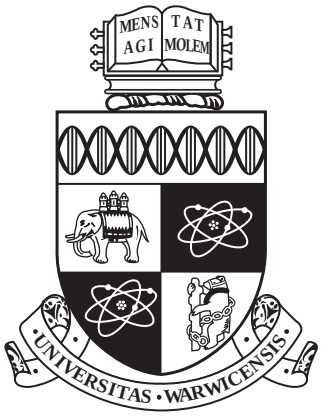

Towards the Design of Efficient Error Detection Mechanisms

by

Matthew Leeke

A thesis submitted to The University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

The University of Warwick

October 2011

Abstract

The pervasive nature of modern computer systems has led to an increase in our reliance on such systems to provide correct and timely services. Moreover, as the functionality of computer systems is being increasingly defined in software, it is imperative that software be dependable. It has previously been shown that a fault intolerant software system can be made fault tolerant through the design and deployment of software mechanisms implementing abstract artefacts known as error detection mechanisms (EDMs) and error recovery mechanisms (ERMs), hence the design of these components is central to the design of dependable software systems. The EDM design problem, which relates to the construction of a boolean predicate over a set of program variables, is inherently difficult, with current approaches relying on system specifications and the experience of software engineers. As this process necessarily entails the identification and incorporation of program variables by an error detection predicate, this thesis seeks to address the EDM design problem from a novel variable-centric perspective, with the research presented supporting the thesis that, where it exists under the assumed system model, an efficient EDM consists of a set of critical variables. In particular, this research proposes (i) a metric suite that can be used to generate a relative ranking of the program variables in a software

with respect to their criticality, (ii) a systematic approach for the generation of highly-efficient error detection predicates for EDMs, and (iii) an approach for dependability enhancement based on the protection of critical variables using software wrappers that implement error detection and correction predicates that are known to be efficient. This research substantiates the thesis that an efficient EDM contains a set of critical variables on the basis that (i) the proposed metric suite is able, through application of an appropriate threshold, to identify critical variables, (ii) efficient EDMs can be constructed based only on the critical variables identified by the metric suite, and (iii) the criticality of the identified variables can be shown to extend across a software module such that an efficient EDM designed for that software module should seek to determine the correctness of the identified variables.

Dedicated to my parents, Leslie and Jennifer.

Declarations

This thesis is presented in accordance with the regulations for the degree of Doctor of Philosophy. It has been composed by the author and has not been submitted in any previous application for any degree. The work described in this thesis has been undertaken by the author except where otherwise stated.

The work presented in this thesis is based on the following publications:

- A. Jhumka, M. Leeke. Early Identification of Locations for Dependability Components in Dependable Software. In Proceedings of the 22nd IEEE International Symposium on Software Reliability Engineering (ISSRE'11), November 29th-December 2nd, 2011, Hiroshima, Japan [77]
- M. Leeke, A. Jhumka. An Automated Wrapper-Based Approach to the Design of Dependable Software. In Proceedings of the 4th International Conference on Dependability (DEPEND'11), August 21-27th, 2011, Nice, France [101]
- M. Leeke, S. Arif, A. Jhumka and S. S. Anand. A Methodology for the Generation of Efficient Error Detection Mechanisms, In Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'11), 27-30th June, 2011, Hong Kong, China [96]

- M. Leeke, A. Jhumka. Towards Understanding the Importance of Variables in Dependable Software. In Proceedings of the 8th European Dependable Computing Conference (EDCC'10), 28-30th April, 2010, Valencia, Spain [100]

- A. Jhumka, M. Leeke. Issues on the Design of Efficient Fail-safe Fault Tolerance. In Proceedings of the 20th IEEE International Symposium on Software Reliability Engineering (ISSRE'09), November 16-19th, 2009, Bengaluru-Mysuru, India [76]

- M. Leeke, A. Jhumka. Evaluating the Use of Reference Run Models in Fault Injection Analysis. In Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'09), 16-18th November, 2009, Shanghai, China [98]

- M. Leeke, A. Jhumka. Beyond The Golden Run: Evaluating the Use of Reference Run Models in Fault Injection Analysis. In Proceedings of the 25th UK Performance Engineering Workshop (UKPEW'09), 6-7th July, 2009, Leeds, UK [97]

In addition, research conducted during the period of registration has also led the the following publications:

- A.P. Chester, M. Leeke, M. Al-Ghamdi, A. Jhumka, S.A. Jarvis. A Framework for Data Center Scale Dynamic Resource Allocation Algorithms. In Proceedings of the 10th IEEE International Conference on Scalable Computing and Communications (SCALCOM'11), August 31st-September 2nd, 2011, Pafos, Cyprus [31]

- A.P. Chester, M. Leeke, M. Al-Ghamdi, S.A. Jarvis, A. Jhumka. A Modular Failure-Aware Resource Allocation Architecture for Cloud Computing. In Proceedings of the 27th UK Performance Engineering Workshop (UKPEW'11), July 9th, 2011, Bradford, United Kingdom [30]

- A. Jhumka, M. Leeke, S. Shrestha. On the Use of Fake Sources for Source Location Privacy: Trade Offs Between Energy and Privacy. The Computer Journal, 54 (6), June 2011 [78]

- S.D. Hammond, S.A. Jarvis and M. Leeke. (Editors). Proceedings of the 26th UK Performance Engineering Workshop (UKPEW'10), July, 2010, Coventry, UK. ISBN: 978-0-9559703-2-0 [69]

- S. Shrestha, M. Leeke, A. Jhumka. On the Tradeoff Between Privacy and Privacy and Power in Wireless Sensor Networks. In Proceedings of the 26th UK Performance engineering Workshop 2010 (UKPEW'10), 9-10th July, 2010, Coventry, UK [99]

Sponsorship and Grants

The research presented in this thesis was made possible by the support of the following benefactors and sources:

- The Department of Computer Science, University of Warwick, Coventry, United Kingdom - Departmental Teaching Award (2008-2011)
- IEEE Computer Science Technical Committee on Dependable Computing and Fault Tolerance, IFIP WG 10.4 on Dependable Computing and Fault Tolerance - IEEE/IFIP Travel Grant for 41st IEEE/IFIP International Conference on Dependable Systems and Networks (2011)

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Dependability has been a concern for mankind throughout history. Be it the availability of food, the reliability of building materials or the security of new settlements, the need for humans to justifiably place trust in some set of services is inherent to the functioning of society. Indeed, the notion of dependability has long been used to capture the properties of services that individuals use on a daily basis, such as the health and transport services. In the context of computer systems, the notion of dependability has, for many decades, been synonymous with specific application domains that have stringent requirements with respect to the functioning of computer system components, including the avionics, automotive, defence and telecommunications industries. However, the pervasive nature of modern computing has led to an increase in the reliance of individuals and organisations on computer systems outside these traditional application domains, thus making computer system dependability a significant issue for systems engineers working across all industries. Moreover, the cost and inflexibility of bespoke hardware development, combined with the flexibility and

availability of general purpose computing platforms, has led to the functionality of computer systems being increasingly defined in software, thus making the issue of more specific issues of software dependability paramount in the design and development of modern computer systems [16].

The earliest electronic computer systems made use of inherently unreliable components, such as relays and vacuum tubes, to provide designated services. The nature of these components and the architecture of these early computer systems made solving even the most modest computational problems relatively challenging [144]. However, in the period since these early years of computer systems engineering, countless advances have been made with respect to the design and development of dependable computer systems, even in situations where systems must be constructed using unreliable components [3] [19] [143]. This is especially true in the context of dependable software systems, where practical techniques such as N-version programming (NVP) [14], checkpointing and rollback [85], error detection and correction codes [123], recovery blocks [131] and modular design [149] have facilitated the design and development of the software systems that underpin the pervasive technologies on which individuals and organisations rely. However, whilst these practical advances have provided software engineers with approaches that aid in the engineering of software systems of unprecedented scale and complexity, theoretical contributions made in this period have provided software engineers with equally important approaches for reasoning about dependable software systems. Indeed, several such contributions are central to the research presented in this thesis. In particular, it has been shown that aspects of dependability, such as fault tolerance, can only be achieved through some form of redundancy in the spatial and temporal domains, e.g., through the replication of software components or the software re-execution of software statements [53]. Further, the process of imparting fault tolerance to a software system can be carried out as post-pass, meaning that an undependable software system can be made dependable once a functionally correct software system has been implemented [71]. Finally, it has also been shown

that techniques and approaches for imparting fault tolerance, with respect to a software system, can be characterised by a combination of abstract artefacts known as error detection mechanisms (EDMs) and error recovery mechanisms (ERMs) [12]. When considered in combination, the main implication of these contributions can be taken to be that a fault intolerant software system can be made fault tolerant through the design and deployment of software mechanisms implementing EDMs and ERMs which incorporate some degree of spatial or temporal redundancy. It is this central premise that underpins the intention of the work presented throughout this thesis.

1.1 Motivation

As it has been demonstrated that techniques and approaches for the provision of fault tolerance in software systems can be reduced to the suitable application of EDMs and ERMs, it is natural to consider how these abstract artefacts can be realised [12]. In the case of an EDM, which is the focus of the research presented in this thesis, it has been shown that this artefact is characterised by (i) the error detection predicate that it implements and (ii) its location in a software system [70]. This characterisation of an EDM gives rise to two related problems; the error detection predicate design problem and the EDM location problem. The focus of the work in this thesis is on the error detection predicate design problem for EDMs, i.e., designing effective error detection predicates, where an error detection predicate is a boolean expression over a some set of programs variables. Specifically, the work presented in this thesis demonstrates that the error detection predicate design problem can be approximated and addressed through the identification and incorporation of a set of critical variables. That is, in order to be effective with respect to the detection of errors, the error detection predicate associated with an EDM should incorporate program variables from this critical set. Throughout this thesis, the terms “critical” and “critical variables” are used to described program variables that, due to them holding

erroneous values, are likely to result in a software system failure.

It is desirable to develop error detection predicates based on the correctness of as few program variables as possible, as this can allow the complexity and overheads of the associated EDMs to be reduced. However, incorporating few program variables typically reduces the efficiency of the associated EDMs. The vast state space associated with real-world software systems typically prohibits the consideration of all program variables in a software module, hence current approaches to the design of error detection predicates rely on the experience of software engineers and software system specifications. In effect these approaches mimic the design of error detection predicates based on critical variables, with some notion of variable criticality being understood with respect to the experience of software engineers or a system specification. As might be expected, such approaches succeed in reducing the state space that must be considered but lack the objectivity and repeatability that could be provided by a systematic approach to error detection predicate design. The identification of critical variables, with respect to the characterisation of criticality proposed in this thesis, serves to reduce the state space that must be considered in the design of error detection predicates for a software system, thus facilitating the design of effective and simple error detection predicates.

1.2 Thesis and Contributions

The contributions made in this dissertation with respect to the design of error detection predicates for EDMs are made based on the thesis that:

Where an efficient EDM exists under the defined system model, that efficient EDM consists of a set of critical variables.

Henceforth, accordance with the defined system model and the existence of an efficient EDM is presumed. In support of this thesis, the following specific contributions are made to the design of efficient error detection predicates for EDMs:

- A metric suite that generates a relative ranking of variables with respect to the notion of criticality applied throughout this thesis is proposed, as well as a fault injection approach for its evaluation. The metric suite is proposed with a view to facilitating the identification of critical variables. This identification is performed through the application of a threshold to the relatively ranking generated, thus allowing a cost-benefit analysis to be undertaken in the design of error detection predicates.

- A systematic approach for the design of efficient error detection predicates is proposed based on the application of data mining techniques to fault injection data sets. As well as providing an effective mechanism for the generation of efficient error detection predicates for real-world, infinite-state software systems, this approach is also used to demonstrate that error detection predicates generated using only critical variables achieve similar levels of efficiency as those generated using all program variables. This result is central to the thesis stated above, as it implies that the set of critical variables identified by the proposed metric suite can be exclusively used in the design of efficient EDMs.

- A methodology for the design of dependable software systems based on the replication of critical variables using software wrappers is proposed and applied to demonstrate that significant dependability enhancements can be achieved through the protection of a relatively small number of critical variables. This result serves to further substantiate the thesis that efficient EDMs consists of a set of critical variables. This is because the protection of relatively few program variables using error detection predicates that are known to be efficient, e.g., the combination of variable replication and majority voting, yields significant improvements in system failure rates.

1.3 Thesis Overview

This chapter has detailed the main motivations and contributions of the research to be presented in this thesis. The remainder of this thesis is organised as follows:

Chapter 2 provides an account of the concepts, principles and terminology in software dependability that are central to the appreciation of the work presented in this thesis. This account includes an overview of dependability attributes, impairments and means, as well as discussion of EDMs and ERMs, and the role they play in the design of fault tolerant software systems.

Chapter 3 describes the models under which the contributions made in this thesis have been developed, including details of the assumed model of software systems, the fault model under which software dependability was considered and the experimental setup applied in dependability evaluation.

Chapter 4 develops a metric suite that yields a relative ranking of program variables with respect a notion of criticality. This metric suite consists of the spatial impact, temporal impact and importance metrics, where importance is a function of spatial impact and temporal impact. In addition to its development, this metric suite is applied to a set of complex software systems and analysed for metric sensitivity to demonstrate the type of results that can be generated.

Chapter 5 presents an approach for the generation of efficient error detection predicates for EDMs based on the application of data mining techniques to data sets obtained during fault injection analysis. Following its development, the proposed approach is applied and the efficiency of the derived predicates measured. The results generated are then used as a basis for validating the capability of the proposed metric suite to identify critical variables.

Chapter 6 demonstrates the potential dependability enhancement that can be achieved through the protection of critical variables. In particular, a software system design methodology based on the replication of critical variables through

the deployment of software wrappers is applied in order to demonstrate that significant dependability enhancements can be achieved through the protection of a relatively small number of critical variables.

Chapter 7 discusses the various implications, applications and limitations of the stated thesis in the context of the research presented. This is done with an emphasis on how these implications, applications and limitations may influence the design and development of dependable software systems.

Chapter 8 concludes this thesis with a summary of the research contributions, a reiteration of the conclusions that can be drawn and a discussion of several issues relating to efficient EDM design that may be addressed by future research.

CHAPTER 2

Software Dependability

The pervasive nature of modern computer systems has led to an increase in our reliance on such systems to provide correct and timely services. Further, as the functionality of computer systems is being increasingly defined in software, it is imperative that software be dependable. In order to give an appropriate context and provide a consistent perspective for the work presented in this thesis, this chapter presents a detailed account of fundamental concepts and terminology in software dependability, with a focus on providing an in-depth discussion of topics and issues that will be developed in subsequent chapters. In particular, a widely accepted taxonomy of dependability is presented as a basis for the consideration of the attributes along which dependability may be considered, the potential impairments to dependability and the possible means for providing dependability. This discussion is then used as a basis for the presentation of several concepts and problems that are central to this thesis, including the fault-error-failure cycle, the erroneous state propagation problem and theoretical basis for achieving fault-tolerance based on EDMs and ERMs.

A set of statements is also provided in this chapter regarding the focus of the work presented in this thesis and how it relates to these concepts and problems.

2.1 Fundamentals of Dependability

The fundamental concepts of dependability adopted throughout this thesis are assumed directly from the comprehensive compilation of concepts developed by Laprie [93]. Hence, the dependability of a computer system is defined as "the trustworthiness of a computer system such that reliance can justifiably be placed in the service it delivers". Further, the dependability of a computer system may be considered with respect to a set of dependability attributes, may be compromised by a set of dependability impairments and may be imparted by a set of dependability means. This characterisation is shown in Figure 2.1, which depicts a dependability taxonomy that was originally synthesised by Laprie in 1985 [92], with work by the members of IFIP-WG 10.4 [67] resulting in its incorporation in the collection of concepts and terminology provided in [93]. Whilst the taxonomy of computer system dependability shown in Figure 2.1 was subsequently refined [16] [94], with further tree levels being added to account for more specific facets of dependability, this tree structure remains widely accepted as a characterisation of computer system dependability.

2.1.1 Dependability Attributes

The concept of computer system dependability is a multifaceted concern that incorporates many properties, each of which relates to a different viewpoint from which the quality of the services provided by a computer system may be evaluated. As shown in Figure 2.1 these properties are availability, reliability, safety, confidentiality, integrity and maintainability. A description and formal definition, where appropriate, of these dependability attributes is given below.

Availability: The availability attribute is concerned with the likelihood that

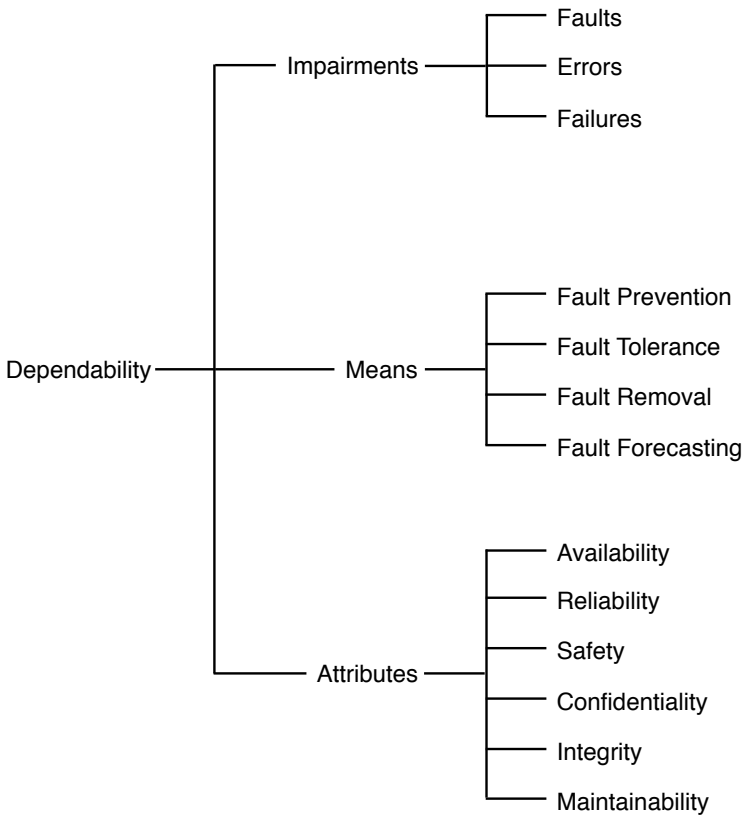

Figure 2.1: The taxonomy of computer system dependability

a service provided by a computer system is ready for use when invoked. More formally, availability is defined as a function of time representing the probability a service provided by a computer system is operating correctly and able to perform its designated function at a given time [82]. Intuitively, the higher the availability of a service provided by a computer system, the more likely that it is to be available when requested. The availability of a service provided by a computer system can be approximated by the ratio of the total time that the computer system has been capable of providing its designated services correctly to the total time that the system has been operational. In [82] it was shown that steady-state availability of a service provided by a computer system was given by Equation 2.1. The MTTF and MTTR terms in Equation 2.1 represent the mean time to failure and the mean time to repair for the service respectively.

$$A_{steady} = \frac{MTTF}{(MTTF + MTTR)} \tag{2.1}$$

Reliability: The reliability of a computer system is a measure of how likely a system is to provide its designated service for a specified period of time. That is, the reliability of a computer system is defined as the conditional probability that the system will provide correct service throughout the interval $[t_0, t]$, given that the system was providing correct service at time t_0. It is typically assumed that time t_0 is the current time, hence $R(t)$ is conventionally used to denote reliability. On the other hand, the unreliability of a computer system is defined as the conditional probability that the system will provide incorrect service throughout the interval $[t_0, t]$, given that the system was providing correct service at time t_0. Again, time t_0 is usually assumed to be the current time. The unreliability of a computer system is conventionally denoted by $Q(t) = 1 - R(t)$.

Safety: The safety attribute of dependability reflects the extent to which a system can operate without damaging or endangering its environment [145]. A safe computer system may deliver correct, incorrect or degraded services but it will never damage or endanger its environment or users. Computer systems where the safety attribute of dependability is the paramount concern, usually due to the potential for the loss of life or high-value resource, are commonly known as safety-critical systems [151].

