PDE will extract traces of the various logged variables and memory areas and can conduct Golden Run Comparisons (i.e. comparing system traces obtained during injection experiments with fault/error free reference traces, details in Section 5.6) to detect whether errors have occurred due to fault injection. Information regarding propagation will be compiled and presented. Also, intermediate extracted data is stored in special files which can subsequently be used in a customized analysis tools which may take into account desired experiment specific information and/or aims, such as coverage estimation of error handling mechanisms, failure classification or other activities which may be target specific.

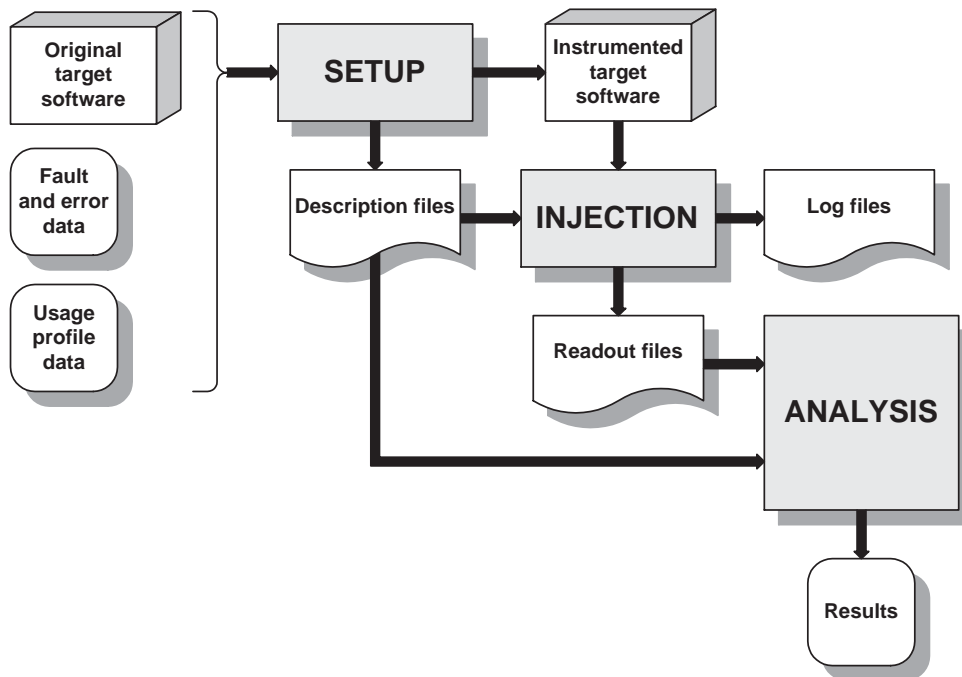


Figure 5.2: The basic work process when using PROPANE.

5.3.2 Work Process for Using PROPANE

The typical work process when using PROPANE can basically be divided into three main phases, namely: 1) the *Setup* phase, 2) the *Injection* phase, and 3) the *Analysis* phase (as illustrated in Fig. 5.2).

Setup phase: In the *Setup* phase, description files are generated and the target system is instrumented. The inputs to this phase include the original source code of the target software, information on distribution and nature of faults and/or errors, and information about target system usage. The fault and error information is used for determining the fault and error sets to be injected in the experiments. The usage information forms the basis for determining the test cases used during the injections in order to provide the target system with a realistic operational profile. In the setup phase, PSI is used for instrumenting the target software with probes for logging variables, memory areas, and events, as well as with high-level software traps for injecting faults/errors. Given basic information about errors and faults, probes, injection details, etc., PSC generates the required description files for PCD/PL and PDE. The description files contain information on which faults are to be injected, which errors are to be injected at which locations, and which test cases are to be used by the environment simulator during the execution. Details regarding the *Setup* phase are provided in Section 5.4.

The output of the phase is a set of description files and the instrumented target system. The description files contain information on which faults are to be injected, which errors are to be injected at which locations, and which test cases are to be used by the environment simulator during the execution. The instrumented target system contains injection locations (which may be regarded as high-level software traps) and probes logging the desired variables, memory areas and events.

Injection phase: During the *Injection* phase, the PROPANE Campaign Driver (PCD) is set up with the description files generated in the *Setup* phase. The PCD invokes the target executable as an individual process and generates readout files containing detailed information on the results of the experiments. During the experiment, the specified faults and/or errors are injected and the specified variables and events are logged. Log-files are generated recording the actions of the PROPANE tool itself. Details regarding the *Injection* phase are provided in Section 5.5.

Faults are injected when the corresponding fault-triggers are activated. Fault injection at this level means that a faulty piece of code is executed instead of the correct piece.

Errors are injected based on the built-in error types, or on user-implemented error types. Thus, it is possible to implement error models which are not included in PROPANE. For example, if some parts of a system work unreliably under extreme temperatures, a user error type could take this into consideration.

Error triggers are boolean expressions and an error is injected when its corresponding error trigger is evaluated to *true*. Error triggers may be based on time, frequency or a probability distribution. In addition to the built-in error triggers, PROPANE also supports user-implemented error triggers. As was the case for user error types, a user error trigger may take into account target specific information, such as system state or the environment. In the example with the temperature-induced error type, a corresponding error trigger may evaluate to *true* when the temperature is below a lower threshold and above an upper threshold.

Analysis phase: The readout files generated in the *Injection* phase are analyzed in the *Analysis* phase to evaluate metrics for the target systems. These metrics may include coverage values, propagation information, etc. One aspect of analysis is to compare traces from two different runs with each other, e.g., compare a golden run(i.e. a reference run) with an injection run. The PROPANE Data Extractor compiles propagation information from the readout files and also generates a set of data-files containing data such as detailed results from Golden Run Comparisons and injection information. These files can then be used for propagation analysis at the variable-level. Further details regarding the *Analysis* phase are provided in Section 5.6.

5.4 Setup: Experiment Design and Target Instrumentation

Setting up experiments with PROPANE requires one to perform the following steps:

1. Select faults to inject.
2. Select error types, injection locations.
3. Select injection triggers (for errors).
4. Select variables, memory areas, and events to log.
5. Select test cases for the environment simulator.
6. Instrument target system software (using PSI)
7. Generate description files (using PSC)

The subsequent sections detail the specifics of the various steps of the setup process for PROPANE.

5.4.1 Selecting Which Faults and Errors to Inject

Before any faults can be selected for injection, one must know which different faults are possible to occur in the target system. Producing a set of possible faults may be done in several ways. The quality of the results obtained from the experiments may depend on how good this set of possible faults, \mathbf{F}_p , is at representing the real world faults that may occur in the target system. The set of possible faults is actually a subset of the entire set of faults \mathbf{F} , i.e., $\mathbf{F}_p \subset \mathbf{F}$. From the set of possible faults, one must select a fault set $\mathbf{F}^* \subseteq \mathbf{F}_p \subset \mathbf{F}$ for injection in the target system. Each fault $\mathbf{f} \in \mathbf{F}^*$ is then manually inserted into the target system together with an activation clause. That is, every fault \mathbf{f} is inserted into the same executable of the target system, and the experiment descriptions then specify which of these faults that are to be activated during the execution of the experiment.

For errors, the situation is similar. There is an abstract set \mathbf{E} containing all errors. Then there is a set $\mathbf{E}_p \subset \mathbf{E}$ containing all the errors that the experimental environment is capable of reproducing. Then, one must select an error set $\mathbf{E}^* \subseteq \mathbf{E}_p \subset \mathbf{E}$ for use in the injection experiments. The selection of errors in \mathbf{E}^* depends on what the objective of the error injections is. If the goal is to mimic the effects of faults, the errors must be selected using the set of selected faults \mathbf{F}^* . From the description of the faults in \mathbf{F}^* , parameters for the errors that are to be injected are obtained. Other

goals may produce other errors. When the set of selected errors is obtained each error $\mathbf{e} \in \mathbf{E}^*$ is analyzed for type and location. The error types are specified in the description files and the locations are specified in the target system by means of indicators (high-level software traps) which tell the PROPANE library when the execution has reached a certain location. Upon reaching an indicated location, PROPANE injects any errors specified for injection at that location.

Faults may be selected in a variety of ways, the PROPANE tool suite does not really have any preference in this regard. In [Christmansson and Rimén, 1997], a tool was developed—the C Fault Locator (CFL)—which, given a C source file, locates all lines that may be modified to contain a fault according to some fault classification. CFL is designed to look for a subset of faults from the Orthogonal Defect Classification (ODC) [Chillarege *et al.*, 1992]. The fault classes we have chosen to look for are those of Assignment (A), Interface (I), and Checking (C). More details on CFL and on how faults are selected are found in [Christmansson and Rimén, 1997].

Error selection can also be made in a variety of ways. One way is to let errors mimic the effects of faults. This eliminates the task of physically instrumenting the code since the system behavior may be the same as for faults. However, it may be difficult to mimic every type of fault by injecting errors. For example, faults directly affecting the control flow of a program may be hard to mimic using only error injection into variables (as was the case in [Christmansson *et al.*, 1998]). Errors may also be selected to resemble intermittent hardware faults or stuck-at faults.

5.4.2 Faults and Fault Triggers

The fault selection process will result in a set \mathbf{F} containing a number of faults that are to be injected into the target software. Each fault $\mathbf{f} \in \mathbf{F}$ has to be manually inserted into the target software with corresponding fault triggers (i.e., PROPANE uses mutation of source code to inject faults). The fault triggers are binary switches (On/Off) and will route the execution trajectory to execute either the correct code or the faulty code. The description files specify which fault triggers are activated (set to On). Faults, in this context, are defects in the source code of the software. Given the wide variety of possible software defects and the inherent complexity such defects may exhibit, PROPANE supports user-defined fault cases.

Once each individual fault and its corresponding fault trigger has been inserted into the target software, the activation of the faults during experiment execution is done as specified in the description files. Thus, even though all faults are inserted into the target software, only those that are of interest in a given experiment are actually activated. Hence, the target software has to be instrumented only once.

Table 5.1: PROPANE Error Types

Error Type	Manipulation Level	Description
Bit-flip	Bit	Flip one or more bits in the binary representation of the error target.
Bit-set	Bit	Set one or more bits in the binary representation of the error target.
Bit-clear	Bit	Clear one or more bits in the binary representation of the error target.
Set-value	Value	Set a variable or memory location to a certain value.
Set-max	Value	Set a variable to its maximum value (as defined by the variable type).
Set-min	Value	Set a variable to its minimum value (as defined by the variable type).
Factor	Value	Multiply the current value of a variable with a certain factor.
Offset	Value	Add an offset to the current value of a variable.
Factor-offset	Value	First multiply the current value of a variable with a factor and then add an offset.
Offset-factor	Value	First add an offset to the current value of a variable and then multiply with a factor.
User-defined	User-defined	User error injectors may inject any kind of error.

5.4.3 Error Types and Injection Locations

The error selection process will generate a set \mathbf{E} containing a number of errors selected for injection during the experiments. Errors in PROPANE differ from faults by not being bound to specific locations in the target system software. Each error $e \in \mathbf{E}$ is an error type which can be injected into several locations in the software. For example, one error may be a bit-flip in bit #2 of the binary representation of a variable. This error may then be injected into one or more locations (variables). Thus, when defining an error set \mathbf{E} for an experiment, one will not have to take into account the locations at that point.

PROPANE supports a variety of error types, both for manipulating individual bits of the binary representations of variable and memory contents and for assigning altered numerical values to variables (as illustrated in Table 5.1).

Bit-level manipulations can be performed on individual variables or memory areas (anything that is addressable within the scope of the target software). Value-level manipulations can only be performed on individual variables. The Set-value

error type may also be used on larger memory areas (set a byte with a certain offset from the start of the area to a certain value).

In addition to injecting errors based on the built-in error types, user-defined customized error models are supported as well (this resembles the light-weight injectors used in [Stott *et al.*, 2000]).

Once all the error types have been defined, the locations where these error types will be injected have to be selected. This will generate a set **L** where each element **l** is a tuple containing the physical location in the source code where the high-level software trap will be placed and the variable or memory area in which the error is to be injected.

The injection locations have to be instrumented in the target software, whereas the error types are defined only in the description files (of course, user error injectors have to be implemented in source code). Thus, once the injection locations are in place, any variety of error types can be injected using the same instrumentation. It should be pointed out here that one error type may be associated with more than one physical location, and vice versa one location may be associated with more than one error type.

5.4.4 Triggering the Error Injections

Once all error types and error injection locations have been selected, the error triggers have to be defined. An error trigger tells PROPANE when and how often to inject errors. The error triggers are basically boolean expressions dictating when a particular combination of location and error type is activated.

A majority of the error triggers are based on the notion of time, either counted as number of times an injection location is reached or based on the internal PROPANE time. The PROPANE time is a clock which has a tick specified by the user (by having the user call a special tick-function with a certain period). Accordingly, we categorize error triggers with respect to timing and frequency of occurrence as well as probability, as detailed in Table 5.2.

Permanent errors are emulated using the *Always* error trigger, transient errors are simulated by using the *Once* error trigger, and intermittent errors are simulated with the *Period* or *Probability* error triggers.

User-defined error triggers are functions designed by the user and may use any expression for triggering an injection. For instance, a user-defined error trigger could make decisions based on the state of the system, similar to the technique used in [Chandra *et al.*, 2000, Cukier *et al.*, 1999].

Table 5.2: PROPANE Error Triggers

Error Trigger	Description
Always	Injection performed every time the trap is reached.
Once-time	Injection performed once when the trap is reached and the PROPANE time is greater than or equal to a certain value.
Once-cycle	Injection performed once after the trap has been reached a certain number of times.
Period-time	Injection performed periodically with a certain PROPANE time period. The first injection is when the trap is reached and the PROPANE time is greater than or equal to the period.
Period-cycle	Injection performed periodically with the period being number of times the trap is reached. The first injection is performed the first time the trap has been reached the number of times specified as the period.
Probability	Injection performed with a certain probability each time the trap is reached. The uniform distribution is used.
User-defined	User-implemented triggers may choose any kind of conditions for triggering an injection.

5.4.5 Logging Variables, Memory Areas, Events

The selection of probes to be used in the experiment is fully dictated by the objectives of the experiments, i.e., the probes must be selected so that the data necessary for obtaining the desired measures in the analysis phase are collected. PROPANE provides two kinds of logging probes: 1) variable probes, and 2) event probes. The variable probes are used for logging the values of variables and memory areas, whereas the event probes are used for logging certain pre-defined events.

The probes must be inserted into the target software and can be considered as high-level software traps, analogous to fault triggers and error injection locations. For event probes, the actual event detection has to be implemented by the user, the probe can only be used for adding an entry in the readout files.

The basic rule is that an entry in the readout files is made every time a logging trap is reached. However, for variable probes this is only true when logging a memory area. When logging a variable, the current value of the logged item is compared to the value it had the previous time it was logged. If the value has not changed, no new entry will be made in the readout files. PROPANE supports two different kinds of probes: variable probes and event probes. The variable probes are used for tracing the value of a variable, and the event probes are used for logging the occurrences of events. The target system must be instrumented with the probes.

Entries created by a variable probe contain the name of the probe, a time stamp (the PROPANE time) and the value of the logged item and entries created by event probes contain only the name of the probe and a time-stamp.

5.4.6 Environment Simulators and Test Cases

The test cases that are to be used for the experiments are highly application specific and depend on the target system as well as the intended operational environment. Since the environment simulator is developed separately from the PROPANE tool, the parameters included in a test case may be different between different target systems, and are not limited by the tool. Generally, it is important to obtain a set of test cases that closely resembles the intended usage profile of the target system. Using test cases that are not representative of how the target system is used in reality will obviously decrease the utility of the obtained results.

PROPANE enables the user to include the handling of environment simulator and test cases in the setup of the tool, i.e., the files used for setting up PROPANE may also be used for setting up environment simulators with user-defined test cases. As the link to the environment simulator is implemented specifically for each environment simulator by the user of PROPANE, there are no special requirements on the interface of the simulator. Using a layer of wrappers as interface between PROPANE and the environment simulators allows for virtually any simulator to be linked with PROPANE.

The environment simulator may also be linked into the final executable. However, PROPANE does not explicitly require this. Actually, any kind of interaction between the target software and the environment simulator may be handled by PROPANE as this interaction is entirely designed and implemented by the user. PROPANE only provides means for integrating control and handling of the environment simulator with the other experiment activities.

5.4.7 Target System Instrumentation

In order to make use of the support for probes and injections provided by PROPANE, the target system must be instrumented, as mentioned previously. Instrumenting a target system includes the following: 1) inserting probes to log variables and events, 2) inserting faults and fault triggers, and 3) inserting injection locations (a form of high-level software traps) for error injection. In addition to these activities in the target source code, PL must be linked together with the target system software and user-implemented error types and error triggers (if required) have to be developed. Instrumentation is currently a manual activity, i.e., probes and injection locations

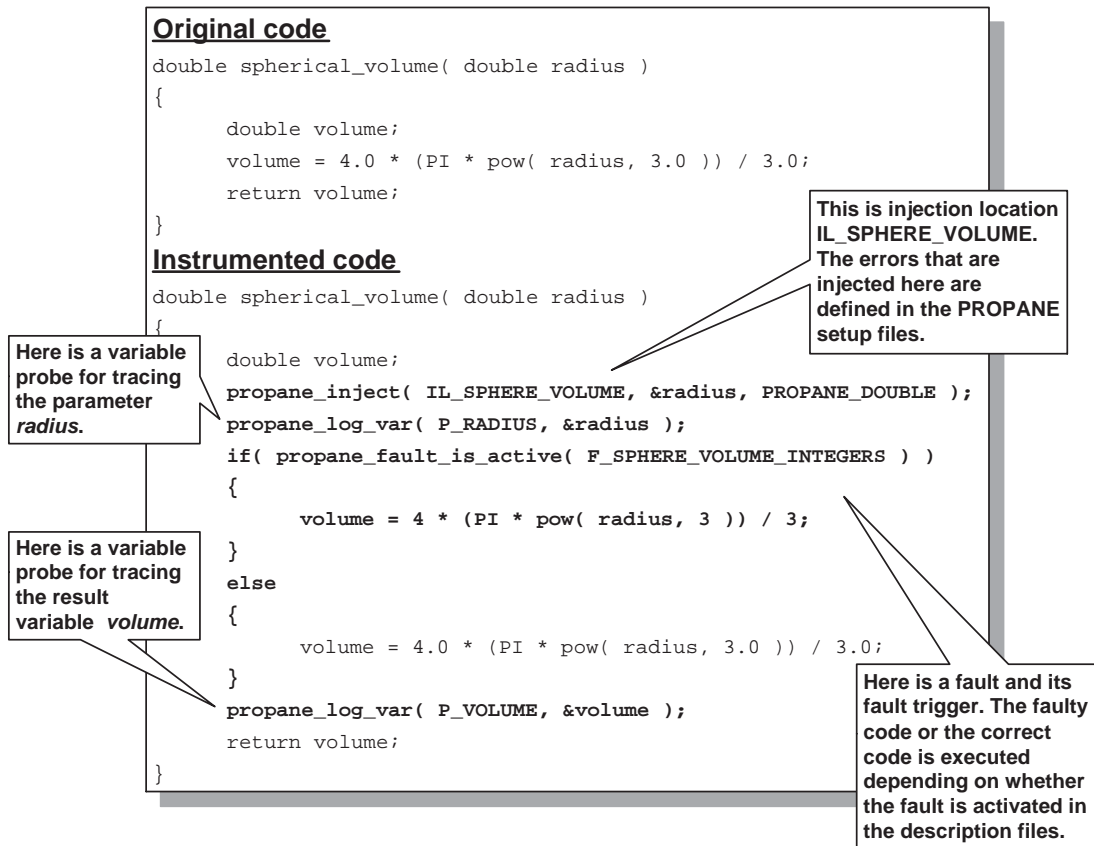


Figure 5.3: An example of instrumented code.

have to be inserted manually into the target source code. However, efforts are currently being made to automate this part. A sample fragment of instrumented code is shown in Fig. 5.3. This example shows a function for calculating the volume of a sphere, given the radius. The instrumented code shows the added code lines in bold-face. Here we have added an injection location, two variable probes and inserted a fault and corresponding fault trigger. We will not describe the details of the various API functions provided by PL here, instead the reader is referred to Appendix A.

Although it is possible to do the instrumentation manually, it is recommended that PSI be used for automatic instrumentation of source code. Currently there exist two main methods for automatically instrumenting the source code: i) adding annotations to the original source code, and/or ii) specify the modular composition of the system and for each module the I/O and internal characteristics. PSI will, given this information, generate instrumented system source code, the PROPANE configuration source files (standard ANSI C files) used by PL and setup files used by PSC. More information on instrumentation of the target system is found in Appendix A.

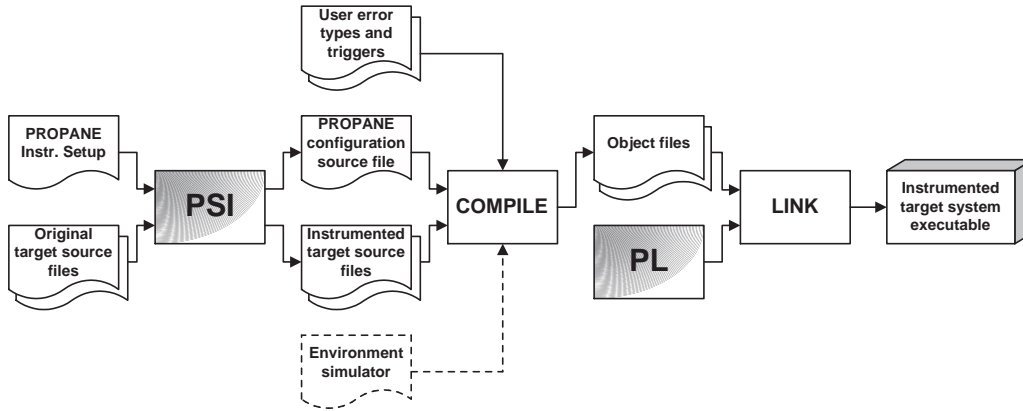


Figure 5.4: The basic work-flow of target system instrumentation.

Information regarding the low-level characteristics (such as data type, identifier, name, etc.) of probes, injection locations and faults are entered in a PROPANE configuration source file (standard C-file, generated by PSI). In the description files, the names are then used when referring to the probes, injection locations and faults. Details about the PROPANE configuration source file are described in Appendix A.

An illustration of the work required for constructing an instrumented target system executable is presented in Fig. 5.4. A description of the system composition and the desired instrumentation, along with the original target source code is used by PSI to generate the PROPANE configuration source file and instrumented target source code. The configuration source file contains information needed in the PL, such as probes, faults, locations, etc. This information is constant information that will remain the same between different experiments. The PROPANE configuration source file, the instrumented target source files, and the user error types and error triggers (written by the user) are compiled and linked together with PL to form the instrumented target system executable. This executable is then used by the PCD when conducting experiments.

The environment simulator may also be linked into the final executable. However, PROPANE does not explicitly require this. Actually, any kind of interaction between the target software and the environment simulator may be handled by PROPANE as this interaction is entirely designed and implemented by the user. PROPANE only provides integration of control and handling of the environment simulator with the other experiment activities by means of wrapper functions for initiating and shutting down the environment simulator. The wrapper functions are called at the start and completion of each individual experiment, respectively.

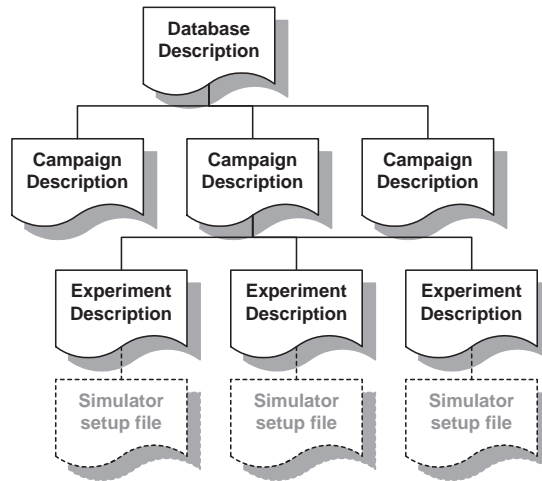


Figure 5.5: Organization of description files for PROPANE setup.

5.4.8 Setup Using Description Files

A target system executable will contain logging probes, faults and fault triggers, and injection locations. However, in order to activate probes, inject faults and/or errors, a set of description files have to be written which contain experimental details. These files are organized as illustrated in Fig 5.5.

At the top level, we have the *Database Description* containing information on where the remaining setup files are found and where the obtained readouts are to be stored. In the *Database Description* is also a list of the campaigns that make up the database. Each campaign has a *Campaign Description* containing information regarding the execution of the experiments, such as the name of the executable file that shall be used and a listing of the experiments that make up the campaign. For each experiment, there is an *Experiment Description* containing details for the experiment, such as which probes shall be activated, which injections shall be performed (i.e., which error type(s) in which injection location(s)), and which setup file (if any) is to be used for the configuration of the environment simulator. These files may have different formats for each environment simulator and are specified by the designer of the simulators. The file formats for the various description files are detailed in Appendix A.

5.5 Injection: Running Experiments

Following the experiment setup and specification phase is the injection phase. For every description file—*Database Description*, *Campaign Descriptions*, and *Experi-*

```
// The file begins with a section containing information
// regarding the various channels, namely channel names and
// entry structures.
#info: Experiment Readout File. Created Wed Dec 06 11:36:14 2000
// Channels created by variable probes
#channel vp.<probe name> VARIABLE_PROBE <type> <size>
#info vp.<probe name> [time, value]
// Channels created by event probes
#channel ep.<probe name> EVENT_PROBE
#info ep.<probe name> [time]
// Channels created by error injections
#channel ei.<location name> ERROR_INJECTION
#info ei.<location name> [time, value before, error, value after]
// Channels created by fault injections
#channel fi.<fault name> FAULT_INJECTION
#info fi.<fault name> [time]
// The entries always come after the channel information section
<channel name> <entry>
```

Figure 5.6: The format of channels in the Experiment Readout Files.

ment Descriptions—a corresponding log file and a corresponding readout file will be generated during the actual execution of experiments. The log files contain records of the actions performed by PROPANE including error messages if anything should go wrong during the execution of the experiments. The readout files contain data obtained from probes, injections and simulators, and form the input for the analysis phase.

The data gathered by the logging probes and injections is stored in the *Experiment Readout Files* (one for each Experiment Description). In these files, data will be organized in *channels*. Each channel contains the readouts produced by one probe, one fault trigger or one injection location. The environment simulator may also store data in the file. This data will also be stored in a channel, but the format is defined by the user. An experiment readout file is formatted as illustrated in Fig. 5.6.

The PCD will supervise the automatic execution of all experiments that are specified in the description files. During the actual execution, PCD will continuously provide information on the current status and estimated completion time for all experiments. At this stage, the user can choose to pause the execution, abort it altogether or skip ahead one campaign.

5.6 Analysis: Obtaining Propagation Characteristics

In the analysis phase of an injection experiment, activities such as estimating coverage values, extracting failure and error data, establishing propagation paths of the errors, etc., are performed. To this end, PROPANE contains the Data Extractor (PDE). PDE enables conducting Golden Run Comparison (GRC), compile propagation information, extract injection information, and create trace files of the channels in the experiment readout files. For plotting channels, the trace files can be imported into spreadsheet programs, such as MS Excel. This section describes details about the actions performed by the PDE.

5.6.1 Golden Run Comparisons

A Golden Run Comparison (GRC) is performed by comparing the readouts produced by the Golden Run (GR), i.e., a reference run with no faults and/or errors injected, with the readout produced by an injection run (IR), i.e., a run in which faults and errors were actually injected. In a GRC, only channels produced by variable probes are considered, as the propagation of data errors only is possible in variables and memory areas.

The PDE treats the the first campaign in a series of campaigns (in a database readout file) as the GR campaign and all others as IR campaigns. It is important that the number of channels in each experiment in the GR campaign is the same as the number of channels in the IR campaigns. If this is not the case, the GRC will not be completed and error messages will be displayed.

During the comparison, each GR channel is compared to the corresponding IR channel. The comparison is performed sample by sample and the first mismatch between the GR and the IR is flagged as an error. The first mismatch will be marked with the documented time stamp of that sample and the GRC for that channel is ended. The comparison can be performed requiring total equality between GR channels and IR channels or using error margins.

The results of the GRC will be written in extraction result files—one such file for each individual campaign (except for the GR campaign). An extraction result file will contain one line of information for each experiment in the campaign. This information includes for each channel the time stamp of the sample that was found not to match the golden run and information about the the mismatch per se, i.e., the golden run value and the injection run value.

From the GRC we will also get a summary of the error propagation showing how errors propagated through the system. This information includes that propagation

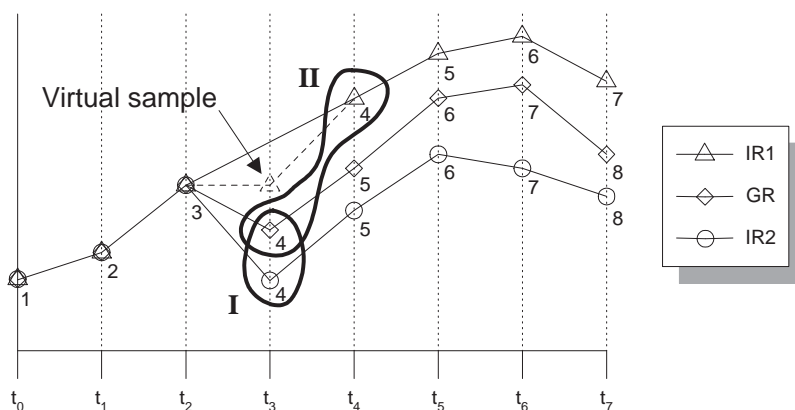


Figure 5.7: An example of GRC performed by the PROPANE Data Extractor.

rate, i.e., how many of the injected errors propagated to various parts of the system as well as how long time it took them.

Virtual Samples

The logging performed by the variable probes is such that only changes in the variables that are being probed are actually entered into the readout files. This reduces the amount of data stored for each experiment, but requires some extra effort during analysis in order to recreate the “ignored” samples since the k^{th} sample for a GR channel may not correspond in time to the k^{th} sample of an IR channel.

Consider the example in Fig. 5.7; here we have the trace of a GR channel and the traces of two corresponding IR channels. There are two different scenarios that can arise when comparing the samples of two corresponding channels: **I**) two samples correspond temporarily, and **II**) two samples do not correspond temporarily. Scenario **I** is illustrated in Fig. 5.7 where sample #4 for the GR channel (at time t_3) corresponds temporally with sample #4 (also at time t_3) of IR channel 2. In the same figure, scenario **II** is illustrated by sample #4 of the GR channel not corresponding temporally with sample #4 of IR channel 1.

In scenario **I**, GRC is simply performed as a comparison between the values of the two sample. Scenario **II**, however, requires a little more work before a comparison can be made. Here a *virtual sample* is created using the time stamp of sample #4 of the GR channel and the value of the previous IR sample, in this case sample #3 of IR channel 1. This can be done because we know that the samples only show the changes of their channels, so at time t_3 , IR channel 1 must have had the same value it had at the previous sample point. After the comparison between the virtual sample and sample #4 of the GR channel is complete, sample #4 of IR channel 1 will

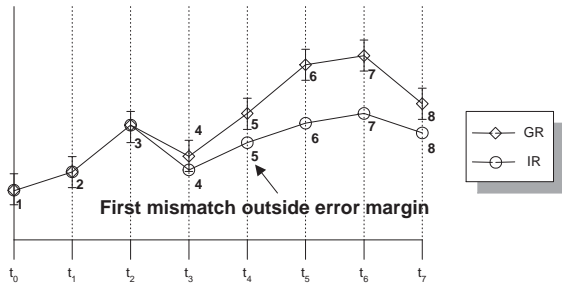


Figure 5.8: Comparison of a Golden Run and corresponding Injection Run using absolute margins.