Confidentiality: The confidentiality attribute is concerned with the non-disclosure of undue information to unauthorised entities [83]. The confidentiality attribute serves as a measure of the extent to which a computer system will allow those without sufficient privilege to obtain information that should be not be made available.

Integrity: The integrity attribute relates to the capacity of a computer system

to ensure the absence of improper system alterations, with regard to the withholding, modification and deletion of information [16]. The concept of integrity is typically interpreted such that "improper system alterations" relates only to information alterations performed by an unauthorised entity, though it also encompasses acts where an unauthorised party prevents modifications or causes information corruption. The composite of the availability, confidentiality and integrity attributes is usually considered to account for the computer system security aspect of computer system dependability.

Maintainability: Formally, maintainability is defined as a function of time representing the probability that a failed computer system will be repaired in t time or less [41]. The maintainability attribute is conventionally denoted by $M(t)$. Where a constant rate of repair, μ, can be assumed, the maintainability of a system can be estimated by Equation 2.2

$$M(t) = 1 - \exp^{-\mu t} \tag{2.2}$$

A computer system satisfying all dependability attributes for a given application domain can be said to be dependable in that application domain. However, the process of developing dependable computer systems is made challenging by the presence of impairments to dependability.

2.1.2 Dependability Impairments

A computer system is said to provide correct service when the service it provides complies with its functional specification. Conversely, a computer system is said to provide incorrect service, i.e., a system failure is said to have occurred, when the service it provides differs from its a functional specification. In general, such a failure occurs due to the presence of impairments to dependability. As shown in Figure 2.1, dependability impairments are faults, errors and failures. A description of each of these dependability impairments is given below.

Fault: In the design, development, deployment and operation of a dependable computer system, events can occur that can potentially reduce computer system dependability, i.e., the trustworthiness of services provided by the computer system. Such occurrences are termed faults. A fault is the hypothesised cause of an error, thus in a fault-free system there can be no errors. A fault may originate from within a system boundary or may find its origins in the surrounding environment. A fault is considered to be dormant until it is activated, at which points it results in an error. The period of time between fault activation and the manifestation of the associated error is know as the latency of the fault, whilst the notion of removal latency corresponds to the period of time between occurrence and removal.

Error: An error is the manifestation of a fault, i.e., a fault that has been activated. An error may be detected or undetected in a computer system. Following its activation, the erroneous state induced by an error may cause other errors to occur in a system. This process is known as error propagation, i.e., the propagation of erroneous state. The period of time between the occurrence of an error and the detection of that error, or the manifestation of the associated failure if the error is not detected, is known as the latency of the error.

Failure: A failure is the result of an error propagating beyond a system boundary, i.e., an erroneous state becoming visible to the environment in which a computer system operates. Given this definition, a failure can also be considered to be an observable deviation from an agreed system specification. Distinctions have often been made between different types of failure. For example, the failure characterisation in [38] classified failures as omission failures, timing failures, response failures or crash failures. However, whilst this categorisation is often relevant and useful in dependability analysis, it is not required for the appreciation of the research presented in this thesis.

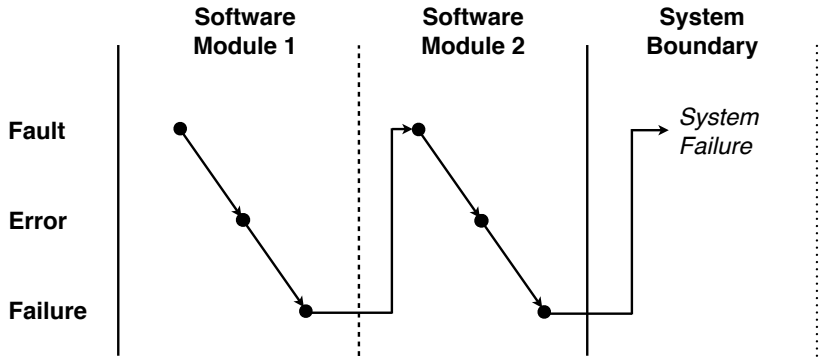

Figure 2.2: The fundamental cycle of computer system dependability

The Fundamental Chain: The fault-error-failure causality cycle, represented below with arrows to indicate causality, is known as the fundamental chain [92], as all dependability attributes are impacted by the interplay of dependability impairments, i.e, faults, errors and failures, at various levels of abstraction.

$$fault \rightarrow error \rightarrow failure$$

As shown in Figure 2.2, the fundamental chain is recursively defined, which means that a failure at one level of abstraction can represent a fault at another level of abstraction. Note that these levels of abstraction are considered to be software modules in Figure 2.2. This recursive nature leads to the definition extended chains of causality to represent the error propagation process, such as the causality chain shown below.

$$\ldots fault \rightarrow error \rightarrow failure \rightarrow [fault \rightarrow error \rightarrow failure] \rightarrow fault \ldots$$

A fundamental capability of any dependable system is to limit the extent of error propagation. Given the nature of the fundamental chain this capability can be characterised as the system being able to prevent chains of causality from growing to the point where a system failure occurs.

2.1.3 Dependability Means

There are four dependability means that may be employed in combination to provide dependability attributes in the presence of dependability impairments. As shown in Figure 2.1 and described below, these dependability means are fault prevention, fault tolerance, fault removal and fault forecasting.

Fault Prevention: The intention of fault prevention is to hinder and obstruct the occurrence, introduction and spread of faults. Established examples of fault prevention techniques include modular software design, software development methodologies and process quality assurance.

Fault Tolerance: Techniques that actively handle the occurrence of faults and errors, thus ensuring that a computer system is able to provide its designated service regardless of these dependability impairments, are termed fault tolerance techniques. In general, such techniques focus on the recognition of an erroneous state in a computer system and restoring a suitably correct state, or at least a safe system state, following the occurrence of an error.

Fault Removal: It is the intention of fault removal techniques to reduce the number, likelihood of activation and wider consequences of faults in a computer system. Typically fault removal is a three stage process, where these steps are validation, diagnosis and system correction. In particular, the validation stage seeks to determine whether a system adheres to a set of defined properties, the diagnosis stage identifies problems, i.e., faults, which prevent these properties from being fulfilled and the system correction stage seeks to modify the system to allow the defined properties to be fulfilled.

Fault Forecasting: Fault forecasting techniques are primarily concerned with estimating the number, likelihood of activation and wider consequences of faults in a computer system. The fault forecasting process typically involves the iden-

tification and classification and analysis of modes by which a computer system can fail, as well as an evaluation of dependability attributes using probabilistic and analytical approaches. Fault injection analysis is a commonly adopted approach when attempting to establish dependability measures and forecast fault proneness. Fault injection is a dependability validation approach whereby the response of a system to the artificial insertion of faults or errors is analysed so that insights can be gained regarding the dependability of the system [8]. Fault injection analysis is commonly used in the validation of dependable software systems [24] [25], hence many specialised fault injection tools and approaches have been developed for the validation of specific software system architectures and implementation paradigms [4] [152].

The contributions made in this thesis are primarily related to the areas of fault tolerance and fault forecasting. More specifically, the research presented in this thesis is concerned with demonstrating that fault tolerance be achieved through the design of EDMs based on a set of critical variables, whilst fault forecasting approaches, including software metrics and fault injection analysis, are used for dependability assessment and validation.

2.2 Achieving Fault Tolerance

Computer systems that are capable of providing their designated services in the presence of faults and errors are considered to be fault tolerant. The idea of deliberately designing computer systems that are able to tolerate the presence of impairments to dependability can be traced back to systems that were implemented entirely in hardware [134]. In this domain it was accepted that hardware components would fail, whether due to specific component properties or their operating environment, thus a degree of tolerance with respect to dependability impairments was necessitated. In general, the activities that a system undertakes to provide fault tolerance can be characterised by two processes; error

processing and fault treatment [7]. These system processes are described below.

Error Processing: The processing of errors involves three sub-processes; error detection, error diagnosis and error recovery. Error detection involves detecting the existence of an erroneous system state, ideally before that erroneous system state results in a system failure. The diagnosis of an error relates to assessing the impact that an erroneous state has had on a system, including the erroneous states that were present before error detection. Once detection and diagnosis have taken place, error recovery is performed in order to replace an erroneous system state with a suitably correct system state.

Fault Treatment: The process of fault treatment focuses on preventing faults that have been activated from being reactivated. In general, fault treatment is composed of fault diagnosis and fault passivation, where fault diagnosis involves the identification of the cause of errors with respect to a location and an underlying fault, and fault passivation relates to the actions taken to prevent a fault from being reactivated. Fault diagnosis is subtly different from error diagnosis, as the focus of fault diagnosis is on the origin and type of fault, whilst error diagnosis focuses on the impact of an error. Typically fault passivation may be encompassed by steps taken in error processing, more specifically during error recovery, though it is likely that some level of system reconfiguration will be required during fault passivation to ensure that the system state accounts for the any actions taken to make faults passive.

The research presented in this thesis is primarily related to the error processing aspect of fault tolerance. In particular, the contributions made are related to the design of software components that are sufficient for the detection of errors.

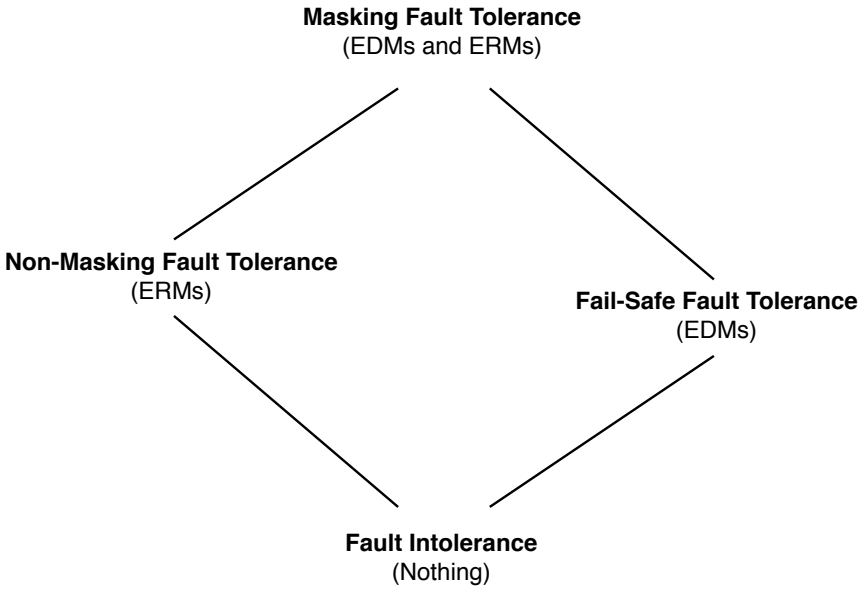

Figure 2.3: The partial ordering of fault tolerant system types established by the presence of EDMs and ERMs

2.3 Error Detection and Recovery

It has been shown that two dependability artefacts, known as EDMs and ERMs, are integral to the design of fault tolerant software systems [12]. In particular, it has been shown that EDMs are necessary and sufficient for the design of fail-safe fault tolerance, whilst ERMs are necessary and sufficient for the design of non-masking fault tolerance. That is, a software system equipped with only EDMs can satisfy the safety component of a system specification, whilst a software system equipped with only ERMs can satisfy the liveness component of a system specification. To provide masking fault tolerance both EDMs and ERMs are necessary and sufficient. Masking fault tolerance is strictly stronger than both fail-safe and non-masking fault tolerance, hence a partial order is established among types of fault tolerant software systems based on the presence of EDMs and ERMs. This partial order of fault tolerance types is depicted in Figure 2.3.

When a dependable software system is executing, an EDM in the software system will attempt to detect whether the state at a given time during execution

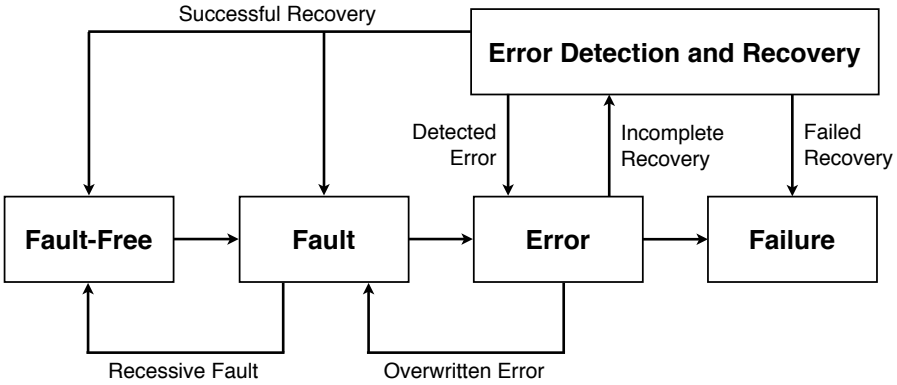

Figure 2.4: The states transition system of a fault tolerant system expressed in the context of EDMs, ERMs and the fundamental cycle

can threaten the proper functioning of the system. Such a state is generally referred to as an erroneous state, i.e., EDMs attempt to detect whether a state is erroneous. If a state is found to be erroneous, then an EDM can be said to have detected an error [93]. When an EDM detects an error, an ERM may attempt to restore a suitable system state, i.e., to recover from the erroneous state, so that the error is contained within a certain boundary and does not propagate throughout a software system. A failure to contain the propagation of errors within a software system is known to make recovery more difficult and increase the likelihood of a system failure [10]. The actions undertaken by EDMs and ERMs can be related to the fundamental cycle and the provision of fault tolerance as shown in Figure 2.4. This figure captures the general states that a fault tolerant system can transition between when equipped with EDMs and ERMs. It is interesting to note that the interplay between dependability impairments remains intact despite the presence of EDMs and ERMs.

The effectiveness of EDMs and ERMs is usually evaluated using measures such as coverage and latency [126]. The effectiveness of an EDM has been shown to depend on two factors. These factors are (i) the error detection predicate that it implements and (ii) its location in a software system [70]. This gives rise to two related problems, i.e., the error detection predicate design problem and the EDM location problem. The location of an EDM is the program statement it is

protecting, i.e., an EDM ensures that the software state in which a statement is executed will not result in an erroneous state or system failure. For some program statements this predicate, which is a boolean predicate over a set of program variables, is non-trivial [11]. The properties of this non-triviality have been shown to be accuracy and completeness [70]. In contrast to an EDM, an ERM will seek to restore a suitable safe state for a software system, i.e., an ERM focuses upon error recovery rather than error detection. When system state is characterised by the assignment of values to variables, this recovery process implies that a number of corrupted variables must be overwritten with appropriate values. The described observations regarding EDMs and ERMs imply that, in order to maintain correct and timely software execution, some set of variables must hold suitable values to make the software state safe for further execution. Clearly, if these variables are known, it is easier to develop error detection predicates and to determine appropriate locations for EDMs. Thus, knowledge of this set of variables is of significance when addressing the location and design problems associated with EDMs.

2.3.1 The EDM Location Problem

The EDM location problem focuses on the identification of software locations at which an EDM will be most effective or is most required, i.e., statements where a located EDM would be effective or statements that should be protected in order to achieve some level of dependability. Often this problem is interpreted as the error containment problem. Indeed, many approaches to the containment of errors have focused on experimentally evaluating the coverage and latency of EDMs [9] [59] [159], often using fault injection analysis [66]. Through these approaches it was established that EDMs exhibiting high coverage and low latency reduced the propagation of errors. These observations served to inform research relating to the location of EDMs [64] [71] [84]. Indeed, the approach developed in [64] extended work in [61] and [62] to provide a framework for the identification of vulnerabilities in software. This framework is based upon a soft-

ware measure, known as error permeability, which uses data-flow information to capture how likely errors are to propagate from a module input to a module output. In situations where it can be employed, i.e., where data flow information is available, this framework identifies modules with a high permeability, which then become candidate locations for EDMs. In contrast, approaches such as [160] and [161] developed error propagation analysis techniques that aimed to estimate the probabilities of given source code locations propagating data errors, when starting from an initially erroneous system state, with a view to informing the location of EDMs. This research was subsequently extend, largely in [111], to aid in the design of test cases that would reveal significant defects and vulnerabilities in software components, again informing the location of EDMs.

2.3.2 The EDM Design Problem

The EDM design problem concerns the derivation of a boolean predicate over program variables that can be used for the detection of erroneous system state, i.e., errors. In contrast to the substantial body of knowledge that relates to the location of EDMs, comparatively little is known that pertains to the design of efficient error detection predicates for EDMs, particularly for real-world, infinite state software systems. Note that the phrase "infinite-state software systems" is used to refer to software systems whose state space makes finite-state modelling impractical. Indeed, most real-world software systems will be of this kind, as given a software system with a reasonably small number of program variables, even where these variables have a limited set of associated states, the state space of the software system can be too large for a human to interpret and analyse as a set of actions and state transitions.

The efficiency properties of an error detection predicate can be characterised along two dimensions; accuracy and completeness. The accuracy property reflects the capability of an error detection predicate to avoid detecting non-erroneous states as errors, whilst completeness refers to the capability of an error detection predicate to detect erroneous states as errors [70]. In the con-

text of finite state systems, which are typically represented as state transition systems, the issue of error detection predicate design has been well considered. In particular, research in [70] and [91] developed polynomial-time algorithms for the automatic refinement of error predicates predicates, i.e., algorithms capable of improving the efficiency properties of an existing error detection predicate in polynomial-time. However, as finite-state systems can be viewed as an approximation to real-world, infinite state software systems, these approaches are not applicable for most real-world software systems. Indeed, the EDM design problem has received little attention in existing research literature.

The research presented in this thesis is primarily concerned with addressing the EDM design problem for real-world, infinite state software systems. More specifically, the work presented provides novel approaches for (i) the automated identification of program variables that must be captured by efficient error detection predicates, (ii) the generation of efficient error detection predicates, and dependability enhancement through the protection of identified program variables with software-wrappers implementing efficient EDMs. Crucially, when taken in contribution these contribution serve to support the thesis that an efficient EDM consists of a set of critical variables.

The concepts described in this chapter are by no means exhaustive in their coverage of topics in software dependability. However, the overview of topics presented provides a sufficient perspective on software dependability to allow the contributions of this thesis to be appreciated.

CHAPTER 3

Models

The models described in this chapter provide information about the context in which the contributions made by this thesis have been developed. In particular, the software system model assumed in the development of these contributions, and the fault model under which they were evaluated are detailed in this chapter. Information regarding the experimental setup that embodied these models is also provided in this chapter, including full details of all target systems, test cases, system failure specifications, software system instrumentation procedures, and dependability validation techniques used throughout this thesis.

3.1 System Model

A software system S is considered to be a tuple, consisting of a set of software modules, $M_1 \dots M_n$, and a set of connections. A software module M_k consists of an import interface I_k, an export interface E_k, a set of non-composite program variables V_k and a sequence of actions $A_{k1} \dots A_{ki}$. Each program variable in V_k has a specific domain of values. Each action in $A_{k1} \dots A_{ki}$ may read or write to a subset of V_k. Two software modules M_k and M_l are connected if the export interface of M_k is matched with the import interface of M_l, i.e., a connection exists if E_k is matched with I_l. Thus, a software system $S = (MOD, CON)$, where $MOD = \{M_1 \dots M_n\}$, and $CON = \{(M_k^a, M_l^a)\}$, where M_k exports to the import interface of M_l over connection a. We assume a software module ENV that exports inputs to the software system and imports output from the software system. Figure 3.1 shows an example of the assumed system model. Relating this example to the definitions given, it follows that if a software system $S = (MOD, CON)$, then $MOD = \{ENV, M_1, M_2, M_3, M_4\}$ and $CON = \{(M_1^b, M_2^b),\ (M_1^c, M_3^c),\ (M_1^d, M_3^d),\ (M_2^e, M_3^e),\ (M_2^f, M_4^f),\ (M_3^g, M_4^g),\ (ENV^a, M_1^a),\ (M_4^h, ENV^h)\}$.

A software system is assumed to be grey box, meaning that access to source code is permitted, but knowledge of functionality, implementation details and structure is not assumed. It is the variable-centric focus of the contributions made in this thesis that necessitates this white box access, whilst the adoption of a generic model of software systems is motivated by the desire to ensure that the contributions made are widely applicable.

3.2 Fault Model

A fault model has been shown to contain two parts; a local part, and a global part [165]. The local fault model, known as the impact model, states the type of faults likely to occur in the system, while the global model, known as the rely specification, states the extent to which the local fault model can occur. In

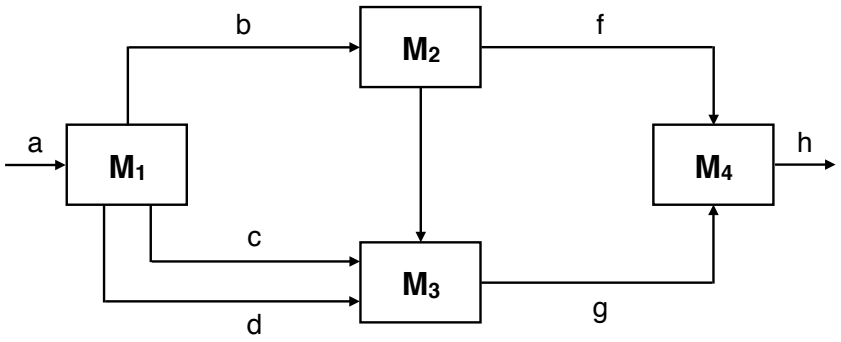

Figure 3.1: A software system represented under the adopted system model

general, the rely specification constrains the occurrence of the local model so that dependability can be imparted. For example, a rely specification will state that "at most f of n nodes can crash" or "faults can occur only finitely often".

In this thesis it is assumed that a software system has to tolerate a transient data value fault model [125]. Here, the local fault model is the transient data value failure, i.e., a program variable value is corrupted and this corruption may never occur again. The global fault model is that we assume that any program variable can be affected by transient faults. A transient fault model is generally used to model hardware faults in which bit flips occur in memory areas that cause instantaneous changes to values held in memory. The transient fault model has also been shown to mimic the presence of software bugs [33]. Thus, through the assumption of a transient data value fault model, consideration is effectively being give to random hardware failures in which random single bit flips may occur and software bugs whose effect is to cause program variables to hold erroneous data values. A transient data value fault model is often assumed during dependability analysis due to the fact that it can be used to mimic several more severe fault models [125]. It is this characteristic that makes the transient data value fault model a good candidate for a base fault model.

3.3 Target Systems

The target systems used throughout this thesis were selected based on their complex nature, widespread usage, modular structure and varying development profiles. An overview of each target system used in this thesis is provided below.

7-Zip Archiving Utility: The 7-Zip utility is a high-compression archiver which supports many file archiving and encryption formats [1]. The 7-Zip utility is widely-used, modular, written in C/C++ and has been designed, developed and maintained by a community of software engineers. Most source code and resources associated with the 7-Zip utility are freely available under the GNU Lesser General Public License.

FlightGear Flight Simulator: The FlightGear Flight Simulator project is an open source project that aims to develop an extensible yet sophisticated flight simulator to serve the needs of the academic and hobbyists communities [50]. The software is modular, contains over 220,000 lines of C/C++ and simulates a situation where dependability is of utmost importance. All source code and resources associated with the FlightGear Flight Simulator project are available under the GNU General Public License.

MP3 Gain: The MP3Gain analyser is an open-source volume normalisation suite for MP3 files [112]. The MP3Gain analyser is modular, written in C/C++ and has been predominantly developed by a single software engineer. All source code and resources associated with the MP3Gain analyser are available under the GNU General Public License.

3.4 Test Cases

Fault injection analysis forms a basis for the formulation and evaluation of the contributions made in this thesis. To perform fault injection analysis on a given

target system a set of test cases must first be identified. The test cases used in the analysis of all target systems in this thesis are described below.

7-Zip Archiving Utility: An archiving procedure was executed in all test cases. A set of 25 files of varying file formats were input to the procedure. Each of these files was then compressed to form an archive and then decompressed to recover the original content. The temporal impact of faults, an issue that will be discussed at length in Chapter 4, was measured with respect to the number of files processed. For example, if a fault were injected during the processing of file 15 and persisted until the end of a test case, then its temporal impact would be 10. To create a varied system load, the experiments associated with each instrumented program variable were repeated for 25 distinct test cases, where each test case involved a distinct set of 25 input files.

FlightGear Flight Simulator: A takeoff procedure was executed in all test cases. This procedure executed for 2700 iterations of the main simulation loop, where the first 500 iterations correspond to an initialisation period and the remaining 2200 iterations relate to pre-fault injection and post-fault injection periods. The temporal impact of injected faults was measured with respect to iterations of the main simulation loop. A control module was used to provide a consistent input vector at each iteration of the simulation. To create a varied and representative system load, the experiments associated with each instrumented program variable were repeated for 9 distinct test cases; 3 aircraft masses and 3 wind speeds uniformly distributed across 1300-2100lbs and 0-60kph respectively.

MP3 Gain: A volume-level normalisation procedure was executed in all test cases. The procedure took a set of 25 MP3 files of varying sizes as input and normalised the volume across each file. The temporal impact of injected faults was measured with respect to the number of files processed. To create a varied system load, the experiments for each instrumented program variable were run

for 3 distinct test cases, where each test case used a distinct set of input files.

3.5 System Failure Specifications

Conducting any form of software system dependability analysis will typically necessitate the definition of a system failure specification. In the case of fault injection analysis, such a specification is particularly important, as it captures the worst-case impact of an injected fault, i.e., a software system failure. The system failure specifications for all target systems are described below.

7-Zip Archiving Utility: A test case execution was considered a failure if the set of archive files and recovered content files were different from those generated during the corresponding golden run, i.e., a fault free test case execution.

FlightGear Flight Simulator: A failure specification was established using golden run observation and relevant aviation information. A failure in the execution of a test case was considered to fall into at least one of three categories; speed failure, distance failure and angle failure. A run was considered a speed failure if the aircraft failed to reach a safe takeoff speed after first passing through critical speed and velocity of rotation. A run was considered a distance failure if the takeoff distance exceeds that specified by the aircraft manufacturer, where the specified distance is increased by 10 meters for every additional 200lbs over the aircraft base-weight. A run was considered an angle failure if a Pitch Rate of 4.5 degrees is exceeded before the aircraft is clear of the runway or the aircraft stalls during climb out.

MP3 Gain: A test case execution was considered a failure if the set of output files, normalised by the MP3 Gain analyser, were different from those generated by the corresponding golden run, i.e., if these files differed from those produced under a fault free test case execution.

Table 3.1: Summary of fault injection experiment totals for 7-Zip

Module	**8-bit Injections**	**32-bit Injections**	**64-bit Injections**	**Total Injections**
7Z1 (ZDecode)	10000	5508000	1680000	7198000
7Z2 (ZInput)	24000	10608000	4144000	14776000
7Z3 (ZHandle)	2000	480000	48000	530000
	36000	16596000	5872000	22504000

Table 3.2: Summary of fault injection experiment totals for FlightGear

Module	**8-bit Injections**	**32-bit Injections**	**64-bit Injections**	**Total Injections**
FG1 (FMass)	864	38880	774144	813888
FG2 (FProp)	77760	743904	155520	977184
FG3 (FGear)	46656	12096	399168	457920
	125280	794880	1328832	2248992

3.6 System Instrumentation

Instrumented software modules in each target system were chosen randomly from all software modules used in the execution of the test cases defined in Section 3.4. Summaries of the fault injection experiments performed for the target modules in each target system are shown in Tables 3.1-3.3. All program variables in each target system were instrumented.

3.7 Fault Injection and Data Logging

The Propagation Analysis Environment (PROPANE) tool was used for all fault injection and system state logging experiments undertaken in this thesis [63]. A golden run was created for each test case, where a golden run is a reproducible fault-free run of the system for a given test case, capturing information about the state of the system during execution. In-line with the adopted fault model, bit flip faults were injected at each bit-position for all instrumented program variables. Each fault injected test case execution entailed a single bit flip in a program variable at one bit-position, i.e. no multiple fault injections were

Table 3.3: Summary of fault injection experiment totals for MP3 Gain

Module	**8-bit Injections**	**32-bit Injections**	**64-bit Injections**	**Total Injections**
MG1 (MLaunch)	86400	4454400	259200	4800000
MG2 (MGain)	9000	201600	7833600	8044200
MG3 (MAnalysis)	0	806400	504000	1310400
	95400	5462400	8596800	14154600

performed during a single test case execution. In the case of the FlightGear Flight Simulator each single bit flip experiment was performed at 3 distinct injection times uniformly distributed across the 2200 simulation loop iterations that follow system initialisation, i.e. 600, 1200 and 1800 control loop iterations after the initialisation period of 500 iterations. In that cases of 7-Zip and MP3 Gain, each single bit-flip experiment was performed at 25 distinct injection times uniformly distributed across the 25 time units associated with each test case. The state of all software modules used in the execution of all test cases was monitored and recorded during each fault injection experiment. The data logged during fault injection was compared with the corresponding golden run, with deviations being deemed erroneous subject to the adopted fault model. Any module containing an erroneous variable, i.e., a variable whose value deviates from its corresponding value during a golden run, is considered to be corrupted.

CHAPTER 4

Towards the Identification of Critical Variables

The identification of program variables that should be incorporated by an error detection predicate is central to the design of an effective EDM. To this end, this chapter proposes a metric suite that can be used to generate a relative ranking of the program variables in a software with respect to their criticality. The intention is then for a threshold to be subsequently applied to the relative ranking generated, in a cost-benefit analysis, to allow for the identification of program variables that should be incorporated by an error detection predicate. The proposed metric suite is composed of three metrics, namely spatial impact, temporal impact and importance, where the importance metric is a function of spatial impact and temporal impact. In order to demonstrate the type of results that the metric suite is capable of generating, the proposed approach is applied to three complex software systems and analysed for sensitivity with respect to parameterisation. Analysis of the results presented indicates that the relative ranking of program variables generated by the metric suite is consistent with the rationale underpinning its development and that this ranking is robust

with respect to parameterisation, indicating that the metrics proposed in this chapter are suitable for use in the identification of critical variables.

4.1 The Identification of Critical Variables

The identification of the program variables that should be captured by error detection predicates, i.e., critical variables, is fundamental to the thesis an EDM consists of a set of critical variables. It is the intention of the metric suite developed in this chapter to provide a systematic approach for the generation of a relative ranking among program variables that can be thresholded in order to identify critical variables and, hence, aid in the design of efficient error detection predicates for EDMs. Such a variable-centric focus is, in-part, motivated by research which has demonstrated that a relatively small number of variables can be effective in fault forecasting and fault tolerance. For example, while not directly concerned with the identification of critical variables, research in [65] demonstrated that the use of very few system variables could allow a predictive accuracy in excess of 80% to be achieved in call availability predication for telecommunication systems. Moreover, it was shown in [117] that adopting a variable-centric focus in the placement of EDMs can allow a detection coverage in excess of 80% to be achieved through the protection of an extremely modest number of program variables, e.g., 10 program variables in the cases of the studies presented.

To demonstrate how the metric suite proposed in this chapter differs from approaches already used in the design and development of dependable software systems, Sections 4.1.1-4.1.4 discuss current approaches to the identifications of critical software components and vulnerabilities, with a particular emphasis on how this identification can aid in the detection of errors and the notion of criticality employed by these approaches. Note that the phrase “software components” is used to capture approaches that are not focused on the identification of program variables, e.g., approaches focusing on vulnerable software modules

or code locations. Following this discussion of current approaches, the novelty and distinctive characteristics of the proposed metric suite are then discussed in Section 4.1.5. The discussion shown in Sections 4.1.1-4.1.4 deliberately avoids the consideration of approaches for the design and composition of error detection predicates, as this topic is addressed at length in Section 5.1 of Chapter 5. More specifically, in considering the identification of critical variables, Sections 4.1.1-4.1.4 classify techniques as being related to experimental evaluation, experience and heuristics, system specifications or static analysis.

4.1.1 Experimental Evaluation

Experimental approaches to the identification of critical software components and vulnerabilities typically involve the application of a dependability validation approach and an assessment of the results derived with respect to some notion of criticality. Such notions of criticality often incorporate aspects of measures such as coverage and latency, particularly when software system has been developed with EDMs [126]. Dependability validation techniques such as fault injection analysis have been shown to be a particularly effective means for estimating difficult to establish software measures, including coverage and latency, where EDMs already exist in a software system [61]. In [126] it was shown that if n_i faults are injected into a software system and n_d of these are detected by an EDM, an unbiased estimate of the detection coverage for that EDM, c_d, is given by Equation 4.1.

$$c_d = \frac{n_d}{n_i} \tag{4.1}$$

A caveat on this evaluation of detection coverage is that the faults or errors injected into a software system should be representative of the errors that could be experienced during its operation. Indeed, the issue of representativeness in fault injection analysis, and more generally in software testing, remains an open issue in research.

When evaluating an existing EDM, the latency associated with that EDM is typically interpreted as detection latency, which corresponds to the period of time between the occurrence of an error and that EDM detecting the existence of the error. As well as being used to estimate software measures that can be difficult to establish, fault injection analysis can also permit more analytical and software system specific analyses to be undertaken with respect to the identification of critical software components and vulnerabilities. For example, Khoshgoftaar *et al.* proposed a set of metrics to identify software modules that do not propagate errors. More specifically, it was the goal in [84] to determine whether data errors could propagate from a code location of interest, i.e., a potentially vulnerable location or component, to the system boundary when testing using inputs drawn from a representative operational profile. The intention here was to identify situations where a fault in a program variable at a location may not be detected during software testing, i.e., a vulnerability in a software component that was likely to go undetected. Further, Steininger and Scherrer demonstrated that fault injection techniques can be used to find optimal combinations of EDMs in hardware, though little attention was given specifically to the identification of what should be incorporated by the error detection predicates of these EDMs [148]. Instead, the focus of this work was on determining the most efficient combination of EDMs when a set of EDMs is already available. A set of measures were also introduced to assist in systematically forming and evaluating the efficiency of sets of EDMs, potentially offering various notions of criticality based on detection coverage and latency. Similar to this approach, the experimental techniques detailed in [161] and [164], which largely focused on locating executable assertions, i.e., specific instances of EDMs, proposed to use sensitivity analysis conducted during fault injection analysis or program mutation, in order to identify component vulnerabilities; a technique subsequently applied in [163]. However, the aims of this technique are not focused on the identification of what should be incorporated by error detection predicates, rather it is concerned with the rationale underlying the

identification of locations where EDMs should be located. In [120] and [121], Pattabiraman *et al.* developed a technique for the prevention of data errors through the derivation of fine-grained, i.e., variable-centric, EDMs. This was based on dynamic application traces and the use of six generic rule templates, where a single rule template corresponded to a different class of errors that could be detected. For example, a single template is used capture situations where a variable should have a constant value. As the EDMs derived in this work are based on the value of a single program variable at a specific location, rather than a single predicate capturing the values of multiple variables at a specific location, a multitude of effective EDM locations may need to be identified in order to achieve the coverage and efficiency. Further, if a class of error falls outside the set of rule templates employed, then any errors associated with that class may not be detected. Hence, the effectiveness of this approach depends on the capability of a software engineer to generate representative rule templates.

4.1.2 Experience and Heuristics

A variety of guidelines and heuristics have been proposed to characterise the criticality of software components, which has led to the definition of various notions of criticality, though often not with respect to programs variables or groups of program variables. For example, the high-level approaches described in [60] and [62] advocate a rigorous approach to code location analysis, combined with the use of an established methodology, such as failure mode and effects analysis (FMEA) or failure mode, effects and criticality analysis (FMECA) [20] [51] [56], to aid in identification of critical software components. Such advocation presumes the existence of a criticality heuristic with little consideration for what form that heuristic might take or what it might incorporate. Other research in software dependability measures has more explicitly set out a notion of software component criticality. For example, with a view to quantifying error propagation between interacting modules, the *inter-module influence* and *separation* measures, originally proposed in [153], were augmented and evaluated in [72]

through the introduction of software measures known as *error transmission probability* and *error transparency*. These related measures focused on characterising the error propagation between interacting modules at a high-level of abstraction, such that identified modules can be aggregated and located on distinct processors in order to confine errors to these processors. The problem of error propagation was investigated at the operating system level in [80], where the authors profiled the propagation of errors due to failures of device drivers. This lead to the development of a measure, known as *driver error diffusion*, that aims to capture the impact of a given device driver on operating system services. Although it is analogous to the metric suite proposed in this chapter, the fact that driver error diffusion focuses on quantifying the impact of device drivers means that the approach can be considered to operate at the level of software modules, rather than at the level of program variables.

4.1.3 System Specifications

A formal specification can be used in the identification of critical software components, not least because the constraints that are commonly defined in a formal specification often relate to aspects of functional correctness. In [132] and [150] it was proposed that a formal specification can be used to derive programatic tests which capture aspects of functional correctness. This is similar in intent to the identification of critical software components, as tests derived will be based on the components and values stored in some model-based representation of a software system. In [132], Richardson *et al.* concluded that specification-based checks, i.e., boolean predicates over program variables derived from a model-based specification, should be combined with self-checks, i.e., code-based assertions defined without a model-based specification, in order to facilitate the detection of a wider range of errors. The reason for this was based on the assertion that self-checks were more able to consider internal system state, whilst specification-based checks were removed from such implementation issues. Despite this suggestion it was also shown that it can be difficult to design effective

error detection predicates for programatic tests on the basis of a formal specification [102]. In support of the assertion that specification based EDMs are insufficient by themselves, work in [130] suggested that online checks should consider both intermediate data values and the correctness of control flow, which may or may not be discerned from a system specification. Interestingly, the process by which error detection predicates can be derived using [130] requires the identification of critical software components for EDM enhancement, a process that necessitates the definition of some notion of software component criticality.

4.1.4 Static Analysis

Static analysis refers to any type of analysis that can be performed without the execution of the software system being analysed. Static analysis is known to be complete, though the potential for false positives can make the approach less appealing in several application domains [29]. With respect to the identification of critical software components, static analysis has commonly been used in the identification of software components that have specific vulnerabilities. For example, research by Zheng *et al.* demonstrated that static analysis techniques were an economic approach to the detection of a range of software system implementation issues [173]. More specifically, it was shown that static analysis is a particularly effective mechanism for the identification of vulnerabilities relating to assignment and checking faults. Based on these findings, it was asserted that static analysis should be used to identify vulnerabilities early in the development of dependable software, thereby allowing subsequent development phases to focus on functional and algorithmic issues. Similar to this approach, Nagappan and Ball proposed an approach to the early prediction of defect density based on the number of vulnerabilities identified through static analysis [113]. More closely aligned with the identification of critical variables, research by Pattabiraman *et al.* adopts a notion of variable criticality based on the sensitivity of variables with respect to random data errors in a software system [118] [119]. This measure of variable criticality can be evaluated based

on the highest dynamic fan-out of variables, which can be computed using backward application slices. The premise underlying the use of highest fan-out for variable criticality is that program variables with a high fan-out, with respect to other program variables in a function in the case of [118] and [119], are more capable of propagating errors than program variables with a low fan-out, thus these high fan-out program variables can be considered to be more critical.

In some situations static analysis has been used to dramatically simplify or circumvent the problem of identifying critical software components. For example, several approaches have used static analysis to compute control-flow graphs and other abstract representations in order to base EDMs directly on the control-flow and structure of a software system, potentially obviating the need for the explicit identification of critical software components or vulnerabilities [6] [168]. Indeed, the approaches developed in [2] and [115] operate by ensuring that a statically generated control-flow graph is preserved during the execution of a software system. This is achieved by associating software checks with all program statements that result in branching or a transition between different program statement blocks. In contrast to this use of static analysis, the approach described in [130] aimed to detect control-flow errors through the application of a time trace technique that allowed expected behaviour to be specified by a software engineer and monitored by a generic control-flow automaton. However, as a study in [158] showed that around 33% of transient faults result in a control-flow error, the general applicability and effectiveness of the approaches discussed here is questionable, even if it is assumed that perfect coverage can be achieved by the associated EDMs.