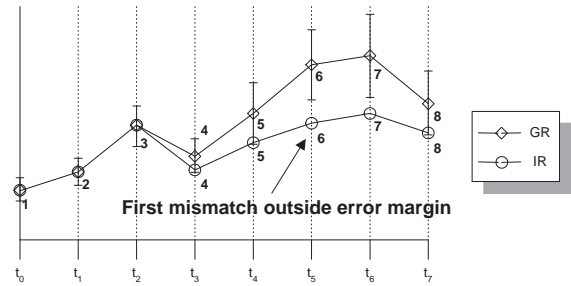


Figure 5.9: Comparison of a Golden Run and corresponding Injection Run using relative margins.

be compared to sample #5 of the GR channel. Virtual samples would of course also be created if the GR channel would “miss” a sample as compared to the sequence of samples in the IR channel.

Error Margins

PDE is capable of applying error margins to the comparison performed during a GRC. With error margins, the two values that are compared do not have to be identical to be considered correct. This can be useful when comparing two channels which inherently are “noisy”, i.e., slight variations when using the same test case may be normal. Each channel can be given individual error margins and the margins can be either absolute or relative (percentages).

An absolute margin will set upper and lower boundaries on the absolute error between the GR channel and the IR channel. For example, if a channel has an absolute margin of 5 up and 10 down, and a golden run sample of that channel has the value 100, then the injection run sample of that channel will be considered correct as long as it is within the range $100 - 10$ and $100 + 5$, i.e. within 90 and 105. If the golden run sample were instead 200, the range would be between 190 and 205.

A relative margin will set upper and lower bounds on the relative error between the GR channel and the IR channel. For example, if we have a relative error margin for a channel with 0.05 upwards and 0.10 downwards, and a golden run sample of that channel has the value 100, then the injection run sample of that channel will be considered correct as long as it is within the range $100 \cdot (1.0 - 0.10)$ and $100 \cdot (1.0 + 0.05)$, i.e. within 90 and 105. If the golden run sample would instead be 200, the range would be between 180 and 210.

In Figs. 5.8 and 5.9 the two types of margins are illustrated. Here we have a Golden Run and a corresponding Injection Run. Fig. 5.8 illustrates a GRC using

absolute margins and Fig. 5.9 illustrates a GRC using relative margins. As can be seen when comparing the two figures, given the same experiment readout file, errors may be detected at different times depending on the type of error margin (if any) used in the analysis. In this example, with absolute error margins an error will be detected at sample #5, whereas with relative error margins, an error is not detected until sample #6.

5.6.2 Channel Logs

Channel logs may be useful for doing a more detailed analysis of different channels than the PDE can provide. However, one should bear in mind that one file will be generated for each individual channel of each individual experiment, i.e. for 10 experiments with 10 channels each, 100 channel logs containing the samples of the individual channels will be generated. The files are delimited text files that are easily imported into a spreadsheet tool, such as Microsoft Excel, where further analysis or graphical representation may be performed.

5.6.3 Injection Information

When injection information files are generated, the PDE creates one file for each campaign (except the GR campaign). The injection information files contain information for each injection run regarding the times at which errors were actually injected. This information is sometimes useful for filtering out values and events that are logged before the actual injection, since these may not be of any interest.

5.6.4 Propagation Information

The Golden Run Comparison performed by the PDE will for each channel identify the first discrepancy between the Golden Run and an Injection Run. These discrepancies can be ordered temporally to give a propagation signature for that particular Injection Run. Every individual Injection Run produces such an error propagation signature, i.e., a set of timestamps showing when the various variable probe channels were erroneous the first time (in comparison with the Golden Run). This signature shows where and when an error was injected and which channels were subsequently affected and the time at which the propagated error occurred.

The signatures from individual experiment runs can be combined into propagation graphs (directed graph with weights on the arcs) showing propagation times and propagation rates. One propagation graph will be created for each unique channel in which errors are injected. In the propagation graph, the channel in which errors were

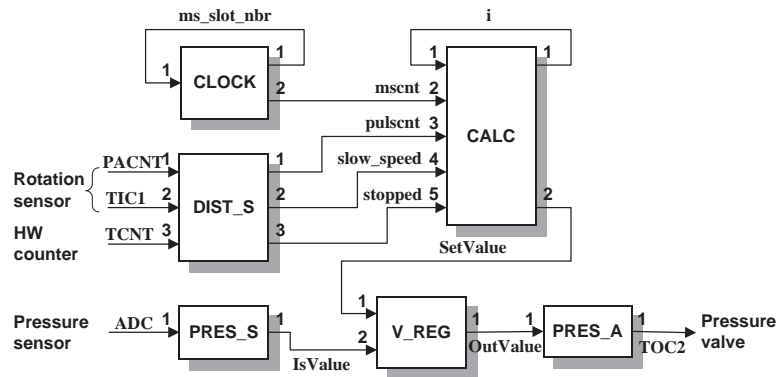


Figure 5.10: Software structure of the example target system (an aircraft arrestment system). For details, see Chapter 3.

injected will be a source node, i.e., all arcs connected to that node will be outgoing. All experiment runs in which an error was injected into that source node will be used to generate the remaining nodes (all channels that were affected at least once by a propagating error will generate nodes) and the weights of the arcs (associated with each arc is the number of errors that propagated along that path and the minimum, average and maximum propagation times).

The propagation graph is also collapsed to a propagation summary. This is a table showing, for each channel, the error count and error rate (i.e., a normalized measure between 0 and 1 of how many errors propagated into that channel) as well as the minimum, average and maximum propagation times.

5.7 Example Results Generated by PROPANE

This section presents example results obtained using PROPANE. The target system used in this example is the aircraft arrestment system described in 3. To aid the reader, the software structure shortly described here, as well.

The structure of the software is illustrated in Fig. 5.10. The numbers shown at the inputs and outputs are used for numbering the signals. For instance, *PACNT* is input #1 of *DIST_S*, and *SetValue* is output #2 of *CALC*. The software is composed of six modules of varying size and input/output signal count. The system receives input from a number of sensors at *PRES_S* and *DIST_S*. The output of the system is provided at *PRES_A*. The remaining modules (*CALC*, *V_REG* and *CLOCK*) provide internal/intermediate signals. The module specifics are provided in Chapter 3.

In this example, bit-flip errors are injected in each of the signals (one at a time) and all signals are monitored. For logging and injection, the target system was

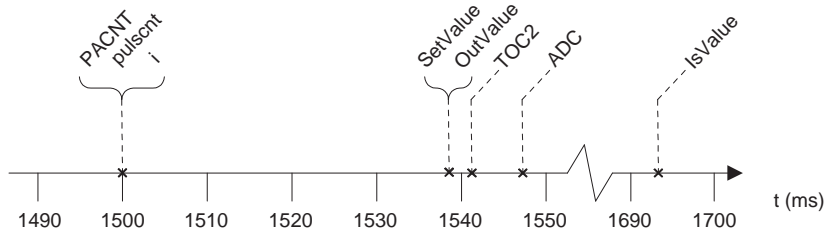


Figure 5.11: The propagation of an error injected into PACNT.

instrumented with high-level software traps. As a trap is reached during execution, an error is injected and/or data logged. The traces obtained during execution have millisecond resolution for every logged variable. Also, the software of the original target system is ported to run on a desktop system, so the intrusion of the traps is non-existent in this setup as it runs in simulated time.

First, a Golden Run (GR) was generated for each test case. Then, errors were injected into the system input signal *PACNT*, which is 8 bits wide. Bit-flips were injected in each bit position (which is the equivalent of injecting offsets of ± 1 , ± 2 , ± 4 , ± 8 , and so on). The bit-flips were injected at 10 different time instances distributed in half-second intervals between 0.5 seconds and 5.0 seconds from start of arrestment (although only at one time in each IR). In order to get a realistic load on the system and the modules, we subjected the system to 25 test cases: 5 masses and 5 velocities of the incoming aircraft uniformly distributed between 8000–20000 *kg*, and between 40–80 *m/s*, respectively. Thus, $8 \cdot 10 \cdot 25 = 2000$ injections were made into *PACNT*. Of these 2000 injections, 1840 were actually injected while an aircraft was still being arrested, i.e., the system had not yet completed its operational run. The remaining 160 errors were injected after the aircraft had stopped completely and, thus, are not considered in this example.

In one of the injection runs, we injected a bit-flip in *PACNT* 1500 milliseconds after system startup, while an aircraft is being arrested. To find how this error propagates through the system we compared the injection run with a golden run (using PDE) as described in Section 5.6. Fig. 5.11 illustrates the data analysis performed by PDE, showing when the other signals were affected by the injected error.

In Fig. 5.11 we can see that at $t = 1500$ milliseconds, *PACNT* is erroneous. The error propagates immediately through *DIST_S*, rendering also *pulsCnt* erroneous at $t = 1500$ milliseconds. This error then immediately affects *i*. At $t = 1539$ milliseconds, the error propagates out of *CALC* via *SetValue* and then immediately through *V_REG* into *OutValue*. At $t = 1541$ milliseconds, the error finally propagates out of the system (via *TOC2*) and affects the environment, leading to potential failure

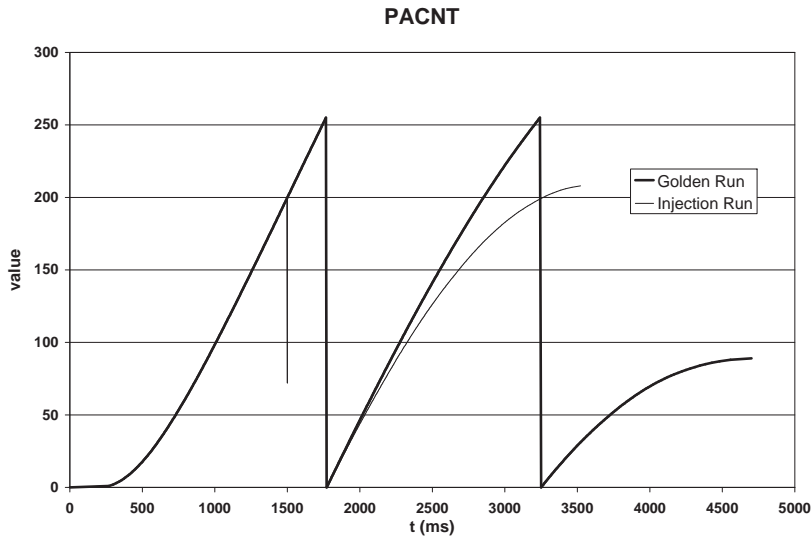


Figure 5.12: Plot of PACNT for a golden run and an injection run.

of the system. At $t = 1548$ milliseconds, the environment is affected so much that that the input to the system (*ADC*) is starting to differ from the golden run, and at $t = 1693$ milliseconds the applied pressure, as seen by the system (*IsValue*), is also different.

From PDE we can also extract data from the individual channels (probes) and plot them to see details about the propagation of errors. In Figs. 5.12 and 5.13 we have the plots for *PACNT* and for *SetValue*, respectively. Each plot contains the golden run and an injection run (the same as used in Fig. 5.11). The aircraft in the injection run was arrested in a much shorter time than in the golden run due to the high output value that resulted from the injected error. Thus, the injection run ends at $t = 3522$ milliseconds (in both figures) as compared to the golden run which ends at $t = 4701$ milliseconds.

In the plot for *PACNT* (Fig. 5.12), we clearly see that the golden run and the injection run are equal up to $t = 1500$ milliseconds where the injection run has a large dip in value. The value recovers almost instantaneously and the injection run follows the golden run closely for another half-second. However, the damage has already been done since the error propagated out of *DIST_S*. We can see how *SetValue* (Fig. 5.13) is radically different from the golden run at $t = 1539$ milliseconds and never recovers throughout the remainder of the operational time.

Apart from the detailed results shown in Figs. 5.11, 5.12 and 5.13 the PDE also generates concise information pertaining to the propagation of the injected errors in the system. For each signal that is subjected to error injections, a prop-

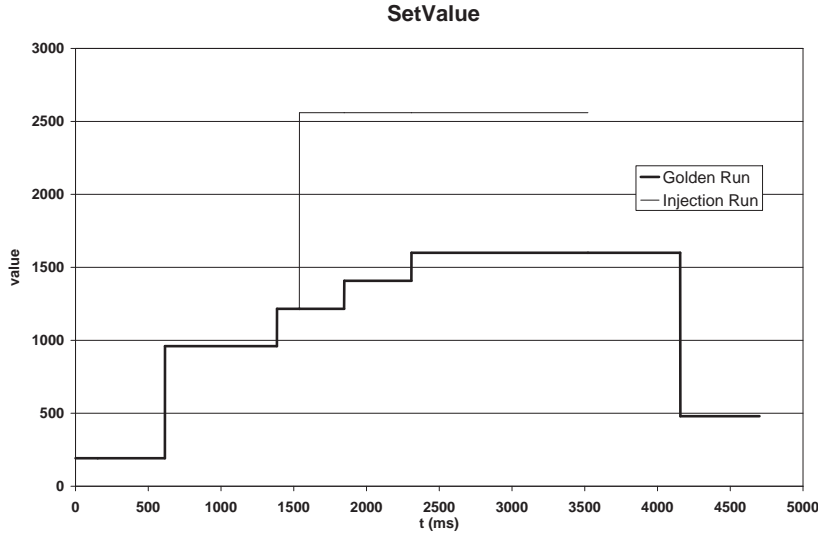


Figure 5.13: Plot of SetValue for a golden run and an injection run.

agation graph and propagation summary will be generated. The PDE stores the propagation graph in two different file formats: i) *dot* [Graphviz, web-link], and ii) *GML* [GML, web-link]. As these formats are common for graph representation, there is a range of applications that can be used for plotting and manipulating the propagation graphs. In Fig. 5.14 we can see the propagation graph for errors injected into the signal *PACNT* in the example system used in this section. The graph is generated using the *dot* tool.

The propagation graph illustrates the propagation characteristics of the errors injected into the signal *PACNT*. The label on an arc from one node to another tells how many errors propagated along this arc (top value), and the minimum, average and maximum propagation times (bottom values) for these errors. The graph shows the temporal order between errors in different signals. For example, if we consider the errors detected (during the Golden Run Comparison) in *i*, we can see that for 1120 of them, there were no errors detected earlier in other signals (although errors were detected in *pulscent* at the same point in time), whereas for 691 of the detected errors, there were errors detected earlier in *pulscent*.

Using the same example experiment as above, we show the generated propagation summary for errors in *PACNT* in Table 5.3. The summary is obtained by collapsing all ingoing arcs of each node in the propagation graph. Thus, for example, the summary for *i* is obtained by adding its two ingoing arcs in the propagation graph, which gives us a total of 1811 errors. The propagation times are obtained from the combined set of propagation times for the errors detected in *i*.

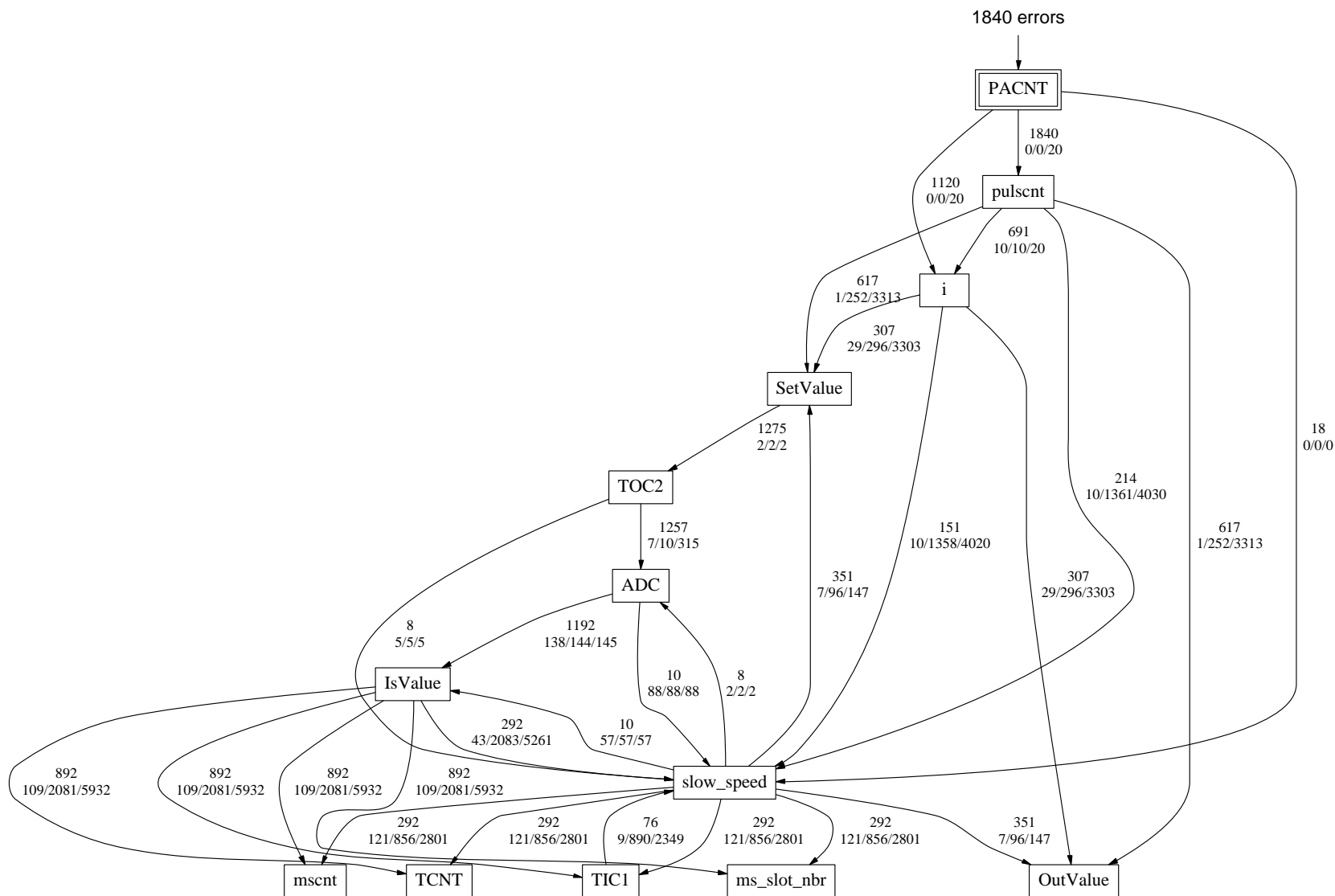
Figure 5.14: Propagation graph (generated by the *dot* tool) for errors injected in PACNT.

Table 5.3: Propagation of errors injected into PACNT.

error count is the number of errors detected using Golden Run Comparison and the error rate is the same information normalized. The propagation times are all in milliseconds.					
Signal	error count	error rate	t_{min}	t_{avg}	t_{max}
PACNT	1840	1.000	0	0	0
pulscnt	1840	1.000	0	0	20
i	1811	0.984	0	4	20
OutValue	1275	0.693	1	613	4159
SetValue	1275	0.693	1	613	4159
TOC2	1275	0.693	3	615	4161
ADC	1265	0.688	10	629	4168
IsValue	1202	0.653	155	682	3467
slow_speed	769	0.418	0	2004	5890
mscnt	1184	0.643	476	2982	6201
ms_slot_nbr	1184	0.643	476	2982	6201
TCNT	1184	0.643	476	2982	6201
TIC1	1184	0.643	476	2982	6201

In the summary shown in Table 5.3 we see the number of errors in *PACNT* that caused errors in other signals (count and rate), as well as the minimum, average and maximum propagation time for these errors (the rows are ordered according to their average propagation time). In this particular example we can see that all of the 1840 errors injected into *PACNT*, propagated to *pulscnt* with an average propagation time of 0 milliseconds. 1275 errors made it all the way to the output signal *TOC2* with an average propagation time of 615 milliseconds. From the software structure shown in Fig. 5.10 we can see that errors in the signals listed below *TOC2* in Table 5.3 (except *slow_speed*), must be indirect, since there is no direct path from *PACNT*. Thus, errors in this signal must have propagated out of the system into the environment and then back into the system again.

The results presented give information on how errors propagate through the system, identifying which modules and signals that may be in need of special mechanisms for protection against propagating errors. For example, from the results in Table 5.3 we see that errors in *PACNT* mainly propagate through *DIST_S* into *CALC* using *pulscnt*. From the propagation graph in Fig. 5.14 we see that propagation into *CALC* is fast, whereas propagation out of *CALC* takes a little longer. Thus, *CALC* seems to delay the propagation of errors. We also see that after *CALC*, error propagation again is swift. These results would indicate that system reliability could increase if *pulscnt* were to be equipped with EDM's (error detection mechanisms) and ERM's (error recovery mechanisms), as this would likely break the propagation at an early stage.

These examples demonstrate PROPANE's capabilities for generating pertinent information for propagation analysis. However, the level of detail required may generate very large amounts of raw data. In order to analyse this raw data (further than done by the PDE) and find likely propagation paths and vulnerable modules, additional actions can be performed to reduce the raw data into useful information. We refer the reader to [Hiller *et al.*, 2001, Hiller *et al.*, 2002(a), Jhumka *et al.*, 2001], where details of actual results, as well as two different data analysis frameworks (with different objectives) are described.

5.8 PROPANE's Attributes and Comparison with Other FI-tools

FI has been used for various purposes for many years (see, e.g., [Arlat *et al.*, 1990], [Arlat *et al.*, 1993], [Chillarege and Bowen, 1989], and [Iyer, 1995]). FI-tools may inject errors in a number of ways, e.g., physical fault injection, radiation, etc. As PROPANE uses SWIFI (SoftWare-Implemented Fault Injection) we will focus this comparison on other tools in that area. Tables 5.4 and 5.5 is a summarized comparison of the tools we have taken a closer look at. First, however, we will summarize the main characteristics of PROPANE as described over the previous sections.

5.8.1 Main Characteristics of PROPANE

The goal of designing PROPANE was to provide an environment with four main characteristics: 1) high flexibility regarding fault/error models and injection options, 2) high portability, 3) low-cost operation, and 4) high observability in target software. These characteristics were achieved in the following way:

Flexibility PROPANE's main logging and injection functionality is provided in a function library which makes it target independent. Thus, the range of applications where PROPANE can be used for error propagation analysis is vast. Also, PROPANE has the capability to handle user-defined error types and injection triggers. The fact that the final data analysis is performed by a separate tool lets the user define exactly the kind of measures each experiment is to produce. Thus, raw data from one experiment may be used for multiple purposes. The only requirement for using PROPANE is that the source language used in the software system is able to interface with libraries implemented in C.

Portability PROPANE is entirely implemented in C. Furthermore, the PROPANE System Instrumentor, the PROPANE Setup Creator, the PROPANE Library

and the PROPANE Data Extractor have no links to the OS making it very easy to port them to other platforms as it only requires recompiling. The PROPANE Campaign Driver is also implemented in C but has a minor link to the OS and that is for spawning new processes. As all desktop operating systems provide an API for process handling, this part will also be easy to port.

Low-cost No external hardware or network is required in order for PROPANE to be able to function properly (although the target system may require it). This, together with the high level of portability, makes PROPANE an attractive low-cost alternative for error propagation analysis for many different platforms.

Observability PROPANE is capable of gathering detailed data on the internal variables and events of software systems during execution and fault injection. This level of detail is necessary when analyzing the propagation of errors in order to locate vulnerable software modules and/or variables and to identify likely propagation paths.

Having described the main characteristics of the PROPANE environment, we present a comparison with other contemporary FI-tools.

5.8.2 Comparison Details

Here we compare PROPANE with a number of other FI-tools. In Tables 5.4 and 5.5, we first categorize the FI-tools into four main categories based on their aim and injection capabilities, namely: 1) tools for evaluation of system architectures, 2) tools for evaluation of systems when subjected to HW faults only, 3) tools for evaluating systems when subjected to HW and SW faults, and 4) tools for evaluating software and software components when subjected to errors in internal data (which may be induced by both HW or SW faults). All tools in categories 1 through 3 are aimed at evaluation at system level, i.e., the granularity of the observed results are at system level (or node level for distributed systems). In order to investigate the error propagation characteristics of software, it is imperative that the individual variables of the software system can be observed during the experiments. The tools in category 4 are all able to observe the target system at a very low level. As PROPANE is in this category, we will focus our comparison on these tools.

MAFALDA is aimed at evaluating the robustness of micro-kernels and investigating the effect of software faults and software errors on the operation of these kernels. This means that the tool is able to inject at the OS-level. PROPANE is

Table 5.4: Summarized comparison (part 1) of different FI-tools — see text in Section 5.8 for details.

Key: Obs Res = observation resolution, FIM = fault insertion method, Err Ty = error types (B = Bit manipulation, AB = Address bus, AR = Arithmetic, U = User-defined), Err Tr = error triggers (B = Built-in, U = User-defined), Ext Req = external requirements, Instr = instrumentation needs, LoD = level of detail of readout data							
Tool	Obs Res	FIM	Err Ty	Err Tr	Ext Req	Instr	LoD
Tool Category/Driver: Evaluate architecture (category 1)							
DEPEND [Goswami, 1997]	System, architecture	None	Component level errors	B	OS support, source code of architecture simulation	Logging, injection	Injection details and predefined system-level data.
Loki [Chandra <i>et al.</i> , 2000] [Cukier <i>et al.</i> , 1999]	System, architecture	None	B, U	U	Target source code, function library	Logging, injection	Injection details and predefined system/node-level data.
Tool Category/Driver: Evaluate tolerance against HW faults (category 2)							
FERRARI [Kanawati <i>et al.</i> , 1995]	System	Op code switch	B, AB	B	UNIX	No	Injection details and predefined data.
DOCTOR [Han <i>et al.</i> , 1995]	System	Mutation	B, Communication	B	HW and OS support	Logging	Injection details and predefined data.
Xception [Carreira <i>et al.</i> , 1995] [Carreira <i>et al.</i> , 1998]	System architecture	Op code switch	B, AB	B	Processor with debugging port (e.g., BDM)	No	Injection details and predefined data.
Tool Category/Driver: Evaluate tolerance against both HW and SW faults (category 3)							
DEFINE [Kao and Iyer, 1995]	UNIX-network	Op code switch	B, AB	B	UNIX, target source code	Logging, injection	Injection details, predefined data and memory locations.

Table 5.5: Summarized comparison (part 2) of different FI-tools — see text in Section 5.8 for details.

Key: Obs Res = observation resolution, FIM = fault insertion method, Err Ty = error types (B = Bit manipulation, AB = Address bus, AR = Arithmetic, U = User-defined), Err Tr = error triggers (B = Built-in, U = User-defined), Ext Req = external requirements, Instr = instrumentation needs, LoD = level of detail of readout data							
Tool	Obs Res	FIM	Err Ty	Err Tr	Ext Req	Instr	LoD
Tool Category/Driver: Evaluate tolerance against both HW and SW faults (category 3, cont'd)							
FIAT [Barton <i>et al.</i> , 1990] [Segall <i>et al.</i> , 1988]	System architecture	Bit-flips in task image	B	B	HW and OS support, target source code	Injection	Injection details and predefined data.
FTAPE [Tsai and Iyer, 1996]	System	No	B, Disk	B	HW and OS support	No	Injection details and predefined data.
Tool Category/Driver: Evaluate effect of SW faults and errors and error propagation (category 4)							
MAFALDA [Fabre <i>et al.</i> , 1999]	Micro-kernels	Bit-flips in text segment	B	B	HW support	No info available	Injection details, predefined data and special events.
NFTAPE [Stott <i>et al.</i> , 2000]	System down-to variable	User-defined injectors	B, AB, AR, U	B, U	LAN-based, target source code, function library	Logging, injection	Injection details, predefined data and user-defined data. Events and individual variables may be logged.
PROPANE	System down-to variable	Mutation, User-defined injectors	B, AR, U	B, U	Target source code, function library	Logging injection	Injection details and user defined data. Events and individual variables may be logged.

aimed at software at the USER-level, hence it is not suited for these types of investigations. However, as far as we know, MAFALDA lacks comprehensive logging facilities for examining the propagation of errors in a micro-kernel. NFTAPE is, in our opinion, a very versatile tool which can perform the same investigations PROPANE can. NFTAPE, just like PROPANE, has support for user-defined injectors as well as user-defined triggers, and is capable of observing the target system at the variable level. As both tools have support for user-defined injectors, both may be extended to handle physical fault injection as well as SWIFI. However, NFTAPE is designed to run on a LAN, and has therefore a separate control host and a target node.

PROPANE, is a single software package primarily designed to run on a single node. This makes the setup time required for experiments using PROPANE very short (e.g., the experiment in [Hiller *et al.*, 2001], which also functions as the example used in this chapter, was set up in just a few hours, where the main part of the time was spent on instrumenting the target software). Also, the system requirements for using PROPANE are very low, the main requirements are set by the target system. On account of the fact that PROPANE requires no special support from either HW or the OS (PCD requires that the OS has an API for spawning new processes), porting the tool to other platforms is an easy task which mainly just requires a recompilation. This also makes PROPANE less expensive than NFTAPE.

Based on this comparison we can argue for the use of PROPANE in early stages of software development before HW platforms have been finalized or when the entire system may be simulated on a single node. NFTAPE may be used when development has come so far that functional setups of the entire target system are available. In our opinion, the remaining tools listed here are not capable of generating data that may be used in a detailed investigation of error propagation in software.

The injection and logging functionality of PROPANE is provided as a static library, which is linked with the target system. This offers flexibility in choosing target systems, facilitating wide applicability of the tool. For example, in addition to the way it is used in the example provided in this chapter, it may be used in a manner similar to DEPEND where entire system architectures are simulated.

In our opinion, PROPANE is well suited for use as a design stage tool which gives valuable insights into the error propagation characteristics of a software system such that resources for error detection and recovery efforts can be directed to those parts which require it the most, i.e., those parts which let errors propagate and those parts which attract propagating errors. This makes PROPANE a good complement to other available analysis tools.

5.9 Summary and Conclusions

This chapter presents PROPANE, the Propagation Analysis Environment, which is a software design-stage profiling tool developed for analyzing the propagation and effect of errors in software systems. PROPANE is a desktop environment and contains support for conducting fault and error injections in target software systems. The tool also provides support for automatically inserting probes into the target system enabling the logging of variables and events during injection experiments.

PROPANE is target system independent, i.e., it may be used on any target system provided that one can execute it in a desktop environment. Also, PROPANE does not require any HW or OS support and is easily ported to other operating systems (the current version is available for Windows 2000/XP-based computers). As PROPANE is implemented using ANSI C, porting it is mostly just a question of recompiling for the desired environment.

The injection capabilities include fault injection by mutation of source code as well as SWIFI-based injection of errors. PROPANE supports user-defined injectors and triggers which makes it capable of supporting other injection techniques than SWIFI (for example, physical fault injection).

PROPANE supports observations down to the variable level, i.e., individual variables may be logged during injection experiments. This enables the detailed examination of error propagation in software and is a valuable help in finding vulnerable software modules and/or variables.

For analysis, the toolkit contains the PROPANE Data Extractor (PDE), which can perform Golden Run Comparisons for each channel created by a variable in the readout files. The results will be stored in a text file with a spreadsheet format that is easily imported into other tools for further analysis. The results from the GRC are also compiled to show where errors propagate through the system and how long time it takes.