4.1.5 Evaluation of Existing Predicate Design Approaches

Current approaches to the identification of critical software components and vulnerabilities in software systems generally suffer from one of two deficiencies. Firstly, many current approaches operate at a level of abstraction that prohibits the identification of program variables that must be incorporated by error detec-

tion predicates. For example, approaches which operate at the level of software modules generally do not provide a sufficient level of granularity to aid in the design of error detection predicates within a software module, meaning that the process of predicate implementation may be entered without knowledge of the program variables that must be captured. Secondly, many approaches fail to provide a systematic, objective and independently-repeatable method by which the prescribed analysis can be undertaken. For example, approaches which focus on providing guidelines, metrics and heuristics, even where these have been shown to be effective for some software systems, rely on the ability of the software engineer applying them. As a result of this it is possible for two independent engineers to apply the same approach and produce entirely different results, which means that the program variables identified may not be composed to form error detection predicates with the levels of accuracy and completeness that are required in dependable software systems. Despite these criticisms of existing techniques, the approach presented in [118] and [119] avoids both pitfalls, i.e., it is both variable-centric and can be applied systematically to provide objective, independently-repeatable results. However, despite being the closest existing work to the proposed metric suite, the notion of variable criticality is based on statically identified variables that may or may not have a high criticality when the associated software system is in operation. Put differently, the use of static analysis results in a potential for false positives, which may lead to the unnecessary identification and protection of variables that do not have a high criticality when the software system is in operation. To circumvent this issue a dynamic measure of criticality must to used in order to capture variable criticality with respect to the execution of a software system.

The metric suite developed in this chapter provides the first dynamic, variable-centric framework that can, through the application of a threshold, be used for the identification of program variables that should be incorporated by the error detection predicates of EDMs. The benefits of this metric suite, with respect to error detection predicate design, are that (i) error detection predicates can

be simplified through the identification of the set of variables that should be captured, (ii) the location of EDMs is informed based on the occurrences of the program variables identified, and (iii) a cost-benefit analysis can be undertaken when determining where the dependability enhancement efforts of software engineers should be focused. In contrast to many current approaches, which are open to some degree of interpretation or subjectivity, the proposed approach is systematic, objective and independently-repeatable. These characteristics mean that the proposed metric suite has the potential to provide both transparency and accountability in the design of dependable software systems.

The proposed metric suite is composed of three metrics; spatial impact, temporal impact and importance, where the importance metric is a function of spatial and temporal impact. The reason that the spatial and temporal impact of variables is considered, as opposed to any other such impact that could be defined, is the fact that it has been shown that fault tolerance can only be incorporated along these two dimensions [53]. The spatial impact metric captures the extent to which a system is corrupted when a given variable is corrupted. In contrast, the temporal impact metric captures the duration during which a system remains corrupted when a given variable is corrupted. To minimise the likelihood of a software failure, each of these aspects must be handled, i.e., the number of corrupted variables and the duration of the corruption must be taken into account. In the following sections the spatial impact, temporal impact and importance metrics are introduced alongside a fault injection approach for their evaluation based on the experimental setup described in Chapter 3.

4.2 The Spatial Impact Metric

The aim of the spatial impact metric is to quantify the extent of the affected area when a particular program variable is corrupted. The intention is then for the value of this metric for each program variable in a software module to contribute to the ranking of program variables in that software module with

respect to criticality. Intuitively, program variables that cause a higher extent of perturbation should contribute more to a chosen measure of criticality than program variables that cause a lower extent of perturbation.

4.2.1 A Definition for Spatial Impact

Given a software system whose functionality is logically distributed over a set of distinct software modules, the spatial impact of program variable v of module M in a run r, denoted by $\sigma^r_{v,M}$, is defined as the number of software modules that are corrupted in r as a result of a corruption in v. The spatial impact of a program variable v of software module M, denoted by $\sigma_{v,M}$, can then be defined as:

$$\sigma_{v,M} = max\{\sigma^r_{v,M}\}, \forall r \tag{4.2}$$

Thus, $\sigma_{v,M}$ captures the extent of the affected area whenever a program variable v in a module M is corrupted. The higher $\sigma_{v,M}$ is, the more difficult it may be to recover from the extent of corruption in the spatial domain. Observe that it is possible to have alternative definitions for spatial impact, e.g., by accounting for the average number of corrupted modules, rather than accounting for the worst-case situation by using the maximum extent of corruption in the way that is done in this thesis.

4.3 The Temporal Impact Metric

The aim of the temporal impact metric is to quantify for the duration of the perturbation when a particular variable is corrupted. The intention is then for the value of this metric for each program variable in a software module to contribute to the ranking of program variables in that software module with respect to criticality. Intuitively, program variables that cause a higher duration of perturbation should contribute more to a chosen measure of criticality that program variables that cause a lower duration of perturbation.

4.3.1 A Definition for Temporal Impact

Given a software system whose functionality is logically distributed over a set of distinct software modules. The temporal impact of program variable v of module M in a run r, denoted as $\tau^r_{v,M}$, is defined as the number of time units over which at least one software module of the software system remains corrupted in r following the corruption of v. The temporal impact of a program variable v of software module M, denoted as $\tau_{v,M}$, can then be defined as:

$$\tau_{v,M} = max\{\tau^r_{v,M}\}, \forall r \tag{4.3}$$

Thus, $\tau_{v,M}$ captures the period of time that the software system state remains affected whenever a program variable v in a software module M is corrupted. The higher $\tau_{v,M}$ is, the more difficult it may be to recover from the extent of corruption in the spatial domain. As with the spatial impact metic, it is possible to have different definitions for temporal impact, e.g., by accounting for the average duration of system corruption, rather than the worst-case situation by using the maximum duration of corruption in the way that is done in this thesis.

Having defined the spatial and temporal impact metrics it remains to provide a definition for the importance metric, which will form the basis for the rankings that can be used, following the application of a suitable threshold, to identify variables that should be incorporated by the error detection predicates of EDMs. Note that "important" and "critical" are not used interchangeably. The notion of criticality is considered to be more abstract than the notion of importance, which refers specifically to definition and application of the importance metric.

4.4 The Importance Metric

As argued earlier, both spatial and temporal impact must be accounted for when assessing the criticality of program variables. Hence, a general form for a soft-

ware metric that accounts for the described factors in expressing the criticality of a program variable v in a software module M, using arbitrary functions G, K and L, can be taken to be:

$$I_{v,M} = G[K(\sigma_{v,M})), L(\tau_{v,M}))] \tag{4.4}$$

Equation 4.4 captures the notion that a metric to measure importance should be a function of spatial impact and temporal impact, which is the role of G. Of course, as these impact metrics capture different aspect of error propagation it is reasonable to expect that they might be adjusted before combination, hence the presence of K and L in Equation 4.4. Further, the general form given in Equation 4.4 does not preclude the inclusion of measures other than spatial impact and temporal impact. Indeed, the inclusion of such information into an instantiated form of this general form can serve to enrich the importance metric. In particular, as spatial impact and temporal impact capture specific notions of error propagation it would be reasonable to include measures that capture other aspects of dependability. For example, the instantiation of Equation 4.4 proposed in this chapter includes system failure rate, derived through fault injection, as a supplementary measure, largely because metric provides information not afforded by spatial impact and temporal impact. In general, instantiations of the general form should seek to find a combination of spatial impact and temporal impact that is consistent with the notion of importance to be applied. For example, where the quantification of vulnerability expose is a concern it may make sense to combine spatial impact and temporal impact unevenly, and perhaps to include measures such as coverage, it should be noted that the variable centric focus of the general form, as imposed by $\sigma_{v,M}$ and $\tau_{v,M}$, should be maintained.

In Section 4.4.1 the general form shown in Equation 4.4 is instantiated. An approach, based on fault injection, for the evaluation of this instantiation is subsequently described in Section 4.4.2.

4.4.1 A Definition for the Importance Metric

Having set out a general form for an importance metric that incorporates both spatial and temporal impact, a concrete instantiation of this general form is now provided. This instantiation will be used throughout this thesis to identify variables that should be incorporated by the error detection predicates of EDMs.

Instantiated Importance Metric: Numerous instantiations of Equation 4.4 could be conceived. For example, an instantiation may incorporate a definition of erroneous state that is based on deviations from expected values, as was done in [61]. In this thesis a more general instantiation of the importance metric is adopted, whereby both erroneous states and system failure are taken in to account. This view, which is motivated by a focus on the detection of erroneous states that lead to system failure, means that the adopted instantiation of the importance metric should enable the generation of a relative ranking where program variables are ordered based on their association with system failure and the propagation of erroneous state. With this in mind, the importance of a variable v in a module M is defined as:

$$I_{v,M} = \frac{1}{(1-f)^n}\Big(\frac{1}{2}\Big(\frac{\sigma_{v,M}}{\sigma_{max}} + \frac{\tau_{v,M}}{\tau_{max}}\Big)\Big)^m \tag{4.5}$$

The importance metric instantiation shown in Equation 4.5 accounts for factors which influence the importance of a variable but are not directly captured by the spatial and temporal impact metrics. Specifically, the instantiation ensures that the observed rate of failures f, associated with variable v in module M is accounted for in the definition of importance, allowing the definitions of $\sigma_{v,M}$ and $\tau_{v,M}$ to remain independent of failure rate. Normalisation of the spatial and temporal impact factors is performed in order to ensure that the combination of these quantities does not mask or enhance the significance of the spatial impact or temporal impact metric. This normalisation is achieved by expressing the spatial impact and temporal impact metrics as a proportion of the maximum

possible extent of corruption, σ_{max}, and duration of corruption, τ_{max}, respectively. As each normalised impact metric can not be greater than 1, the $\frac{1}{2}$ term in Equation 4.5 served to bound the sum of normalised impact metrics above by 1. The definition of importance given in Equation 4.5 allows a balance to be reached between the need to detect errors and recover from them. Specifically, this can be achieved by varying the values assigned to n and m, which dictate how much emphasis is placed upon the need to avoid failures or the need to prevent widespread system corruption in the spatial and temporal domains. By setting $n = 0$ and $m > 0$ the importance metric can be made to focus solely upon state corruption in the spatial and temporal domains, i.e., erroneous states, whilst setting $m = 0$ and $n > 0$ focuses the importance metric solely on system failures. In general, system specific analysis requirements can be met through the parameterisation of Equation 4.5. Note that this refocusing of the importance metric can be achieved with or without system specific knowledge with respect to system structure and functionality.

4.4.2 Evaluating the Importance Metric

To evaluate the importance metric for all instrumented program variables, the failure rates and the values of the spatial and temporal impact metrics, σ and τ, associated with these program variables must be determined. In this thesis these values are experimentally estimated using fault injection analysis.

Spatial Impact: Applying the definition given in Section 4.2.1, the spatial impact of a program variable is the maximum number of software modules that were corrupted during any run where that program variable was the target of a fault injection, i.e., when that program variable was corrupted.

Temporal Impact: Similarly, applying the definition given in Section 4.3.1, the temporal impact of a program variable is the maximum duration over which at least one software module remains corrupted during any run where that pro-

gram variable was the target of a fault injection, i.e., when that program variable was corrupted. In the analysis shown, corruption duration is measured by the number of iterations of a main simulation loop or the number of files processed by an application. For example, if a program variable has a temporal impact of 1, it is an indication that the software system remained corrupted for exactly 1 simulation loop iteration or file processing tasks. In other words, a temporal impact of 1 is an indication that, when an error was injected into a program variable in iteration n, the state corruption remained in iteration $n+1$ but was such corruption was not present in iteration $n+2$.

Failure Rate: The system failure rate, f, associated with a program variable v is the number of failure executions where v was the target of a fault injection expressed as a proportion of the total number of executions where v was the target of a fault injection.

Weightings: The instantiation of the importance metric shown in Equation 4.5 is adaptable, in the sense that the instantiation can be configured to yield an importance ranking that focuses on the need to avoid failures or the need to prevent widespread corruption in the spatial and temporal domains. In order to demonstrate the type of results that can be generated using the instantiation shown in Equation 4.5, an initial parameterisation of $n = 1$ and $m = 2$ is adopted. Hence, this initial analysis attempts to balance the identification of program variables that have been implicated in widespread system corruption against the identification variables that are implicated in system failure. Several alternative parameterisations of the instantiation shown in Equation 4.5 are considered in Section 4.6, where the sensitivity of the instantiation is analysed.

$$I_{v,M} = \frac{1}{(1-f)^1}\Big(\frac{1}{2}\Big(\frac{\sigma_{v,M}}{\sigma_{max}} + \frac{\tau_{v,M}}{\tau_{max}}\Big)\Big)^2 \tag{4.6}$$

Once the spatial and temporal impact metrics have been estimated for each

Table 4.1: Importance ranking for 7Z1 variables ($\sigma_{max} = 34$, $\tau_{max} = 25$)

	Identifier	f	σ	τ	I
1	processedPosition	0.012869	2	1	0.002473
2	remainLen	0.010028	2	1	0.002466
3	distance	0.010085	1	1	0.001217
4	posState	0.008381	1	1	0.001215
5	ttt	0.006903	1	1	0.001213
6	matchByte	0.005063	1	1	0.001211
7	probLit	0.004625	1	1	0.001210
8	dicPos	0.004438	1	1	0.001210
9	range	0.002125	1	1	0.001207
10	kMatchLen	0.001600	1	1	0.001206

program variable, the system failure rates associated with program variables have been determined and the configuration of the importance metric has been set, the value of the importance metric for each program variable can be directly calculated using Equation 4.6. Section 4.5 presents several case studies in order to demonstrate the capability of the importance metric to generate a relative ranking of program variables that, through the use of an appropriate threshold, may be used as a means to identify critical variables.

4.5 Importance Metric Case Studies

The importance ranking of program variables in all target software modules are shown in Tables 4.1-4.9. Tables 4.10-4.12 show the importance rankings of the most important program variables, with respect to the importance metric, for each target system. In all tables the top 10 highest ranking program variables, with respect to the importance metric, are shown. Information used in the calculation of the importance metric for each program variable can also be found in these tables. Specifically, an identifier, system failure rate, spatial impact and temporal impact is shown for each program variable. Unique identifiers are assigned here to account for situations where two program variables at different levels of scope have an identical name.

The importance rankings shown in Tables 4.10-4.12 demonstrate the type of

Table 4.2: Importance ranking for 7Z2 variables ($\sigma_{max} = 34$, $\tau_{max} = 25$)

	Identifier	f	σ	τ	I
1	numberStreams	0.013882	5	15	0.141488
2	highPart	0.015526	3	15	0.120285
3	unpack	0.010994	3	9	0.050787
4	sizeIndex	0.002756	2	8	0.035976
5	i_unpack	0.002699	3	4	0.015447
6	attribute	0.018011	2	2	0.004906
7	numInStreams	0.002443	2	1	0.002448
8	numSubstream	0.002386	2	1	0.002447
9	unpackSize	0.002375	2	1	0.002447
10	nextHeaderOffset	0.002313	2	1	0.002447

Table 4.3: Importance ranking for 7Z3 variables ($\sigma_{max} = 34$, $\tau_{max} = 25$)

	Identifier	f	σ	τ	I
1	seekInStreamSint	0.009250	2	8	0.036212
2	wMode	0.008188	2	8	0.036173
3	res	0.001417	2	6	0.022356
4	oSize	0.001375	2	2	0.004825
5	moveMethod	0.001292	2	2	0.004824
6	CFIp	0.000958	2	2	0.004823
7	pos	0.000417	1	2	0.002994
8	lenghR	0.004000	1	1	0.001209
9	pHandle	0.000104	1	1	0.001205
10	cSize	0.000083	1	1	0.001205

Table 4.4: Importance ranking for FG1 variables ($\sigma_{max} = 312$, $\tau_{max} = 2000$)

	Identifier	f	σ	τ	I
1	Weight	0.003472	13	2000	0.272212
2	EmptyWeight	0.011905	2	2000	0.256266
3	bixx	0.000992	2	2000	0.253467
4	bixy	0.000992	2	2000	0.253467
5	bixz	0.000992	2	2000	0.253467
6	bizz	0.000868	2	2000	0.253435
7	biyz	0.000868	2	2000	0.253435
8	biyy	0.000772	2	2000	0.253411
9	Mass	0.011905	12	1432	0.144018
10	PMTotalWeight	0.000771	7	1432	0.136427

results that the importance metric is capable of generating. Relating the variables ranked in these tables to the component information in Tables 4.1-4.9, it is evident that the importance metric generally attributes a higher importance value to variables which are implicated in high levels of spatial and temporal

Table 4.5: Importance ranking for FG2 variables ($\sigma_{max} = 312$, $\tau_{max} = 2000$)

	Identifier	f	σ	τ	I
1	currentThrust	0.010417	8	2000	0.265753
2	hasInitEngines	0.003472	3	2000	0.255719
3	numTanks	0.004630	1	2000	0.252775
4	totalFuelQuantity	0.004167	1	2000	0.252658
5	firsttime	0.001736	1	2000	0.252043
6	dt	0.005208	3	983	0.063108
7	electricEng	0.122272	8	799	0.051481
8	throttleAdd	0.114087	2	600	0.026494
9	enme	0.120536	1	600	0.026133
10	te	0.009425	1	600	0.023202

Table 4.6: Importance ranking for FG3 variables ($\sigma_{max} = 312$, $\tau_{max} = 2000$)

	Identifier	f	σ	τ	I
1	compressLen	0.013889	17	1311	0.127795
2	groundSpeed	0.001984	5	831	0.046646
3	steerAngle	0.011111	15	8	0.000686
4	contractType	0.062066	8	48	0.000657
5	bDampRebound	0.027778	12	8	0.000464
6	eDampType	0.027344	11	8	0.000396
7	serviceRe	0.032407	4	43	0.000304
8	GearPos	0.120370	4	4	0.000062
9	rfrv	0.100694	3	4	0.000038
10	retractable	0.009549	1	6	0.000010

Table 4.7: Importance ranking for MG1 variables ($\sigma_{max} = 17$, $\tau_{max} = 25$)

	Identifier	f	σ	τ	I
1	selfWrite	0.028650	4	1	0.019506
2	bitridx	0.012650	4	1	0.019189
3	whiChannel	0.008400	4	1	0.019107
4	gainA	0.016700	3	1	0.011914
5	curFrame	0.015300	3	1	0.011897
6	inf	0.014925	3	1	0.011892
7	cuFile	0.005850	1	1	0.002456
8	wrdpntr	0.004167	1	1	0.002452
9	inbuffer	0.003906	1	1	0.002451
10	done	0.000156	1	1	0.002442

corruption, thus demonstrating that the results generated by the importance metric are consistent with the intentions underpinning its design. However, it should be remembered that the importance metric takes into account failure rate, as well as the extent and duration of software corruption. For example,

Table 4.8: Importance ranking for MG2 variables ($\sigma_{max} = 17$, $\tau_{max} = 25$)

	Identifier	f	σ	τ	I
1	sampleWin	0.194400	2	24	0.360391
2	batchSample	0.031100	2	21	0.236631
3	curSamples	0.008350	2	20	0.212292
4	first	0.006250	2	20	0.211843
5	op	0.014375	8	4	0.100860
6	linpre	0.071250	4	1	0.020400
7	rinpre	0.037917	4	1	0.019693
8	totsamp	0.034583	4	1	0.019625
9	cursamples	0.136458	3	1	0.013566
10	cursamplepos	0.129792	3	1	0.013462

Table 4.9: Importance ranking for MG3 variables ($\sigma_{max} = 17$, $\tau_{max} = 25$)

	Identifier	f	σ	τ	I
1	maxAmpOnly	0.011825	2	25	0.316021
2	dSmp	0.009200	2	14	0.115867
3	winCont	0.000800	2	14	0.114893
4	sum	0.009583	3	12	0.108781
5	mSamp	0.001375	4	10	0.101039
6	bandPtr	0.000250	4	8	0.077107
7	window	0.010179	3	8	0.062254
8	windowSL	0.014643	2	8	0.048595
9	sBuffs	0.002143	1	1	0.002447
10	b0	0.001964	1	1	0.002446

Table 4.10: Importance ranking for instrumented modules in Z-Zip

Rank	**Identifier**	**Module**	I
1	numberStreams	7Z2	0.141488
2	highPart	7Z2	0.120285
3	unpack	7Z2	0.050787
4	seekInStreamSint	7Z3	0.036212
5	wMode	7Z3	0.036173
6	sizeIndex	7Z2	0.035976
7	res	7Z3	0.022356
8	i_unpack	7Z2	0.015447
9	attribute	7Z2	0.004906
10	oSize	7Z3	0.004825

7Z2 program variables *attribute* and *numInStreams* both have a spatial impact of 2, paired with a temporal impact of 2 and 1 respectively, making these program variables comparable in terms of the widespread corruption that they can incur. However, the fact that their associated failure rates are 0.018011 and

Table 4.11: Importance ranking for instrumented modules in FlightGear

Rank	**Identifier**	**Module**	I
1	Weight	FG1	0.272212
2	currentThrust	FG2	0.265753
3	EmptyWeight	FG1	0.256266
4	hasInitEngines	FG2	0.255719
5	bixx	FG1	0.253467
6	bixy	FG1	0.253467
7	bixz	FG1	0.253467
8	bizz	FG1	0.253435
9	biyz	FG1	0.253435
10	biyy	FG1	0.253411

Table 4.12: Importance ranking for instrumented modules in MP3 Gain

Rank	**Identifier**	**Module**	I
1	sampleWin	MG2	0.360391
2	maxAmpOnly	MG3	0.316021
3	batchSample	MG2	0.236631
4	curSamples	MG2	0.212292
5	first	MG2	0.211843
6	dSmp	MG3	0.115867
7	winCont	MG3	0.114893
8	sum	MG3	0.108781
9	op	MG2	0.100860
10	mSamp	MG3	0.101039

0.002443 respectively, means that the importance metric considers *attribute* to be significantly more important than *numInStreams*, as indicated by the fact that the importance value of *attribute* is double that of *numInStreams*. This is because the failure rate associated with *attribute* is significant enough, given values of n and m in Equation 4.6, to enhance the importance of *attribute* far beyond that of *numInStreams*. Allowing the supposedly dominant component of Equation 4.6, i.e., the spatial and temporal impact metrics, to be led in this way is entirely desirable, as the potential for system failure when *attribute* is corrupted justifies it being afforded a higher value. It is also interesting to note how spatial and temporal impact contribute to the value of the importance metric for each program variable. For example, the *electricEng* variable in FG2 has a spatial impact of 8 combined with a seemingly high temporal impact of 799. However, as the importance metric interprets these values as proportions of the

associated maximum possible values, the overall importance of *electricEng* is lower than might otherwise be expected. Most crucially, the case studies associated with the relative rankings shown in Tables 4.10-4.12 provide a means for assessing the capability of the importance metric to identify critical variables, a process that is undertaken later in this thesis.

4.6 Sensitivity Analysis

In order to demonstrate the robustness of the importance metric definition, details of the rankings that would be generated by changing the values of n and m in Equation 4.5 are shown in Tables 4.13-4.21. The tuple (p_1, p_2) is used to represent a single parameterisation of Equation 4.5, where $p_1 = n$ and $p_2 = m$. Tables 4.13-4.21 show how the relative rankings for each module differ when $(4, 1)$, $(2, 1)$, $(1, 1)$, $(1, 2)$ and $(1, 4)$ parameterisations are used. For example, in Table 4.17 the variable *dt* is considered the 7th most important program variable under $(4, 1)$, whilst variable *electricEng* is considered to be the 7th most important variable under the $(1, 1)$ parameterisation. If the definition of the importance metric given in Equation 4.5 is robust there should be little variation in the ranking of program variables across parameterisations, unless there exists a justification, with respect to spatial impact, temporal impact and system failure rate, for such variation. Table rows where there is any discrepancy in the ranking across parameterisation are indicated by a $*$ symbol in the *Identifier* column.

The sensitivity analysis presented in Tables 4.13-4.21 demonstrates that the importance metric given in Equation 4.5 is robust across many possible parameterisations. Observe that the relative ranking generated by the $(1, 2)$ parametrisation of the importance metric is often replicated by all parameterisations considered. In cases where the ranking is not replicated, discrepancies are minor, localised and sensible given the parameterisation of the importance metric. For example, Table 4.17 shows that the ranking of program variables *throttleAdd* and

Table 4.13: Sensitivity analysis of the importance ranking for 7Z1

Identifier	**(4,1)**	**(2,1)**	**(1,1)**	**(1,2)**	**(1,4)**
processedPosition	1	1	1	1	1
remainLen	2	2	2	2	2
distance	3	3	3	3	3
posState	4	4	4	4	4
ttt	5	5	5	5	5
matchByte	6	6	6	6	6
probLit	7	7	7	7	7
dicPos	8	8	8	8	8
range	9	9	9	9	9
kMatchLen	10	10	10	10	10

Table 4.14: Sensitivity analysis of the importance ranking for 7Z2

Identifier	**(4,1)**	**(2,1)**	**(1,1)**	**(1,2)**	**(1,4)**
numberStreams	1	1	1	1	1
highPart	2	2	2	2	2
unpack	3	3	3	3	3
sizeIndex	4	4	4	4	4
i_unpack	5	5	5	5	5
attribute	6	6	6	6	6
numInStreams	7	7	7	7	7
numSubstream	8	8	8	8	8
unpackSize	9	9	9	9	9
nextHeaderOffset	10	10	10	10	10

Table 4.15: Sensitivity analysis of the importance ranking for 7Z3

Identifier	**(4,1)**	**(2,1)**	**(1,1)**	**(1,2)**	**(1,4)**
seekInStreamSint	1	1	1	1	1
wMode	2	2	2	2	2
res	3	3	3	3	3
oSize	4	4	4	4	4
moveMethod	5	5	5	5	5
CFIp	6	6	6	6	6
pos	7	7	7	7	7
lenghR	8	8	8	8	8
pHandle	9	9	9	9	9
cSize	10	10	10	10	10

enme, as well as *dt* and *electricEng*, are interchanged under the $(4, 1)$ and $(2, 1)$ importance metric parameterisations. This discrepancy is minor and localised, as these program variables represent the only changes to the ordering generated by all other parameterisations for the software module. Indeed, the top

Table 4.16: Sensitivity analysis of the importance ranking for FG1

Identifier	**(4,1)**	**(2,1)**	**(1,1)**	**(1,2)**	**(1,4)**
Weight	1	1	1	1	1
EmptyWeight	2	2	2	2	2
bixx	3	3	3	3	3
bixy	4	4	4	4	4
bixz	5	5	5	5	5
bizz	6	6	6	6	6
biyz	7	7	7	7	7
biyy	8	8	8	8	8
Mass	9	9	9	9	9
PMTotalWeight	10	10	10	10	10

Table 4.17: Sensitivity analysis of the importance ranking for FG2

Identifier	**(4,1)**	**(2,1)**	**(1,1)**	**(1,2)**	**(1,4)**
currentThrust	1	1	1	1	1
hasInitEngines	2	2	2	2	2
numTanks	3	3	3	3	3
alpha	4	4	4	4	4
firsttime	5	5	5	5	5
$*$ dt	7	7	6	6	6
$*$ electricEng	6	6	7	7	7
$*$ throttleAdd	9	9	8	8	8
$*$ enme	8	8	9	9	9
te	10	10	10	10	10

Table 4.18: Sensitivity analysis of the importance ranking for FG3

Identifier	**(4,1)**	**(2,1)**	**(1,1)**	**(1,2)**	**(1,4)**
compressLen	1	1	1	1	1
groundSpeed	2	2	2	2	2
$*$ steerAngle	4	4	4	3	3
$*$ contractType	3	3	3	4	4
bDampRebound	5	5	5	5	5
eDampType	6	6	6	6	6
serviceRe	7	7	7	7	7
GearPos	8	8	8	8	8
rfrv	9	9	9	9	9
retractable	10	10	10	10	10

5 program variables remain consistent across all parameterisation considered. The discrepancy observed is sensible given the orientation of the importance metric, since under $(4, 1)$ and $(2, 1)$ the importance metric attributes greater weight to program variables with a higher system failure rate, which is why the

Table 4.19: Sensitivity analysis of the importance ranking for MG1

Identifier	**(4,1)**	**(2,1)**	**(1,1)**	**(1,2)**	**(1,4)**
selfWrite	1	1	1	1	1
bitridx	2	2	2	2	2
whiChannel	3	3	3	3	3
gainA	4	4	4	4	4
curFrame	5	5	5	5	5
inf	6	6	6	6	6
cuFile	7	7	7	7	7
wrdpntr	8	8	8	8	8
inbuffer	9	9	9	9	9
done	10	10	10	10	10

Table 4.20: Sensitivity analysis of the importance ranking for MG2

Identifier	**(4,1)**	**(2,1)**	**(1,1)**	**(1,2)**	**(1,4)**
sampleWin	1	1	1	1	1
batchSample	2	2	2	2	2
curSamples	3	3	3	3	3
first	4	4	4	4	4
op	5	5	5	5	5
$*$ linpre	8	6	6	6	6
$*$ rinpre	9	7	7	7	7
$*$ totsamp	10	8	8	8	8
$*$ cursamples	6	9	9	9	9
$*$ cursamplepos	7	10	10	10	10

Table 4.21: Sensitivity analysis of the importance ranking for MG3

Identifier	**(4,1)**	**(2,1)**	**(1,1)**	**(1,2)**	**(1,4)**
maxAmpOnly	1	1	1	1	1
dSmp	2	2	2	2	2
$*$ winCont	4	3	3	3	3
$*$ sum	3	4	4	4	4
mSamp	5	5	5	5	5
bandPtr	6	6	6	6	6
window	7	7	7	7	7
windowSL	8	8	8	8	8
sBuffs	9	9	9	9	9
b0	10	10	10	10	10

program variable *enme* rises in these rankings. In some cases, such as the $(4, 1)$ parameterisation for module MG2, the metric is more sensitive to changes in parameterisation, though when interpreted alongside the associated values for spatial impact, temporal impact and failure rate, it is clear that this apparent

sensitivity can be explained by the orientation of the importance metric in these situations. Specifically, as this parameterisation emphasises the importance of variable with a high failure rate, those variables with a failure rate that is sufficiently large will rise above variables whose importance ranking is mainly the result of a high spatial and temporal impact. For example, variables *cursamples* and *cursamplepos* have risen in the $(4, 1)$ ranking for module MG2 due to them having failure rates of 0.136458 and 0.129792 respectively. Overall, the sensitivity analysis in Tables 4.13-4.21 has shown the importance metric to be robust with respect to parameterisation, with all ranking variations being accounted for by the orientation of the metric.

4.7 Implications and Discussion

The identification of program variables that should be incorporated by error detection predicates can simplify and guide the design of efficient EDMs. The importance metric, when used in conjunction with an appropriate threshold, can enable this identification process through the generation of an importance ranking of program variables. The application of a threshold to an importance ranking would essentially entail the selection of an importance value or ranking position, where all program variables with an importance metric value greater than the selected value or occupying a position greater than the selected position would be considered critical variables. Such an application of the importance metric would allow software engineers to precisely target those variables which have the most significant impact upon the correct functioning of a software system. For example, from Table 4.12 it can be observed that program variables *processedPosition* and *remainLen* have the highest importance values. If a threshold were set such that these program variables could be considered critical then it would follow that an EDM must ascertain that these program variables hold appropriate values during the execution of the associated software system.

The relative rankings generated by the importance metric can also be used

to focus the efforts of software engineers on specific software modules. For example, from Tables 4.10-4.12 it can be seen that program variables associated with software modules *7Z1*, *FG2* and *MG2* feature heavily in the importance rankings for their respective software systems, suggesting that these software modules should be equipped with dependability mechanisms, re-engineered or closely monitored. This use of the importance metric is particularly applicable in the context of commercial software system development, where subsequent software system releases can not address all known issues and, hence, software engineers will seek to address software vulnerabilities in order of severity. In such a situation it would be possible to threshold the relative rankings generated as part of a cost-benefit analysis, thus ensuring that the most severe vulnerabilities are readily addressed. Further, observe that the relative rankings presented in Tables 4.1-4.12 have been generated without prior knowledge of software system structure or functionality. As the importance measure does not take into account composition or communication paths, it can be readily applied in situations where dependability is to be assessed post-implementation or by software engineers who were not directly involved in the implementation of a software system. Again, this is consistent with current approaches to commercial software engineering, where dedicated teams often deal with the maintenance and on-going support for previously developed software systems.

Over 38 million fault injection experiments were performed in order to estimate values for failure rate, spatial impact and temporal impact of program variables in all target software modules. The PROPANE fault injection suite allowed these experiments to be conducted in an automated fashion. Provided that the instrumentation of the target system is concerned with maximising the number of program variables instrumented in a given module, which will generally be the case for most software systems, there is no reason why this level of automation can not be achieved for any given target software system. Once the values for the spatial and temporal impact metrics have been determined, the importance metric can be automatically calculated for each instrumented

variable. As the definition of the importance metric is fixed for a software system, rather than individual variables or modules, the cost of employing the proposed approach, in terms of engineering effort, is relatively low. Further, in the case of dependable software systems, the fact that fault injection analysis is a commonly adopted dependability validation techniques means that the information required to evaluate the importance metric is likely to be available during software system development.

There are limitations associated with the use of the importance metric. In particular, the relative ranking generated by the importance metric is sensitive to the set of program variables under consideration. Ideally, all variables within a software module should be analysed, thus ensuring that no variable with a potentially high importance, which may subsequently lead it it being considered a critical variable, is overlooked. However, performing such a comprehensive analysis may be impractical in some situations, as it may require a prohibitive number of fault injection experiments, particularly if all the software modules in a software system are to be analysed. However, this limitation is, to a large extent, mitigated by the extended validation and testing periods associated with dependable software systems and the automated nature of the analysis process. Another perceived limitation of the importance metric is that its value for a particular program variable is not an objective representation of the real-world importance of that program variable. Rather it is a value which, when viewed relative to others produced during the same analysis, can focus the efforts of dependability engineers and allow a meaningful cost-benefit analysis to be undertaken. Finally, although it is not a directly limitation of the metric itself, the fault injection analysis approach proposed for estimating the importance metric suffers from the inherent limitations of the fault injection process, such as a dependence upon a the identification of a representative set of test cases. As these are limitations of the evaluation mechanism, and more generally of fault injection analysis as a dependability validation technique, they do not negatively impact on the use of the importance metric as applied in this thesis.

4.8 Conclusion

In this chapter a variable-centric, dynamic metric suite that can be used, in conjunction with an appropriate threshold, for the identification of program variables that should be incorporated by the error detection predicates of EDMs has been proposed. The proposed metric suite is composed of three metrics, namely spatial impact, temporal impact and importance. The spatial and temporal impact metrics capture the spatial and temporal degree to which a system is corrupted respectively, whilst the importance metric was defined as a function of the spatial and temporal impact metrics. After a specific instantiation of the importance metric was provided, an experimental approach to evaluate the importance metric was presented in order to demonstrate the type of results that can be generated. These results took the form of a relative ranking amongst the program variables in a software module. Through the application of a threshold on importance values or ranking position, this relative ranking can be used in the engineering of dependable software systems to facilitate the design of EDMs based on critical variables, as well as informing the positioning of EDMs based on the premise that critical variables should always hold appropriate values and permitting a cost-benefit analysis to be undertaken when deciding where the dependability enhancement efforts of software engineers should be focused.

Having defined, applied and demonstrated the type of results that can be generated by the proposed metric suite, it is important to consider the capability of the importance metric to identify variables that should be incorporated by the error detection predicates of EDMs. To this point it has been presumed that accounting for the extent and duration of corruption in the spatial and temporal domains respectively, coupled with consideration of system failure rate and analysis alignment, will necessarily ensure that the importance metric identifies critical variables. However, as this capability can not be presumed, a mechanism must be found for its evaluation. In the next chapter we develop an approach for the generation of efficient error detection predicates for EDMs.

Crucially, as this approach is independent of the metric suite developed in this chapter and does not rely on the experience of software engineers or a system specification, it can be used in evaluating the suitability of using the importance metric in the identification of critical variables.

CHAPTER 5

Generating Efficient Error Detection Mechanisms

In the previous chapter a dynamic metric suite was proposed to facilitate the identification of program variables that should be captured by the error detection predicates of EDMs. In order to validate the capability of this metric suite to identify program variables that are critical, when used in conjunction with an appropriate threshold, an approach for the generation of efficient error detection predicates is necessitated. To be fit for this purpose, the developed approach must be unrelated to the proposed metric suite, systematic and repeatable. This is in contrast with current approaches for the design of error detection predicates for EDMs, which generally rely on system specifications and the experience of software engineers. This chapter develops the first systematic approach for the generation of efficient error detection predicates for real-world, infinite-state software systems. More specifically, the proposed approach employs data mining techniques, including decision tree induction and rule induction, for the analysis of fault injection data sets, in order to discover efficient error detection predicates. The results presented demonstrate that this

approach can be used to generate error detection predicates that are efficient by design, removing the reliance on system specifications and the experience of software engineers. Analysis of the error detection predicates generated by the proposed approach serves to validate the capability of the metric suite developed in Chapter 4 to identify critical variables.

5.1 The Design of Error Detection Predicates

Predicates for EDMs are commonly designed based on a system specification [60] or the experience of software engineers [102]. As highlighted previously, it has been shown that the efficiency properties of EDMs can be classified along two dimensions; (i) completeness and (ii) accuracy [70]. The completeness of an EDM relates to its ability to detect erroneous states, i.e., to flag true positives, whilst accuracy relates to its ability to avoid making incorrect detections, i.e., to avoid false positives. Throughout this thesis it is assumed that an erroneous state is one that will lead to a system failure if the error is not handled. A system failure is characterised as a violation of a behavioural specification. An EDM that is both complete and accurate is known as a perfect detector. However, due to implementation constraints, such as read and write restrictions, it is, in general, not possible to develop perfect detectors [76]. A perfect detector at a given location in a program is therefore the most efficient detector for that location. As previously stated, the term efficient EDM is used to refer to EDMs that implement detection predicates with high completeness and high accuracy.

Research that has addressed the systematic design of efficient detectors has generally focused on finite-state software systems. However, little work has focused on the systematic design of efficient detectors for real-world, infinite-state software systems. To address this issues, this chapter proposes a systematic approach to the design of efficient error detection predicates. Most significantly, the proposed approach is applicable in the context of real-world, infinite-state software systems and generates error detection predicates whose efficiency is

guaranteed by design. The premise of the proposed approach is that, since fault injection analysis captures relationships among the program variables and the success of software system executions, data mining techniques can be applied to learn these relationships and how they impact the success of software executions, with a view to applying these derived relationships as error detection predicates for failure-inducing software system states.

In order to illustrate the novelty and significance of the proposed approach, Sections 5.1.1-5.1.5 discuss current approaches to the design of error detection predicates for EDMs. The work discussed in these sections is also intended to complement the research surveyed in Chapter 4, which focused on the identification of critical software components and vulnerabilities, rather than approaches for the derivation or composition of error detection predicates.

5.1.1 Heuristics and Experience

Many approaches to the error detection predicate design problem have focused on the refinement of proposed error detection predicates based on evaluations carried out with respect to coverage and latency. Often such approaches have concentrated on the refinement of error detection predicates through the assessment of executable assertions using fault injection analysis [17] [162]. Through approaches such as these it was established that EDMs with high coverage and low latency reduced error propagation. However, designing error detection predicates for EDMs is difficult and error-prone, as highlighted in [102], where it was remarked that "...the process of writing self checks is obviously difficult". To remedy this, the authors suggested that "...more training or experience might be helpful". Indeed, the use of experience in the design of error detection predicates for EDMs is commonplace in software engineering, not least due to a shortage of systematic approaches for the design of efficient error detection predicates.