PDE can also extract injection information from the readout files and store this in separate files, and create channel logs for each individual channel of each individual experiment if a more detailed analysis or graphical representation is desired. Also, PDE creates propagation graphs and summaries which visualize the propagation characteristics of the software system.

To demonstrate the tool we have shown detailed results from an injection experiment performed on an actual embedded control system used for arresting aircraft (similar to the cable-and-hook systems found on aircraft carriers).

In a comparison with other FI-tools, we comment that PROPANE is unique in the level of its provision of detailed data necessary for error propagation analysis.

Also, the portability and flexibility uniquely distinguish PROPANE as a simple, low-cost error propagation analysis tool which lends itself for early analysis of software systems. It complements other, more expensive, tools which can be used later during development.

CHAPTER 6

Error Propagation and Effect Analysis

Probable impossibilities are to be preferred to improbable possibilities.

— Aristotle (384–322 B.C.)

This chapter presents a novel approach for profiling software by analyzing the propagation and effect of data errors in software. The concept of *error permeability* is introduced as a basic measure upon which we define a set of related measures. These measures guide us in the process of analyzing i) the vulnerability of software to find the modules and signals that are most likely exposed to propagating errors, and ii) find the modules and signals which, when erroneous, tend to cause more damage than others from a systems operation point-of-view. Based on the analysis performed with error permeability and its related measures, we discuss how to select suitable locations for error detection mechanisms (EDM's) and error recovery mechanisms (ERM's). A method for experimental estimation of error permeability, based on fault injection, is described and the software of a real embedded control system analyzed to show the type of results obtainable by the analysis framework. The results highlight the utility of the developed framework in being useful for analyzing error propagation and error effect such that knowledge is gained regarding the vulnerability of the software system at hand, and for driving the process for effective placement of EDM's and ERM's.

6.1 Introduction

Software based functionality in embedded control systems usually comprises numerous discrete modules interacting with each other to provide a specific task or service. With an error (as defined in [Laprie, 1995]) present in a software module, there is a likelihood that this error can propagate to other modules with which it interacts. Knowing where errors propagate in a system is of particular importance for a number of development activities. Propagation analysis may be used to find the module which are most exposed to errors in a system, and to ascertain how different modules affect each other in the presence of errors. In addition to knowing propagation characteristics it is also important to know where errors are likely to do the most damage. Note that those errors which are likely to propagate are not always those that are most likely to cause great damage. Thus, it is important to do an analysis of both these notions to identify the most vulnerable parts of a system. Furthermore, error propagation analysis and error effect analysis also gives an insight on locations in the system that would be best suited for placement of error detection mechanisms (EDM's) and associated error recovery mechanisms (ERM's).

Apart from the technical issues that can be addressed using propagation and effect analysis, there are also issues pertaining to project and resource management. Error propagation and effect analysis may be used as a means of obtaining information for use in decisions on where additional resources for dependability development are necessary and to determine where they would be most cost effective. Software is common not only in applications such as aircraft or other high-cost systems, but also in consumer-based cost-sensitive systems, such as cars. These systems often require both development costs and production costs to be kept low. Analyzing error propagation and error effects can also complement other analysis activities, for instance FMECA (Failure Mode Effect and Criticality Analysis). Consequently, modules and signals found to be vulnerable and/or critical during propagation and effect analysis might be given more attention during design activities. Thus, error propagation and effect analysis, as a means of both system analysis and resource management, may be a very useful design-stage tool in such systems.

In this chapter, an approach for analyzing error propagation and error effect in software based systems is presented. The basic intent of this framework is software level data errors, thus considering distributed software functions resident on either single or distributed hardware nodes. The approach adopts a black-box view on modular software and introduces the measure *error permeability* as well as a set of related measures, and subsequently a methodology is defined for using these measures to obtain information on error propagation and error effect, and also for

identifying candidate locations for placement of error detection and recovery mechanisms. The basic definition of error permeability is the probability of an error in an input signal of a given module permeating to one of the output signals of that module (there is one permeability value assigned to each pair of input/output signals of each module of the software system). The related measures are divided into three categories covering *exposure*, *impact* and *criticality*. Thus, the combined framework is called EPIC (Exposure, Permeability, Impact and Criticality). EPIC provides a consolidated and comprehensive software profiling framework building upon the individual framework components introduced in [Hiller *et al.*, 2001] and [Hiller *et al.*, 2002(a)].

The remainder of this chapter is organized as follows: In Section 6.2 we briefly describe the assumed system model (details are found in Chapter 3). The EPIC framework is described in Section 6.3 and methods for estimating numerical values of the introduced metrics are discussed in Section 6.4. In Section 6.5 the framework is illustrated on an example target system and estimates are produced for the various metrics which are then used in Section 6.6 to select locations for Executable Assertions (which are also evaluated with regard to detection coverage). Some limitations and caveats are identified and discussed in Section 6.8. Finally, Section 6.9 contains a summary and conclusions.

6.2 Software and System Model

The framework presented in this chapter assumes software systems to be constructed according to Chapter 3. In short, the framework assumes software built up of discrete black-box software modules inter-linked with some form of signaling (for information and data exchange, e.g., shared memory or message passing).

6.3 EPIC: Generating Software Profiles

The EPIC framework aims at providing a means of profiling software such that suitable locations for error detection and recovery mechanisms can be identified. To achieve this, EPIC charts the propagation and effect of errors, i.e., how errors propagate through a software system and their effect on system operations. Our focus here is on data errors – erroneous values in the internal variables and signals of a system.

A data error has a probability of affecting the system such that further errors are generated during operation. If one could obtain knowledge of the error propagation



Figure 6.1: A basic black-box software module with m inputs and n outputs

characteristics of a particular system, this would aid the development of techniques and mechanisms for detecting and eventually correcting the error.

Such knowledge can translate to improved effectiveness of error detection and handling and the consequent cost/performance-ratio of these mechanisms, as the efforts can be concentrated to those areas of the system to where errors tend to propagate. The results obtained using the EPIC framework are useful even with minimal knowledge of the distribution of the occurring errors, i.e., if one does not know which errors are most likely to appear. Having such knowledge would certainly improve the value of the results, but performing the analysis without it still provides qualitative insights on system error susceptibility.

This section will describe the framework starting with the conceptual basis of *error permeability*. Upon this we will define a set of measures and techniques which can be used for generating two distinct profiles of a software system: i) error propagation profile and ii) error effect profile.

6.3.1 Error Permeability - Letting Errors Pass

In our approach, we introduce the measure of *error permeability*, and based on it we define a set of related measures that cumulatively provide an insight on the error propagation and effect characteristics and vulnerabilities of a system.

Consider the software module in Fig. 6.1 (at this point only discrete software modules are considered). Starting with a simple definition of error permeability, refinements will follow successively. For each pair of input and output signals, the *error permeability* is defined as the conditional probability of an error occurring on the output given that there is an error on the input. Thus, for input i and output k of a module \mathbf{M} , the *error permeability*, $P_{i,k}^M$, is defined as follows:

$$0 \leq P_{i,k}^M = Pr\{\text{error in output } k | \text{error in input } i\} \leq 1 \quad (6.1)$$

This measure indicates how *permeable* an input/output pair of a software module is to errors occurring on that particular input. One major advantage of this definition of error permeability is that it is independent of the probability of error occurrence on

the input. This reduces the need for having a detailed model of error occurrence. On the other hand, error permeability is still dependent on the workload of the module as well as the type of the errors that can occur on the inputs. It should be noted that if the error permeability of an input/output pair is zero, this does not necessarily mean that the incoming error did not cause any damage. The error may have caused a latent error in the internal state of the module that for some reason is not visible on the outputs. In Section 6.4, we describe an approach for experimentally estimating values for this measure.

Error permeability is the basic measure for characterizing error propagation, upon which we develop related refined measures. This basic measure is defined at the signal level, i.e., an error permeability value characterizes the propagation from one input signal to one output signal in a given module. Going to the module level (Fig. 6.1), we define the *relative permeability*, P^M , of a module \mathbf{M} with m input signals and n output signals, to be:

$$0 \leq P^M = \left(\frac{1}{m} \cdot \frac{1}{n} \right) \sum_i \sum_k P_{i,k}^M \leq 1 \quad (6.2)$$

Note that this does not necessarily reflect the overall probability that an error is permeated from the input of the module to the output. Rather, it is an abstract measure that can be used to obtain a relative ordering across modules. If all inputs are assumed to be independent of each other and errors on one input signal can only generate errors on one output signal at a time, then this measure is the actual probability of an error on the input permeating to the output. However, this is seldom the case in most practical applications.

At this stage, one potential limitation of this measure is that it is not possible to distinguish modules with a large number of input and output signals from those with a small number of input and output signals. This distinction is useful to ascertain as modules with many input and output signals are likely to be central parts (almost like hubs) of the system thereby attracting errors from different parts of the system. In order to be able to make this distinction, we remove the weighting factor in Eq. 6.2, thereby, in a sense, “punishing” modules with a large number of input and output signals. Thus, for a module \mathbf{M} with m input signals and n output signals, we can define the *non-weighted relative permeability*, \hat{P}^M as follows:

$$0 \leq \hat{P}^M = \sum_i \sum_k P_{i,k}^M \leq m \cdot n \quad (6.3)$$

Similar to the relative permeability, this measure does not have a straightforward real-world interpretation but is a measure that can be used during development to

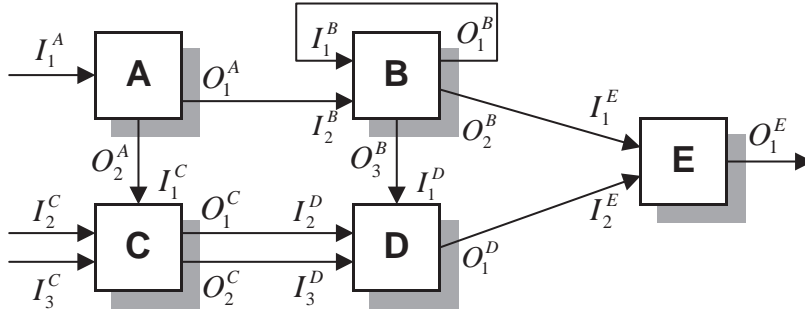


Figure 6.2: An example software system with five modules

obtain a relative ordering across modules. The larger this value is for a particular module the more effort has to be spent in order to increase the error containment capability of that module (which is the same as decreasing the error permeability of the module), for instance by using wrappers as in [Salles, 1999]. Note that, as the maximum value of each individual permeability value is 1, the upper bound for this measure is the product of the number of inputs (m) and outputs (n).

The two measures defined in Eqs. 6.2 and 6.3 are both necessary for analyzing the modules of a system. For instance, consider the case where two modules, **G** and **H**, are to be compared. **G** has few inputs and outputs, and **H** has many. Then, if $P^G = P^H$, then $\hat{P}^G < \hat{P}^H$. And vice versa, if $\hat{P}^G = \hat{P}^H$, then $P^G > P^H$.

6.3.2 Ascertaining Propagation Paths

So far, we have obtained error permeability factors for each discrete software module in a system. Considering every module individually does have limitations; this analysis will give insights on which modules are likely (relatively) to transfer incoming errors, but will not reveal modules likely to be exposed to propagating errors in the system. In order to gain knowledge about the exposure of the modules to propagating errors in the system we define the following process which now considers interactions across modules.

Consider the example software system shown in Fig. 6.2. Here we have five modules, **A** through **E**, connected to each other with a number of signals. The i^{th} input of module **M** is designated I_i^M and the k^{th} output of module **M** is designated O_k^M . External input to the system is received at I_1^A , I_2^C and I_3^C . The output produced by the system is O_1^E .

Once we have obtained values for the error permeability for each input/output pair of each module, we can construct a *permeability graph* as illustrated in Fig. 6.3.

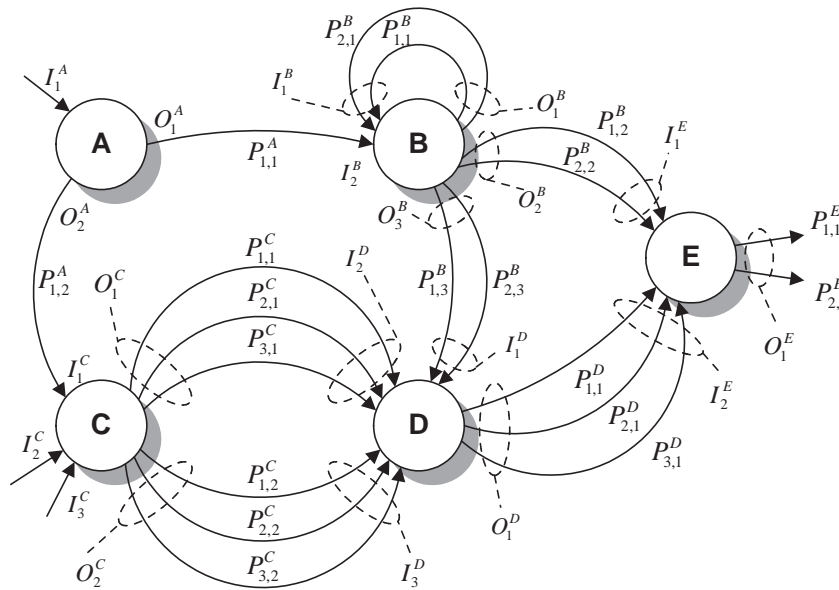


Figure 6.3: Permeability graph for the example software system in Fig. 6.2

Each node in the graph corresponds to a particular module and has a number of incoming arcs and a number of outgoing arcs. Each arc has a weight associated with it, namely the error permeability value. Hence, there may be more arcs between two nodes than there are signals between the corresponding modules. Actually, the maximum number of outgoing arcs for a node is the product of the number of incoming signals and the number of outgoing signals for the corresponding software module (each input/output pair of a module has an error permeability value). Arcs with a zero weight (representing non-permeability from an input to an output) can be omitted. With this permeability graph we can perform two different propagation analyses, namely:

- A** Backtrack from system output signals to system input signals in order to find those paths which have the highest probability of error propagation (*Output Error Tracing*), or
- B** Trace errors from system input signals to system output signals in order to find which paths these errors will most likely propagate along (*Input Error Tracing*).

Output Error Tracing is easily accomplished by constructing a set of *backtrack trees*, one for each system output. These backtrack trees can be constructed quite simply based on the following steps on the permeability graph, namely:

-
- A1. Select a system output signal and let it be the root node of the backtrack tree.
 - A2. For each error permeability value associated with the signal, generate a child node that will be associated with an input signal.
 - A3. For each child node, if the corresponding signal is not a system input signal, backtrack to the generating module and determine the corresponding output signal. Use this signal and construct the sub-tree for the child node from A2. If the corresponding signal is a system input signal it will be a leaf in the tree. If the corresponding signal is an input signal to the same module it will be a leaf in the tree (as opposed to other leaves which are system input signals). We do not follow the recursion that is generated by the feedback.
 - A4. If there are more system output signals, go back to A1.
-

This will, for each system output, give us a backtrack tree where the root corresponds to the system output, the intermediate nodes correspond to internal outputs and the leaves correspond to system inputs (or module inputs receiving feedback from its own module). Also, all vertices in the tree have a weight corresponding to an error permeability value. Once we have obtained this tree, finding the propagation paths with the highest propagation probability is simply a matter of finding which paths from the root to the leaves have the highest weight.

Input error tracing is achieved similarly. However, instead of constructing a backtrack tree for each system output, we construct a *trace tree* for each system input, as follows:

- B1. Select a system input signal and let it be the root node of the trace tree.
 - B2. Determine the receiving module of the signal and for each output of that module, generate a child node. This way, each child node will be associated with an output signal.
 - B3. For each child node, if the corresponding signal is not a system output signal, trace the signal to the receiving module and determine the corresponding input signal. Use this signal and construct the sub-tree of the child node from B2. If the corresponding signal is a system output signal it will be a leaf in the tree. If the input signal is the same module that generated the output signal (i.e. we have a module feedback) then follow this feedback once and generate the sub-trees for the remaining outputs. We do not follow the recursion generated by this feedback.
 - B4. If there are more system input signals, go back to B1.
-

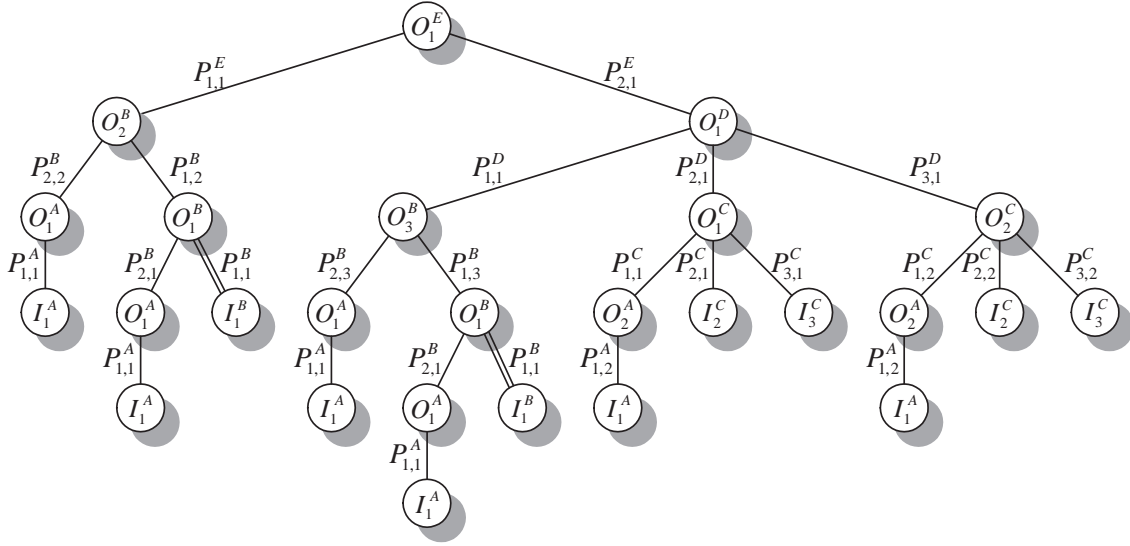


Figure 6.4: Backtrack tree of system output signal O_1^E of the example system.

This procedure results in a set of trace trees - one for each system input. In a trace tree, the root will represent a system input, the leaves will represent system outputs, and the intermediate branch nodes will represent internal inputs. Thus, all vertices will be associated with an error permeability value. From the trace trees we find the propagation pathways that errors on system inputs would most likely take by finding the paths from the root to the leaves having the highest weights.

The case when an output of a module is connected to an input of the same module is handled in the way described in step A3 of the backtrack tree generation script. If we would use recursive sub-tree generation we would get an infinite number of sub-trees with diminishing probabilities. As all permeability values are ≤ 1 , the sub-tree with the highest probability is the one which only goes one pass through the feedback loop and this path is included in the permeability tree. In [Fujiwara and Shimono, 1983], [Goel, 1981] and [Roth, 1980] similar techniques have been utilized for hardware error propagation analysis.

The backtrack tree for system output O_1^E of the example system is shown in Fig. 6.4. Here we observe the double line between I_1^B and O_1^B . This notation implies that we have a local feedback in module **B** (O_1^B is connected to I_1^B) and represents breaking up of the propagation recursion.

The weight for each path is the product of the error permeability values along the path. For example, in Fig. 6.4, the path from O_1^E to I_1^A going straight from O_1^E (connected to I_2^B) to O_2^B (the leftmost path in the tree) has the probability $P = P_{1,1}^A \cdot P_{2,2}^B \cdot P_{1,1}^E$. This is the conditional probability that, given an error in O_1^E and

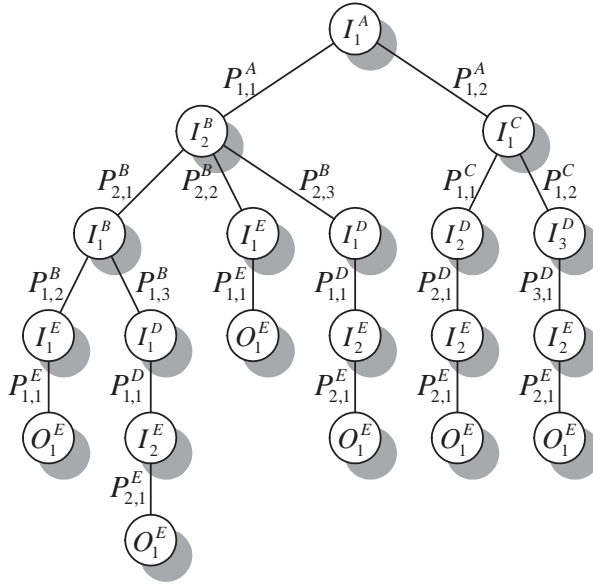


Figure 6.5: Trace tree for system input signal I_1^A of the example system.

the error originated from I_1^A , it propagated directly through O_2^B which is connected to I_1^E and then to O_1^E .

If we have knowledge regarding the probability of errors appearing on the input signals we can use these probabilities as additional weights on the paths. For example, if the probability of an error appearing on I_1^A is $Pr(I_1^A)$, then the P can be adjusted with this factor, giving us $P' = Pr(I_1^A) \cdot P_{1,1}^A \cdot P_{2,2}^B \cdot P_{1,1}^E$. This is the probability of an error appearing on system input I_1^A , propagating through module **B** directly via O_2^B to system output O_1^E .

The trace tree for system input I_1^A is shown in Fig. 6.5. Here we can see which propagation path from system input to system output has the highest probability. As for backtrack trees, the probability of a path is obtained by multiplying the error permeability values along the path. For example, in Fig. 6.5, the probability of an error in I_1^A propagating to module **C** and via its output O_2^C to module **D** and from there via module **E** to system output O_1^E is $P = P_{1,2}^A \cdot P_{1,2}^C \cdot P_{3,1}^D \cdot P_{1,1}^E$. Again, if we know that $Pr(I_1^A)$ is the probability of an error appearing on I_1^A , then we can adjust P to get $P' = Pr(I_1^A) \cdot P_{1,2}^A \cdot P_{1,2}^C \cdot P_{3,1}^D \cdot P_{1,1}^E$.

6.3.3 Assessing the Error Exposure of Modules and Signals

Using the backtrack and trace trees enables determining two specific aspects: (a) the paths in the system that errors will most likely propagate along to get to certain output signals, and (b) which output signals are most likely affected by errors occur-

ring on the input signals. With this knowledge we can start selecting locations for the EDM's and ERM's that we may want to incorporate into our system based on system reliability/safety requirements.

One problem remains though: once we have the most probable propagation paths, we still have to find the modules along that path that are the best to target with EDM's and ERM's. Earlier, in Eqs. 6.2 and 6.3, we had defined two measures, *relative permeability* and *non-weighted relative permeability*, that can guide us in this search.

These measures only consider the permeability values of discrete modules – couplings across modules are disregarded. Using the permeability graph, we now define a set of measures that explicitly consider coupling and aid determining locations for EDM's and ERM's. To find modules most likely to be exposed to propagating errors, we want to have some knowledge of the amount of errors that a module may be subjected to. For this we define the *error exposure*, X^M , of a module \mathbf{M} as:

$$X^M = \frac{1}{N} \sum \text{weight of all incoming arcs of } M \quad (6.4)$$

where N is number of incoming arcs and M is the node in the permeability graph, representing software module \mathbf{M} . This measure does not consider any correlation that may exists between two or more incoming arcs. Since we use this as a relative measure, this is not a concern for us. The *error exposure* is the mean of the weights of all incoming arcs of a node and is bounded as $\frac{1}{N}$. Analogous to the *non-weighted relative permeability*, we can also define the *non-weighted error exposure*, \hat{X}^M , of a module \mathbf{M} as:

$$\hat{X}^M = \sum \text{weight of all incoming arcs of } M \quad (6.5)$$

This measure does not have a real-world interpretation either – it is used only during system analysis to obtain a relative ordering between modules. The two exposure measures (Eqs. 6.4 and 6.5) along with the previously defined permeability measures (Eqs. 6.2 and 6.3) will be the basis for the analysis performed to obtain information upon which to base a decision about locating EDM's and ERM's. As was the case for the two relative permeability measures, the two exposure measures, *error exposure* and *non-weighted error exposure*, are used for distinguishing between nodes with a small number of incoming arcs and those with a large number.

The error exposure measures defined in Eqs. 6.4 and 6.5 indicate which modules will most probably be the ones exposed to errors propagating through the system. If we want to analyse the system at the signal level and get indications on which signals might be the ones that errors most likely will reach and propagate through,

we can define a measure which is the equivalent of the error exposure defined in Eq. 6.4, but is only calculated for one signal at a time. In the backtrack trees we can easily see which error permeability values are directly associated with a signal S . We define the set S_p as composed of all unique arcs going to the child nodes of all nodes generated by the signal S . A signal may generate multiple nodes in a backtrack tree (see for instance signal O_1^B in the backtrack tree in Fig. 4). However, in the set S_p , the permeability values associated with the arcs emanating from those nodes will only be counted once. The *signal error exposure*, X_s^S , of signal S is then calculated as:

$$X_s^S = \sum \text{all permeability values in } S_p \quad (6.6)$$

The interpretation for the signal error exposure is the same as for the error exposure of a module, but at a signal level. That is, the higher a signal error exposure value, the higher the probability of errors in the system being propagated through that signal.

It may be difficult to give strict rules for selecting the EDM and ERM locations. A discussion on how to identify candidate locations is provided in Section 6.3.5, and an example study demonstrating the actual process depicted is provided in Section 6.5.

We have now defined a basic analytical framework for ascertaining measures pertaining to error propagation and software vulnerability. In the following sections we augment the framework with measures for analyzing the effect of errors on the final output of the system as well as for obtaining a measure of criticality of signals. The knowledge gained in the propagation analysis combined with the knowledge gained in the impact analysis will help in finding suitable locations for EDM's and ERM's.

6.3.4 Analyzing the Effect of Errors on System Output

When selecting locations for EDM's and ERM's, it may be insufficient to only take into account the propagation characteristics of data errors for a given software system. Errors that may have a low probability of propagating may still cause severe damage should propagation occur. Taking this into account we now define measures which let us analyse to what extent errors in a signal (system input signal or intermediate signal) affect the system output, i.e., what is the *impact* of errors on the system output signals.

As errors in a source signal can propagate along many different paths to the (destination) system output signal we must consider this in our definition of impact.

In order to calculate the impact of errors in a signal S_s on a system output signal O^{Sys} we must first generate an *impact tree*, which is a generalization of the trace tree described in Section 6.3.2. Instead of generating a trace tree with a system input as root node, we use the signal of interest in our analysis as the root, in this case S_s . An impact tree is generated using the following steps:

-
- C1. Select a signal S_s and let it be the root node of the impact tree.
 - C2. Determine the receiving module of the signal and for each output of that module, generate a child node. This way, each child node will be associated with an output signal.
 - C3. For each child node, if the corresponding signal is not a system output signal, trace the signal to the receiving module and determine the corresponding input signal. Use this signal and construct the sub-tree of the child node from C2. If the corresponding signal is a system output signal it will be a leaf in the tree. If the input signal is the same module that generated the output signal (i.e. we have a module feedback) then follow this feedback once and generate the sub-trees for the remaining outputs. We do not follow the recursion generated by this feedback.
 - C4. Repeat the procedure from C1 for each signal (system input and intermediate signals) in the system.
-

Once we have generated the impact tree for a given signal S_s , we generate all the propagation paths from the root to the leaves containing system output signal O^{Sys} (there may be leaves which are generated by other system output signals). Each path has a weight associated with it which is the product of all permeability values along that path. We define $S_s \rightsquigarrow O^{Sys}$, the *impact* of (errors in) S_s on O^{Sys} , as

$$0 \leq S_s \rightsquigarrow O^{Sys} = 1 - \prod_i (1 - w_k) \leq 1 \quad (6.7)$$

where w_k is the weight of path k from S_s to O^{Sys} . If one could assume independence over all paths, the impact measure would be the conditional probability of an error in S_s propagating all the way to O^{Sys} . However, as independence can rarely be assumed we will treat this as a relative measure by which different signals can be ranked. The general interpretation of this measure is that the higher the impact, the higher the risk of an error in the source signal generating an error in the output of the system. Thus, when placing EDM's and ERM's one may consider placing such mechanisms at signals which have a high impact even though they may have a low error exposure (meaning that errors in this signal are relatively rare but, should they occur, are likely to be costly).

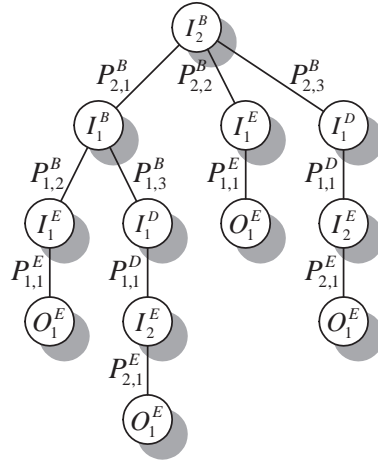


Figure 6.6: Impact tree for intermediate signal I_2^B .

In Eq. 6.7, the measure only considers one system output signal. If a system has multiple output signals, the corresponding impact value which considers all output signals can be defined as:

$$0 \leq S_s \rightsquigarrow O^{Sys} = 1 - \prod_i (1 - (S_s \rightsquigarrow O_i^{Sys})) \leq 1 \quad (6.8)$$

where $S_s \rightsquigarrow O_i^{Sys}$ is the impact of signal S_s on system output signal O_i^{Sys} , i.e., the i^{th} system output signal.

To further illustrate the concept of impact, again consider the example shown in Fig. 6.2. Suppose that we would like to calculate the impact of errors in signal I_2^B on system output O_1^E . First, we will generate an impact tree as shown in in Fig. 6.6.

The impact tree shown in Fig. 6.6 is actually the left sub tree of the trace tree for system input signal I_1^A shown in Fig. 6.5. In order to calculate the impact of errors in I_2^B on system output O_1^E we generate all the propagation paths from the root to the leaves. In this case, with only one system output, all leaves are considered. This gives us four paths as shown in Table 6.1.

Using the weights of the paths we can now calculate $I_2^B \rightsquigarrow O_1^E$, i.e., the impact of (errors in) I_2^B on O_1^E , as

$$\begin{aligned} I_2^B \rightsquigarrow O_1^E &= 1 - \prod_{i=1}^4 (1 - w_i) = \\ &= 1 - (1 - w_1)(1 - w_2)(1 - w_3)(1 - w_4) \end{aligned}$$

where w_i are the weights listed in Table 6.1.

Table 6.1: The four paths generated from the impact tree of signals I_2^B

Path/Product
$P_{2,1}^B P_{1,2}^B P_{1,1}^E = w_1$
$P_{2,1}^B P_{1,3}^B P_{1,1}^D P_{2,1}^E = w_2$
$P_{2,2}^B P_{1,1}^E = w_3$
$P_{2,3}^B P_{1,1}^D P_{2,1}^E = w_4$

The concept of impact as described above considers the impact on system output generated by errors in system input signals and intermediate signals. However, when a system has multiple output signals, these are not necessarily all equally important for the operation of the system, i.e., some output signals may be more critical than others. For cost-efficiency, one may wish to concentrate resources for dependability on the most critical system output signals and therefore needs to know which signals in the system that are “best” (in a loose sense) to equip with EDM’s/ERM’s.

Each system output signal O_i^{Sys} is assigned a criticality $C_{O_i^{Sys}}$, which is a value between 0 and 1, where 0 denotes *not at all critical* and 1 denotes *highest possible criticality*. These criticality values are assigned by the system designer, for example from the specifications of the system or from results from experimental vulnerability analyses.