5.1.2 System Specifications

Aside from the experience of software engineers, approaches to the design of error detection predicates have also used the software system specifications and the constraints placed on signals, parameters and variable to design corresponding executable assertions [60] [159]. However, such executable assertions may not exhibit the high levels of efficiency required in dependable software systems. In particular, it has been shown that the error detection predicates associated with such executable assertions may not flag erroneous states, i.e., false negatives, or may incorrectly flag correct states as being erroneous, i.e., false positives [75]. When a particular EDM does not meet the coverage and latency thresholds required of a software system, it must be redesigned. However, little research has focused on the refinement of such mechanisms in practical software systems. The refinement of EDMs has been investigated in finite-state software systems, which are usually represented as state transition systems [70] [91]. Through these approaches, polynomial-time algorithms were developed to automatically refine existing error detection predicates. In contrast, the approach proposed in this chapter targets the systematic derivation and subsequent optimisation of error detection predicates for real-world, infinite state software systems, a problem that has received little attention in existing research.

5.1.3 Verification and Validation Techniques

A number of software validation and verification techniques have been applied in the design of error detection predicates for EDMs, such as model checking [34], data-flow analysis [138] and abstract interpretation [37]. Many such approaches have relied on some form of static analysis. For example, a static analysis-based approach was used in [118] and [119], specifically to derive detection predicates for the prevention of data error propagation. Model checking approaches, which are typically concerned with software verification, generally consider software systems that have finite-state or may be reduced to finite-state by some degree

of abstraction, whilst data-flow analysis is a lattice-based software validation technique for gathering information regarding sets of permissible values. Abstract interpretation is a software validation approach where the aim is to model the impact that program statements have on the state of an abstract machine, i.e., a software system is executed based on the mathematical properties of each program statement. Such abstract machines are known to over-approximate the behaviours of a software system. The abstract system is therefore made simpler to analyse at the expense of incompleteness, as not every property that is true of the original software system is also true of the abstract system. However, if properly performed, abstract interpretation is sound, which means that every property that is true of the abstract system can be mapped to a property that is true of the original software system. Further, it is well known that, barring some hypothesis that the state space of all computer programs is finite and small, finding all possible run-time errors, or more generally any kind of violation of a specification on the final result of a program, is undecidable. Thus, static analyses-based techniques applied to a software system are, in general, sound, in the sense that the properties they report are true, but not complete.

5.1.4 Data Mining Techniques

With regard to the derivation of error detection predicates, the application of data mining techniques have generally focused on the analysis of failure data and service logs for dependable software systems. For example, research in [124] used a combination of data mining techniques on data recorded during benchmarking to identify key infrastructural factors in determining the behaviour of systems in the presence of faults. These investigations can also serve to help to identify weaknesses or vulnerabilities in a software system. In contrast, the data mining-based approach proposed in this chapter seeks to discover predicates for EDMs in order to enhance dependability and address vulnerabilities in software systems. Data mining techniques have also been applied to address a number of other software dependability issues. For example, in the context of computer security,

data mining has been shown to be an effective approach to intrusion detection and anomaly identification [105] [167].

5.1.5 Likely Program Invariants

A program invariant is a property that holds throughout the execution of a program. It is know that determining all the sound invariants for a program may be undecidable. Further, invariants reported may not be sound, i.e., an invariant may hold for most executions, but not for some. Thus, determining likely program invariants may be the best approximation, though steps must be taken to handle false positives [44]. The use of program invariants is potentially valuable in many aspects of software development, including program design, implementation, testing and maintenance. Unfortunately, explicit invariants are usually absent from programs, depriving programmers and automated tools of their benefits. The seminal work on discovering likely program invariants shows how invariants can be dynamically detected from program traces that capture variable values at specified points of interest [44]. Typically a target program is executed alongside a test suite to create program traces. An invariant detector will then process these traces to determine which properties and relationships hold over program variables. A software tool, called Daikon, exists that supports the discovery of likely program invariants. Subsequently, several applications of the techniques have been proposed. For example, Demsky *et al.* applied these techniques to discover invariants of abstract data types [39]. More recently, these techniques have also been applied to detect permanent hardware failures [135]. Dynamic invariant detection is a machine learning technique that can be applied to arbitrary data. However, program invariants generally do not hold in presence of transient failures. Indeed, the approach proposed in this chapter seeks to detect erroneous states that lead to failure rather than all erroneous states, which contrasts with the intention of likely program invariants.

5.1.6 Evaluation of Existing Design Approaches

In the context of finite-state systems, the design of error detection predicates for EDMs can be considered to be well understood [70] [91]. However, state-of-the-art approaches to the design of EDMs do not provide a systematic approach for the generation of error detection predicates that exhibit high accuracy and completeness for real-world, infinite-state software systems. Indeed, the most efficient EDMs for real-world, infinite-state software systems are often designed based on some interpretation of a system specification and the experience of software engineers, with repeated redesign being an accepted part of the EDM design process. Further, whilst static analysis and the use of likely program invariants are systematic, the theoretical limitations of static analysis, with respect to accuracy and completeness, and the necessary focus of likely program invariants on a constrained definition of erroneous state, i.e., likely program invariants do not consider erroneous software states that do not result in a system failure to be permissible, means that these approaches do not solve the error detection predicate design problem for real-world, infinite-state software systems.

5.2 Data Mining Concepts

Technology related to the modelling, collection, storage and querying of data generated by real-world processes have advanced significantly in recent years. Data pertaining to a real-world process is usually modelled as a set of entities, their attributes and their relationship to other entities. This is commonly known as the relational model of data. Data generated, and hence stored, within such a relational data model is a sample of all the data that may be generated by the process. Often, rather than being interested in the retrieval of stored data, it is more interesting and useful to be able to forecast behaviours of the process not previously encountered or derive knowledge about the process if the process itself is not well understood. For example, in the context of the research presented in

this chapter, it is interesting to understand how a software module under test, and its associated software system, is likely to behave when confronted with an injected fault.

5.2.1 Fundamentals of Data Mining

It is the aim of data mining to learn useful and actionable knowledge from large collections of data. In simple domains, it is common to assume that data exists as a single relation consisting of a set of n input attributes that define an n-dimensional space called the Instance Space, I. Every point in I is a potential state of the process being modelled. In supervised learning a data mining algorithm is tasked with learning a good approximation, $\hat{f}$, of an unknown function f, referred to as the target function, given a training data set, $T \subseteq I$, consisting of the N pairs $\langle x_i, f(x_i) \rangle$. If the function is discrete then the task is referred to as classification. In the case of learning a function from data generated through fault injection analysis, the function is binary as a system state is either going to lead to a system failure or a successful execution. The task of learning a binary function is often referred to as concept learning, which is a special case of classification. Within a data set to be analysed by a data mining algorithm, instances of the class of interest, known as the concept, are referred to as positive instances. In contrast, all instances within a data set that do not belong to the concept are referred to as negative instances.

A number of algorithms have been proposed to solve classification problems, including many that employ naïve Bayes, nearest neighbour methods, support vector machines (SVMs), logistic regression, rule induction, neural networks or decision tree induction. The key difference between algorithmic solutions to classification problems is in the kind of decision boundary that is defined between classes, i.e., their functional form and the set of parameters they fit, and the heuristic they employ in searching for the optimal function, also known as the hypothesis, within the space of possible hypotheses as defined by the functional form of the hypotheses. In this chapter the aim of the proposed

approach is to generate efficient error detection predicates, hence there is a focus on evaluating symbolic pattern learning algorithms, such as decision tree induction and rule induction, as their output can easily be represented as first-order predicates. In addition, results for classifiers based on the naïve Bayes and logistic regression algorithms are also shown in this chapter. This is done to illustrate the levels of error detection predicate efficiency that can be achieved through the application of relatively modest data mining algorithms.

The function approximation learnt, often referred to as the model, by the classification algorithm from training instances needs to be evaluated, in order to obtain a measure of the expected accuracy of the model, on unseen data. Typically the accuracy of a model is measured by the percentage of test data instances correctly classified, hence most algorithms seek to learn hypotheses that minimise the number of errors. However, this implicitly assumes that all types of misclassification incur an equal cost, which is not always the case. For example, in the context of a safety-critical software system, a model incorrectly classifying a failure-inducing state as non-failure-inducing will, in the majority of circumstances, result in a much more significant cost than a non-failure-inducing state being classified as failure-inducing. In such situations, the predictions of a model on a test data set can be cross-tabulated with the actual classes assigned to the instances by the target function to produce a confusion matrix. Table 5.1 shows the general form of a confusion matrix for a concept learning problem. In Table 5.1, TP is the number of positives instances labelled as positive instances by $\hat{f}$, known as true positives, whilst FN is the number of positive instances labelled as negative, known as false negatives. Further, FP is the number of negative instances labelled as positive, known as false positives, whilst TN is the number of negative instances labelled as negative, known as true negatives. Finally, n_{pos}/n_{neg} are the number of positive/negative instances in the test data and $\hat{n}_{pos}/\hat{n}_{neg}$ are the number of instances predicted as positive/negative. In the design of efficient error detection predicates, it natural to seek out models that maximise true positives and minimise false positives, not least because these

Table 5.1: The general form of a confusion matrix for concept learning.

		Predicted Class		
		Pos.	Neg.	Marginal Sums
Actual Class	Pos.	TP	FN	n_{pos}
	Neg.	FP	TN	n_{neg}
	Marginal Sums	$\hat{n}_{pos}$	$\hat{n}_{neg}$	n

correspond closely with the concepts of accuracy and completeness. However, as a balance must be struck between these related concerns, it is appropriate to identify an aggregated measures of model quality.

5.2.2 Measuring Model Quality

A variety of metrics for model evaluation have been proposed based on the structure of the confusion matrix. The most common of these are specificity or true negative rate (TNR), as shown in Equation 5.1, and sensitivity or true positive rate (TPR), as shown in Equation 5.2.

$$specificity = TNR = \frac{TN}{TN + FP} \tag{5.1}$$

$$sensitivity = TPR = \frac{TP}{TP + FN} \tag{5.2}$$

Kubat *et al.* used the geometric mean of the TPR and TNR as an evaluation metric [88]. In contrast, ROC analysis is based on a plot in two dimensions where each model is a point defined by the coordinates $(1-$*specificity*,*sensitivity*$)$, where $(1-$*specificity*$)$ is also referred to as the false positive rate (FPR), as shown in Equation 5.3.

$$1 - specificity = FPR = \frac{FP}{TN + FP} \tag{5.3}$$

For different configurations, the same classification algorithm will produce multiple points on such a plot. The area under the curve (AUC) obtained by joining these points to $(0, 0)$ and $(1, 1)$ is a common measure of the expected accuracy of a classification algorithm. For a single model, the simple trapezium

obtained by connecting the coordinates $(0,0)$, (FPR, TPR), $(1,1)$ and $(1,0)$ has an area given by Equation 5.4.

$$AUC = \frac{TPR - FPR + 1}{2} \tag{5.4}$$

The Euclidean distance from the perfect classifier, which has coordinates $(0,1)$, i.e, $FPR = 0$ and $TPR = 1$, may be used in the ranking of single models. This measure is given by the well known formula in Equation 5.5.

$$distance = \sqrt{(FPR - 0)^2 + (1 - TPR)^2} \tag{5.5}$$

A model quality metric from the domain of information retrieval is the F1 measure that combines precision and recall by computing their harmonic mean, where precision is given by Equation 5.6 and recall is identical to the sensitivity measure shown in Equation 5.2.

$$precision = \frac{TP}{TP + FP} \tag{5.6}$$

When the cost associated with a false positive is different from that of a false negative, a more appropriate measure of the quality of a model is the expected misclassification cost, rather that the expected error. This requires the definition of a cost matrix. Assuming there are m class labels, L_i, an $m \times m$ cost matrix, C, needs to be defined such that the value $C(i,j)$ is the cost of misclassifying an instance of class L_i to the class L_j. Clearly $C(i,i) = 0$ as there should be no cost associated with correctly classifying an instance. Minimising the error is a special case of minimising misclassification cost when the cost matrix is defined as $C(i,j) = 1$, where $i \neq j$ and $C(i,i) = 0$. The expected misclassification cost, *mcost*, can then be calculated as shown in Equation 5.7, where $CM(i,j)$ is represents index access to the associated confusion matrix using i and j.

$$mcost = \sum_{i}^{m} \sum_{j}^{m} C(i,j) * CM(i,j) \tag{5.7}$$

The focus of the approach proposed is this chapter is on the generation of efficient error detection predicates, which means that the measurement of model quality is performed with respect to the efficiency, i.e., the levels of accuracy and completeness, that can be achieved by these predicates. With this in mind, the AUC measure, which represents an aggregate of accuracy and completeness in the form of TPR and FPR, is used in model quality evaluation. However, as misclassification costs are likely to vary in the context of dependable software systems, steps must be taken to ensure that high AUC values are not achieved through the neglect of accuracy or completeness. With this in mind, TPR and FPR are also considered when evaluating the quality of generated models.

5.2.3 Addressing Class Imbalance

The approach proposed in this chapter is founded on the premise that the data generated during fault injection analysis captures aspects of the relationships between system states and system failures. Based on the states sampled and behaviours observed during fault injection analysis, a data mining algorithm can then generate error detection predicates through learning about these captured relationships. However, data sets derived from fault injection analysis are often imbalanced, in the sense that most of the logged states will not lead to a system failure, i.e., only a small proportion of runs lead to failure. Such an imbalance in the data sets to be processed must be addressed for the data mining process to be effective with respect to the generation of efficient error detection predicates.

A key assumption made by concept learning algorithms that are based on error minimisation is that the training data used is well balanced [68]. That is to say, such algorithms assume that the distribution of class labels in training data sets is approximately uniform. However, there are a number of domains, such as network intrusion detection, fraud detection and software reliability, where the number of positive instances are often fewer than the number of negative instances. In addition to this skew in distribution, it is often the case that the minority class is the more interesting class to predict. Indeed, with respect

the examples of generating efficient error detection mechanisms and detecting network intrusion, it is the minority classes, i.e., system failures and network intrusions, that are of most interest.

Two approaches have been used to address problem of class imbalance. The first of these is to act as if there is a higher cost associated with misclassifying instances of the minority class. Specifically, it is possible to define a cost matrix based on the class imbalance and then use the same error minimisation-based concept learning algorithms. However, this approach assumes that such a cost matrix can be incorporated by the learning process. This incorporation can, for example, be achieved by the altered priors technique proposed by Breiman *et al.* [21]. The second approach to addressing the problem of class imbalance is to replace error minimisation metrics with cost minimisation metrics when searching the hypothesis space. However, Pazzani *et al.* showed that using misclassification costs as a greedy selection criteria in decision tree induction does not provide cost minimisation for the model generated [122]. Further, Ting *et al.* compared instance weighting to using minimum expected cost metrics for assigning labels to leaf nodes in a decision tree induced to minimise errors [157]. The results of these experiments suggested that instance weighting is more effective than a cost minimisation-based approach.

The assignment of distinct weights to training examples, in effect, changes the data distribution within the training data [40] [46] [122] [157]. The associated cost matrix must be converted to a cost vector, V, which can be difficult in the context of multi-class classification problems. Breiman *et al.* proposed using the sum of all misclassification costs for instances of the class, though alternatives, such as $V(i) = \arg\max_j(C(i,j))$, have also been proposed [21]. Ting *et al.* assign the same weight to all instances of a particular class, L_j, based on $V(j)$ using the formula shown in Equation 5.8, where N_j is the number of instances in the data labelled L_j and $N = \sum_i N_i$ [157].

$$w(j) = V(j)\frac{N}{\sum_i V(i)N_i} \tag{5.8}$$

An alternative to implicitly changing the data distribution is to resample an original data set, either by oversampling the minority class or undersampling the majority class to make the class distribution more uniform [68] [89] [103]. A variety of resampling approaches have been investigated, with the most common approaches being those which resample with replacement and sample without replacement for undersampling the majority class. Japkowicz also experimented with focussed sampling approaches that oversampled from the boundary regions and undersampled from regions far from the decision boundary but experiments in these investigations suggested that there is little value over random sampling approaches [68]. Chawla *et al.* proposed the generation of synthetic data for minority classes along the line segment joining an example to k minority class nearest neighbours rather than simply sampling with replacement [26]. Empirical tests showed their method, known as Synthetic Minority Oversampling Technique (SMOTE), to outperform simple sampling with replacement. Zadrozny *et al.* proposed the use of a cost-proportionate rejection sampling technique, while Kubat and Matwin suggest undersampling by removing redundant and borderline negative examples [89] [172]. A criticism of the oversampling and undersampling approaches is that it is not clear how much over oversampling and undersampling should be carried out. Chawla *et al.* proposed the use of cross validation for setting the level of oversampling and undersampling of the majority and minority classes automatically, ultimately demonstrating that this process can improve model accuracy [27].

5.3 Error Detection Predicate Generation

The proposed approach for the generation of efficient error detection predicates is a four stage process. In the first stage, fault injection analysis is performed on a target software module in order to generate data logs pertaining to system state that can be used to learn error detection predicates. In the second stage, an appropriate data mining algorithm is selected and data preprocessing is per-

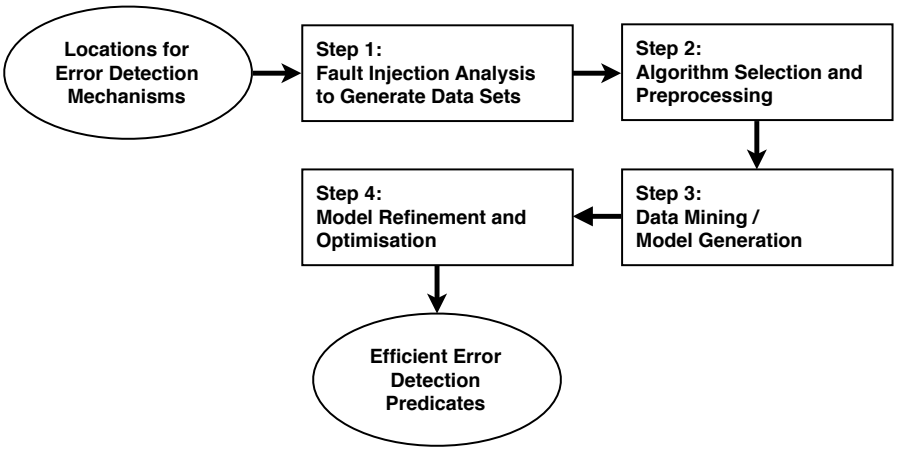

Figure 5.1: An overview of efficient error detection predicate generation

formed on the fault injection logs. The goals of preprocessing are to (i) transform the format of the fault injection data for analysis, (ii) address the class imbalance that is prevalent in fault injection data sets, e.g., using the techniques identified in Section 5.2.3, and (iii) perform any operations that are know to specifically improve the effectiveness of the adopted data mining algorithm. In the third stage, the selected data mining algorithm is used to analyse the transformed fault injection data set in order to generate and evaluate first-attempt error detection predicates. To improve the efficiency properties, i.e., accuracy and completeness of the derived predicates, the final step of the approach is to vary the parameters associated with the selected learning algorithm in search of improved detection efficiency. The four stages of the error detection predicate generation process are depicted in Figure 6.1, with detailed description of each step being provided in Sections 5.3.1-5.3.4.

5.3.1 Step 1: Dataset Generation

The first step of the proposed approach is to perform fault injection analysis on a target system in order to generate fault injection data sets which capture aspects of the relationship between system state and system failure. The specific nature of the fault injection performed will depend on the adopted fault and system

models, which will in-turn depend on the characteristics and requirements of the target software system. It should be noted that there will be a direct relationship between the nature of the fault and system models adopted and the nature of the predicates that can be derived. For example, in this thesis a transient, single bit-flip fault model is assumed, which means that the set of system states from which a relationship to system behaviour can be discerned is constrained. System states not captured by the adopted fault model will not necessarily be accounted for by the generated error detection predicates, which means that the representativeness of the adopted models and test cases is, as always in fault injection analysis, of utmost importance if the results generated are to be relevant and useful in dependability enhancement. A further consideration that must be made when performing fault injection in order to derive data sets for the generation of error detection predicates is the code location at which program state is sampled, as this will determine the code location at which the generated predicate will be relevant and, hence, the code location at which the associated EDM will be effective. In practice this means that the code location where program state is sampled should correspond with the location where an EDM is to be located. Further, extent of system state observed during sampling will govern the set of program variables that can be captured by a generated error detection predicate and, hence, the efficiency of that predicate. The results presented in this chapter are based on sampling all in-scope system state for a given code location, i.e., sampling all variables in scope at a given code location.

5.3.2 Step 2: Data Preprocessing

Following the compilation of fault injection data, an appropriate data mining algorithm must be selected for data analysis. To derive first-order predicates over the program variables whose values were captured during fault injection analysis, the use of symbolic pattern learning algorithms, such as rule induction or decision tree induction, is advocated. This use of symbolic pattern learning algorithms is advocated for the generation of error detection predicate because

these algorithms learn concepts, such as application-specific system failure, by constructing a predicate-like structure, such as a decision tree, that describes a class of objects in a manner that can be easily interpreted as a first-order predicate. Following the identification of an appropriate data mining algorithm, the data set collected during fault injection analysis may be preprocessed in order to maximise the likelihood that an efficient error detection predicate will be generated. In general, the motivations for this process are threefold:

- To transform the format of the data set derived from fault injection analysis for processing by the selected data mining algorithm.
- To address the issues of the class imbalance that is prevalent in data sets obtained through fault injection analysis.
- To perform any and all operations that may be required to improve the effectiveness of the adopted data mining algorithm.

The transformation of fault injection data to a format that is compatible with the adopted data mining analysis software will be specific to the adopted fault injection tool and data mining suite or algorithm. In the case of the results presented in this chapter, the format transformation was between the logging format of PROPANE [63] and the Attribute-Relation File Format (ARFF) used by the Weka Data Mining suite [58].

An imbalance in class distribution, i.e., a skewed distribution of positive and negative instances, is common in fault injection data sets, due to the factors such as the inherent resilience of software and the difficulty in inducing system failures under a given fault model. In order for effective predicates to be generated this imbalance must be addressed through approaches such as undersampling and oversampling with replacement for the minority class. Oversampling can be viewed as a case of SMOTE [26]. In SMOTE, synthetic examples are generated from positive instances in the training data set, t_{i+}. These positive instances are known as seed instances, as they are used to generate new, synthetic, instances. This process occurs as follows. First, the k nearest neighbours, n_{it}'s of t_{i+}

are retrieved. Next, r of these nearest neighbours are chosen through sampling by replacement, where r is the number of synthetic examples that each of the positive training instances will contribute to the new oversampled training data set. For example, if 300% oversampling is to be carried out then $r = 3$. The synthetic data instance s_{ij} is then generated as shown in Equation 5.9, where q is a random number between 0 and 1. Oversampling with replacement is a case of SMOTE where $q = 0$.

$$\vec{s_{ij}} = \vec{t_{i+}} + q.(\vec{n_{ij}} - \vec{t_{i+}}) \tag{5.9}$$

The skewed nature of data sets generated by fault injection analysis, particularly when using a transient data value fault model, means that it is appropriate, when using certain algorithms, to perform some form of attribute transformation before learning begins. For example, when the intention is to use data mining algorithms, such as naïve Bayes or logistic regression, to generate error detection predicates, mapping the original attribute values using the logarithm function shown in Equation 5.10 can improve model quality.

$$g(x_i) = \begin{cases} \log(x_i + 1) & \text{if } x_i \geq 0 \\ -\log(|x_i| + 1) & \text{if } x_i < 0 \end{cases} \tag{5.10}$$

In practice the three stated aims of data preprocessing may not be fully realised at this stage. For example, the transformation of data formats and the learning enhancement techniques are likely to be simple processes that can be contained to the preprocessing stage. However, the task of addressing class imbalance can not completed until data mining has been used to generate some initial model, hence it is an aim that is only realised during the optimisation of the generated predicates, as described in Section 5.3.4.

5.3.3 Step 3: Model Generation

The aim of the third stage of the approach is to generate first-attempt error detection predicates from the transformed fault injection data. To do this a baseline configuration of the data mining algorithm selected in the previous stage is applied the transformed data sets. At this stage the aim is not necessarily to generate highly-efficient error detection predicates, but to establish a baseline model that can be optimised and refined in the next stage of the approach.

The evaluation of the generated error detection predicates may take place by equipping the relevant location in the target system with a runtime assertion that implements the corresponding predicates or by evaluating the effectiveness with which predicates classify unseen instances, i.e., instances not used in error detection predicate generation. In either case, the aim is to evaluate the effectiveness of the predicate on previously unseen data in order to measure its efficiency properties.

5.3.4 Step 4: Model Refinement

Once a baseline predicate has been generated and evaluated, it may be refined in order to improve its level of accuracy and completeness. This can be achieved by varying the parameters associated with the configuration of the adopted learning algorithm. In particular, it is useful to vary the levels of undersampling and oversampling, including the levels and number of nearest neighbours used by any sampling techniques applied, in order to establish an algorithm configuration which yields the most efficient error detection predicates.

It is possible to generate an error detection predicate for a location that will yield a perfect EDM, i.e., a predicate that is both accurate and complete for a given code location. However, due to theoretical constraints, this is not always achievable [76]. In reality it may be the case that an error detection predicate for a given location can not be optimised beyond a certain level of efficiency. Hence, when evaluating refined error detection predicates, it should

be remembered that achieving a perfect TPR, FPR, and hence AUC, may not be possible. Note that this is not a direct limitation of the proposed approach, rather it is an established theoretical limitation of any approach that addresses the EDM design problem for real-world, infinite-state software systems.

5.4 Case Studies

To demonstrate that the proposed approach for the generation of error detection predicates yields efficient error detection predicates, the results of applying each stage of the approach are presented in Sections 5.4.1-5.4.4.

5.4.1 Step 1: Data Set Generation

In order to generate data sets, fault injection analysis was conducted on all target systems under the experimental conditions described in Chapter 3. During fault injection it is possible to inject at the specific code location and then record the state at any subsequent code location. Broadly, the code location at which an injection is performed will govern the set of erroneous states explored, whilst the code location at which program state is recorded is relevant to the location of an EDM. The code locations selected for EDM deployment, i.e., the input to the approach as depicted in Figure 6.1, were chosen based on the need to identify preconditions and postconditions for the execution of instrumented modules. This meant that entry and exit points of each modules were used as code locations for fault injection and program state recording. As illustrated in in Figure 5.2, the fact that a fault injection must be performed before system state is recorded meant that three fault injection data sets were generated for each instrumented module. A description of the data sets, as characterised by injection location and sample location, i.e., whether an entry point or exit point was used for each, is shown in Table 5.2. The results of fault injection analysis were stored in the PROPANE analysis and logging format [63].

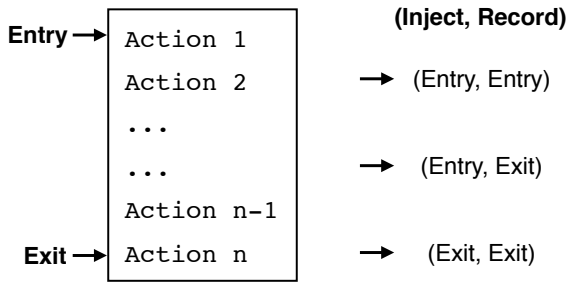

Figure 5.2: An overview of instrumentation for data set generation

```
@RELATION newRelation

@ATTRIBUTE var1 NUMERIC
@ATTRIBUTE var2 NUMERIC
@ATTRIBUTE failure {TRUE,FALSE}

@DATA
32, 3.14    FALSE
33, 3.14    FALSE
34, 3.14    FALSE
```

Figure 5.3: An overview of instrumentation for data set generation

5.4.2 Step 2: Data Preprocessing

A purpose-built tool was used to convert from the PROPANE analysis and logging format to the ARFF format used by the Weka Data Mining Suite [58]. An example of ARFF format, consisting of two program variables and three experiments, is shown is Figure 5.3.

Naïve Bayes: The naïve Bayes classification algorithm estimates the prior probability distribution of the classes, i.e., failure and non-failure in the case of learning error detection predicates, and the class conditional probabilities of input vectors. It assumes conditional independence of input variables given the class. Given an input vector, x, it assigns the class label that has the maximal posterior probability, as shown in Equation 5.11.

$$c_i = \arg\max_{c_i} p(c_i|x) = \arg\max_{c_i} \frac{\prod_{j=1}^{n} p(x_i|c_i)p(c_i)}{\sum_{k} \prod_{j=1}^{n} p(x_j|c_k)p(c_k)} \tag{5.11}$$

Table 5.2: Fault injection location-sample information for all data sets

Software Module	Data Set	Fault Injection Location	State Recording Location
7Z1	7Z1-A	Entry	Entry
	7Z1-B	Entry	Exit
	7Z1-C	Exit	Exit
7Z2	7Z2-A	Entry	Entry
	7Z2-B	Entry	Exit
	7Z2-C	Exit	Exit
7Z3	7Z3-A	Entry	Entry
	7Z3-B	Entry	Exit
	7Z3-C	Exit	Exit
FG1	FG1-A	Entry	Entry
	FG1-B	Entry	Exit
	FG1-C	Exit	Exit
FG2	FG2-A	Entry	Entry
	FG2-B	Entry	Exit
	FG2-C	Exit	Exit
FG3	FG3-A	Entry	Entry
	FG3-B	Entry	Exit
	FG3-C	Exit	Exit
MG1	MG1-A	Entry	Entry
	MG1-B	Entry	Exit
	MG1-C	Exit	Exit
MG2	MG2-A	Entry	Entry
	MG2-B	Entry	Exit
	MG2-C	Exit	Exit
MG3	MG3-A	Entry	Entry
	MG3-B	Entry	Exit
	MG3-C	Exit	Exit

In the case of continuous input attributes, kernel density estimation is used to estimate the class conditional probability density functions as opposed to the common assumption of a single Gaussian distribution. The implementation of the naïve Bayes classification algorithm used to generated the results presented employs the gaussian kernel, g, as shown in Equation 5.12

$$p(X_i = x|c_j) = \frac{1}{n}\sum_k g(x; x_k, \sigma_j) \tag{5.12}$$

The naïve Bayes classifier implementation associated with the results presented is based on the classification process described in [81].

Logistic Regression: As opposed to the naïve Bayes classification algorithm, which uses Bayes' rule to estimate posterior probabilities of the class labels given an input vector, logistic regression assumes a parametric form for the distribution $p(c_j|X_i)$ directly. Specifically, for concept learning Equations 5.13 and 5.14 are assumed.

$$P(c|x) = \frac{1}{1 + \exp(w_0 + \sum_{i=1}^{n} w_i x_i)} \tag{5.13}$$

$$P(\neg c|x) = \frac{\exp(w_0 + \sum_{i=1}^{n} w_i x_i)}{1 + \exp(w_0 + \sum_{i=1}^{n} w_i x_i)} \tag{5.14}$$

The parameterisation of the a posteriori class probabilities in Equations 5.13 and 5.14 results in a simple linear decision boundary where an instance is defined as belonging to the concept if Equation 5.15 is satisfied, where the w_i parameters are chosen to maximise the conditional log-likelihood.

$$w_0 + \sum_{i=1}^{n} w_i x_i > 0 \tag{5.15}$$

The implementation of the logistic regression-based classifier associated with the presented results is based on the model described in [95].

Decision Tree Induction: Decision tree induction is a data mining algorithm that learns a disjunction of conjunctive rules describing a concept. A decision tree consists of two types of nodes, decision nodes and leaf nodes. A decision node contains an input attribute value, and each edge emanating from the decision node is labelled with one of the unique values in the domain of the attribute labelling the decision node. A leaf node is labelled using one of the classification labels. Each path of the tree from the root node to a leaf node is interpreted

as a set of conjunctive expressions that lead to the classification label at the associated leaf node.

The decision tree induction algorithm works by performing a greedy search of the space of all possible trees, choosing decision node attributes that maximise the reduction in entropy of the class label at each stage. In other words, at each stage the decision tree induction algorithm selects the attribute that provides the most information with respect to the class label and uses it to label a decision node. The C4.5 decision tree induction algorithm was used to construct the decision trees that represent error detection predicates [128].

Rule Induction: Rule induction is a desirable approach to learning because the knowledge generated is a set of conjunctive rules that are easy to understand and have a straightforward mapping to first-order logic [127] [129]. The RIPPER algorithm is used as the rule induction implementation for the results presented in this chapter [36]. The RIPPER algorithm is an enhancement of the incremental reduced error pruning (IREP) rule-learning algorithm [52]. Similarly, IREP is based on reduced error pruning (REP), an established pruning technique that has been shown to function effectively when used in rule learning systems [22].

The REP algorithm works by splitting all training data into two sets, a growing set and pruning set. An initial set of over-fitted rules is derived using a heuristic method, before the rule set is iteratively simplified by applying pruning operators, where these operators typically delete a rule-part or delete a rule. At each iteration the pruning operator that most significantly reduces the error on the pruning set is applied. This pruning terminates only when the application of any pruning operator would not reduce the error on the pruning set. In contrast to REP, the IREP rule-learning algorithm constructs a rule set, one rule at a rime, using a greedy choice. When a rule is identified, all instances, both positive or negative, captured by the rule are removed. This process is repeated until the error rate meets a specified threshold or no positive instances

remain. The key difference between IREP and RIPPER is that each identified rule is optimised immediately following identification. That is, RIPPER builds and optimises one rule at a time and post prunes to improve accuracy.

5.4.3 Step 3: Model Generation

Following the application of the four data mining algorithms to each generated fault injection data set, 10-fold cross validation was used in order to generate the confusion matrix for each algorithm on each data set. In 10-fold cross validation the entries in each data set are partitioned into 10 stratified samples, then for each cross validation run, one of these partitions is used as a test sample, whilst the other nine are used as the training set for a particular data mining algorithm.

Tables 5.3-5.6 summarise the results of applying the naïve Bayes, logistic regression, decision tree induction and rule induction data mining algorithms to each fault injection data set. The statistics shown in these tables relate to error detection predicates generated using a baseline configuration of each data mining algorithm, i.e., no attempt was made to search for algorithm parameters that would yield the most effective predicates. In these table, the *FPR* and *TPR* columns give the mean false positive and true positive rates taken across all 10 cross validations. A false positive here corresponds to the situation where a model incorrectly detects a state as being failure-inducing, whilst a true positive corresponds to a model correctly identifying a failure-inducing state. The AUC column shows the area under the ROC curve, as described in Section 5.2.2, whilst the *SD* column gives the standard deviation in AUC across all 10 cross validations.

The results shown in Table 5.3 relate to error detection predicates generated by the naïve Bayes algorithm. Observe that the predicates generated for each data set have varied TPR values, with entries in Table 5.3 being in the range 0.78996 to 0.98797. The value of mean FPR for naïve Bayes are equally diverse across different data sets, with these values being in the range 0.00423 to 0.14218. In general, the TPR and FPR values shown in Table 5.3 mean that

Table 5.3: Predicate efficiencies for naïve Bayes with no sampling

Data Set	TPR	FPR	AUC	SD
7Z1-A	0.78996	0.05100	0.86948	0.02049
7Z1-B	0.82856	0.01811	0.90522	0.03782
7Z1-C	0.96218	0.02446	0.96886	0.02387
7Z2-A	0.94278	0.09042	0.92618	0.00707
7Z2-B	0.94822	0.09940	0.92441	0.03225
7Z2-C	0.92569	0.11010	0.90780	0.01612
7Z3-A	0.92819	0.00711	0.96054	0.04817
7Z3-B	0.97607	0.01881	0.97863	0.05119
7Z3-C	0.81070	0.00423	0.90324	0.00548
FG1-A	0.98610	0.09512	0.94549	0.01414
FG1-B	0.96755	0.11886	0.92435	0.00548
FG1-C	0.89957	0.14218	0.87869	0.02345
FG2-A	0.85122	0.04435	0.90344	0.01000
FG2-B	0.88874	0.05842	0.91516	0.01581
FG2-C	0.86244	0.04684	0.90780	0.01000
FG3-A	0.84521	0.10611	0.86955	0.02280
FG3-B	0.93145	0.01092	0.96027	0.04483
FG3-C	0.83131	0.09440	0.86846	0.04940
MG1-A	0.85461	0.02563	0.91449	0.00316
MG1-B	0.85812	0.02783	0.91515	0.08509
MG1-C	0.87785	0.10835	0.88475	0.02121
MG2-A	0.89966	0.10882	0.89542	0.04712
MG2-B	0.96666	0.11393	0.92636	0.02236
MG2-C	0.84404	0.13106	0.85649	0.04604
MG3-A	0.98797	0.05419	0.96689	0.04472
MG3-B	0.81853	0.04340	0.88757	0.00316
MG3-C	0.88492	0.03537	0.92477	0.04743
	0.8951	0.0663	0.9144	0.0281

the worst performing predicates generated by naïve Bayes may not have the levels of efficiency that are required in the context of dependable software. In contrast, the best performing of these predicates may be useful in the design of dependable software. For example, the predicates associated with *7Z3-B* have a TRP and FRP of 0.97607 and 0.01881 respectively, yielding a promising AUC of 0.97863. Perhaps the most interesting characteristic of the results presented in Table 5.3 is the consistently low standard deviation in mean AUC, which indicates that high levels of detection efficiency, i.e., TPR and FPR rates, were

Table 5.4: Predicate efficiencies for logistic regression with no sampling

Data Set	TPR	FPR	AUC	SD
7Z1-A	0.86935	0.01328	0.92804	0.10025
7Z1-B	0.83851	0.03274	0.90289	0.02915
7Z1-C	0.85139	0.03743	0.90698	0.30232
7Z2-A	0.87851	0.044711	0.91690	0.02000
7Z2-B	0.85862	0.053304	0.90266	0.03240
7Z2-C	0.87938	0.043068	0.91816	0.00632
7Z3-A	0.82177	0.09813	0.86182	0.15103
7Z3-B	0.88461	0.09631	0.89415	0.04593
7Z3-C	0.92025	0.02983	0.94521	0.03332
FG1-A	0.94544	0.08762	0.92891	0.00707
FG1-B	0.92391	0.08175	0.92108	0.00548
FG1-C	0.78287	0.18602	0.79843	0.15388
FG2-A	0.90671	0.03645	0.93513	0.02608
FG2-B	0.89962	0.04620	0.92671	0.01871
FG2-C	0.91344	0.04210	0.93567	0.00632
FG3-A	0.93917	0.00869	0.96524	0.01049
FG3-B	0.97946	0.09830	0.94058	0.03000
FG3-C	0.88357	0.04952	0.91703	0.03302
MG1-A	0.85564	0.05326	0.90119	0.04785
MG1-B	0.88434	0.07482	0.90476	0.00775
MG1-C	0.88471	0.06335	0.91068	0.00632
MG2-A	0.95960	0.02203	0.96879	0.00775
MG2-B	0.87028	0.07887	0.89571	0.00949
MG2-C	0.88397	0.08470	0.89964	0.17745
MG3-A	0.95524	0.03913	0.95806	0.00949
MG3-B	0.88327	0.03726	0.92300	0.00707
MG3-C	0.90740	0.08961	0.90890	0.00775
	0.8911	0.0603	0.9154	0.0479

consistently achieved during each of the 10 cross validations.