The criticality of system input signals and intermediate signals is calculated using the assigned criticality values of the system output signals and the various impact values calculated for the various signals. Each signal S_s has a certain impact, $S_s \rightsquigarrow O_i^{Sys}$, on system output O_i^{Sys} , as calculated according to Eq. 6.7. The criticality of S_s as experienced by system output O_i^{Sys} , $C_{s,i}$, is calculated as

$$0 \leq C_{s,i} = C_{O_i^{Sys}} \cdot (S_s \rightsquigarrow O_i^{Sys}) \leq 1 \quad (6.9)$$

Once we have the criticality of S_s with regard to each system output signal O_i^{Sys} we can subsequently compute a total criticality value. We define the *criticality* C_s of signal S_s as

$$\begin{aligned}
0 \leq C_s &= 1 - \prod_i (1 - C_{s,i}) = \\
&= 1 - \prod_i (1 - C_{O_i^{Sys}} \cdot (S_s \rightsquigarrow O_i^{Sys})) \leq 1 \quad (6.10)
\end{aligned}$$

For each signal, the criticality measure indicates how “expensive” errors are with

regard to the total system operation, i.e., the higher the criticality value, the higher the likelihood of the system not being able to deliver its intended service, should an error occur in the signal. The notion of criticality as defined here also takes into account the “cost” associated with errors in system outputs as defined by the system designer. Thus, while the impact measures are independent of the project policies regarding dependability, the criticality values may change when the project policies for software development change.

Note that if the system only has one output signal, then the obtained criticality will only function as a constant scaling factor of the impact values, i.e., the relative order among the signals of the system will not change. Thus, calculating criticality values is essential only when there are multiple output signals in a system and these are of different “importance”.

At this point, we have only defined *impact* and *criticality* at the signal level. Going up in abstraction levels, we can now define equivalent measures which are based on the signal level measures, but consider entire modules instead. If we consider a module \mathbf{M} in a system with i output signals, we can define the impact of \mathbf{M} on a given system output signal O_i^{Sys} , $M \rightsquigarrow O_i^{Sys}$, as follows:

$$0 \leq M \rightsquigarrow O_i^{Sys} = 1 - \prod_j (1 - (O_j^M \rightsquigarrow O_i^{Sys})) \leq 1 \quad (6.11)$$

where $O_j^M \rightsquigarrow O_i^{Sys}$ is the impact of (errors in) the output signal O_j^M of \mathbf{M} on system output signal O_i^{Sys} . For each output signal of \mathbf{M} , there is one such impact value. In order to get a measure for the impact of \mathbf{M} on the system output as a whole we can define $M \rightsquigarrow O^{Sys}$, the *module impact* of \mathbf{M} on system output, as follows:

$$\begin{aligned} M \rightsquigarrow O^{Sys} &= 1 - \prod_i (1 - (M \rightsquigarrow O_i^{Sys})) = \\ &= 1 - \prod_i (1 - (1 - \prod_j (1 - (O_j^M \rightsquigarrow O_i^{Sys})))) = \\ &= 1 - \prod_i \prod_j (1 - (O_j^M \rightsquigarrow O_i^{Sys})) \leq 1 \end{aligned} \quad (6.12)$$

Going from impact to criticality is not a big step. Instead of using the individual impact values of the outputs of a module, the corresponding criticality values for the chosen system output signal are used. The criticality of module \mathbf{M} , with regard to system output O_i^{Sys} can thus be defined as:

$$0 \leq C_i^M = 1 - \prod_j (1 - C_{O_j^M, i}) \leq 1 \quad (6.13)$$

where $C_{O_j^M, i}$ is the criticality of output O_j^M with regard to system output signal O_i^{Sys} . A total measure regarding all system output signals is then referred to as the *module criticality*, C^M , of \mathbf{M} and is defined as:

$$\begin{aligned} 0 \leq C^M &= 1 - \prod_j (1 - C_{O_j^M}) = \\ &= 1 - \prod_j (1 - (1 - \prod_i (1 - C_{O_j^M, i}))) = \\ &= 1 - \prod_j (1 - (1 - \prod_i (1 - C_{O_i^{Sys}} \cdot (O_j^M \rightsquigarrow S_{o,i})))) = \\ &= 1 - \prod_i \prod_j (1 - C_{O_i^{Sys}} \cdot (O_j^M \rightsquigarrow O_i^{Sys})) \leq 1 \end{aligned} \quad (6.14)$$

Note that the difference between impact and criticality (see Eq. 6.8 and Eq. 6.10 for signals, and Eq. 6.12 and Eq. 6.14 for modules) is the criticality factor $C_{O_i^{Sys}}$ of system output signal O_i^{Sys} which is defined by the system designer. Thus, criticality is a biased version of the impact.

We have now defined a number of measures for analyzing the propagation of errors and the effect of errors on system output. In the following section we discuss how the various measures obtained in the error effect analysis together with the values obtained in the error propagation analysis can be used to identify candidate locations for EDM's and ERM's.

6.3.5 Identifying Candidate Locations for ERM's and EDM's

The EPIC framework introduced in Section 6.3 contains a number of various measures for analyzing the propagation and effect of errors in software, namely:

- error permeability for input/output pairs ($P_{i,k}^M$, Eq. 6.1),
- relative error permeability for modules (weighted, P^M , and non-weighted, \hat{P}^M , Eqs. 6.2 and 6.3, respectively),
- error exposure for modules (weighted, X^M , and non-weighted, \hat{X}^M , Eqs. 6.4 and 6.5, respectively),

- error exposure for signals (X_s^S , Eq. 6.6),
- impact for signals ($S_s \rightsquigarrow O^{Sys}$, Eqs. 6.7 and 6.8),
- criticality for signals (C_s , Eq. 6.10),
- impact for modules ($M \rightsquigarrow O^{Sys}$, Eq. 6.12), and finally,
- criticality for modules (C^M , Eq. 6.14).

In this section we will discuss how candidate locations for EDM's and ERM's may be identified based on the results from the propagation analysis and the effect analysis. It is hard to develop a generalized heuristic for identifying the locations. However, the following rules of thumb or recommendations can be made:

- The higher the error exposure values of a module, the higher the probability that it will be subjected to errors propagating through the system if errors are indeed present. Thus, it may be more cost effective to place EDM's in those modules than in those with lower error exposure. An analogous way of reasoning is valid also for the signal error exposure.
- The higher the error permeability values of a module, the lower its ability to contain errors. Thus, there is an increase in the probability of subsequent modules being subjected to propagating errors if errors should pass through the module. Therefore, it may be more cost effective to place ERM's in those modules than in those with lower error permeability.
- The higher the criticality (or impact if the system only has one output signal) of a signal, the higher the probability of an error in that signal causing damage from a system point-of-view. Thus, it may be more cost effective to equip those signals with EDM's and ERM's which have the highest criticality (impact). An analogous way of reasoning is valid also for the module criticality (impact).

When selecting locations, these rules may not individually yield the same result. Consider the case where a signal has a *low exposure* but a *high criticality*. The low exposure means that there is a low probability of errors propagating to that signal. However, the high criticality means that, should an error find its way into that signal, there is a high probability of that error causing damage which propagates beyond the system barrier into the environment. Thus, one may select signals with low exposure and high criticality as candidate locations for EDM's and ERM's.

One way of having a more manageable approach in a project may be to setup certain conditions which must be met by the software. For example, one may wish to set a minimum level of error containment for all modules, which can be accomplished by setting a maximum level on error permeability values and error exposure values. Thus, if a module exceeds this limit, this indicates that more resources have to be allocated to that module to increase its error containment capabilities. The same argument can be used for error exposure. If a module or signal is highly exposed, this indicates that more resources are required either to protect the exposed modules or signal, or to increase the error containment capabilities of the module or signal responsible for the high degree of exposure.

From a criticality (impact) point-of-view, a project may also set up criticality threshold limits. For example, one may wish to set a maximum level of impact for the various signals. Signals exceeding this threshold limit indicate that the error containment from that signal out to the system output signals is not high enough. As the criticality values of signals are based on the criticality values assigned to system output signals, these can only be indirectly adjusted via the impact values.

The results from the analysis may also aid in the design of EDM's. For example, a situation with low error exposure and high criticality (impact) indicates that any EDM in that location would have to be highly specialized as errors are infrequent and likely to be hard to detect. The opposite situation, i.e., high exposure and low criticality (impact) indicates that a coarser EDM in that location may suffice.

Next, we describe how to obtain experimental estimates of the measures and use of our framework on actual software of an embedded control system.

6.4 Obtaining Numerical Estimates of Error Permeability

Obtaining numerical values for the error permeability may prove to be quite difficult, given that many factors, such as error occurrence probabilities, operational profiles, etc., have to be taken into account. This may render it impossible to get the "real" value of the error permeability values of a software system. Thus, a method of estimating these values is needed. In this section we describe an experimental method based on fault/error injection for obtaining estimates of error permeability values. However, other approaches, such as data flow analysis and other static compiler-assisted approaches, might be investigated in the future.

Our method for experimentally estimating the error permeability values of software modules is based on fault injection (FI). FI artificially introduces faults and/or errors into a system and has been used for evaluation and assessment of dependability for several years (see, e.g., [Chillarege and Bowen, 1989], [Arlat *et al.*, 1990],

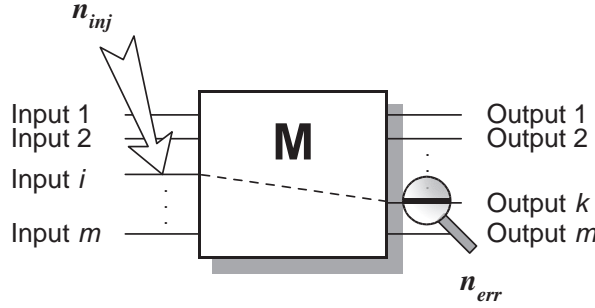


Figure 6.7: A software module where errors are injected into input i and output k is observed to detect permeated errors.

and [Fabre *et al.*, 1999]). A comprehensive survey of experimental analysis of dependability appears in [Iyer and Tang, 1996].

For analysis of raw experimental data, we make use of so-called Golden Run Comparisons (GRC). A Golden Run (GR) is a trace of the system executing without any injections being made, hence, this trace is used as reference and is stated to be “correct”. All traces obtained from the injection runs (IR’s, where injections are conducted), are compared to the GR, and any difference indicates that an error has occurred. The main advantage of this approach is that it does not require any *a priori* knowledge of how the various signals are supposed to behave, which makes this approach less application specific.

Experimentally estimating values for error permeability of a module is done by injecting errors in the input signals of the module and logging its output signals. We only inject one error in one input signal at a time. Consider the module illustrated in Fig 6.7. Suppose, for module **M**, we inject n_{inj} distinct errors in input i , and at output k observe n_{err} differences compared to the GR’s, then we can directly estimate the error permeability $P_{i,k}^M$ to be $\frac{n_{err}}{n_{inj}}$ (see more on experimental estimation in [Cukier *et al.*, 1999] and [Powell *et al.*, 1995]).

Since the propagation of errors may differ based on the system workload, it is generally preferred to have realistic input distributions than randomly generated inputs. This generates permeability estimates that are closer to the “real” values than randomly chosen inputs would.

The type of injected errors can also affect the estimates. Ideally, one would inject errors from a realistic set, with a realistic distribution. However, as the measures in our framework are mainly used as relative measures, the relevance of the realism provided by the error model is decreased, assuming that the relative order of the modules and signals when analyzing permeability is maintained.

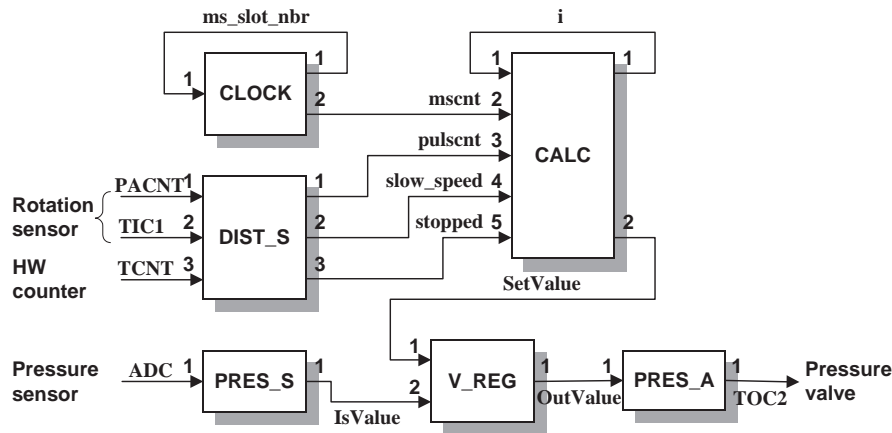


Figure 6.8: Software structure of the example target system (an aircraft arrestment system). For details, see Chapter 3.

6.5 Experimental Analysis: An Example System

For an actual application of our proposed methodology on an embedded control system, we have conducted an example study. This study illustrates the results obtained using the EPIC framework and experimental estimates for error permeability values. First we will shortly describe the target system used in the example, then we will do the pre-experimental analysis of the software to get the permeability graph, trace trees and backtrack trees needed for the subsequent analysis. After that, the experimental estimates of the measures of EPIC are produced.

6.5.1 Target Software System

The target system is an embedded control system used for arresting aircraft on short runways and aircraft carriers and is described in detail in Chapter 3. To aid the reader, the software structure shortly described here, as well.

The structure of the software is illustrated in Fig. 6.8. The numbers shown at the inputs and outputs are used for numbering the signals. For instance, *PACNT* is input #1 of *DIST_S*, and *SetValue* is output #2 of *CALC*. The software is composed of six modules of varying size and input/output signal count. The system receives input from a number of sensors at *PRES_S* and *DIST_S*. The output of the system is provided at *PRES_A*. The remaining modules (*CALC*, *V_REG* and *CLOCK*) provide internal/intermediate signals. The module specifics are provided in Chapter 3.

The system specifications [USAF, 1986] set a number of physical constraints within which the system must operate. These constraints are described in Chapter 3.

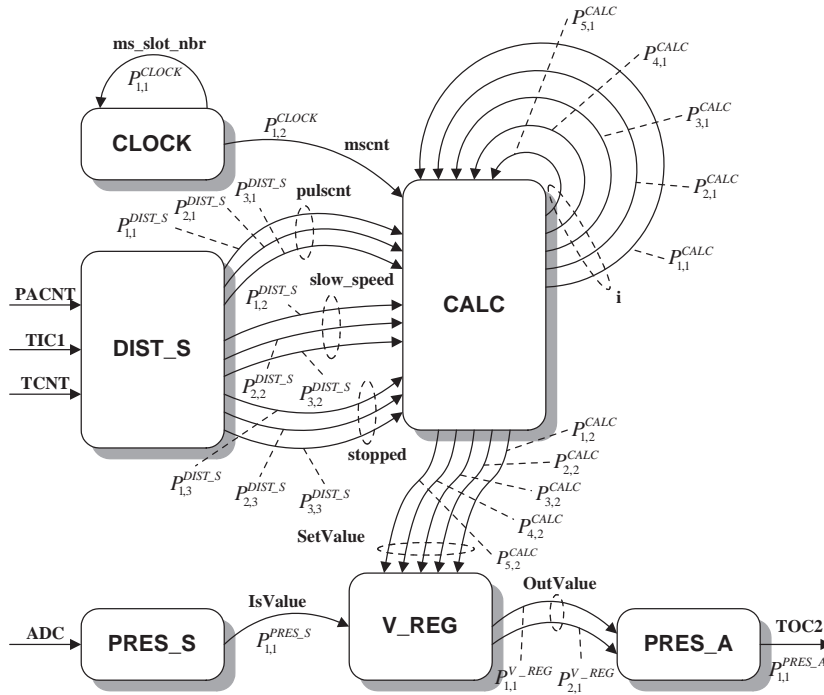


Figure 6.9: Permeability graph of the target system shown in Fig. 6.8

In the experiments described in this chapter, this failure classification has been used when obtaining coverage estimates of error detection mechanisms.

Now we have presented the target system used in our comparisons. The next section will briefly describe the propagation analysis framework with associated measures used for systematic placement of EA's and subsequent sections contain the comparison itself.

6.5.2 System Analysis

Prior to running the experiments, the permeability graph and the backtrack trees and trace trees for the target system were generate as per the process described in Sections 6.3. The permeability graph is shown in Fig. 6.9.

In the graph (Fig. 6.9) we can see the various permeability values (labels on the arcs) that will have to be calculated. The numbers used in the notation refer to the numbers of the input signals and output signals respectively, as shown in Fig. 6.8. For instance, $P_{2,1}^{CALC}$ is the error permeability from input 2 (*mscnt*) to output 1 (*i*) of module CALC. From the permeability graph in Fig. 6.9 we can now generate the backtrack tree for the system output signal *TOC2*, using the steps described in Section 6.3.2. This tree is shown in Fig. 6.10.

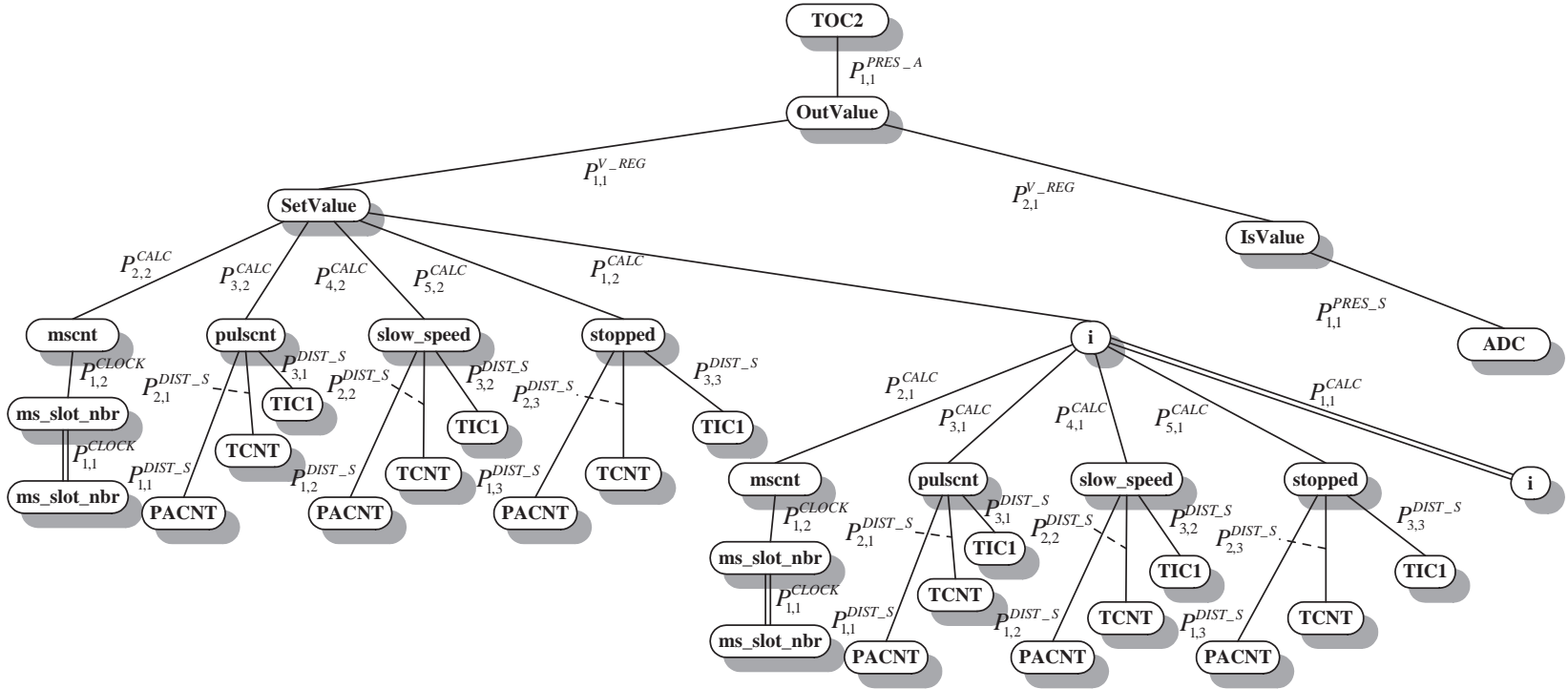
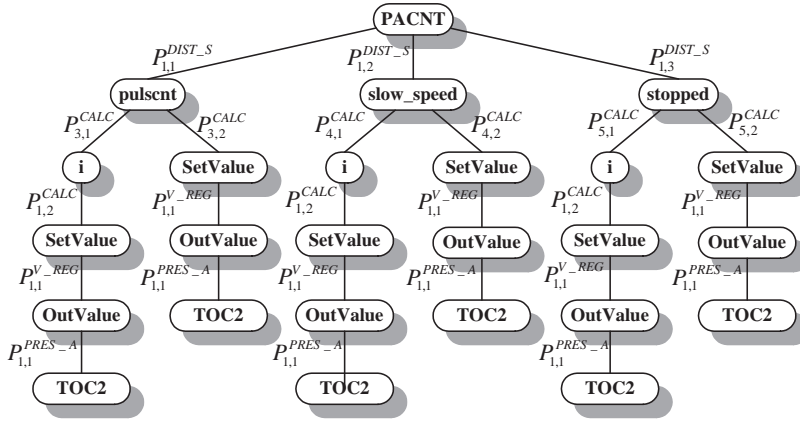
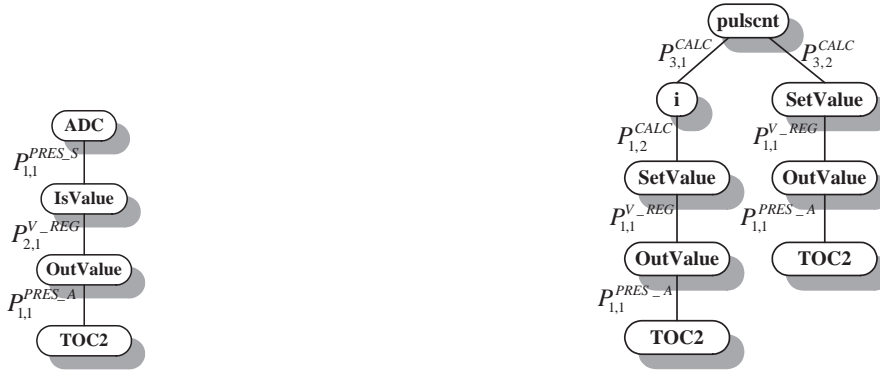
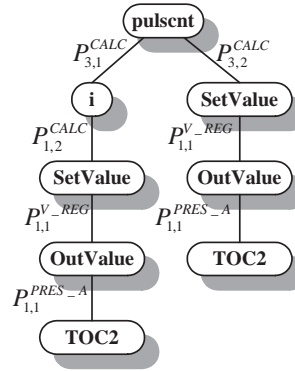


Figure 6.10: Backtrack tree for system output signal $TOC2$

Figure 6.11: Trace tree for system input signal *PACNT*Figure 6.12: Trace tree for system input signal *ADC*Figure 6.13: Impact tree for intermediate signal *pulscnt*.

As illustrated with a double line in the backtrack tree (Fig. 6.10), we have a special relation between the leaves for *ms_slot_nbr* and for *i* and their respective parent. This is because the parent node is also either *ms_slot_nbr* or *i*. Thus, we have an output signal which is connected back to the originating module giving us a recursive relation. In those cases where errors only can enter a system via its main inputs, these branches of the backtrack-trees can be disregarded.

In Figs. 6.11 and 6.12, we have the trace trees for system input signal *PACNT* and system input signal *ADC*, respectively, as obtained by the processes defined in Section 6.3.2. The trees for inputs *TIC1* and *TCNT* are very similar to the tree for *PACNT* so they will not be shown here.

As described in Section 6.3.2, we do not follow the recursion generated by a feedback from a module to itself. In module *CALC* we have a feedback in signal *i*, and as can be seen in Fig. 6.11, we do not have a child node from *i* that is *i* itself.

In order to calculate impact values for the various signals, we generated their respective impact trees. Depicted in Fig. 6.13, we have the impact tree of the signal *pulscnt* (other impact trees have been left out, but are easily generated by the interested reader). The impact tree shown in Fig. 6.13 is actually the left sub tree of the trace tree for system input signal *PACNT* shown in Fig. 6.11. In order to calculate the impact of errors in *pulscnt* on system output *TOC2* we generate all the propagation paths from the root to the leaves. In this case, where we have only one system output, all leaves are considered.

At this point we have generated all trees and graphs required for our analysis of the software. The next step is to estimate numerical values for the individual permeability values such that we can use the analysis results to identify the modules and signals in the system which should be equipped with EDM's and ERM's.

6.5.3 Experimental Setup

For estimating error permeability values, we used the Propagation Analysis Environment (PROPANE, see Chapter 5). This tool enables fault and error injection, using SWIFI (SoftWare Implemented Fault Injection), in software running on a desktop (currently for Windows 2000/XP). The tool is also capable of creating traces of individual variables and different pre-defined events during the execution. Each trace of a variable from an injection experiment is compared to the corresponding trace in the Golden Run. Any discrepancy is recorded as an error.

For logging and injection, the target system was instrumented with high-level software traps. As a trap is reached during execution, an error is injected and/or data logged. The traces obtained during execution have millisecond resolution for every logged variable. Also, we ported the software to run on a desktop system, so the intrusion of the traps is non-existent in our setup as it runs in simulated time.

In this study, the aim was to produce an estimate of the *error permeability* of the modules of the target system. As described in Section 6.4 we produced a Golden Run (GR) for each test case. Then, we injected errors in the input signals of the modules and monitored the produced output signals. For each injection run (IR) only one error was injected at one time, i.e., no multiple errors were injected.

The input signals are all 16 bits wide, except *PACNT* which is 8 bits wide. We injected bit-flips in each bit position at 10 different time instances distributed in half-second intervals between 0.5 seconds and 5.0 seconds from start of arrestment (although only at one time in each IR). In order to get a realistic load on the system and the modules, we subjected the system to 25 test cases: 5 masses and 5 velocities of the incoming aircraft uniformly distributed between 8000–20000 *kg*,

Table 6.2: Estimated error permeability values of the input/output pairs

Input → Output	Name	Value
ms_slot_nbr → ms_slot_nbr	$P_{1,1}^{CLOCK}$	1.000
ms_slot_nbr → mscnt	$P_{1,2}^{CLOCK}$	0.000
PACNT → pulscnt	$P_{1,1}^{DIST-S}$	0.957
TIC1 → pulscnt	$P_{2,1}^{DIST-S}$	0.000
TCNT → pulscnt	$P_{3,1}^{DIST-S}$	0.000
PACNT → slow_speed	$P_{1,2}^{DIST-S}$	0.010
TIC1 → slow_speed	$P_{2,2}^{DIST-S}$	0.000
TCNT → slow_speed	$P_{3,2}^{DIST-S}$	0.000
PACNT → stopped	$P_{1,3}^{DIST-S}$	0.000
TIC1 → stopped	$P_{2,3}^{DIST-S}$	0.000
TCNT → stopped	$P_{3,3}^{DIST-S}$	0.000
ADC → IsValue	$P_{1,1}^{PRES-S}$	0.000
i → i	$P_{1,1}^{CALC}$	1.000
mscnt → i	$P_{2,1}^{CALC}$	0.000
pulscnt → i	$P_{3,1}^{CALC}$	0.494
slow_speed → i	$P_{4,1}^{CALC}$	0.000
stopped → i	$P_{5,1}^{CALC}$	0.013
i → SetValue	$P_{1,2}^{CALC}$	0.056
mscnt → SetValue	$P_{2,2}^{CALC}$	0.530
pulscnt → SetValue	$P_{3,2}^{CALC}$	0.000
slow_speed → SetValue	$P_{4,2}^{CALC}$	0.892
stopped → SetValue	$P_{5,2}^{CALC}$	0.000
SetValue → OutValue	$P_{1,1}^{V-REG}$	0.885
IsValue → OutValue	$P_{2,1}^{V-REG}$	0.896
OutValue → TOC2	$P_{1,1}^{PRES-A}$	0.875

and between 40–80 m/s , respectively. Thus, for each input signal, we conducted $16 \cdot 10 \cdot 25 = 4000$ injections (2000 for *PACNT*).

The raw data obtained in the IR's was used in a Golden Run Comparison where the trace of each signal (input and output) was compared to its corresponding GR trace. The comparison stopped as soon as the first difference between the GR trace and the IR trace was encountered. In our experimental setup—real software running in simulated time, in a simulated environment, and on simulated hardware—this is a valid way of comparing traces even for continuous signals where fluctuations between similar runs in a real environment may be normal.

When calculating the individual error permeability values, we only took into account the direct errors on the outputs. We did not count errors originating from errors that propagated via one of the other outputs and then came back to the original input producing an error in the first output.

Table 6.3: Estimated relative permeability, error exposure and impact values of the modules

Module	P^M	\hat{P}^M	X^M	\hat{X}^M	$M \rightsquigarrow TOC2$
CLOCK	0.500	1.000	1.000	1.000	0.410
DIST_S	0.107	0.966	-	-	0.698
PRES_S	0.000	0.000	-	-	0.784
CALC	0.299	2.986	0.165	2.473	0.784
V_REG	0.890	1.781	0.247	1.479	0.875
PRES_A	0.875	0.875	0.890	1.781	-

6.5.4 Experimental Results and Obtained Profiles

In the target system, we have 25 input/output pairs for which we produced an estimate of the error permeability measure (see Eq. 6.1) using the method from Section 6.4. These estimated values are shown in Table 6.2, and they form the basis for subsequent results, which are calculated as described in Section 6.3.

In Table 6.3, we obtain weighted and non-weighted relative permeability values (P^M and \hat{P}^M , respectively), weighted and non-weighted error exposure values (X^M and \hat{X}^M) and module impact values ($M \rightsquigarrow TOC2$) for each module.

The modules DIST_S and PRES_S have no error exposure values as they only receive system input signals, i.e., from external sources. This does not mean that these modules will never be exposed to errors on their inputs, but rather that the error exposure is dependent on the probability of errors occurring in the various external data sources. The modules with the highest non-weighted error exposure are the CALC module and the V_REG module. This indicates that these two modules are central in the system and that they are good candidates for error detection and recovery mechanisms.

The module PRES_A has no impact value since the impact is calculated with regard to its output. One could perhaps say that this module has an impact of 1.0, as an error in its output signal (*TOC2*) is guaranteed to generate an error in the system output signal (also *TOC2*). When calculating module impact, one may also view the environment as a module and calculate its impact on system output. In this case, the system input signals are viewed as the outputs of the environment and calculations are performed as described in Eq. 6.12. The system only has one output signal. Thus, no criticality values are calculated as these would only be scaled impact values.

From the backtrack tree in Fig. 6.10, we can generate 22 propagation paths from the system output signal to an input signal. Each of these paths has a total weight, which is the product of the permeability values of the arcs in the path. Ordering the

Table 6.4: The three non-zero propagation paths and their weights

Path/Product	Weight
$P_{1,1}^{CALC} P_{1,2}^{CALC} P_{1,1}^{V_REG} P_{1,1}^{PRES_A}$	0.04337
$P_{1,1}^{DIST_S} P_{3,1}^{CALC} P_{1,2}^{CALC} P_{1,1}^{V_REG} P_{1,1}^{PRES_A}$	0.02050
$P_{1,2}^{DIST_S} P_{4,2}^{CALC} P_{1,1}^{V_REG} P_{1,1}^{PRES_A}$	0.00691

Table 6.5: Estimated signal error exposures and impacts on *TOC2*

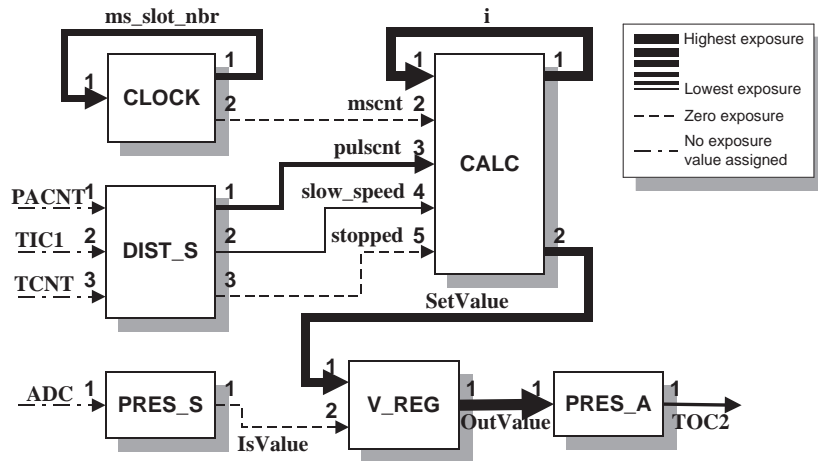
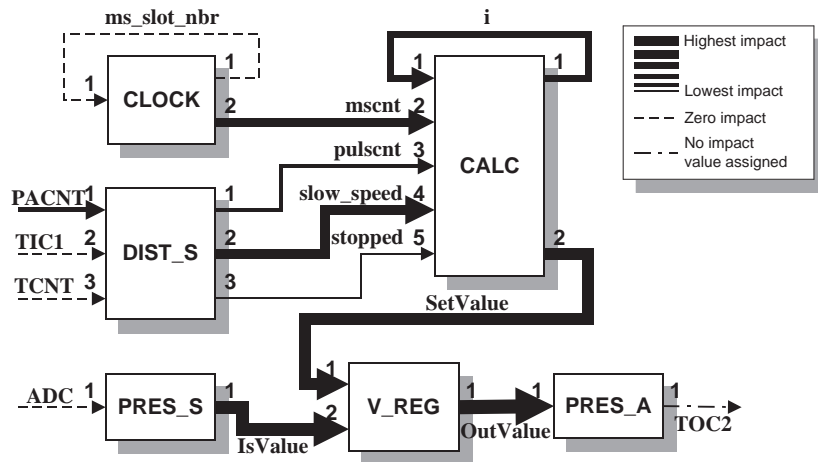
Signal (s)	X_s^S	$s \rightsquigarrow TOC2$
PACNT	-	0.027
TCNT	-	0.000
TIC1	-	0.000
ADC	-	0.000
OutValue	1.781	0.875
i	1.507	0.043
SetValue	1.478	0.774
ms_slot_nbr	1.000	0.000
pulsent	0.957	0.021
TOC2	0.875	-
slow_speed	0.010	0.691
IsValue	0.000	0.784
mscnt	0.000	0.410
stopped	0.000	0.001

paths according to their total weight gives us some knowledge of the more probable paths for error propagation. Table 6.4 depicts the three paths that acquired weights greater than zero (the paths along which errors might propagate).

In Table 6.5, we have both exposure values, X_s^S , and impact values, $s \rightsquigarrow TOC2$, of the various signals of the target system. Signal *TOC2* has no impact value associated with it as this is the system output signal (one could say that the impact is 1.0 in this case). Having this granularity (i.e., signal level information) will help us in deciding which signals we should equip with EDM's or ERM's.

The information in Table 6.5 is depicted graphically in Figs. 6.14 and 6.15. Here we can clearly see the difference between the two profiles of the system. The thickness of a line now depicts the value of the respective measure—the thicker the line, the higher the value. A dashed line indicates a zero value and a dashed-dotted line indicates that no value is assigned to that signal (either because the signal is a system input or output signal).

In Figs. 6.14 and 6.15, an example of how the rules-of-thumb for selection of locations for EDM's and ERM's can yield different results is highlighted. Consider

Figure 6.14: Propagation analysis: *Exposure* profileFigure 6.15: Effect analysis: *Impact* profile

the signal *IsValue* going from PRES_S to V_REG. With the propagation analysis, we obtained a zero error exposure value (see Fig. 6.14) indicating that errors never (or at least rarely) propagate into this signal. This suggests that *IsValue* may not be the best selection for an EDM or ERM. On the other hand, with the effect analysis, we obtained a very high error impact value. This means that an error in *IsValue* could have a high impact should it occur and may cause severe system failure, which would suggest that *IsValue* may be a very good location for an EDM or ERM. Thus, the propagation analysis and the effect analysis may yield different sets of locations for EDM's and ERM's and corresponding input to system designers regarding cost/benefit trade-offs and implications of EDM/ERM placement and design.

Table 6.6: EA-locations based on propagation profile (the P-set)

Signal	Producer	Consumer	Test location
SetValue	CALC	V_REG	V_REG
i	CALC	CALC	CALC
pulsent	DIST_S	CALC	DIST_S
OutValue	V_REG	PRES_A	PRES_A

6.6 Selecting Locations for EDM's

In this section we will select locations for EDM's and evaluate the coverage obtained. We will start by assuming that errors are only introduced to the system via its main input signals (i.e., by faulty sensor readings etc.) and then adopt a more severe error model where errors are introduced in random memory locations and signals throughout the system. This will illustrate how propagation analysis and effect analysis complement each other.

The mechanisms we have chosen to use for this study are so called Executable Assertions (EA's) and are commonly used in embedded software (see, for example, [Saib, 1978], [Mahmood *et al.*, 1984], and [Rabéjac *et al.*, 1996]). EA's are usually small snippets of code which are executed on-line to check that certain constraints on the values of variables are not violated, such as minimum and maximum values and rate change limitations. The specific EA's used in this chapter are generic parameterized mechanisms aimed at individual signals and are described in Chapter 4.

6.6.1 Propagation-Based Selection of Locations

Here we assume that errors are introduced only through the system input signals and that they are transients, i.e., an error originally appears only for a very short time (one calculation round of the selected target system). Thus, we only take into account those signals which are in the way of propagating errors from the input signals.

As the EA's we have chosen for our system are aimed at individual signals, we take a closer look at individual permeability values and the signal error exposure values (see Tables 6.2 and 6.5, respectively) in order to select the signals to equip with EA's.

The experimentally ascertained exposure of *IsValue* is zero, meaning that errors in *ADC* are unlikely to propagate through *PRES_S*. Thus, although the permeability of errors from *IsValue* to *OutValue* is quite high (0.896), we do not select *IsValue* as a location for an EA.

Table 6.7: EA-locations based on both propagation and effect analysis (the P&E-set)

Signal	Producer	Consumer	Test location
SetValue	CALC	V_REG	V_REG
IsValue	PRES_S	V_REG	V_REG
i	CALC	CALC	CALC
pulscnt	DIST_S	CALC	DIST_S
ms_slot_nbr	CLOCK	CLOCK	CLOCK
mscnt	CLOCK	CALC	CLOCK
OutValue	V_REG	PRES_A	PRES_A

We do not select *ms_slot_nbr* as errors in this signal do not propagate into *mscnt*. We do not select *TOC2* either, as this is a hardware register and any errors here would most probably come from the *OutValue* signal. We do not select *mscnt* as this signal has a zero error exposure. We do not select *slow_speed* as this signal has a low error exposure, and the mechanisms we have chosen are not particularly geared at detecting errors in boolean values.

Based on the results obtained here, we select the following signals as locations for EA's: *SetValue*, *i*, *OutValue*, and *pulscnt*. The first three are selected based on their high signal error exposure values and the last one as this is the signal which is most likely to be affected by errors in system input. The selection of EA's is summarized in Table 6.6 and we will refer to this set of locations as the P-set.

In the *Signal*-column are the names of the signals we have selected to equip with EA's. The *Producer*- and *Consumer*-columns contain the names of the source and sink module, respectively. The *Test*-column contains the name of the module where the EA was physically placed (this was selected based on perceived implementation simplicity).

6.6.2 Adding the Effect Profile to the Selection Process

So far, we have only used the profile generated by propagation analysis to determine locations for EA's. This may be sufficient as long one assumes that errors will only enter the system via system input signals. In this section we will, in addition to the profile provided by the propagation analysis, also make use of the profile provided by the error effect analysis of the software system. We will also adopt a more "severe" error model where errors are introduced not only via system input signals but also in intermediate signals and/or internal variables and memory structures.

Thus, we will now consider not only where errors tend to propagate but also what effect errors have (regardless of whether these errors are likely to occur or

Table 6.8: EA-setup and sum of ROM/RAM requirements

Signal	EA	P-set	P&E-set	ROM (bytes)	RAM (bytes)
SetValue	EA1	✓	✓	50	14
IsValue	EA2	-	✓	50	14
i	EA3	✓	✓	25	13
pulscnt	EA4	✓	✓	25	13
ms_slot_nbr	EA5	-	✓	37	13
mscnt	EA6	-	✓	25	13
OutValue	EA7	✓	✓	50	14
Total ROM/RAM (bytes)		150/54	262/94		

not). Previously we had ascertained that signals *SetValue*, *i*, *pulscnt* and *OutValue* were to be guarded by EA's because of their high exposure to propagating errors. If we now take into account the impact of the signals on system output, we see that signals *IsValue*, *mscnt* and *slow_speed* may be considered for being guarded by EA's as well as these have very high impact values (see Table 6.5). The mechanisms we have chosen are implemented in such a way that it is difficult to detect errors in a boolean value, thus setting an EA on the signal *slow_speed* is not efficient in this case. Therefore, when taking into account also the impact values of the signals we can decide to place EA's on *IsValue* and *mscnt* as well. Also, as the permeability-value of *ms_slot_nbr* is 1, and the assumed error model now introduces errors in the entire memory space of the system (as opposed to only system input signals as was the case before) we also select that signal. The new selection is summarized in Table 6.7 and we will refer to this set of locations as the P&E-set.

In Table 6.7, for each signal, the *Producer* column indicates the module where the signal originates and the *Consumer* column where the signal is used. The *Test location* column indicates which of these two was chosen as the module where the corresponding EA was placed.

6.7 Comparing the Two Location Selections

This section will compare the two sets of selected locations with regard to resources required and also with regard to the coverage obtained when the systems is subjected to errors at system input signals and in random locations.

6.7.1 Memory and Execution Time Requirements

For comparison of resource requirements using the -approach and the PA-approach, Table 6.8 presents a summary of the two sets of locations/mechanisms and their respective requirements on memory resources (ROM contains constant parameters defining allowed behavior, and RAM contains run-time data). As expected, the requirements for the P-set, {EA1, EA3, EA4, EA7}, is less than the requirements for the P&E-set, {EA1, EA2, EA3, EA4, EA5, EA6, EA7}, as the former is a subset of the latter (as seen in Table 6.8). Specifically, there is a 40% reduction in memory requirements when for the P-set over the P&E-set.

The overhead in terms of execution time is also reduced. The tool used for obtaining these results does not provide a means for measuring execution time, thus we were not able to quantitatively assess the reduction. However, the EA's are all functions which are executed sequentially, i.e. the software is not executed in a truly parallel manner as only one processor is used. Also, they are invoked with roughly the same period and require roughly the same execution time for each invocation. Thus, the reduction in execution time overhead is likely to be in the order of the reduction in number of EA's, i.e., about 40%.

6.7.2 Error Detection Coverage

In this section we compare the two sets of EA's with regard to error detection coverage, and we do this using two distinct error models: one where errors are introduced at the system input signals only, and one where errors are introduced in random locations in memory.

Errors in System Input Signals

After having added the EA's to the system, we performed a set of injection experiments. In these experiments we used the same tool (PROPANE) and setup as we used for obtaining the estimates of the individual error permeability values (as described in Section 6.5.3), i.e., we injected transient single-bit errors in system input signals.

In Table 6.9 we summarize the results from the injection experiments. The results are shown for each input signal that was targeted during the experiments. The n_{err} column shows how many errors that were active after injection (e.g., we injected a total of 2000 errors in *PACNT*, and of those 1856 were active, i.e., injected before the arrestment of an aircraft was not completed). The various *EAx* columns show the obtained coverage for each individual EA (a dash indicates zero coverage),

Table 6.9: Obtained detection coverage for errors injected in system input - P-based and P&E-based placements

Signal	n_{err}	EA1	EA2	EA3	EA4	EA5	EA6	EA7	Total
Member of P-set		✓	-	✓	✓	-	-	✓	
Member of P&E-set		✓	✓	✓	✓	✓	✓	✓	
PACNT	1856	0.218	0.105	-	0.975	-	-	0.005	0.975
TIC1	3712	-	-	-	-	-	-	-	-
TCNT	3712	-	-	-	-	-	-	-	-
All	9280	0.062	0.040	-	0.195	-	-	< 0.001	0.195

calculated as $\frac{n_{det}}{n_{err}}$. The *Total* column is the combined coverage considering all EA's. Each row contains the data for errors injected into one signal except for *All* which shows the coverage obtained for the various EA's considering all signals. The rows containing tick-marks indicate which EA's were part of the P-set and which were part of the P&E-set (a tick-mark, ✓, indicates membership).

As was indicated by the obtained zero error permeability of the PRES_S module, no errors propagated from input signal *ADC* to intermediate signal *IsValue*. Therefore, no errors could be detected by any of the EA's.

In Table 6.9 we can see that only those errors that were injected into *PACNT* were detected. This is on par with the results obtained in the propagation analysis which showed that errors injected into those signals with a very low probability propagated into any of the signals that were selected to be guarded with an EA. Those errors that propagate are likely to be hard to detect by the selected mechanisms. However, 97.5 percent of the errors injected into *PACNT* were detected. All errors detected by EA1 (*SetValue*), EA2 (*IsValue*) or EA7 (*OutValue*) were also detected by EA4 (*pulscnt*).

It may seem odd that EA2, which guards *IsValue*, has a non-zero coverage for errors in *PACNT*, while no errors injected into *ADC* could propagate into *IsValue*. This, however, is a result of errors in *PACNT* propagating all the way through the system and out beyond the system barrier where they eventually affect the environment to such a degree that *ADC* is affected in a way the PRES_S module cannot fully mask or contain, and the errors are then detected by the EA guarding *IsValue*.

From Table 6.9 we can observe that the coverage obtained with the P&E-set of EA's (EA1 through EA7) is the same as that obtained with the P-set set of EA's (EA1, EA3, EA4, and EA7).

From this we can conclude that if errors can only enter a system via its inputs signals, making a selection of EA locations based on the propagation profile only is sufficient from an error detection point of view. From Table 6.8 we can see that this

Table 6.10: Detection coverage for errors injected periodically in system RAM and stack for both sets of EA's

Measure	RAM		Stack		Total	
	P&E	P	P&E	P	P&E	P
c_{tot}	0.128	0.056	0.042	0.018	0.106	0.046
c_{fail}	0.811	0.418	0.137	0.031	0.394	0.253
$c_{no\ fail}$	0.111	0.038	0.029	0.017	0.092	0.033
P&E = Propagation & Effect analysis, P = Propagation analysis only						

may reduce the resource requirements. The next step in our comparison will investigate the effect of varying error model on the obtained error detection coverage.

Errors in Random Locations in Memory

Now we take both sets of EA's—the one selected using only the profile provided by propagation analysis (Table 6.6) and the one using profiles provided by both the propagation and the effect analysis (Table 6.7)—and compare them using a more severe error model. We still use single bit flips to generate data errors, but now the target will not only be system input signals but also intermediate signals and module state (a total of 150 locations in RAM and 50 locations in the stack) of the system. The errors are injected not only at one point in time but periodically with a period of 20 milliseconds. The same 25 test cases were used giving us a total of $200 \cdot 25 = 5000$ runs with injections. An error is said to be detected if it is detected at least once during the arrestment. These experiments were performed on a real setup of the arrestment system (real hardware, real software, simulated environment—not a simulation run on a desktop computer) using the FIC³-tool (see [Christmansson and Rimén, 1997] and [Christmansson *et al.*, 1998] for details).

The results are summarized in Table 6.10. The *RAM*-column contains the coverage values for errors injected into the RAM areas of the modules, and the *Stack*-column the coverage values for errors injected into the stack area. The *Total*-column contains the coverage for all errors. The measure c_{tot} is the total coverage of the EA-set. The measure c_{fail} is the coverage when considering only those errors that led to system failure (according to the classification of Section 6.5), $c_{no\ fail}$ is for errors that did not lead to system failure. The same data is depicted in Fig 6.16.

In the columns marked with *P&E* in Table 6.10 we can see the coverage values obtained for the EA's selected by utilizing both the propagation profile and the effect profile. The coverage values for the system equipped with the EA's selected using only the propagation profile are shown in the columns marked *P*.

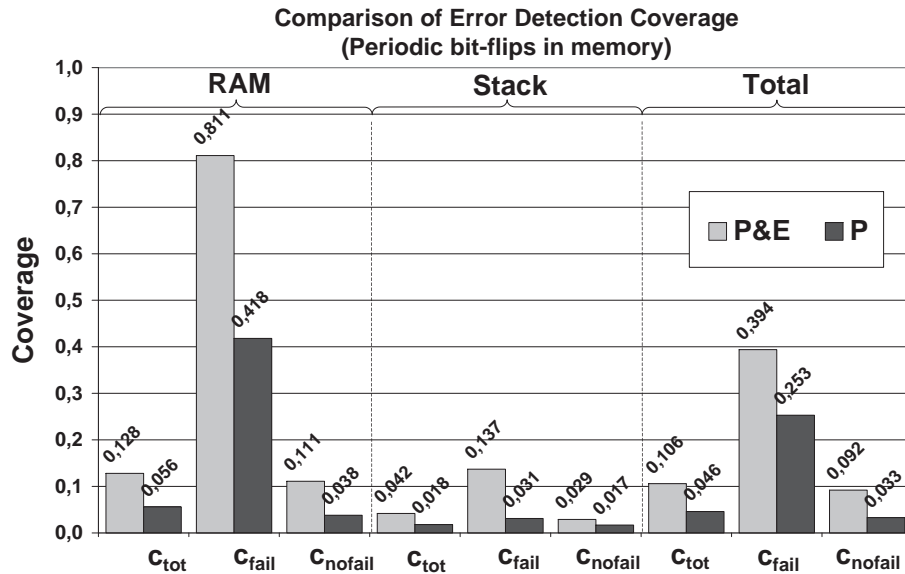


Figure 6.16: Comparison of coverage values

The first observation we make when comparing the results for the two sets is that the coverages for the P-set of EA's is lower than the coverage obtained using the P&E-set of EA's. For errors injected into RAM the coverage is just over half that obtained using the full set of EA's and for errors in stack the decrease is even greater. This indicates that using only the propagation profile may prove a weakness if errors are introduced not only via the inputs of a system, but also via internal variables and structures.

The results illustrate the important distinction between permeability/exposure and impact/criticality, where the former is used for profiling software with regard to its error propagation characteristics and the latter to profile software with regard to the effect errors would have if they were present in different parts of the system. This fact is also highlighted and discussed in conjunction with Figs. 6.14 and 6.15.

From these results we can conclude that if errors are introduced not only via the system input signals, selecting locations for EDM's and ERM's based on the propagation profile alone is insufficient. The effect profile must also be considered. Even though the profiles have been generated with an error model that introduces errors only in the inputs of the modules, the combination of the propagation profiles and the effect profile lessens the impact of this. That is, the profiles are useful even in situations where the system under consideration is subjected to an error model which is different from the one used for profiling.

6.8 Discussion on Framework Limitations and Caveats

In this chapter we have presented an analysis framework for error propagation and effect analysis. We have also used it in an example study illustrating its ability to indicate locations for EDM's and ERM's. However, there are limitations to the framework which we highlight in this section.

One limitations is that, when defining the basic *error permeability* measure, we have only considered direct influence between one input signal and one output signal of a module. It may be (and in most practical cases, probably is) the case that the probability of an error propagating from a given input signal to a given output signal of a module is not independent of errors on other input signals. Thus, the *error permeability* values may differ when multiple input signals are erroneous as compared to when only one input signal is erroneous.

Moreover, the framework currently only considers error propagation and error effect in a “direct” manner, i.e., we only take into account errors that propagate from their original locations to system output directly. This means that we do not consider errors which might at one point propagate out of the system, affect the environment (the controlled entity) and then get fed back into the system via the system input signals. It could be argued that, if the controlled environment has an inherent inertia (i.e., minor perturbations on the output signals of the control system do not affect the environment very much), the probability for this is small. Thus, this may not present a serious problem, however, one should bear this possibility in mind.

Also, at this point we have in our experiments only considered the basic error model *transient bit flips*, i.e., an error was introduced by flipping the value of one bit in one calculation round. Should other error models be used, the results obtained by propagation and effect analysis may need to be re-estimated using the basic approach outlined in EPIC. Even though the usage of two distinct profiles lessens the impact of varying error models, one should aim at using an error model as close to the one assumed for the “real” environment in which the system is designed to operate.

We have not specifically considered cross-linked modules in the system, i.e., situations where the outputs of a module M_1 are used as inputs to another module M_2 and vice versa. Such a setup may prove to have an amplifying or a dampening effect and may have to be modeled separately.

Future work on the EPIC framework will address the limitations mentioned here. The initial focus, however, will be on the limitation of single input errors and error models, i.e., we plan to investigate how the results provided by the EPIC framework is affected when multiple input signals are erroneous simultaneously and when changing the error model to something other than transient bit-flips.

6.9 Summary and Conclusions

In this chapter we have presented the EPIC framework for analysis of the propagation and effect of data errors in software. The system model assumed in this framework is that software is composed of a set of modules which take in data as signals and produce output, also as signals. Specifically, the main contributions described here are:

Software Profiling: The EPIC framework is able to produce distinct profiles of a given modular software system allowing the assessment of the vulnerability of software modules and signals to upcoming data errors. These profiles are i) an error propagation based profile, and ii) an error effect based profile. We introduced the basic measure *error permeability* defined on a input/output signal pair basis from which a set of related measures (both at the signal level and the module level) can be calculated. The framework has four basic measures all related to data errors: i) exposure, ii) permeability, iii) impact, and iv) criticality. The first two measures relate to the analysis of error propagation, whereas the last two measure relate to the analysis of error effect. Thus, the framework is capable of providing a software designer with two distinct profiles regarding the software system at hand, upon which design decisions regarding error detection and recovery can be based. As the framework assumes a black-box view of the software, its applicability is not limited to software developed in-house, i.e., it can also be used for software which is provided in libraries where only interface specifications are provided (e.g. COTS components).

Placement of Error Detection and Recovery Mechanisms: Using the profiles obtained by using the EPIC framework, we have shown how one may interpret the results, using a set of guide-lines, in order to identify and select locations in a software system appropriate for error detection mechanisms (EDM's) and error recovery mechanisms (ERM's). These methods can also pinpoint critical signals and paths in a system. Trade-offs regarding these location selection guide-lines that might have to be considered are also discussed.

Experimental Estimation Method: We have described an experimental method based on fault injection for obtaining estimates for the error permeability values. This method is based on transient bit-flips occurring at single input signals of the various software modules of the software system. Propagation to the output signals of the given module is detected by doing a comparison with "error-free" reference runs (Golden Run Comparisons).

Example Study: We have conducted an experimental assessment on the software of an embedded control system for aircraft arrestment. The results clearly show that using the presented framework generates knowledge on error propagation and error effect giving insights into software vulnerabilities that are very useful when designing dependable systems, especially regarding the selection of locations for EDM's and ERM's. The results also illustrate that by incorporating less efficient detection and recovery mechanisms in locations which have high error exposure instead of very efficient mechanisms which are seldom exposed to errors one would most likely get a better cost/performance ratio. An important conclusion that can be made from the obtained results is that in addition to the software profiles, the assumed error model should also be considered in the selection of locations for EDM's and ERM's.

Discussion on limitations: We have identified and discussed some limitations and caveats of the EPIC framework. This discussion also suggests directions for future work regarding the EPIC framework.

Concluding this chapter, we state that the presented analysis framework, EPIC, provides means for software profiling which may provide knowledge pertinent to dependability engineering in software systems, such that the project resources available can be allocated in a cost-efficient manner.

CHAPTER 7

Summary and Conclusions

By three methods we may learn wisdom: First, by reflection, which is noblest; second, by imitation, which is easiest; and third by experience, which is the bitterest.

— Confucius (around 551–479 B.C.)

This thesis investigates some aspects of techniques and mechanisms for analyzing and designing dependable software for embedded control systems. Specifically, mechanisms for the detection and recovery of data errors in variables/signals of embedded software is investigated, as well as methods for analyzing the propagation and effect of data errors. Furthermore, a tool for analyzing propagation and effect of data errors is described. This chapter briefly summarizes the contributions put forward in this thesis and attempts to draw conclusions from the obtained results.

7.1 Summary of Research Contributions

Here are brief summaries of the results and contribution in this thesis. More detailed accounts can be found in the respective chapters as indicated.

7.1.1 Error Detection and Recovery Mechanisms

In Chapter 4, mechanisms for error detection and recovery are described and evaluated. Error detection is based on the concept of *executable assertions*, i.e., pieces of code checking the validity of a certain variable/signal given a set of constraints. A violation of the specified constraints is defined as an error and when such a violation is detected, the mechanisms apply *forced validity* to the variable/signal, i.e., replacing the erroneous data value with one which is as close to the original value as possible but still within the valid domain of that variable/signal.

In order to lessen the impact of lack-of-experience on behalf of the system designer regarding the use of software implemented mechanisms for data error detection and correction, the presented mechanisms are designed as generalized mechanisms which are instantiated with parameters. The parameters for each signal are predefined according to a certain *signal classification* where each class requires a certain set of parameters. At the top-most level, signals are divided into *continuous* and *discrete* signals. Below that, there are sub-classes which further narrow down the required set of parameters or put constraints on them. The actual values of the parameters are set by the system designer using, e.g., FMECA (Failure Mode, Effect and Criticality Analysis) or other specifications.

The main limitation for the proposed mechanisms is for discrete signals which do not have any restrictions in transition between values, i.e., one can only check that the current value of the signal is actually within the defined domain, and not the transitions between the values. Such signals may not be very common if the valid domain contains more than two values. However, a typical, and common, special case is a boolean signal which can be either *true* or *false*. As a transition in a boolean signal is in most cases triggered by other signals, checking multiple values within the same test or assertion is likely to detect more errors.

7.1.2 Evaluation of Mechanisms

In order to evaluate the effectiveness of the mechanisms for error detection and recovery presented in Chapter 4, two evaluations were conducted. The chosen example target system was an aircraft arresting system, i.e., a system which aids landing

aircraft to stop on short runways. This particular system was implemented as a cable strapped across the runway. A landing aircraft grabs hold of that cable using a hook and the system then pulls the aircraft opposite to the direction of movement in order to slow it down to a complete halt. More information on this target is found in Chapter 3.

In *Evaluation 1*, the combined effects of error detection and error recovery (i.e., error tolerance) was the focus. The most critical signals of the system were identified and equipped with mechanisms. The system was then run with 25 different test cases (combinations of aircraft mass and engaging velocity) while being subjected to error injections in the monitored signals and random locations in memory, stack and registers. The results of this evaluation showed that the failure rate for errors injected into the monitored signals was decreased by 32.56%, while the decrease in failure rate for errors in random memory, stack and registers was only roughly 4.69%.

The focus of *Evaluation 2* was on error detection only. Again, the target system was equipped with mechanisms (this time with recovery turned off) and run for the same 25 test cases as before. Errors were injected into the monitored signals and in random locations in memory and stack areas. The results showed that, on a whole, errors in the monitored signals were detected with a probability of 74%. If only those errors that subsequently lead to system failure were taken into account, the probability of detection was 99.6%. For errors in memory locations, the overall detection probability was only 12.8% and for errors that lead to system failure it was 81.1%. Errors in stack were detected with an overall probability of 4.2% and a probability of 13.7% if only those errors that lead to system failure were taken into account.

The results of the two evaluations performed show that they are best at dealing with errors in the monitored signals. Handling errors in non-monitored areas requires these errors to propagate into the monitored areas in order to be detected and recovered. Thus, from the results here one can conclude that in order to make error detection and recovery as efficient as possible (with regard to the cost of these mechanisms) it is important to know how errors propagate and what their subsequent effect on system output is.

7.1.3 Error Propagation and Effect Analysis

The results obtained in the evaluation of the error detection and recovery mechanisms made it clear that in order obtain a high coverage, not only the effectiveness of the mechanisms is important, but also, how error propagate. That is, not only must

the available mechanisms be good at detecting and recovering errors, they must also be placed at locations where errors occur or propagate. This is especially important if a limited amount of resources is available for dependability purposes (which is the case for most consumer and/or low-cost systems). Analyzing the propagation of errors allows the profiling of software with regard to weaknesses and hot-spots—locations in the software which let errors pass through and/or location which attract propagating errors.

A propagation profile of a software system can in some cases be inadequate for selecting where to place error detection and recovery mechanisms. A location which attracts errors with a very small probability may be such that should an error occur at that location it would cause very much damage. Thus, the propagation profile of a software system has to be complemented with an effect profile, showing how much damage an error in different locations could cause.

In Chapter 6, an analysis framework is presented which enables system designers to profile software systems such that vulnerable modules and signals/variables can be identified. The framework introduces four basic measures: (i) *Exposure*, (ii) *Permeability*, (iii) *Impact*, and (iv) *Criticality*, and thus is called EPIC. This framework is able to produce the two distinct quantitative profiles mentioned above, namely: (i) a propagation profile (using *exposure* and *permeability*), and (ii) an effect profile (using *impact* and *criticality*). The propagation profile shows how errors propagate through the software system, and the effect analysis shows to what extent errors in the various signals/variables affect system output.

EPIC is mainly focused towards black-box modular software, i.e., modular software where only the I/O-characteristics and basic functionality is known and the internals of the modules is either unknown or unchangeable.

A fault-injection approach for estimating the measures of the framework is also introduced. Here, errors are injected into each individual input signal of a module and the output signals are observed for any differences compared to reference runs (i.e., using classical Golden Run Comparison).

7.1.4 Evaluation of Analysis Framework

The analysis framework, EPIC, presented in Chapter 6 is evaluated on real software in order to illustrate its applicability. The target system is again the aircraft arresting system used in the evaluation of the error detection and recovery mechanisms. The introduced approach for estimating the various measures in the EPIC framework was used and two distinct profiles created. Using the profiles, two sets of mechanisms for error detection were created. One set was selected based on the propagation profile

only and contained four mechanisms, and one set was selected based on both the propagation profile and the effect profile and contained seven mechanisms.

Two versions of the target system were created, one for each set of mechanisms. The two versions were then subjected to error injection with two different error models: i) errors are introduced at the system input signals only, and ii) error are introduced in random locations in memory and stack areas. The results show that the error detection coverage obtained for the first error model was the same for both sets of error detection mechanisms. Thus, in this case the first set is to be preferred as it consumes less resources than the second set (four mechanisms versus seven mechanisms). For the second error model, the error detection coverage was higher for the second set of mechanisms than the first set. In this case errors were introduced in random locations in memory and the second set of mechanisms monitored also those signals into which errors were not likely to propagate. Thus, low probability errors were detected more easily by the second error set than the first.

The results of the experiment with EPIC show that software profiling with regard to error propagation and error effect can facilitate rigorous selection of locations for error detection and recovery mechanisms. It also shows that the error model one assumes the system will be subject to during operation does greatly affect the error detection coverage obtained for a given setup of mechanisms. This shows that a propagation profile alone may not always for be sufficient and that other profiles have to be used.

7.1.5 Tool for Analyzing Error Propagation

To be able to perform the evaluation of EPIC, a tool-suite called PROPANE, the Propagation Analysis Environment, was developed. PROPANE is a so called SWIFI (SoftWare Implemented Fault Injection) tool, i.e., it injects faults and errors into its target using software, and is capable of tracing the values of variables in software such that error propagation and error effect can be analyzed. PROPANE can also log events which enables the evaluation of error detection and recovery mechanisms. There are vast extension possibilities, enabling users to construct their own injectors and logging probes, making PROPANE a very versatile tool.

The results that can be obtained from PROPANE “out-of-the-box” contain basic propagation analysis in the form of propagation graphs and propagation summaries. However, PROPANE will also compile the raw readouts from experiments such that further analysis can be performed using other tools. For instance, in the evaluation of EPIC, Matlab and MS Excel were used for the final analysis and generation of estimates of the measures.

PROPANE is a pure software-based tool, implemented in ANSI C, which facilitates porting it to different platforms. At this point in time, PROPANE is available for WIN32-systems (such as Windows 2000/XP). The injection and logging functions are packaged in a static library which is linked together with the target system. Therefore, PROPANE can be used together with a vast range of applications of varied types.

7.2 Conclusions

As software is more and more becoming that part of a computer system which defines its functionality, and as computer systems are used more frequently in consumer and other low-cost products, the demands for inexpensive dependability is increasing. Therefore, the search for techniques and methods for designing and analyzing dependable software for computer nodes in embedded systems is an active area. Thus, as stated in Chapter 1, the main goal of the work presented in this thesis has been to find and evaluate new construction methods for dependable computer nodes with little amount of redundancy in (possibly distributed) control systems.

The mechanisms for error detection and error recovery proposed and evaluated in Chapter 4 seem to be able to provide good error detection coverage for errors in the monitored areas, although they do not perform any intricate tests. Thus, executable assertions should be viewed as viable mechanisms for detection of data errors in embedded software.

The recovery on the other hand seems to have room for improvement. One reason for the low recovery rate exhibited by these mechanisms can be the error model used in the evaluation. As this error model was a very aggressive (almost vicious) one where errors were injected periodically with no relation to the period of the software, any recovery efforts are sure to have been in vain in many cases. Further evaluation with other error models may show other results. For example, one evaluation to perform would be recovery of one single error at one point in time during the arresting of an aircraft.

From the results of the evaluation of the mechanisms one can also conclude that adding error detection and recovery mechanisms to a given software system can benefit greatly from having knowledge of the propagation characteristics of the software. Therefore, the EPIC framework was introduced in Chapter 6. In experiments with this framework, it was shown that it can generate software profiles for error propagation characteristics and for error effect characteristics. Using these profiles, a system developer can select locations in the software which should be considered for placement of detection and recovery mechanisms, i.e., locations which attract

or promote propagating errors and/or locations which, when erroneous, have a high probability of affecting the output (and therefore the behavior) of the system.

The limitations of EPIC include the fact that the framework is based on the notion of *error permeability* which at this point only considers single errors individual signals. That is, only one signal was assumed to erroneous at any one time when defining the expressions for the measures. In reality, of course, multiple signals can be erroneous simultaneously. However, the profiling capabilities of EPIC should still be useful for system developers.

Another limitation, or question-mark rather, regarding EPIC is the issue of internal feedback loops across modules (cross-linked modules). At this point, the framework handles feedback loops within a single module. However, if there is feedback spanning multiple modules (i.e., a situation where the output of a module M_1 is used as input to another module M_2 and vice versa) this may have to be handled separately.

PROPANE, the tool developed for the evaluation of EPIC, and described in Chapter 5, is tool which can be useful on its own, without having to use the EPIC framework. An example study performed with the tool showed that basic propagation analysis can be performed with the tool. however, this propagation analysis is at the signal level only, i.e., it may not scale so easily if many signals are considered in the analysis. On the other hand, the tool is automated to such a degree that the problems with scaling mainly are with regard to the amount of data that is produced in the evaluations. Thus, it is more of a hardware resource/storage problem than a problem with the approach used by the tool.

CHAPTER 8

Outlook on Future Work

Science... never solves a problem without creating ten more.

— George Bernard Shaw (1856–1950)

In the process of investigating various questions and trying to find solutions to problems, one of course always stumbles across new questions and problems. Unfortunately (or fortunately?), the time allotted for a PhD thesis is limited and thus, some things have to either be skipped or have to wait until an opportunity presents itself where these new questions and problems can be addressed. This chapter contains a brief summary of the questions that had to be put aside while doing the work in this thesis. Hopefully, a chance to dive further into these topics will come up in the future.

8.1 The Future and Executable Assertions

As shown by the results in this thesis, executable assertions can be very efficient as error detection mechanisms in software. The method used here for design of these methods is to have generic parameterized mechanisms and instantiate them for each given variable/signal.

An interesting problem is whether the design of executable assertion can be further automated. This problem has been addressed in [Jhumka *et al.*, 2002(a)] and will be further investigated in the future.

The executable assertions in this thesis are aimed at individual signals. Perhaps a better result can be obtained if multiple signals are considered together as intricate relations between several values can be monitored? An initial investigation of this is also part of [Jhumka *et al.*, 2002(a)].

In [Askerdal *et al.*, 2002], work on model based analysis of control systems has been investigated. This is a path that should be further investigated as this is independent of the implementation details of the system and only considers the actual functionality.

8.2 The Future and Software Analysis

Analyzing software to find weaknesses and create various profiles of the characteristics should be investigated further as this can provide a substantial help for system designers when deciding upon design issues of dependability mechanisms, on resource allocations, on architectural and policy issues, etc.

An interesting question to investigate further is whether static analysis of software (source code, designs, etc.) can be used to identify potential weaknesses. If, for example, a statistical correlation can be found between some static metric obtained from the source code of a system and the measures obtained by the EPIC framework, perhaps the dynamic analysis performed in this thesis can be made statically instead? The benefits here would be shortened analysis time and thus easier adoption by industry.

Another take on static analysis is to investigate whether estimates of *error permeability* can be obtained by static analysis, for instance methods related to data flow analysis, can be used instead. Again, not having to run long and resource demanding experiments will probably make industry adopt such an approach more easily.

At this point, EPIC is mainly focused towards single-node software. Even though no explicit assumptions have been made that limit its use in a distributed

setting, this has not been fully investigated. Thus, applying the proposed software analysis methods on distributed systems should be part of future endeavors.

The work up to this point has been focused on data errors and on reliability and safety. Future directions should contain also control flow errors, and also gear towards security (e.g., data integrity, confidentiality) etc.

The analysis frame work presented in this thesis assumes a black-box view on modular software. Relaxing this assumption to include gray-box, or even white-box, knowledge of software may perhaps make it possible to provide even further information to software designers in the development and composition of software modules. This direction has been initialized in [Jhumka *et al.*, 2002(b)].

8.3 The Future and PROPANE

One aim for the future is to further develop PROPANE and make it available for free (including source code) for academic use. This way, the tool would likely be improved as more people take a look at its innards. Furthermore, experimental techniques for dependability evaluations are distributed such that more people have easy access to such methodologies.

Even though PROPANE can be augmented by the user to include virtually any kind of error model, it will be expanded to provide more built-in error types and error triggers than it does today. For example, one may want to inject a set of different errors in a given sequence or at various points in time (not just at one point in time or periodical).

Another issue to further expand in PROPANE is automation. At this point, PROPANE has automated the instrumentation, basic setup, experiment execution, and analysis. However, each step has to be initiated manually. Thus, one aspect that should be addressed is to make the total chain automatic, i.e., the user provides source code and basic description of the architecture of the system, and PROPANE then automatically produces the propagation and effect profiles and suggests main locations where (increased) EDM and ERM capability should be considered.

8.4 The Future and The Rest

Going to the distributed world opens up many areas which may not be closely related to the work presented in this thesis. For instance, communication protocols, distributed fault tolerance and adaptive systems, self-stabilization, etc.

In the area of experimental validation of fault-tolerance, there are open issues in representativeness of fault and error models used in the injection experiments. Perhaps also other injection approaches can be devised.

There is really no end to the possibilities...

Bibliography

- [Ammann and Knight, 1988] Ammann P.E. and Knight J.C., “Data Diversity: An Approach To Software Fault Tolerance”, IEEE Transactions on Computers, Vol. 37, No. 4, pp. 418–425, 1988
- [Anderson and Lee, 1982] Anderson T. and Lee P.A., “Fault Tolerance Terminology Proposals”, Proceedings of the 12th International Symposium on Fault-Tolerant Computing, pp. 29–33, 1982
- [Andrews, 1979] Andrews D.M., “Using Executable Assertions for Testing and Fault Tolerance”, Proceedings 9th International Symposium on Fault-Tolerant Computing, pp. 102–105, 1979
- [Arlat *et al.*, 1990] Arlat J., Aguera M., Amat L., Crouzet Y., Fabre J.-C., Laprie J.-C., Martins E., and Powell D., “Fault Injection for Dependability Validation: A Methodology and Some Applications”, IEEE Transactions On Software Engineering, Vol. 16, No. 2, pp. 166–182, Feb., 1990
- [Arlat *et al.*, 1993] Arlat J., Costes A., Crouzet Y., Laprie J.-C., and Powell D., “Fault Injection and Dependability Evaluation of Fault Tolerant Systems”, IEEE Transactions on Computers, Vol. 42, No. 8, pp. 913–923, 1993
- [Askerdal *et al.*, 2002] Askerdal Ö., Gäfvert M., Hiller M., and Suri N., “A Control Theory Approach for Analyzing the Effects of Data Errors in Safety-Critical Control Systems”, *to appear in* Proceedings of the Pacific Rim International Symposium on Dependable Computing (PRDC), 2002
- [Avizienis and Chen, 1977] Avizienis A. and Chen L., “On The Implementation Of N-Version Programming for Software Fault-Tolerance During Program Execution”, Proceedings of the 1977 International Conference on Computer Software and Applications, pp. 149–155, 1977
- [Avizienis, 1985] Avizienis A., “The N-Version Approach to Fault-Tolerant Software”, IEEE Transactions on Software Engineering, Vol. 11, No. 12, pp. 1491–1501, 1985
- [Barton *et al.*, 1990] Barton J.H., Czeck E.W., Segall Z.Z., and Siewiorek D.P., “Fault Injection Experiments Using FIAT”, IEEE Transactions on Computers, Vol. 39, No. 4, pp. 575–582, 1990

- [Carreira *et al.*, 1995] Carreira J., Madeira H., and Silva J., “Xception: Software Fault Injection and Monitoring in Processor Functional Units”, International IFIP Conference on Dependable Computing for Critical Applications (DCCA-5), pp. 135–149, 1995
- [Carreira *et al.*, 1998] Carreira J., Madeira H., and Silva J.G., “Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers”, IEEE Transactions on Software Engineering, Vol. 24, No. 2, pp. 125–136, Feb., 1998
- [Chandra *et al.*, 2000] Chandra R., Lefever R.M., Cukier M., and Sanders W.H., “Loki: A State-Driven Fault Injector for Distributed Systems”, Proceedings of the International Conference on Dependable Systems and Networks (DSN’00), pp. 237–242, 2000
- [Chillarege and Bowen, 1989] Chillarege R. and Bowen N.S., “Understanding Large System Failures - A Fault Injection Experiment”, Proceedings of the 19th International Symposium on Fault-Tolerant Computing (FTCS-19), pp. 356–363, 1989
- [Chillarege *et al.*, 1992] Chillarege R., Bhandari I.S., Chaar J.K., Halliday M.J., Moebius D.S., Ray B.K., and Wong M.-Y., “Orthogonal Defect Classification - A Concept for In-Process Measurements”, IEEE Transactions on Software Engineering, Vol. 18, No. 11, pp. 943–956, 1992
- [Chillarege *et al.*, 2002] Chillarege R., Goswami K., and Devarakonda M., “Error Propagation Decreases Confirming the Effect of Failure Acceleration in Fault Injection” (to appear in) Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE-02), 2002
- [Christmansson and Chillarege, 1996] Christmansson J. and Chillarege R., “Generation of an Error Set that Emulates Software Faults Based on Field Data”, Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26), pp. 304–313, 1996
- [Christmansson and Santhanam, 1996] Christmansson J. and Santhanam P., “Error Injection Aimed at Fault Removal in Fault Tolerance Mechanisms”, Proceedings of the 7th International Symposium on Software Reliability Engineering (ISSRE-96), pp. 175–184, 1996
- [Christmansson and Rimén, 1997] Christmansson J. and Rimén M., “A Fault Injection Control Campaign Computer (FIC³)”, Technical Report CE/298, Department of Computer Engineering, Chalmers University of Technology, 1997
- [Christmansson *et al.*, 1998] Christmansson J., Hiller M., and Rimén M., “An Experimental Comparison of Fault and Error Injection”, Proceedings of the 9th International Symposium on Software Reliability Engineering (ISSRE-98), pp. 369–378, 1998

- [Clarke and Wing, 1996] Clarke E. M. and Wing J. M. “Formal Methods: State of the Art and Future Directions”, ACM Computing Surveys, Vol. 28, No. 4, pp. 626–643, 1996
- [Clegg and Marzullo, 1997] Clegg M. and Marzullo K., “Predicting Physical Processes in the Presence of Faulty Sensor Readings”, Proceedings of the 27th, International Symposium on Fault-Tolerant Computing (FTCS-27), pp. 373–378, 1997
- [Csertán *et al.*, 1995] Csertán Gy., Pataricza A., and Selényi E., “Dependability Analysis in HW-SW Codesign”, Proceedings of the International Computer Performance and Dependability Symposium (IPDS’95), pp. 306–315, 1995.
- [Cukier *et al.*, 1999] Cukier M., Chandra R., Henke D., Pistole J., and Sanders W.H., “Fault Injection Based on a Partial View of the Global State of a Distributed System”, Proceedings of the Symposium on Reliable Distributed Systems (SRDS’99), pp. 168–177, 1999
- [Devarakonda *et al.*, 1990] Devarakonda M., Goswami K., and Chillarege R., “Failure Characterization of the NFS Using Fault Injection”, IBM Research Report, RC 16342, 12/5/90, 1990
- [Echtle and Masum, 1996] Echtle K. and Masum A., “A multiple bus broadcast protocol resilient to non-cooperative Byzantine faults”, Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26), pp. 158–167, 1996
- [Fabre *et al.*, 1999] Fabre J.-C., Salles F., Rodriguez Moreno M., and Arlat J., “Assessment of COTS Microkernels by Fault Injection”, Proceedings of the International IFIP Conference on Dependable Computing for Critical Applications (DCCA-7), pp. 25–44, 1999
- [Folkesson, 1999] Folkesson P., “Assessment and Comparison of Physical Fault Injection Techniques”, Ph.D. thesis, Technical Report No. 377, Department of Computer Engineering, Chalmers University of Technology, Sweden, 1999
- [Fujiwara and Shimono, 1983] Fujiwara H. and Shimono T., “On the Acceleration of Test Generation Algorithms”, Proceedings of the 13th International Symposium on Fault-Tolerant Computing (FTCS-13), pp. 98–105, 1983
- [Geoghegan and Avresky, 1996] Geoghegan S. J. and Avresky D., “Method for Designing and Placing Check Sets based on Control Flow Analysis of Programs”, Proceedings of the International Symposium on Software Reliability Engineering, (ISSRE’96), pp. 256–265, 1996.
- [GML, web-link] Information about *GML*, the *Graph Modelling Language*, and related tools is found at <http://www.infosun.fmi.uni-passau.de/Graphlet/GML>

- [Goel, 1981] Goel P., “An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits”, *IEEE Transactions on Computers*, Vol. 30, No. 3, pp. 215–222, 1981
- [Goswami and Iyer, 1991] Goswami K.K. and Iyer R., “A Simulation-Based Study of a Triple Modular Redundant System using DEPEND”, *Proceedings of the 5th International Tests, Diagnosis, Fault Treatment Conference*, pp. 300–311, 1991
- [Goswami, 1997] Goswami K.K., Iyer R.K., and Young L., “DEPEND: A Simulation-Based Environment for System Level Dependability Analysis”, *IEEE Transactions on Computers*, Vol. 46, No. 1, pp. 60–74, Jan., 1997
- [Gough and Klaeren, 1997] Gough K.J. and Klaeren H., “Executable Assertions and Separate Compilation”, *Proceedings of the Joint Modular Languages Conference (JMLC’97)*, pp. 41–52, 1997
- [Graphviz, web-link] Information about the tool suite to which *dot* belongs is found at <http://www.research.att.com/sw/tools/graphviz>
- [Gunneflo *et al.*, 1989] Gunneflo U., Karlsson J., and Torin J., “Evaluation of Error Detection Schemes Using Fault Injection By Heavy-Ion Radiation”, *Proceedings of the 19th International Symposium on Fault-Tolerant Computing (FTCS-19)*, pp. 340–347, 1989
- [Han *et al.*, 1995] Han S., Shin K.G., and Rosenberg H.A., “DOCTOR: An Integrated Software Fault InjeCTiOn EnviRonment for Distributed Real-Time System”, *Proceedings of the International Computer Performance and Dependability Symposium (IPDS’95)*, pp. 204–213, 1995
- [Hecht, 1976] Hecht H., “Fault-Tolerant Software for Real-Time Applications”, *ACM Computing Surveys*, Vol. 8, No. 4, pp. 391–407, 1976
- [Hiller, 1998] Hiller M., “Software Fault Tolerance Techniques from a Real-Time Systems Point of View: An Overview”, *Technical Report CE/98-16*, Department of Computer Engineering, Chalmers University of Technology, Sweden, (available at <http://www.ce.chalmers.se/staff/hiller/>), 1998
- [Hiller, 1999] Hiller M., “Error Recovery Using Forced Validity Assisted by Executable Assertions for Error Detection: An Experimental Evaluation”, *Proceedings of the 25th EUROMICRO Conference*, Vol. II, pp. 105–112, 1999
- [Hiller, 2000(a)] Hiller M., “Executable Assertions for Detecting Data Errors in Embedded Control Systems”, *Proceedings of the International Conference on Dependable Systems and Networks (DSN’00)*, pp. 24–33, 2000

- [Hiller, 2000(b)] Hiller M., “A Tool for Examining the Behaviour of Faults and Errors in Software”, *Technical Report CE/00-19*, Department of Computer Engineering, Chalmers University of Technology, Sweden, (available at <http://www.ce.chalmers.se/staff/hiller/>), 2000
- [Hiller *et al.*, 2001] Hiller M., Jhumka A., and Suri N., “An Approach for Analysing the Propagation of Data Errors in Software”, *Proceedings of the International Conference on Dependable Systems and Networks (DSN'01)*, pp. 161–170, 2001
- [Hiller *et al.*, 2002(a)] Hiller M., Jhumka A., and Suri N., “On the Placement of Software Mechanisms for Detection of Data Errors”, *Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)*, pp. 135–144, 2002
- [Hiller *et al.*, 2002(b)] Hiller M., Jhumka A., and Suri N., “PROPANE: An Environment for Examining the Propagation of Errors in Software”, *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'02)*, ACM Software Engineering Notes (SEN), Vol. 27, No. 4, pp. 81–85, 2002
- [Horning *et al.*, 1974] Horning J.J., Lauer H.C., Melliar-Smith P.M., and Randell B., “A Program Structure for Error Detection And Recovery”, *Lecture Notes in Computer Science (LNCS)*, Vol. 16, pp. 172–187, 1974
- [Hudak *et al.*, 1993] Hudak J.J., Suh B.-H., Siewiorek D.P., and Segall Z., “Evaluation and Comparison of Fault-Tolerant Software Techniques”, *IEEE Transactions on Reliability*, Vol. 42, No. 2, pp. 190–204, June, 1993
- [Iyer *et al.*, 1990] Iyer R.K., Young L.T., and Iyer P.V.K., “Automatic Recognition of Intermittent Failures: An Experimental Study of Field Data”, *IEEE Transactions on Computers*, Vol. 39, No. 4, pp. 525–537, 1990
- [Iyer, 1995] Iyer R.K., “Experimental Evaluation”, Special Issue FTCS-25 Silver Jubilee, 25th International Symposium on Fault-Tolerant Computing (FTCS-25), pp. 115–132, 1995
- [Iyer and Tang, 1996] Iyer R.K. and Tang D., “Experimental Analysis of Computer System Dependability”, Chapter 5 in *Fault-Tolerant Computer System Design* (ed. D.K. Pradhan), Prentice Hall, 1996
- [Jenn *et al.*, 1994] Jenn E., Arlat J., Rimén M., Ohlsson J., and Karlsson J., “Fault Injection into VHDL Models: The MEFISTO Tool”, *Proceedings of the 24th International Symposium on Fault-Tolerant Computing (FTCS-24)*, pp. 66–75, 1994
- [Jhumka *et al.*, 2001] Jhumka A., Hiller M., and Suri N., “Assessing Inter-modular Error Propagation in Distributed Software”, *Proceedings of the Symposium on Reliable Distributed Systems (SRDS'01)*, pp. 152–161, 2001

- [Jhumka *et al.*, 2002(a)] Jhumka A., Hiller M., and Suri N., “On Systematic Design of Consistent Executable Assertions for Distributed Embedded Software”, Proceedings of the ACM Joint Conference Languages Compilers and Tools for Embedded Systems/Software and Compilers for Embedded Systems (LCTES/SCOPES), pp. 74–83, 2002
- [Jhumka *et al.*, 2002(b)] Jhumka A., Hiller M., and Suri N., “Component-Based Synthesis of Dependable Embedded Software”, Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT’02), Lecture notes on Computer Science (LNCS) 2469, pp. 111–128, 2002
- [Kanawati *et al.*, 1995] Kanawati G.A., Kanawati N.A., and Abraham J.A., “FERRARI: A Flexible Software-Based Fault and Error Injection System”, IEEE Transactions on Computers, Vol. 44, No. 2, pp. 248–260, February, 1995
- [Kao, 1994] Kao W.L., “Experimental Study of Software Dependability”, Ph.D. thesis, Technical report CRHC-94-16, Department of Computer Science, University of Illinois at Urbana-Champaign, Illinois, 1994
- [Kao and Iyer, 1995] Kao W.-l. and Iyer R.K., “DEFINE: A Distributed Fault Injection and Monitoring Environment”, Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, pp. 252–259, 1995
- [Kim, 1989] Kim K.H. and Welch H.O., “The Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults in Real-Time Applications”, IEEE Transaction on Computers, Vol. 38, No. 5, pp. 626–636, 1989
- [Knight and Leveson, 1986] Knight J.C. and Leveson N.G., “An Experimental Evaluation of the Assumption of Independence in Multiversion Programming”, IEEE Transactions on Software Engineering, Vol. 12, No. 1, pp. 96–109, 1986
- [Koga and Kolasinski, 1984] Koga R. and Kolasinski W., “Heavy-ion Induced Single Event Upsets of Microcircuits; A Summary of the Aerospace Corporation Test Data”, IEEE Transaction on Nuclear Science, Vol. 31, No. 6, pp. 1190–1195, 1984
- [Laprie *et al.*, 1987] Laprie J.-C., Arlat J., Béounes C., Kanoun K., and Hourtolle C., “Hardware- and Software-Fault Tolerance: Definition and Analysis of Architectural Solutions”, Proceedings of the 17th International Symposium on Fault-Tolerant Computing (FTCS-17), pp. 116–121, 1987
- [Laprie (ed.), 1992] Laprie J.-C. (ed.), “Dependability: Basic Concepts and Terminology”, Dependable Computing and Fault-Tolerant Systems series, Vol. 5, Springer-Verlag, 1992

- [Laprie, 1995] Laprie J.-C., “Dependable Computing: Concepts, Limits, Challenges”, Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25), pp. 42–54, 1995.
- [Lee, 1978] Lee P.A., “A Reconsideration of the Recovery Block Scheme”, Computer Journal, Vol. 21, No. 4, pp. 306–310, 1978
- [Lee and Anderson (ed.), 1990] Lee P.A and Andersson T. (ed.), “Fault Tolerance - Principles and Practice”, Dependable Computing and Fault-Tolerant Systems Series, Vol. 3, Second Edition, Springer Verlag, 1990
- [Leveson *et al.*, 1990] Leveson N.G., Cha S.S., Knight J.C., and Shimeall T.J., “The Use of Self Checks and Voting in Software Error Detection: An Empirical Study”, IEEE Transactions on Software Engineering, Vol. 16, No. 4, pp. 432–443, 1990
- [Lyu (ed.), 1995] Lyu M.R. (ed.), “Handbook of Software Reliability Engineering”, ISBN 0-07-039400-8, McGraw-Hill, 1995
- [Madeira *et al.*, 1994] Madeira H., Rela M., Moreira F., and Silva J.G., “RIFLE: A General Purpose Pin-level Fault Injector”, Proceedings of the 1st European Dependable Computing Conference (EDCC-1), pp. 199–216, 1994
- [Mahmood *et al.*, 1984] Mahmood A., Andrews D.M., and McCluskey E.J., “Executable Assertions and Flight Software”, Proceedings of the 6th AIAA/IEEE Digital Avionics Systems Conference (DASC-6), pp. 346–351, 1984
- [McMillin and Ni, 1988] McMillin B.M. and Ni L.M., “Executable Assertions Development for the Distributed Parallel Environment”, Proceedings of the 23th International Symposium on Fault-Tolerant Computing (FTCS-23), pp. 228–237, 1993
- [Michael and Jones, 1997] Michael C. C. and Jones R. C., “On the Uniformity of Error Propagation in Software”, Proceedings of the International Conference on Computer Assurance (COMPASS’97), pp. 68–76, 1997
- [Morell *et al.*, 1997] Morell L., Murrill B., and Rand R., “Perturbation Analysis of Computer Programs”, Proceedings of the International Conference on Computer Assurance (COMPASS’97), pp. 77–87, 1997
- [Powell *et al.*, 1995] Powell D., Martins E., Arlat J., and Crouzet Y., “Estimators for Fault Tolerance Coverage Evaluation”, IEEE Transactions on Computers, Vol. 44, No. 2, pp. 261–274, 1995
- [Rabéjac *et al.*, 1996] Rabéjac C., Blanquart J.-P., and Queille J.-P., “Executable Assertions and Timed Traces for On-Line Software Error Detection”, Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26), pp. 138–147, 1996

- [Randell, 1975] Randell B., “System Structure for Software Fault-Tolerance”, IEEE Transactions on Software Engineering, Vol. 1, No. 2, pp. 220–232, 1975
- [Randell *et al.*, 1978] Randell B., Lee P.A., and Treleaven P.C., “Reliability Issues in Computing System Design”, Computing Surveys, Vol. 10, pp. 123–165, 1978
- [Randell and Xu, 1995] Randell B. and Xu J., “The evolution of the recovery block concept”, Chapter 1 in *Software Fault Tolerance*, Lyu M.R. (ed.), Wiley & Sons, 1995
- [Rimén *et al.*, 1994] Rimén M., Ohlsson J., and Torin J., “On Microprocessor Error Behavior Modeling”, Proceedings 24th International Symposium on Fault-Tolerant Computing (FTCS-24), pp. 76–85, 1994
- [Rosenblum, 1995] Rosenblum D.S., “A Practical Approach to Programming with Assertions”, IEEE Transactions on Software Engineering, Vol. 21, No. 1, pp. 19–13, 1995
- [Roth, 1980] Roth J.P., *Computer Logic, Testing and Verification*, Computer Press, 1980.
- [Saib, 1978] Saib S.H., “Executable Assertions - An Aid To Reliable Software”, Proceedings of the 11th Asilomar Conference on Circuits, Systems and Computers, pp. 277–281, 1978.
- [Salles, 1999] Salles F., Rodriguez M., Fabre J.C., and Arlat J., “MetaKernels and Fault Containment Wrappers”, Proceedings of the 29th International Symposium on Fault-Tolerant Computing (FTCS-29), pp. 22–29, 1999.
- [Scott *et al.*, 1983] Scott R.K., Gault J.W., and McAllister D.F., “The Consensus Recovery Block”, Proceedings of the Total System Reliability Symposium, pp. 74–85, 1983
- [Segall *et al.*, 1988] Segall Z., Vrsalovic D., Siewiorek D., Yaskin D., Kownacki J., Barton J., Dancy R., Robinson A., and Lin T., “FIAT – Fault-Injection based Automated Testing environment”, Proceedings 18th International Symposium on Fault-Tolerant Computing (FTCS-18), pp. 102–107, 1988
- [Shin and Lin, 1988] Shin K. G. and Lin T.-H., “Modeling and Measurement of Error Propagation in a Multimodule Computing System”, IEEE Transactions on Computers, Vol. 37, No. 9, pp. 1053–1066, 1988
- [Steininger and Scherrer, 1997] Steininger A. and Scherrer C., “On Finding an Optimal Combination of Error Detection Mechanisms Based on Results of Fault Injection Experiments”, Proceedings 27th International Symposium on Fault-Tolerant Computing (FTCS-27), pp. 238–247, 1997

- [Sieh *et al.*, 1997] Sieh V., Tschäche O. and Balbach F., “VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions”, Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS-27), pp. 32–36, 1997
- [Sinha and Suri, 1999] Sinha P. and Suri N., “On the Use of Formal Techniques for Analyzing Dependable Real-Time Protocols”, Proceedings of the Real-Time Systems Symposium (RTSS’99), pp. 126–135, 1999
- [Stott *et al.*, 2000] Stott D.T., Floering B., Burke D., Kalbarczyk Z., and Iyer R., “NF-TAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors.”, Proceedings of the 4th International Computer Performance and Dependability Symposium (IPDS’00), pp. 91–100, 2000
- [Stroph and Clarke, 1998] Stroph R. and Clarke T., “Dynamic Acceptance Tests for Complex Controllers”, Proceedings of the 24th EUROMICRO Conference, pp. 411–417, 1998
- [Sullivan and Chillarege, 1991] Sullivan M. and Chillarege R., “Software Defects and their Impact on System Availability - a Study of Field Failures in Operating Systems”, Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS-21), pp. 2–9, 1991
- [Suri and Sinha, 1998] Suri N. and Sinha P., “On the Use of Formal Techniques for Validation”, Proceedings of the 28th International Symposium on Fault-Tolerant Computing (FTCS-28), pp. 390–399, 1998
- [Tsai and Iyer, 1996] Tsai T.K. and Iyer R.K., “An Approach towards Benchmarking of Fault-Tolerant Commercial Systems”, Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26), pp. 314–325, 1996
- [USAF, 1986] US Air Force - 99, “MIL-SPEC: Aircraft Arresting System BAK-12A/E32A; Portable, Rotary Friction”, MIL-A-38202C, Notice 1, US Dept. of Defense, Sept. 2, 1986
- [Voas and Morell, 1990] Voas J. and Morell L. J., “Propagation and Infection Analysis (PIA) Applied to Debugging”, Proceedings of Southeastcon’90, pp. 379–383, 1990
- [Voas, 1992] Voas J., “PIE: A Dynamic Failure-Based Technique”, IEEE Transactions on Software Engineering, Vol. 18, No. 8, pp. 717–727, 1992
- [Voas *et al.*, 1998] Voas J., Charron F., and Beltracchi L., “Error Propagation Analysis Studies in a Nuclear Research Code”, Proceedings of the IEEE Aerospace Conference, Vol. 4, pp. 115–121, 1998

- [Walter, 1990] Walter C.J., “Evaluation and Design of an Ultra-Reliable Distributed Architecture for Fault-Tolerance”, IEEE Transactions on Reliability, Vol. 39, No. 4, pp. 492–499, 1990
- [X-by-wire, 1998] X-By-Wire Team, “X-By-Wire: Safety Related Fault Tolerant Systems in Vehicles”, Final Report, XbyWire-DB-6/6-24, Version 2.0.0, November 26, 1998
- [Yin and Bieman, 1994] Yin H. and Bieman J.M., “Improving software Reliability with Assertion Insertion”, Proceedings of the International Test Conference (ITC), pp. 831–839, 1994

Appendix A.

PROPANE – Details

In this appendix, details about instrumentation of target systems and setup of experiments for PROPANE are described. The PROPANE tool suite itself is described in Chapter 5.

A.1 Instrumentation of Target Systems

In order to generate readouts and inject faults and errors, a target system has to be instrumented. This instrumentation consists of inserting probes for logging variables and events as well as inserting injection locations for faults and errors. The basic work-flow of target system instrumentation is depicted in Fig. A.1.

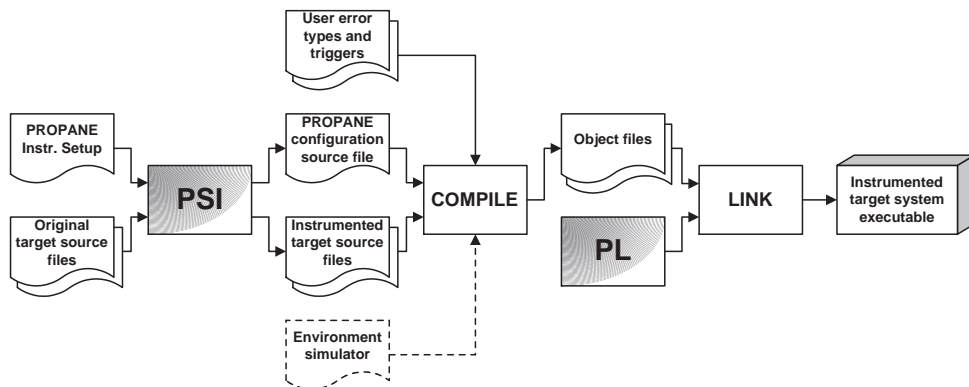


Figure A.1: The basic work-flow of target system instrumentation.

The PROPANE configuration source file contains static information required during the execution of experiment by the PROPANE Library. This information contains the static setup of variable probes, event probes, faults, error injection locations, user error types, and user error triggers. As this information is constant, it will remain the same between different experiments. Dynamic information required for experiment execution is provided in a set of description files (described in Section A.5 of this appendix).

The PROPANE configuration source file and the instrumented target source files are compiled and linked together with the PROPANE Library to form the instrumented target system executable. This executable is then used by the PROPANE Campaign Driver when conducting experiments.

Details about the PROPANE configuration source file is given here in order to facilitate manual construction in case a PROPANE user wants to do that. However, the PROPANE tools suite contains the PROPANE System Instrumentor (the PSI), which can automatically generate instrumented source code and the PROPANE configuration source file. In general, there are four ways of instrumenting the target source code: i) by going the hard-core programmer way and doing everything manually, ii) be using PSI generate the PROPANE configuration source code and inserting the probes in the target source manually, iii) by annotating the code with variable probes, event probes and error injection locations and then letting PSI generate the PROPANE configuration source file and instrumented target source, and iv) specifying the modular composition of the system and for each module specify I/O and internal characteristics.

Method i) is clearly the most time consuming way to go, but may prove fun for some people. Method ii) is somewhat more pleasant as the PROPANE configuration source file is generate automatically. Methods iii) and iv) are of course the recommended methods for instrumenting target systems. Method iii) is semi-automatic as the locations for variable probes, event probes and error injection locations have to be chosen manually. However, the actual insertion of the corresponding calls to the PL API is done by PSI. Method ii) on the other hand is fully automatic, i.e., PSI chooses which variable probes to and error injection locations to insert and where to insert them. Note that event probes must be inserted either manually or by means of annotations as PSI cannot detect locations where certain events are to be flagged. Fault locations must always be inserted manually as this requires additional (faulty) code to be inserted into the target system.

This section describes how to instrument the target system using PSI and how to generate the PROPANE configuration source file manually.

A.1.1 Instrumenting for Probes

In order to be able to log variables and events, probes must be inserted into the target system. First we describe how to insert probes manually into the target system and how to create the necessary structures in the PROPANE configuration source file and then we describe how to annotate the target system such that variable probes and event probes can be inserted using PSI.

Manual insertion of probes

Probes are inserted in two steps. First, the required probes must be set up in the PROPANE configuration source file. Then, special calls to the PROPANE Library must be made from the application in order to actually log the variable or event.

If variable probes are used, the PROPANE configuration source file must contain the following data items:

```
/* This variable contains the number of defined variable probes */
unsigned int propane_no_of_log_vars = n;

/* This array contains information about the defined probes */
PROPANELogVarInfo propane_log_var_info[n] =
{
    /* Each probe must be defined with a line as shown below. */
    { handle, type, name, size, channel },
    ...
};

/* This array is used by the PROPANE Library during run-time */
PROPANELogVarData propane_log_var_data[n];
```

The variable `propane_no_of_log_vars` holds the number of defined variable probes in the variable probe information and data arrays. It is very important that the arrays contain exactly the number of entries as specified in this variable.

The array `propane_log_var_info` contains information about the defined variable probes that will not change over time. For every variable probe that is to be inserted into the system, there must be one entry containing the following information:

- handle
- probe type
- name
- size (size of logged data in bytes - only for probes of type `PROPANE_AREA`)
- channel

The *channel* will be created automatically during the PROPANE Library setup-phase and is not entered in the PROPANE configuration source file. The *handle* is an integer value and is used in the function call made from the target system to the PROPANE Library performing the actual logging of the variable. The *handle* must be equal to the index in the array, i.e., the first probe must have handle 0, the second

handle 1, and so on. The n^{th} probe must have handle $(n - 1)$. It is a good idea to create a pre-processor constant (i.e., a `#define` constant) for each handle and then use that constant instead of the actual numerical values in the function calls in the target system. The *probe type* indicates the type of the variable that is logged by that probe. The *probe type* may be one of the following:

- PROPANE_CHAR – The variable is of type `char`.
- PROPANE_UCHAR – The variable is of type `unsigned char`.
- PROPANE_SHORT – The variable is of type `short`.
- PROPANE_USHORT – The variable is of type `unsigned short`.
- PROPANE_INT – The variable is of type `int`.
- PROPANE_UINT – The variable is of type `unsigned int`.
- PROPANE_LONG – The variable is of type `long`.
- PROPANE_ULONG – The variable is of type `unsigned long`.
- PROPANE_FLOAT – The variable is of type `float`.
- PROPANE_DOUBLE – The variable is of type `double`.
- PROPANE_AREA – The variable is a pointer to a memory area. This is a special type which, for instance, may be used for injecting errors into random locations in the memory areas of the target software.

The *name* of the probe may be any normal C-string not containing white space. This name must then be used in the PROPANE description files (described in Section A.5 of this appendix) to activate the probe during experiment execution. The *size* value is only required for probes of type PROPANE_AREA. This value indicates the size (in bytes) of the area. For all other probe types, the size will be automatically calculated during the setup process of the PROPANE Library as they are standard data types with defined sizes.

The array `propane_log_var_data` contains information about the defined event probes that will be defined during the setup process of the PROPANE Library and may change during the execution of experiments. This array only has to be declared in the PROPANE configuration source file, initialization will be done during the setup process of the PROPANE Library.

The following is an example showing how variable probes are defined in the PROPANE configuration source file and how they are inserted into the source code of the target system. This example defines three variable probes named `SetValue`, `IsValue`, and `t[]`.

```

/* We have three variable probes. This is entered in the
 * PROPANE configuration source file.
 */
#define P_SETVALUE          (0)
#define P_ISVALUE           (1)
#define P_T_ARRAY           (2)

unsigned int propane_no_of_log_vars = 3;

PROPANELogVarInfo propane_log_var_info[3] =
{
    { P_SETVALUE, PROPANE_INT, "SetValue", 0, "", },
    { P_ISVALUE,  PROPANE_INT, "InValue",  0, "" },
    { P_T_ARRAY,  PROPANE_AREA, "t[]",      sizeof(Type_t)*10, "" },
};

PROPANELogVarData propane_log_var_data[3];

```

When all the desired variable probes have been defined in the PROPANE configuration source file, they must be inserted into the target system. The function in the API of the PROPANE Library that is used for variable probes has the following prototype:

```

PROPANEReturnCode
propane_log_var( PROPANESignalID handle,
                void * value );

```

The parameter `handle` is the handle of the probe created in the PROPANE configuration source file. The pointer `value` is the address to the variable that is to be logged. As the type of the variable that is logged is provided in the static information in the PROPANE configuration source file, the same function can be used for all types of variables. The function returns either `PROPANE_OK` if everything went fine, or `PROPANE_FAILURE` if something went wrong. Errors during this function call can only occur if the contents of the probe information and data structures have been corrupted. Here is an example of how the instrumentation for variable probes may look in the source code of the target system:

```

/* This is how the function calls are made in the
 * target system source code.
 */
int SetValue;
PROPANEReturnCode probe_rc;
...

```

```

probe_rc = propane_log_var( PROBE_SETVALUE, &SetValue );
if( PROPANE_OK == probe_rc )
{
    /* Everything went fine! */
}
else
{
    /* Something went wrong! */
}

```

Note that the steps above are not sufficient for getting a working variable probe during experiment execution, the probe must also be activated in the Experiment Description (see Section A.5 of this appendix).

The actions required for event probes are similar to those required for variable probes. The following structures for information and for data have to be declared in the PROPANE configuration source file:

```

/* This variable contains the number of defined event probes */
unsigned int propane_no_of_log_events = n;

/* This array contains information about the defined probes */
PROPANLogEventInfo propane_log_event_info[n] =
{
    /* Each probe must be defined with a line as shown below. */
    { handle, name, channel },
    ...
};

/* This array is used by the library during run-time */
PROPANLogEventData propane_log_event_data[n];

```

The variable `propane_no_of_log_events` contains the number of defined event probes in the event probe information and data arrays. It is very important that the arrays contain exactly the number of entries as specified in this variable.

The array `propane_log_event_info` contains information about the defined event probes that will not change over time. For every event probe that is to be inserted into the system, there must be one entry containing the following information:

- handle
- name
- channel

The only difference between variable probes and event probes from this point of view is that event probes do not have a *type*. That is, all events are considered to be type-less. The remaining information works as described for variable probes above.

The array `propane_log_event_data` contains information about the defined event probes that will be defined during the setup process of the PROPANE Library and may change during the execution of experiments. This array only has to be declared in the PROPANE configuration source file, initialization will be done during the setup process of the PROPANE Library.

The following shows how event probes are defined in the PROPANE configuration source file and how they are inserted into the source code of the target system. This example defines two probes, `EA1_detection` and `EA2_detection`.

```

/* We have two event probes. This is entered in the
 * PROPANE configuration source file.
 */
#define P_EA1_DETECT                (0)
#define P_EA2_DETECT                (1)

unsigned int propane_no_of_log_events = 2;

PROPANELogEventInfo propane_log_event_info[2] =
{
    { P_EA1_DETECT, "EA1_detection", "" },
    { P_EA2_DETECT, "EA2_detection", "" },
};

PROPANELogEventData propane_log_event_data[2];

```

When all the desired event probes have been defined in the PROPANE configuration source file, they must be inserted into the target system. The function in the API of the PROPANE Library that is used for variable probes has the following prototype:

```

PROPANEReturnCode
propane_log_event( PROPANEEventID handle );

```

This function only takes one parameter, `handle`, which is the handle of the probe created in the PROPANE configuration source file. The function returns either `PROPANE_OK` if everything went fine, or `PROPANE_FAILURE` if something went wrong. Errors during this function call can only occur if the contents of the probe

information and data structures have been corrupted. Here is an example of how the instrumentation for event probes may look in the source code of the target system.

```

/* This is how the function calls are made in the
 * target system source code.
 */
PROPANE_ReturnCode probe_rc;

...
/* Run Executable Assertion 1 */
if( assertC( SetValue ) == ERROR_DETECTED )
{
    probe_rc = propane_log_event( PROBE_EA1_DETECT );
    if( PROPANE_OK == probe_rc )
    {
        /* Everything went fine! */
    }
    else
    {
        /* Something went wrong! */
    }
}

```

Note that the steps above are not sufficient for getting a working variable probe during experiment execution, the probe must also be activated in the Experiment Description (see Section A.5 of this appendix).

PSI can be used to create the PROPANE Configuration source file. For variable probes and event probes, all that is needed is entries in the PROPANE Instrumentation Setup file for PSI. In this file, a variable probe is specified as follows:

```

// This is entry for a variable probe
>begin variable probe
    >handle <handle>
    >name <name>
    >type <probe type>
    >size <size_count> <size_type>
>end

```

Note that the target variable of the probe is not specified here. The link between the probe and the actual variable that is to be logged has to be made manually, i.e., using PSI to generate the PROPANE configuration source file still requires that the calls to the PL API be inserted manually. An event probe is specified using an entry in the PROPANE Instrumentation Setup file as follows:

```
// This is entry for an event probe
>begin event probe
    >handle <handle>
    >name <name>
>end
```

As was the case for the variable probes, the calls to the PL API must still be inserted manually.

Annotated insertion of probes

By annotating the target system, PSI can be used to automatically insert calls to the PL API in the target source code and to generate the PROPANE configuration source file. An annotation for variable probes and event probes look as follows:

```
/* This is an annotation for a variable probe */
/*@P>vprobe <handle> <target> <type> <size_count> <size_type> */

/* This is an annotation for an event probe */
/*@P>eprobe <handle> */
```

By having the annotations within ordinary C comments, a standard C compiler can be used for the original target source code and the annotations will not affect the software. If the annotated source is run through PSI instead, an instrumented version of the file and entries in the PROPANE configuration source file will be generated.

The annotation for variable probes and event probes do not have a separate field for the name of the probe (remember that the handle and the name of a probe need not be the same). For variable probes PSI will automatically set the name of the probe to the same as the name of the target and for event probes, the name will be the same as the handle.

For example, the annotations:

```
/*@P>vprobe P_SETVALUE SetValue PROPANE_INT */
/*@P>vprobe P_ISVALUE IsValue PROPANE_INT */
/*@P>vprobe P_T_ARRAY t[] PROPANE_AREA 10 Type_t */
```

would generate the same structures in the PROPANE configuration file as those shown for variable probes in the previous section on manual insertion. Note that size information is only necessary if the probed variable is of the type PROPANE_AREA. In that case, the size is specified with a number and a type. The number specifies

how many items of the specified type the chosen target represents. In the example above, `t[]` is thus the size of 10 structures of type `Type_t`. One may also specify the type as being `bytes` if no specially defined type exists.

When it comes to event probes, the annotations:

```
/*@P>eprobe P_EA1_DETECT */
/*@P>eprobe P_EA2_DETECT */
```

would create similar structures in the PROPANE configuration source file as those shown for event probes in the previous section on manual insertion. The main difference is that the handle and the name of the event probe will be the same.

If several annotations were to be made for the same variable probe or event probe, only one entry will be made in the PROPANE configuration source file. However, each annotation will generate a call to the PL API at the same place as the annotation.

A.1.2 Instrumenting for Fault Injection

Fault injection with PROPANE requires three tasks:

1. Defining the faults in the PROPANE configuration source file,
2. Inserting the faults and fault activation guards into the target source code, and
3. Activating faults in the *Experiment Descriptions* (described in Section A.5 of this appendix).

As the injected faults are modifications of the actual source code of the target system, the only limitation on what a fault can actually be is set by the imagination of the experimenter. This, however, makes the instrumentation of the target system, with regard to faults, a manual task. PSI can only be used for generating the necessary data structures in the PROPANE configuration source file. All faults are always present in the system at run-time, however, in an inactive state. The faults that are to be injected in an experiment are activated in the *Experiment Descriptions*.

If fault injection is used, each fault must be specified in the PROPANE configuration source file using the following data items:

```
/* This variable contains the number of defined faults */
unsigned int propane_no_of_faults = n;

/* This array contains info about the defined faults */
PROPANEFaultInfo propane_fault_info[n] =
{
```

```

/* Each fault must be defined with a line as shown below. */
{ handle, name, channel },
...
};

```

The variable `propane_no_of_faults` contains the number of defined faults in the fault information array. It is very important that the array contains exactly the number of entries as specified in this variable.

The array `propane_fault_info` contains information about the defined faults that will not change over time. For every fault, there must be one entry containing the following information:

- handle
- name
- channel

The *channel* will be created automatically during the PROPANE Library setup-phase and is not entered in the PROPANE configuration source file. The *handle* is an integer value and is used in the function call made from the target system to the PROPANE Library checking the fault activation guard. The *handle* must be equal to the index in the array, i.e., the first fault must have handle 0, the second handle 1, and so on. The n^{th} fault must have handle $(n - 1)$. It is a good idea to create a pre-processor constant (i.e., a `#define` constant) for each handle and use that instead of the numerical values in the function calls in the target system.

The *name* of the fault may be any normal C-string not containing white space. This name must then be used in the PROPANE description files (described in Section A.5 of this appendix) to activate the fault during experiment execution.

The following is an example showing how faults are defined in the PROPANE configuration source file.

```

/* We define two faults. This is entered in the
 * configuration source file.
 */
#define F_001                (0)
#define F_002                (1)

unsigned int propane_no_of_faults = 2;

PROPANEFaultInfo propane_fault_info[2] =
{
    { F_001, "Fault_f001", "", NULL },

```

```
{ F_002, "Fault_f002", "", NULL },
};
```

When all the desired faults have been defined in the PROPANE configuration source file, the corresponding faulty code must be inserted into the target system and guarded by a fault activation guard. The function in the API of the PROPANE Library that is used as the activation guard has the following prototype:

```
PROPANEReturnCode
propane_fault_is_active( PROPANEFaultID handle );
```

This function only takes one parameter, `handle`, which is the handle of the fault defined in the PROPANE configuration source file. The function returns either `(0u)` if the fault *is not* activated (or the specified handle does not exist in the information structure), or `(1u)` if the fault *is* activated.

The following is an example showing how a fault is inserted into the target system, and how the fault activation guard is placed:

```
/* This is how the function calls are made in the
 * target system source code.
 */

int SetValue
...
if( (1u) == propane_fault_is_active( F_001 ) )
{
    /* Fault F_001 is activated. Execute faulty code. */
    ...
}
else
{
    /* Fault F_001 is not activated. Execute correct code. */
}
```

Note that the steps above are not sufficient for injecting faults. The faults that are to be injected must also be activated in the Experiment Description (see Section A.5 of this appendix).

Although the faults have to be manually inserted into the target source code, PSI can be used to generate the required structures in the PROPANE configuration source file by adding entries for faults in the PROPANE Instrumentation Setup as follows:

```
// This is an entry for a fault
>begin fault location
    >handle <handle>
    >name <name>
>end
```

The calls to the PL API as well as the fault code must still be inserted as described above.

A.1.3 Instrumenting for Error Injection

Error injection with PROPANE requires the specification of four things:

1. Error types
2. Error triggers
3. Error targets, and
4. Injection locations.

Error types describe the types of errors that are to be injected and are normally specified in the Experiment Descriptions (see Section A.5 of this appendix). However, PROPANE allows the user to implement his or her own error types, in case the built-in types should not be sufficient. How this is done is described in Section A.4.

Error triggers say when a specific injection is to take place and, if applicable, with which period. The error triggers are normally specified in the Experiment Descriptions (see Section A.5 of this appendix). However, as was the case for error types, PROPANE allows the user to implement his or her own error triggers, in case the built-in triggers should not be sufficient. How this is done is described in Section A.4.

The *error target* is the variable (or rather, the memory location) where the error is to be injected, and the injection location is where the injection itself is to be performed. The target is specified in the call to the API function of the PROPANE Library performing the actual error injection.

Injection locations are logical locations in the target system where error injections can take place. The link between these logical locations and the physical locations in the target source code are handled by the user instrumenting the target system. The same injection location may actually correspond to several physical locations in the target source code.

By dividing the setup of error injections into these four parts, the number of items that have to be specified can be reduced and the experimental freedom is increased. For example, if an error type is very common for many different error targets, it is sufficient to specify one error type and use this error type on all error targets.

Manual insertion of error locations

If error injection is used, the configuration source file must contain the following data items:

```
/* This variable contains the number of defined locations */
unsigned int propane_no_of_locations = n;

/* This array contains info about the defined locations */
PROPANELocationInfo propane_location_info[n] =
{
    /* Each location must be defined with a line as shown below. */
    { handle, name, filename, file pointer },
    ...
};
```

The variable `propane_no_of_locations` contains the number of defined injection locations in the error information array. It is very important that the array contains exactly the number of entries as specified in this variable.

The array `propane_location_info` contains information about the defined injection locations that will not change over time. For every injection location, there must be one entry containing the following information:

- handle
- name
- filename
- file pointer

The *handle* is an integer value and is used in the function call made from the target system to the PROPANE Library to perform the actual error injection. The *handle* must be equal to the index in the array, i.e., the first injection location must have handle 0, the second handle 1, and so on. The n^{th} injection location must have handle $(n - 1)$. It is a good idea to create a pre-processor constant (i.e., a `#define`

constant) for each handle and then use that constant instead of the actual numerical values in the function calls in the target system.

The *name* of an injection location may be any normal C-string not containing white space. This name must then be used in the PROPANE description files (described in Section A.5 of this appendix) when defining which injections are to be made during experiment execution.

The *filename* and *file pointer* will be created automatically during the setup process of the PROPANE Library and are not entered in the PROPANE configuration source file.

The following is an example of how event probes are defined in the PROPANE configuration source file and how they are subsequently inserted into the source code of the target system.

```

/* We have two error injection locations. This is entered in the
 * configuration source file.
 */
#define EL_CALC                (0)
#define EL_V_REG               (1)

unsigned int propane_no_of_locations = 2;

PROPANELocationInfo propane_location_info[2] =
{
    { EL_CALC,  "Location_CALC",  "", NULL},
    { EL_V_REG, "Location_V_REG", "", NULL},
};

```

When all the desired locations for error injections have been defined in the PROPANE configuration source file, the corresponding high-level software traps must be inserted into the target system, i.e., the logical locations defined in the PROPANE configuration source file have to be linked to physical locations in the target source code. The function in the API of the PROPANE Library that is used for error injections has the following prototype:

```

PROPANEReturnCode
propane_inject( PROPANELocationID handle,
               void * value,
               PROPANEValueType type );

```

The parameter *handle* is the handle of the injection location created in the PROPANE configuration source file. The pointer *value* is the address to the variable that is to be subjected to error injection, i.e., the error target. The parameter

type tells PROPANE the type of the error target. This is needed because targets of different types require different actions for error injection. For instance, injecting an offset of +5 into an integer is different from injecting the same offset into a floating-point value. The various types available in PROPANE are described in Section A.1.1 of this appendix. The function returns either PROPANE_OK if everything went fine, or PROPANE_FAILURE if something went wrong. Errors during this function call can only occur if the contents of the information and data structures for the injection locations have been corrupted.

If, in the experiment descriptions, several errors are defined in for the same injection location, all of these errors will be injected with the same call to the injection function. Also, note that the same injection location may be used for several error targets, and that each logical injection location may correspond to several physical location.

Here is an example of how the instrumentation for an error injection may look in the source code of the target system:

```

/* This is how the function calls are made in the
 * target system source code.
 */
int SetValue
PROPANE_ReturnCode injection_rc;
...
injection_rc = propane_inject( L_CALC,
                             &SetValue,
                             PROPANE_INT );

if( PROPANE_OK == probe_rc )
{
    /* Everything went fine! */
}
else
{
    /* Something went wrong! */
}

```

Note that the steps above are not sufficient for injecting errors, the combinations of error types, error triggers and locations (such a triplet is called an *injection*) must also be specified in the Experiment Description (see Section A.5 of this appendix).

Here, PSI can be used to create the PROPANE configuration source file. All that is needed is entries in the PROPANE Instrumentation Setup file for PSI. In this file, an error injection location is specified using an entry as follows:

```
// This is entry for an error injection location
>begin error location
    >handle <handle>
    >name <name>
>end
```

Note that the target data area of the error location is not specified here. The link between the error location and the actual data area in which errors are to be injected has to be made manually, i.e., using PSI to generate the PROPANE configuration source file still requires that the calls to the PL API be inserted manually.

Annotated insertion of error locations

By annotating the target system, PSI can be used to automatically insert calls to the PL API in the target source code and to generate the PROPANE configuration source file. An annotation for error locations looks as follows:

```
/* This is an annotation for an error location */
/*@P>elocation <handle> <target> <type> */
```

By having the annotations within ordinary C comments, a standard C compiler can be used for the original target source code and the annotations will not affect the software. If the annotated source is run through PSI instead, an instrumented version of the file and entries in the PROPANE configuration source file will be generated.

The annotations for error locations do not have a separate field for the name of the location (remember that the handle and the name of a location need not be the same). Instead, PSI will automatically set the name of the location same as the handle. For example, the annotations:

```
/*@P>elocation EL_CALC SetValue PROPANE_INT */
/*@P>elocation EL_V_REG IsValue PROPANE_INT */
```

would generate the same structures in the PROPANE configuration file as those shown for error locations in the previous section on manual insertion. Note that size information is not provided in the annotation, not even if type of the target is PROPANE_AREA. This is due to the fact that all information on the error that is injected at a particular location is provided in the experiment setup files, i.e., this is dynamic information.

If several annotations were to be made for the same error location only one entry will be made in the PROPANE configuration source file. However, each annotation will generate a call to the PL API at the same place as the annotation.

A.2 Fully Automated Instrumentation of Target Systems

The previous sections describe how to instrument the source code of a target system either completely manually or by using annotations, and how to use PSI to generate the PROPANE configuration source file. Fully automated instrumentation of a target system with regard to variable probes and error locations can also be done by PSI. All that is needed is a description of the modular composition of the system and for each module the I/O and internal characteristics (i.e., the input and output signals, and the internal static and temporary signal) and a list of source files that implement the module. Each module in the system is specified in the PROPANE Instrumentation Setup file as follows:

```
// This is entry for a module of the target system
>module
    >name <name>

    >file <file1.c>
    >file <file2.c>
    ...
    >file <filen.c>

    >input
        >name      <name>
        >symbol    <symbol name>
        >function  <funcation name>
        >type      <type>
        >pointer   (yes|no)
    >end

    >output
        ...
    >end

    >state
        ...
    >end

    >temporary
        ...
    >end

>end
```

The parameters are the following:

- `name` – This is the name of the module. It is only used in the setup and description files of PROPANE and does not require a corresponding symbol in the target source code.
- `file` – Each file containing source code that implements the module has to be specified with a `file` parameter.
- `input` – This subsection is used to specify an input signal of the module.
- `output` – This subsection is used to specify an output signal of the module.
- `state` – This subsection is used to specify a state signal (i.e., an internal static variable) of the module.
- `temporary` – This subsection is used to specify a temporary signal (i.e., an internal non-static variable) of the module.

The signals of the module (inputs, outputs, states and temporaries) each are defined using the following parameters:

- `name` – This is the name of the signal.
- `symbol` – This is actual symbol in the target source code that implements the signal.
- `function` – This is the function (i.e., scope) in which the symbol can be found. If no function is specified, PSI will look for the symbol in all scopes (including the global scope).
- `type` – This is the type of the signal. This can be one of the standard types used by PROPANE.
- `pointer` – This tells PSI whether the symbol is a pointer to a data area or a data area itself.

For each signal specified in in the module, PSI will add variable probes for logging and error locations for injecting errors. The name of a signal will be the basis of both the handles and the names of these probes and error locations. The `symbol` and the `type` will be used to generate the necessary data structures in the PROPANE configuration source file and as arguments of the inserted variable probes and error locations (i.e., the inserted calls to the PL API). If a signals is set to be a `pointer`, PSI will use the specified symbol directly instead of adding a `&` to get the address of a symbol when using it in the API calls.

A.3 Interfacing with Environment Simulators

When executing target systems in a possibly artificial environment, an environment simulator has to be provided that can provide stimuli to and can react on the output produced by the target system. PROPANE can be used to control initialization and shut-down the environment simulator if the user provides functions which the PROPANE Library calls during its setup process. These functions must have the following interface:

```
PROPANE_ReturnCode  
my_init_simulator( char * readout_directory,  
                  unsigned int experiment_id,  
                  char * sim_config_file );  
  
PROPANE_ReturnCode  
my_shutdown_simulator( void );
```

Three parameters are passed to the initialization function and are taken from the description files of the experiments (see Section A.5). The `readout_directory` parameter is a string containing the path to where readouts are to be stored. The parameter `experiment_id` is the identifier of the current experiment. The parameter `sim_config_file` is the name of the file with which the environment simulator is to be initialized.

The shut-down function does not take any parameters. This function is responsible for shutting down any activity in the simulator and performing any required clean-up (e.g., closing files that have been opened by the initialization function).

Both functions shall return either `PROPANE_OK` if they are successful in initializing/shutting down the environment simulator, or `PROPANE_FAILURE` if an error occurred. All necessary types and constants are provided in the header file `propane.h` included in the PROPANE tool suite.

There are no restrictions as to how the simulators are designed or how the simulator setup file is formatted. PROPANE only calls the provided functions with the parameters and expects a return code as described above. Thus, these functions may be implemented as an interface/wrapper level between PROPANE and the environment simulator, i.e., using the interface to PROPANE on one side (as the prototype of the function) and the interface of the environment simulator on the other side (in the body of the function).

In order to let PL know which functions to call, the following information has to be provided in the PROPANE configuration source file:

```

/* Pointers to the initialization function and the shut-down
 * function of the environment simulator.
 */
PROPANEEnvSimInitFptr      propane_init_env_sim_fptr =
                           my_env_sim_init_function;

PROPANEEnvSimShutdownFptr propane_shutdown_env_sim_fptr =
                           my_env_sim_shutdown_function;

```

These data structures can be automatically added to the PROPANE configuration source file by adding the following to the PROPANE Instrumentation Setup file used by PSI:

```

// This is the entry for the environment simulator
>environment simulator
  >name SIM_TEST
  >init my_env_sim_init_function
  >shutdown my_env_sim_shutdown_function
>end

```

The parameter name is used by PROPANE in the readout files to identify the channel created by the environment simulator. The remaining two parameters, `init` and `shutdown`, are used to specify the user provided functions.

A.4 Adding Error Types and Error Triggers

In addition to the built-in error types and error triggers provided by PROPANE, a user may specify user error types (called error injectors) and user error triggers. Both error injectors and error triggers are specified as functions. Error injectors have the following interface:

```

unsigned char
my_error_injector( PROPANELocationID location,
                  void * value,
                  PROPANEValueType type,
                  PROPANEErrorID error_id,
                  double par1,
                  double par2 );

```

The parameter `location` is the identifier of the injection location where this trigger is used in this particular call. The target for injection is pointed to by `value`, and the target type is specified in `type`. This type is one of the standard PROPANE

types. The parameter `error_id` is the identifier of the error that is to be injected. The parameters `par1` and `par2` are used to send special parameters to the error type. The interpretation of the parameters `par1` and `par2` is totally user-defined. The function shall return 1 if the injection was successful, and 0 otherwise.

Error triggers have the following interface:

```
unsigned char
my_error_trigger( unsigned int location,
                  void * value,
                  PROPANEValueType type );
```

The parameter `location` is the identifier of the injection location where this trigger is used in this particular call. The source value for the trigger is pointed to by `value`, and the type of the source is specified in `type`. This type is one of the standard PROPANE types. The function shall return 1 if an error is to be injected, i.e., if the trigger is released, and 0 otherwise.

User error injectors and user error triggers have to be specified in the PROPANE configuration source file. Error injectors are specified with the following data structures:

```
unsigned int propane_no_of_user_injectors = n;

PROPANEUserInjectorInfo propane_user_injector_info[n] =
{
    /* Each user injector must be defined with a line as below. */
    { handle, name, function },
    ...
};
```

The variable `propane_no_of_user_injectors` contains the number of defined injectors in the injector information array. It is very important that the array contains exactly the number of entries as specified in this variable.

The array `propane_user_injector_info` contains information about the defined injectors that will not change over time. For every user injector, there must be one entry containing the following information:

- handle
- name
- function

The *handle* is an integer value and is used to identify the injector internally in PROPANE and must be equal to the index in the array, i.e., the first injector must have handle 0, the second handle 1, and so on. The n^{th} injector must have handle $(n - 1)$. It is a good idea to create a pre-processor constant (i.e., a `#define` constant) for each handle and then use that constant instead of the actual numerical values in the function calls in the target system.

The *name* of a user injector location may be any normal C-string not containing white space. This name must then be used in the PROPANE description files (described in Section A.5 of this appendix) when defining which errors to inject during experiment execution.

Finally, the *function* is the name of the function implementing the user injector. This function is of course written by the user.

Error triggers are specified with the following data structures in the PROPANE configuration source file:

```
unsigned int propane_no_of_user_triggers = n;

PROPANEUserTriggerInfo propane_user_trigger_info[n] =
{
    /* Each user trigger must be defined with a line as below. */
    { handle, name, function },
    ...
};
```

As can be seen, the structures are very similar to the structures used for user injectors, and the definitions of the various parameters are actually the same. Thus, please see the explanation of the parameters in the user injector structures for details.

Instead of entering the data structures required for user injectors and user triggers manually into the PROPANE configuration source file, the following entries in the PROPANE Configuration Setup file can be used with PSI:

```
// This is an entry for a user injector
>user injector
    >handle <handle>
    >name <name>
    >function <function>
>end

// This is an entry for a user trigger
>user trigger
    >handle <handle>
```

```

>name <name>
>function <function>
>end

```

A.5 Description Files for PCD and PL

The PROPANE Campaign Driver and the PROPANE Library are set up using a number of description files as illustrated in Fig. A.2.

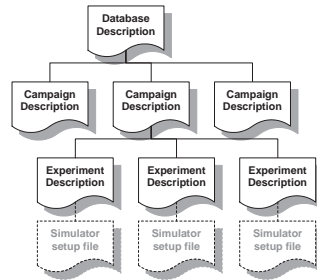


Figure A.2: Organization of description files for PROPANE setup.

This section will describe the format these description files. Note that as environment simulators are external to PROPANE, the format of these files is not specified by PROPANE, but rather the designer of the simulator or the user of PROPANE. The description files are all plain ASCII text files and contain a set of parameters, one parameter per line. If several parameters are specified on the same line in a file, only the first one on that line will be recognized. All commands start with a '>' in the leftmost position of the line followed by the parameter name. Lines not starting with a '>' are ignored by PCD and PL.

A.5.1 Database Descriptions

The top file in the hierarchy is the *Database Description*. This file is used for setting up the PROPANE Campaign Driver in order to execute experiments. The format of this file is as follows:

```

>database          <identifier>
>work directory    <directory>
>readout directory <directory>
>campaign          <campaign description file>

```

The parameters are the following:

- `database` – This parameter is a numerical value which is used for giving different databases individual identifications. The actual value has no practical meaning to PROPANE. It exists as a convenience to the user.
- `work directory` – This parameter tells PROPANE where all the remaining setup files can be found. This is also where all log files will be stored during execution of experiments.
- `readout directory` – This parameter tells PROPANE where to store all obtained readouts.
- `campaign` – The name specified here will be used by PROPANE as the file name of a *Campaign Description*.

The first three parameters (i.e, `database`, `work directory`, and `readout directory`) have to be specified before any campaigns can be listed. PROPANE will look in the specified work directory for the *Campaign Descriptions* referred to (with `campaign`) in the *Database Description*. During experiment execution, all log files will be placed in the work directory and all readout files will be placed in the readout directory. The file extension used for *Database Descriptions* is `.pdd`.

A.5.2 Campaign Descriptions

The second level in the hierarchy contains the *Campaign Descriptions*. These files are also read by the PROPANE Campaign Driver, as was the case with the *Database Descriptions*. The format of these files is as follows:

<code>>campaign</code>	<code><identifier></code>
<code>>executable</code>	<code><name></code>
<code>>execution width</code>	<code><number></code>
<code>>experiment</code>	<code><experiment description file></code>
<code>>use</code>	<code><filename></code>

The parameters are the following:

- `campaign` – This parameter is a numerical value which is used for giving different campaigns individual identifications. The actual value has no practical meaning to PROPANE. It exists as a convenience to the user.
- `executable` – This parameter tells PROPANE the name of the executable file to be used during execution of experiments.

- `execution width` – This parameter is the maximum number of experiment processes that will be run simultaneously. On computers where the target executable does not take 100 percent of the CPU, a width greater than 1 may shorten the time required to execute the entire campaign.
- `experiment` – The name specified after this parameter will be used by PROPANE as the file name of an *Experiment Description*.
- `use` – This parameter tells PROPANE to use the specified file in the campaign setup process. This parameter works similar to the `#include` pre-processor directive in C.

The first three parameters (i.e, `campaign`, `executable`, and `execution width`) have to be specified before any experiments can be listed. PROPANE will look in the directory specified as the work directory in the *Database Description* to find any setup files referred to in the *Campaign Description*. The *Experiment Descriptions* listed in the campaign will be used during the execution experiments in order to set up the PROPANE Library in the target executable. The file extension used for *Campaign Descriptions* is `.pcd`.

A.5.3 Experiment Descriptions

The third level (and final) in the hierarchy consists of the *Experiment Descriptions*. These files are read by the PROPANE Library (linked to the target executable) in order to set up a particular experiment. The file extension used for *Experiment Descriptions* is `.pxd` and the format of these files is as follows:

<code>>experiment</code>	<code><identifier></code>
<code>>simulator</code>	<code><filename></code>
<code>>probe</code>	<code>(<name> ALL)</code>
<code>>error</code>	<code><name> <type> <parameter 1> (<parameter 2>)</code>
<code>>error injection</code>	<code><location> <error name> <type> <parameter></code>
<code>>fault injection</code>	<code><fault name></code>
<code>>use</code>	<code><filename></code>

The parameters are the following:

- `experiment` – This parameter is a numerical value which is used for giving different experiments individual identifications. The actual value has no practical meaning to PROPANE. It exists as a convenience to the user.

- `simulator` – If an Environment Simulator (or at least an interfacing layer) has been linked to the target executable, this parameter can be used to tell PROPANE the name of the setup file to be used in the initialization of the Environment Simulator for this particular experiment.
- `probe` – This parameter is used to activate a probe that has been inserted into the source code of the target system. The user may choose to activate each probe individually (using multiple `probe` parameters, one for each probe to activate) or activate every available probe at by using `ALL` instead of a probe name. Although variable probes and event probes are two distinct entities in PROPANE, the activation of them looks identical.
- `error` – This parameter defines a specific error type that may be injected into the target system during the experiment. more information regarding this parameter is found below.
- `error injection` – This parameter is used for setting up the injection of an error during the experiment. More information regarding this parameter is found below.
- `fault injection` – This parameter is used for activating a fault that has been inserted into the source code of the target system.
- `use` – This parameter tells PROPANE to use the specified file in the experiment setup process. This parameter works similar to the `#include` pre-processor directive in C.

The `experiment` parameter must be specified before any other parameters are specified. The `simulator` parameter is optional (depending on whether an Environment Simulator is linked to the target executable or not). During initialization, the PROPANE Library will call an externally (i.e., user) provided function in order to set up the Environment Simulator. The file name specified after `simulator` is passed as a parameter to this function. If no probes are activated using `probe`, the readout files generated during experiment execution will not contain any data gathered by the probes in the target system. The `use` parameter may help in modularizing the *Experiment Descriptions* as one may choose to have, for instance, probe activations and error types in separate files. If multiple experiments are to use the same set of probes and error types, they can all just use the same files for probe activations and error type definitions. This increases the ability to overview (and change) the total setup.

The `error` parameter has several information fields. The `<name>` field is the alphanumeric name of the error type and follows the specifications of normal C-strings. The `<type>` can be one of the following:

- **E_SETMIN** – Set error target to the minimum value that the type of the target in question can hold. This error type does not take any parameters.
- **E_SETMAX** – Set error target to the maximum value that the type of the target in question can hold. This error type does not take any parameters.
- **E_SETVALUE** – Set error target to a constant value. The constant value is specified in `<parameter 1>`. `<parameter 2>` is not used.
- **E_FACTOR** – Multiply the current value of the error target with a factor. The factor is specified in `<parameter 1>`. and `<parameter 2>` is not used.
- **E_OFFSET** – Add an offset to the current value error target. The offset is specified in `<parameter 1>`. `<parameter 2>` is not used.
- **E_FACTOR_AND_OFFSET** – First multiply the current value of the error target with a factor, specified in `<parameter 1>`, and then add an offset, specified in `<parameter 2>`.
- **E_OFFSET_AND_FACTOR** – First add an offset to the current value of the error target and then multiply with a factor. In `<parameter 1>` the offset is specified, and the factor is specified in `<parameter 2>`.
- **E_BITFLIP** – Flip bits in the bit-vector representation of the error target. The bits to flip are specified as a bit-mask in `<parameter 1>`. This error type does not use `<parameter 2>`.
- **E_BITSET** – Set bits in the bit-vector representation of the error target. The bits to set are specified as a bit-mask in `<parameter 1>`. This error type does not use `<parameter 2>`.
- **E_BITCLEAR** – Clear bits in the bit-vector representation of the error target. The bits to clear are specified as a bit-mask in `<parameter 1>`. This error type does not use `<parameter 2>`.
- **E_SETVALUE_A** – The equivalent of **E_SETVALUE** but for variables that are of type **PROPANE_AREA**. The value specified in `<parameter 1>` is the offset from the start of the area, and `<parameter 2>` is the value to which the memory location shall be set. The start of the area is specified in the call to the injection function.
- **E_BITFLIP_A** – The equivalent of **E_BITFLIP** but for variables that are of type **PROPANE_AREA**. The value specified in `<parameter 1>` is the offset from the start of the area, and `<parameter 2>` contains a bit-mask indicating which bits to flip. The start of the area is specified in the call to the injection function.
- **E_BITSET_A** – The equivalent of **E_BITSET** but for variables that are of type **PROPANE_AREA**. The value specified in `<parameter 1>` is the offset

from the start of the area, and `<parameter 2>` contains a bit-mask indicating which bits to set. The start of the area is specified in the call to the injection function.

- `E_BITCLEAR_A` – The equivalent of `E_BITCLEAR` but for variables that are of type `PROPANE_AREA`. The value specified in `<parameter 1>` is the offset from the start of the area, and `<parameter 2>` contains a bit-mask indicating which bits to clear. The start of the area is specified in the call to the injection function.
- `E_USER` – Instead of using one of the built-in error injectors, the user may implement his or her own injectors. These injectors are implemented as functions and the name of the function is specified in `<parameter 1>`. This error type does not use `<parameter 2>`.

The parameter error injection specifies an actual injection to be performed during the execution of an experiment. An injection is specified with a number of information fields. The `<location>` specifies where the error is to be injected, i.e., in which of the predefined injection locations in the target system this particular injection is to be made. The `<error name>` is the name of an error type specified as described above. The `<type>` can be one of the following:

- `I_ALWAYS` – The error is injected every time the specified location is reached. The information field `<parameter>` is not used.
- `I_ONCE_TIME` – The error will be injected once at the specified location when the `PROPANE` timer reaches the value specified in `<parameter>`.
- `I_ONCE_CYCLE` – The error will be injected once at the specified location when that location has been reached a certain number of times. This number is specified in `<parameter>`.
- `I_PERIOD_TIME` – The error will be injected periodically at the specified location with a period specified in `<parameter>`. The period is counted based on the `PROPANE` timer. The first injection will be made the first time the `PROPANE` timer reaches the specified period value.
- `I_PERIOD_CYCLE` – The error will be injected periodically at the specified location with a period specified in `<parameter>`. The period is counted as the number of times the location is reached. The first injection will be made when the location has been reached as many times as specified in the period value.
- `I_PROBABILITY` – The error will be injected at the specified location with the probability specified in `<parameter>`. The `PROPANE` Library uses

the built in random number generator of the C Standard Library (specifically, PROPANE uses the functions `srand()` and `rand()`) to calculate a probability value for comparison with the specified probability.

- `I_USER` – Instead of using one of the built-in error triggers, the user may implement his or her own triggers. These triggers are implemented as functions and the name of the function is specified in `<parameter>`.

A.6 Analysis Scripts for PDE

The PROPANE Data Extractor is used for analyzing and extracting data in experiment readouts. As it can perform a number of different tasks, it requires a description file for analysis. One such description file per experiment database to be analyzed is required. The file extension used for *Analysis Scripts* is `.pas` and the format of the file is as follows:

<code>>analysis directory</code>	<code><directory></code>
<code>>database readouts</code>	<code><filename></code>
<code>>golden run comparison</code>	<code>(yes no) [(NONE <error margins>)]</code>
<code>>propagation information</code>	<code>(yes no)</code>
<code>>campaign range</code>	<code><first> <last> <injection channel></code>
<code>>interest channel</code>	<code><channel name></code>
<code>>injection information</code>	<code>(yes no)</code>
<code>>event information</code>	<code>(yes no) (<start> start) (<end> end)</code>
<code>>channel logs</code>	<code>(yes no)</code>

The `analysis directory` tells PDE where the files generated by the analysis and extraction actions are to be placed. The `database readouts` is the name of the top file in the readout hierarchy (i.e., the `.pdr`-file). After these two parameters there are a number of parameters regarding the analysis actions.

The parameter `golden run comparison` tells PDE whether it shall perform *Golden Run Comparisons* or not (choose either `yes` or `no`). If this action is chosen, one may then choose to use *error margins* by providing the name of the file containing the error margin setup (the format of this file is described in Section A.6.1). If no error margins are to be used, `NONE` is specified instead.

Important: PDE always treats the first campaign listed in the database readout file as the Golden Run campaign. All other campaigns will be compared to this one. Also, all campaigns (Golden Run as well as Injection Run) have to contain the same number of experiments, and all experiments have to contain the same channels (otherwise, the GRC will fail).

The parameter `propagation information` is set to `yes` if propagation graphs are requested. In this action, PDE will gather propagation information from all experiments and generate propagations summaries and propagation graphs. In order to produce propagation information, at least one `campaign range` and at least one `interest channel` have to be specified.

A `campaign range` is specified with three sub-parameters. The `<first>` specifies the index of the campaign that is the first in the range. The Golden Run campaign (i.e., the first campaign referenced in the database readout file) has index 0. Thus, the first injection campaign (i.e., the second reference in the database readout file) will have index 1. The `<last>` specifies the index of the last campaign in the range. The `<injection channel>` is the name of the channel that is to be considered the original injection location. All propagations will be measured with this channel as starting point. PDE will generate propagation information for each range specified in the description file.

In addition to the campaign ranges, PDE also needs to know which channels to consider in the propagation analysis. For this, the parameter `interest channel` is used. Each channel that is to be considered is specified as an `interest channel`. Thus, the experiment readouts may contain a large number of channels, while a selection of these may be used in the propagation analysis.

If information regarding error injections (i.e., which errors that were injected, locations, and injection times) is requested, the the `injection information` parameter is set to `yes`. This will generate one file for each campaign in the database readout file (excluding the Golden Run campaign).

The parameter `event information` tells PDE whether readouts generated by event probes are to be extracted. If the parameter is set to `yes`, PDE will for each campaign generate one file containing event information for each experiment in that campaign. The sub-parameters tell PDE the range, in time, for which the event information is requested. The `<start>` parameter sets the timestamp at which extraction is started. Setting this timestamp to `start` will make PDE start at the very beginning. Similarly, the `<end>` parameter tells PDE where to stop looking for events. To have PDE go to the very end of the readouts, set `<end>` to `end`.

The parameter `channel logs` tells PDE whether to generate log files of individual channels. If this parameter is set to `yes`, then PDE will generate a file which could easily be imported into a spreadsheet tool (such as Microsoft ExcelTM).

Warning: if channel logs are to be created, PDE will generate one file for each channel and each experiment and each campaign (thus, if there are, for instance, 10 campaigns with 10 experiments each, and in each experiment there are 10 channels, PDE will generate $10 \cdot 10 \cdot 10 = 1000$ files).

A.6.1 Error Margins for Golden Run Comparisons

When performing Golden Run Comparison, PDE can be set up to use error margins for the individual variable probe channels (except for PROPANE_AREA channels). These error margins are specified in a separate error margin file. Each entry in that file corresponds to the error margin of a channel. The format used is the following:

```
>margin <channel> <margin type> <up> <down>
```

The <channel> is the name of the variable probe channel that this error margin is intended for. The <margin type> can be either ABSOLUTE or RELATIVE, and <up>/<down> are the limits used (the actual margin).

An absolute margin will set upper and lower boundaries on the absolute error between the golden run (GR) channel and the injection run (IR) channel. For example, if a channel has an absolute margin of 5 up and 10 down, and a golden run sample of that channel has the value 100, then the injection run sample of that channel will be considered correct as long as it is within the range $100 - 10$ and $100 + 5$, i.e. within 90 and 105. If the golden run sample were instead 200, the range would be between 190 and 205.

A relative margin will set upper and lower bounds on the relative error between the GR channel and the IR channel. For example, if we have a relative error margin for a channel with 0.05 upwards and 0.10 downwards, and a golden run sample of that channel has the value 100, then the injection run sample of that channel will be considered correct as long as it is within the range $100 \cdot (1.0 - 0.10)$ and $100 \cdot (1.0 + 0.05)$, i.e. within 90 and 105. If the golden run sample would instead be 200, the range would be between 180 and 210.

A.7 Setup Scripts for PSC

When setting up PROPANE, one may choose to create all description files manually as these are normal text files. However, as a large number of description files is needed this may be a time consuming (and, frankly, not too exciting) task. For this, we have the PROPANE Setup Creator which can generate description files and analysis scripts given a smaller set of setup information.

```
#name      <database name>
#wdir      <directory>
#rodir     <directory>
#exec      <filename>
```

```
#width      <number>
#prbcfg     <filename>
#errcfg     <filename>
#errlst     <error list file>
#loclst     <location list file>
#tclst      <testcase list file>
#mtdlst     <method list file>
```

The database name is used for naming the various description files generated by PSC. All files will have a name starting with this name and followed by, if needed, some running number and the corresponding file extension for that file type. For instance, if the name is set to `my_exp`, then the database description would get the name `my_exp.pdd`, the first campaign description would be called `my_exp_0000.pcd`, the first experiment in the first campaign would be called `my_exp_0000_0000.pxd`, and the analysis script for the PDE would be called `my_exp.pas`.

The `wdir` specifies the working directory, i.e., the directory where PCD (see parameter `working directory` for database descriptions) will check for description files when running experiments. The description files generated by PSC will not automatically be placed there, this has to be done by the user.

The `rodir` specifies where readout files generated during experiment execution should be placed by PCD and PL (see parameter `readout directory` for database descriptions).

The executable file to be used for the setup is specified with parameter `exec`. The value specified here will be used with the parameter `executable` in the campaign descriptions generated by PSC. The execution width, i.e., the maximum number of processes spawned by PCD at any one time, is specified with the `width` parameter which will be used with the `execution width` parameter in the generated campaign descriptions.

The parameters `prbcfg` and `errcfg` specify which files to use for probes and error definitions, respectively. The probe configuration file shall contain a list of those probes which are to be activated during the experiments and thus, shall only contain a list of probe parameters (see experiment description for a description of the probe parameter). Analogously, the error configuration file shall contain a list of those error definition to use in the injections of the experiment. This file will, therefore, only contain a list of error parameters (see experiment description for a description of the error parameter). These two file, the probe configuration file and the error configuration file will be included in the setup with the `use` parameter in the experiment descriptions.

The next four parameters—`errlst`, `loclst`, `tc.lst`, and `mtdlst`—are names of files containing lists of errors, locations, test cases and injection methods to use, respectively. The errors specified in `errlst` must be defined in the error configuration file specified earlier. The locations listed must be defined in the instrumentation of the target system. The test cases listed will be used as filenames with the `simulator` parameter of the generated experiment descriptions. The injection methods listed must be the methods described for experiment descriptions above. All necessary parameters for each method have to be provided.

Using these for lists, PSC will create a setup where each experiment is running the target executable for one test case, injecting one error in one locations using one method. That is, if one specifies 10 errors, 10 locations, 10 test-cases and 10 methods, PSC will create a setup containing $10 \cdot 10 \cdot 10 \cdot 10 = 10000$ individual experiments. These will be organized in $10 \cdot 10 \cdot 10 = 1000$ campaigns where in each campaign, the same error is injected in all experiments. In addition to the injection campaigns, PSC will also create one Golden Run campaign, i.e., one campaign where the target system is run with the specified test cases and no errors are injected. These runs will be used as reference runs by PDE. The algorithm used for creating description files is the following:

```
-- Create the database description file
create_database_description();

-- Create the campaign description for Golden Run
create_golden_campaign_description();

-- Create one Golden Run for each test case
for each t in TEST CASE
  create_golden_experiment_description( t );
end for

campaign_number := 0;

for each l in LOCATION

  for each e in ERROR

    for each m in METHOD

      -- Create one campaign for each combination of
      -- location, error and injection method.
      campaign_number := campaign_number + 1;
      create_campaign_description( campaign_number )
```

```
experiment_number := 0;

for each TEST CASE
  -- In each campaign, we have one experiment run
  -- for each test case.
  experiment_number := experiment_number + 1;
  create_experiment_description( experiment_number, l, e, m, t )
end for

end for

end for

end for
```

The filenames of the generate files will start with the name specified in the name parameter described above and then have numerical values shown the campaign number and experiment number. For instance, the Golden Run campaign will be named `my_exp_0000.pcd`, and the third experiment of the fourth injection campaign will be named `my_exp_0004_0003.pxd`.

A.8 PROPANE Architecture

This section describes the internal architecture of the parts of PROPANE that perform the actual experiments, i.e., the PROPANE Campaign Driver (PCD) and the PROPANE Library (PL). We will first describe PCD and then continue with PL.

A.8.1 The PROPANE Campaign Driver

The PROPANE Campaign Driver is the main desktop part of the PROPANE tool and consists of six objects (see Fig. A.3):

1. Menu Handler
2. Database Manager
3. Executor
4. Controller
5. Log Unit
6. Readout Unit

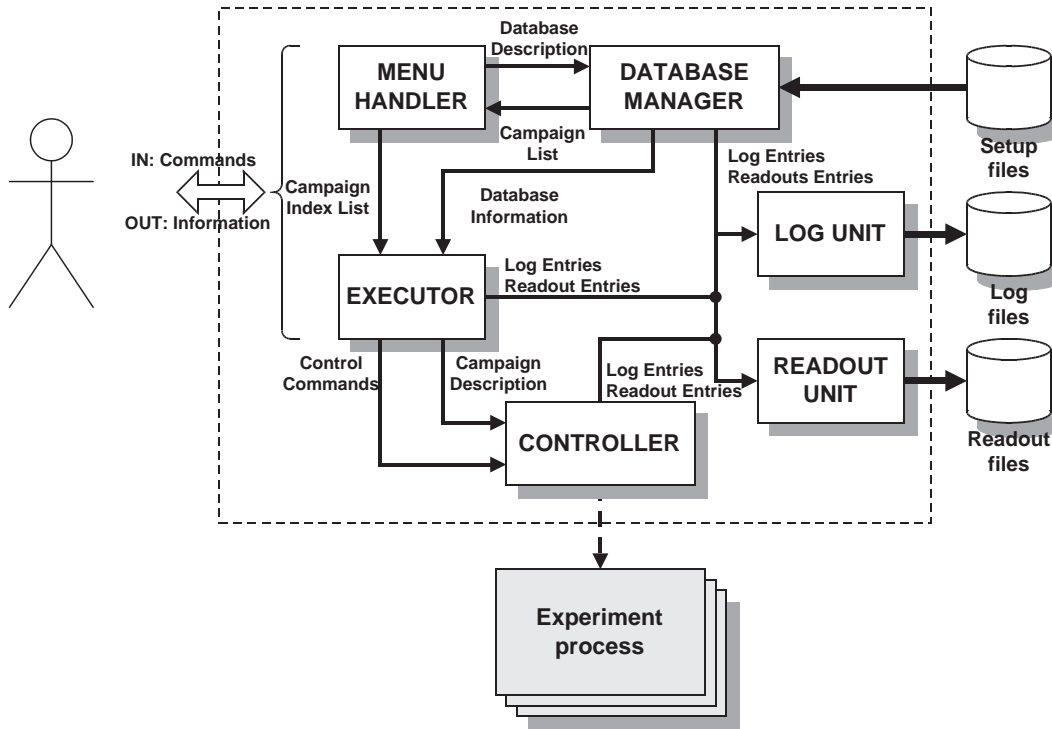


Figure A.3: The internal architecture of the PROPANE Campaign Driver.

In Fig. A.3, the objects belonging to the PROPANE Campaign Driver are found inside the dashed box. The other objects are external and not described here.

The Menu Handler is in charge of the menus presented to the user. From here, the user can load Database Descriptions, select campaigns, and initiate campaign execution. When a Database Description is to be loaded, the filename specified by the user is passed to the Database Manager, which reads the file and sets up the internal database. If the database is successfully set up, a list of the available campaigns is returned to the Menu Handler. From this campaign list, the user may choose to select a subset of campaigns to execute or to execute all campaigns.

When a set of campaigns has been selected for execution, a list containing information about these campaigns is passed to the Executor, which then starts the actual execution. During the execution it displays an information screen and allows the user to control the campaigns. For each campaign the Executor sets up the Controller and starts a separate thread for the Controller.

The Controller reads information about the campaigns from the Database Manager and uses this information when executing the experiments. For each Experiment Description in a campaign, the Controller will spawn a new process in which the target system executable file is executed. The Controller passes the Experiment

Description and the Readout Directory on the PROPANE Library, which is a part of the executable. Executing each experiment in its own process guarantees that the target system is reset for each experiment so that the starting conditions are the same for all experiments. Several processes may be started simultaneously, depending on the execution width specified in the description files.

During the execution of campaigns, the user may choose to send control commands via the Executor to the Controller in order to manipulate the execution of campaigns. The user may choose to pause and continue execution, or to skip the current campaign or abort all campaigns.

The Database Manager, the Executor and the Controller all use two support units: the Log Unit and the Readout Unit. The Log Unit handles the database log files and campaign log files, and the Readout Unit handles the database readout file and campaign readout files. The other units send entries to the Log Unit and the Readout Unit, which are then stored in the log files and the readout files correspondingly.

A.8.2 The PROPANE Library

The PROPANE Library is a function library enabling the PROPANE Campaign Driver to communicate with the experiment processes. It also contains everything necessary for the user to instrument a target system for variable and event logging, fault and error injection, and environment simulator control. The library is to be linked together with the target system and is mainly a passive component. The experiment executable may be executed manually outside of the control of the Campaign Driver in which case it asks on the console for the information it would otherwise receive from the Campaign Driver.

The PROPANE Library consists of 5 objects (see Fig. A.4):

1. Experiment Handler
2. Probe Manager
3. Injector
4. Log Unit
5. Readout Unit

In Fig. A.4, the objects belonging to the PROPANE Library are found inside the dashed box. The other objects are external and not described here.

The Experiment Handler is the main interface unit. It receives information from the Campaign Driver on which experiment to run and where to put the generated

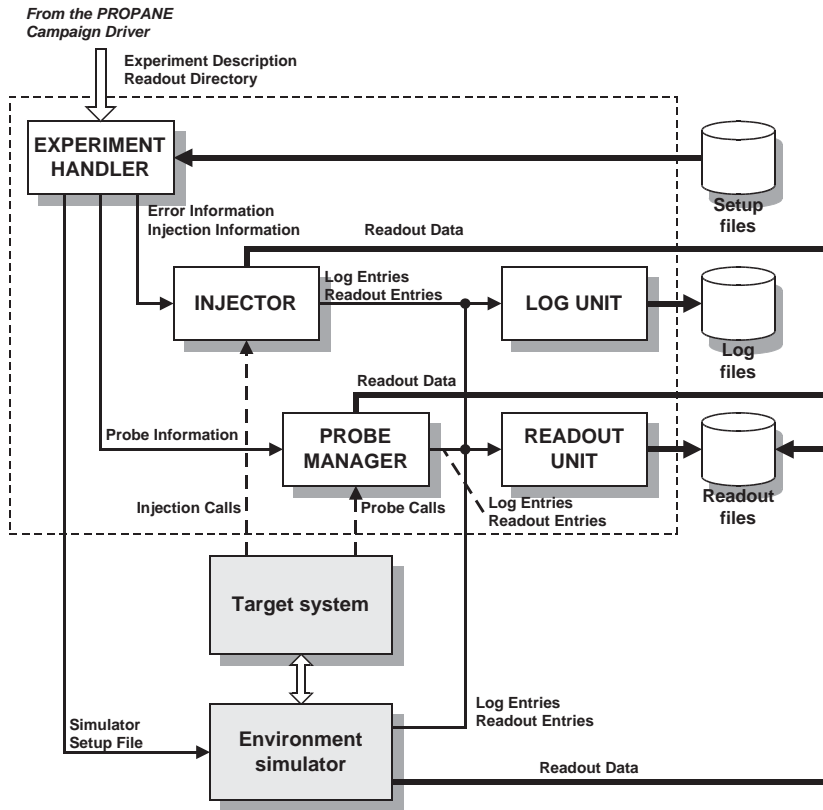


Figure A.4: The internal architecture of the PROPANE Library.

readouts. The Experiment Handler reads the specified Experiment Description and extracts the information needed for the experiment. Information regarding faults and errors is passed to the Injector, information regarding activated probes is passed to the Probe Handler, and the name of the simulator setup-file is passed to the external environment simulator. The Experiment Handler also initiates the Log Unit and the Readout Unit so that the experiment log file and experiment readout file is generated. Note that the experiment readout file is not the file in which the actual readout data is stored. This data is stored in a number of files, one for each readout collection point (i.e., variable probe, event probe, decision point, or injection location).

The Injector receives fault and error information from the Experiment Handler and uses this information to set up the injections that are to be performed during the experiment. Once it is activated, it will wait for the target system to call either the fault activation check routine or the error injection routine. When the fault activation check routine is called, it will decide which path the execution shall take, based on fault activation information in the setup of the experiment. When the error injection routine is called the errors specified for the location from which the routine is called

will be injected. Whenever an injection is performed, an entry with readout data in the readout file for the fault or the error location will be made.

The Log Unit and the Readout Unit are support units and work in much the same way as their equivalents in the PROPANE Campaign Driver do, i.e., they handle the experiment log file and the experiment readout file respectively. These two units are used by the internal PROPANE units but can also be used by the external environment simulator if it is programmed to do so.

(Gratuitous free space. Write notes, doodle, spill coffee...do whatever you want!)