The results presented in Table 5.4 demonstrate that the error detection predicates generated by the logistic regression classifier are comparable with, but marginally less efficient than, those generated using naïve Bayes, with all mean AUC values being in the range 0.79843 to 0.96879. Indeed, the naïve Bayes classifier surpassed the logistic regression classifier, with respect to mean AUC, for all but 7 of the data sets. Logistic regression also yielded the worst results for a single data set, with *FG1-C* having a TRP and FPR of 0.78287 and 0.18602

Table 5.5: Predicate efficiencies for rule induction with no sampling

Data Set	TPR	FPR	AUC	SD
7Z1-A	0.96456	0.00157	0.98150	0.00010
7Z1-B	0.94018	0.03494	0.95262	0.00008
7Z1-C	0.94009	0.02147	0.95931	0.00014
7Z2-A	0.94648	0.04554	0.95047	0.00063
7Z2-B	0.91891	0.02253	0.94819	0.00447
7Z2-C	0.92471	0.02635	0.94918	0.00141
7Z3-A	0.93912	0.07671	0.93120	0.00026
7Z3-B	0.92937	0.09719	0.91609	0.00006
7Z3-C	0.90103	0.06047	0.92028	0.00045
FG1-A	0.94151	0.09568	0.92291	0.00045
FG1-B	0.98804	0.01560	0.98622	0.00028
FG1-C	0.92306	0.04026	0.94140	0.00045
FG2-A	0.97342	0.03998	0.96672	0.00200
FG2-B	0.97722	0.02533	0.97595	0.00084
FG2-C	0.98387	0.03895	0.97246	0.00063
FG3-A	0.94786	0.00033	0.99376	<0.00000
FG3-B	0.98716	0.02340	0.97188	0.00045
FG3-C	0.95952	0.09413	0.93269	0.00045
MG1-A	0.95481	0.00922	0.97280	0.00632
MG1-B	0.93365	0.02219	0.95573	0.00837
MG1-C	0.92935	0.02006	0.95465	0.00224
MG2-A	0.96559	0.00332	0.98114	0.00004
MG2-B	0.97930	0.09288	0.94321	0.00032
MG2-C	0.99393	0.07091	0.96151	0.00014
MG3-A	0.90587	0.04206	0.93190	0.00024
MG3-B	0.93431	0.05055	0.94188	0.00004
MG3-C	0.94874	0.04487	0.95193	0.00022
	0.9493	0.0414	0.9544	0.0012

respectively. Interestingly, the standard deviation in mean AUC remains consistently low, again indicating the consistency with which similarly efficient error detection predicates are generated during cross validation.

The results presented in Table 5.5 indicate that the error detection predicates generated using rule induction surpass those generated under naïve Bayes and logistic regression with respect to the level of efficiency achieved, with all mean AUC values in Table 5.5 being in the range 0.91609 to 0.99376. The standard deviation in AUC is also markedly lower than for the naïve Bayes and logistic

Table 5.6: Predicate efficiencies for decision tree induction with no sampling

Data Set	TPR	FPR	AUC	SD
7Z1-A	0.94347	0.00012	0.97168	0.01732
7Z1-B	0.96912	0	0.98456	0.00003
7Z1-C	0.96541	0	0.98271	0.00003
7Z2-A	0.99001	0.00048	0.99477	0.00020
7Z2-B	0.98922	0.00201	0.99361	<0.00000
7Z2-C	0.99111	0.00181	0.99465	0.00014
7Z3-A	0.99792	0.00002	0.99895	0.00017
7Z3-B	0.99792	0	0.99896	0.00010
7Z3-C	0.99868	0	0.99934	0.00010
FG1-A	0.79282	0.00011	0.89636	<0.00000
FG1-B	0.95842	0.00001	0.97920	0.00100
FG1-C	0.82232	0.00014	0.91109	0.00024
FG2-A	0.98622	0.00010	0.99306	0.00017
FG2-B	0.99218	0.00021	0.99599	0.00001
FG2-C	0.98108	0	0.99054	0.00008
FG3-A	0.99063	0.00021	0.99521	0.00026
FG3-B	0.98071	0.00317	0.98877	0.00173
FG3-C	0.98780	0.00060	0.99360	0.00173
MG1-A	0.97922	0.00092	0.98915	0.00014
MG1-B	0.98084	0.00010	0.99037	0.00008
MG1-C	0.97990	0.00210	0.98890	<0.00000
MG2-A	0.97404	0	0.98702	<0.00000
MG2-B	0.97404	0	0.98702	<0.00000
MG2-C	0.97280	0	0.98640	<0.00000
MG3-A	0.99381	0	0.99691	0.00003
MG3-B	0.99381	0.00032	0.99675	0.00026
MG3-C	0.99890	0	0.99945	<0.00000
	0.9697	0.0005	0.9846	0.0009

regression classifiers, with the highest observed standard deviation being less than the lowest value associated with naïve Bayes and logistic regression. In general, the results associated with rule induction are promising with respect to the generation of efficient error detection predicates, not least because these results relate to a baseline configuration of the rule induction algorithm.

Table 5.6 suggests that decision tree induction is the most effective of the data mining algorithms applied to this point. Observe from Table 5.6 that the mean AUC of all baseline predicates generated through decision tree induction

is greater than 0.97168. As this measure reflects both FPR and TPR, this is an indication that the predicates generated are effective classifiers for failure inducing states. Observe also that, aside from data sets *FG1-A* and *FG1-C*, the mean TPR for all predicates is greater than 0.94347, with the maximum observed being 0.99890. Further, the mean FPR is extremely low in all cases, with the maximum observed value being 0.00317. This indicates the discriminatory nature of the predicates generated by the decision tree induction algorithm. It is also interesting to note that the standard deviation of the predicates generated, regardless of the data mining algorithm applied, is consistently low, which demonstrates the consistency with which efficient predicates can be generated when using a decision tree induction-based approach.

Interestingly, despite the differing levels of efficiency achieved by the data mining algorithms applied, the results presented for each algorithm generally outperform existing approaches to the error detection predicate design. Indeed, in [75] it was shown that predicates, realised as executable assertions, designed on the basis of a system specification and domain knowledge could have an accuracy as low as 0.75 and a completeness as low as 0.75. Further, it was shown in [159] that, when component replication and repeated execution was employed for error detection and correction, the number of transient value failures could only be reduced to a minimum of 3%. In most cases, these levels of efficiency are surpassed by the predicates evaluated in Tables 5.3-5.6, each of which was generated under a baseline configuration. Moreover, even under such a baseline configuration, the levels of efficiency achieved by the error detection predicates generated by rule induction and decision tree induction, i.e., the advocated symbolic pattern learning algorithms, are appropriate for use in dependable software systems, i.e., they have high accuracy and completeness.

5.4.4 Step 4: Model Refinement

Having generated and evaluated a set of baseline error detection predicates, these predicates can now be refined by varying the parameters associated with

the applied data mining algorithms. In particular, it is interesting to vary parameters that are independent of any data mining algorithm, such as data set sampling levels prior to learning. This allow the same refinement process to be applied regardless of the selected data mining algorithm.

The results of model refinement process for the presented case studies are summarised in Tables 5.7-5.10. The columns of Tables 5.7-5.10 are identical to those as those given in Tables 5.3-5.5, except for the Sampling and N columns, which show the sampling level and the number of nearest neighbours used in sampling to generate the associated predicates respectively. Each entry in the *Sampling* column also shows the type of sampling performed, where an O indicates oversampling and a U indicates undersampling. A total of 20 undersampling and 15 oversampling percentage levels were used in model refinement. These levels were uniformly distributed over [5,100] and [100,1500] for undersampling and oversampling respectively, giving increments of 5 and 100 respectively. The number of nearest neighbours considered in the sampling process were uniformly distributed over [1,15] with increments of 1. The values in the *Sampling* and N columns of Tables 5.7-5.10 represent optimal observed values, with regard to achieved AUC, across all candidate values considered.

All entries in Table 5.7 improve on the results presented for the naïve Bayes classifier in Table 5.3, clearly indicating that varying the sampling parameters associated with the application of naïve Bayes can improve the efficiency of the error detection predicates generated. More specifically, all mean TPR and FPR values have been improved, which lead to a increase in mean AUC. The standard deviation in AUC is consistently low and remains comparable with the results generated under a baseline configuration of the naïve Bayes classifier.

Similar to the results derived from varying the sampling parameters of the naïve Bayes classifier, Table 5.8 indicates that varying the parameters associated the logistic regression classifier yields a universal improvement in the level of efficiency that can be achieved by generated predicates. Again, the mean AUC is improved and the associated standard deviation remains low in all cases.

Table 5.7: Predicate efficiencies for naïve Bayes with sampling

Data Set	Sampling	N	TPR	FPR	AUC	SD
7Z1-A	200 (O)	8	0.85697	0.00375	0.92661	0.08473
7Z1-B	900 (O)	-	0.90012	0.00680	0.94666	0.02646
7Z1-C	400 (O)	-	0.96784	0.00033	0.98376	0.05657
7Z2-A	1100 (O)	-	0.94912	0.07931	0.93491	0.02145
7Z2-B	1200 (O)	4	0.95865	0.08899	0.93483	0.03619
7Z2-C	700 (O)	-	0.94933	0.07554	0.93690	0.03317
7Z3-A	200 (O)	4	0.96851	0.00037	0.98407	0.24958
7Z3-B	100 (O)	2	0.99043	0.00339	0.99352	0.08497
7Z3-C	500 (O)	2	0.83297	0.00257	0.91520	0.18163
FG1-A	200 (O)	-	0.99118	0.07037	0.96041	0.00775
FG1-B	300 (O)	2	0.98234	0.00000	0.99117	0.14856
FG1-C	35 (U)	-	0.94226	0.10326	0.91950	0.00775
FG2-A	200 (O)	6	0.87420	0.03232	0.92094	0.07450
FG2-B	1300 (O)	4	0.89994	0.03893	0.93051	0.07280
FG2-C	100 (O)	11	0.89825	0.03089	0.93368	0.03146
FG3-A	500 (O)	9	0.87823	0.00000	0.93911	0.04919
FG3-B	600 (O)	3	0.97470	0.01004	0.98233	0.04506
FG3-C	20 (U)	-	0.83824	0.03941	0.89942	0.01844
MG1-A	500 (O)	12	0.86274	0.02019	0.92128	0.04405
MG1-B	200 (O)	10	0.96827	0.02000	0.97414	0.01703
MG1-C	1100 (O)	2	0.85213	0.03607	0.90803	0.09808
MG2-A	1100 (O)	5	0.95668	0.04734	0.95467	0.04147
MG2-B	600 (O)	3	1	0.00576	0.99712	0.08556
MG2-C	100 (O)	9	0.88544	0.11873	0.88336	0.06411
MG3-A	900 (O)	-	0.99620	0.00153	0.99734	0.08485
MG3-B	800 (O)	-	0.85319	0.00119	0.92600	0.09690
MG3-C	200 (O)	-	0.91393	0.01157	0.95118	0.06542
			0.9238	0.0314	0.9462	0.0677

A notable improvement can be seen in the predicates associated with *FG1-C*, where the mean TRP and FPR have increased to 0.85503 and 0.005 respectively. This is a particularly notable improvement, as the value of TPR and FPR were previously 0.78287 and 0.18602 respectively, which resulted in the lowest mean AUC observed for any baseline configuration of any data mining algorithm employed.

The results presented in Table 5.9 demonstrate that the refinement process has improved the efficiency properties of the error detection predicates gener-

Table 5.8: Predicate efficiencies for logistic regression with sampling

Data Set	Sampling	N	TPR	FPR	AUC	SD
7Z1-A	900 (O)	14	0.89954	0.03100	0.94094	0.08136
7Z1-B	500 (O)	2	0.90605	0.04017	0.93233	0.04817
7Z1-C	500 (O)	12	0.88466	0.00620	0.93893	0.03082
7Z2-A	200 (O)	-	0.88193	0.00404	0.93895	0.00949
7Z2-B	100 (O)	-	0.88843	0.00209	0.94317	0.01095
7Z2-C	900 (O)	2	0.90612	0.00099	0.95257	0.09586
7Z3-A	300 (O)	3	0.89451	0.00044	0.94704	0.01549
7Z3-B	800 (O)	-	0.90967	0.00237	0.95365	0.04336
7Z3-C	100 (O)	2	0.92072	0.00073	0.96000	0.17378
FG1-A	100 (O)	8	0.94857	0.00525	0.97166	0.09143
FG1-B	200 (O)	-	0.98993	0.04967	0.97013	0.01789
FG1-C	800 (O)	-	0.85503	0.00538	0.92483	0.08894
FG2-A	400 (O)	-	0.91415	0.02554	0.94431	0.02098
FG2-B	200 (O)	5	0.90290	0.02712	0.93789	0.02881
FG2-C	100 (O)	3	0.92941	0.02920	0.95011	0.04669
FG3-A	1000 (O)	-	0.94612	0.01542	0.96535	0.09965
FG3-B	500 (O)	12	0.98173	0.00111	0.99031	0.02983
FG3-C	700 (O)	11	0.89350	0.00049	0.94651	0.04817
MG1-A	400 (O)	-	0.85906	0.04914	0.90496	0.08781
MG1-B	1300 (O)	4	0.90654	0.06572	0.92041	0.08025
MG1-C	800 (O)	-	0.89453	0.06003	0.91725	0.02864
MG2-A	35 (U)	-	0.96054	0.00001	0.98027	0.08136
MG2-B	900 (O)	6	0.87469	0.00629	0.93420	0.06519
MG2-C	900 (O)	7	0.93115	0.00039	0.96538	0.03362
MG3-A	500 (O)	7	0.96200	0.00001	0.98100	0.09418
MG3-B	200 (O)	11	0.89273	0.00463	0.94405	0.05000
MG3-C	100 (O)	10	0.91559	0.00350	0.95605	0.04940
			0.9130	0.0162	0.9486	0.0575

ated using rule induction. Indeed, the results show an improved mean AUC in all cases, though in several cases this improvement is less than a 0.000001 increase. The standard deviation in mean AUC is easily comparable with standard deviation observed under a baseline configuration of rule induction, with some generated predicates even yielding a reduction in standard deviation with respect to mean AUC.

Despite being the best performing algorithm under a baseline configuration, the entries in Table 5.10 show consistent improvements, with respect to the mean

Table 5.9: Predicate efficiencies for rule induction with sampling

Data Set	Sampling	N	TPR	FPR	AUC	SD
7Z1-A	100 (O)	2	0.96456	0.00157	0.98150	0.02646
7Z1-B	1200 (O)	-	1	0.03494	0.98253	0.00245
7Z1-C	900 (O)	-	0.94009	0.02147	0.95931	0.02449
7Z2-A	400 (O)	4	0.98554	0.01241	0.98657	0.00316
7Z2-B	800 (O)	-	0.99521	0.00470	0.99526	0.01414
7Z2-C	200 (O)	8	0.98529	0.00955	0.98787	0.01000
7Z3-A	200 (O)	3	0.93912	0.07671	0.93120	0.00245
7Z3-B	700 (O)	9	0.92937	0.09719	0.91609	0.00095
7Z3-C	100 (O)	-	0.92103	0	0.96052	0.00141
FG1-A	25 (U)	-	0.94151	0.09568	0.92291	0.02236
FG1-B	1100 (O)	8	0.99804	0.01560	0.99122	0.08367
FG1-C	200 (O)	-	0.92306	0	0.96153	0.02646
FG2-A	1100 (O)	3	0.98244	0.00420	0.98912	0.00707
FG2-B	800 (O)	-	0.98974	0.01000	0.98987	0.00141
FG2-C	200 (O)	2	0.99877	0.00711	0.99583	0.00141
FG3-A	1100 (O)	5	0.98786	0.00033	0.99376	0.00894
FG3-B	400 (O)	3	0.96716	0.02340	0.97188	0.07746
FG3-C	300 (O)	8	0.95952	0.09413	0.93269	0.00089
MG1-A	100 (O)	-	0.97130	0.00001	0.98565	0.00632
MG1-B	1200 (O)	3	0.96972	0.00055	0.98459	0.00316
MG1-C	300 (O)	2	0.96818	0.00034	0.98392	0.00300
MG2-A	10 (U)	-	0.99559	0	0.99780	0.00300
MG2-B	900 (O)	2	0.97930	0.09288	0.94321	0.00020
MG2-C	200 (O)	-	0.99393	0.07091	0.96151	<0.00000
MG3-A	100 (O)	-	0.90587	0.04206	0.93190	0.00084
MG3-B	600 (O)	-	0.93431	0.05055	0.94188	0.00001
MG3-C	500 (O)	9	0.94874	0.04487	0.95193	0.03000
			0.9658	0.0300	0.9679	0.0134

AUC measure, during the error detection predicate refinement process. In some cases this improvement is small, again less than a 0.000001 increase in some cases, though in the context of an error detection mechanism this magnitude of increase can be significant. In almost all cases the standard deviation of all predicates is increased, though it should be noted that these values remain extremely low, particularly in comparison to the results shown for other data mining algorithms.

In addition to evaluating the efficiency of the error detection predicates gen-

Table 5.10: Predicate efficiencies for decision tree induction with sampling

Data Set	Sampling	N	TPR	FPR	AUC	SD
7Z1-A	300 (O)	12	0.99849	0.00100	0.99875	0.00077
7Z1-B	900 (O)	6	0.98768	0.00035	0.99367	0.00775
7Z1-C	700 (O)	7	0.99998	0.00007	0.99996	0.00017
7Z2-A	1200 (O)	4	0.99914	0.00009	0.99953	0.00028
7Z2-B	900 (O)	2	0.99901	0.00005	0.99948	0.00003
7Z2-C	500 (O)	5	0.99091	0.00006	0.99543	<0.00000
7Z3-A	85 (U)	-	0.99826	0.00002	0.99912	0.00004
7Z3-B	300 (O)	4	0.99836	0.00005	0.99916	0.00022
7Z3-C	500 (O)	14	0.99919	0	0.99960	<0.00000
FG1-A	35 (U)	-	0.79633	0.01311	0.89161	0.00447
FG1-B	500 (O)	-	0.96280	0.00024	0.98128	0.00002
FG1-C	500 (O)	-	0.82292	0.00020	0.91136	0.00002
FG2-A	100 (O)	3	0.99982	0	0.99991	0.00001
FG2-B	200 (O)	8	0.99950	0	0.99975	0.00100
FG2-C	300 (O)	7	0.99121	0.00011	0.99555	0.00100
FG3-A	500 (O)	12	0.99662	0.00111	0.99776	0.00028
FG3-B	900 (O)	1	0.99952	0.00405	0.99774	0.00010
FG3-C	500 (O)	11	0.99631	0.00151	0.99740	0.00032
MG1-A	200 (O)	2	1	0.00990	0.99505	<0.00000
MG1-B	35 (U)	2	1	0.00995	0.99503	<0.00000
MG1-C	200 (O)	-	0.99904	0.00009	0.99948	0.00004
MG2-A	30 (U)	-	0.97403	0	0.98702	<0.00000
MG2-B	5 (U)	-	0.97403	0	0.98702	<0.00000
MG2-C	5 (U)	-	0.97281	0	0.98641	<0.00000
MG3-A	100 (O)	2	0.99380	0	0.99690	<0.00000
MG3-B	40 (U)	-	0.99380	0	0.99690	<0.00000
MG3-C	5 (U)	-	0.99892	0	0.99946	<0.00000
			0.9794	0.0016	0.9889	0.0006

erated by different data mining algorithms, it is meaningful to consider the efficiency of predicates across the same fault injection and state recording locations, e.g., data sets where an module entry point was used for fault injection and state recording. As explained previously, each target software module is associated with three data sets, where each data set related to a distinct pair of fault injection and state recording locations, i.e., entry-entry, entry-exit and exit-exit data sets. Table 5.11 shows the average efficiency achieved by each data mining algorithm across all data set using the same fault injection and

Table 5.11: Fault injection location-sample information for all data sets

Locations	**Mean AUC**			
	NB	**LR**	**RI**	**DTI**
Entry-Entry	0.94882	0.95272	0.96893	0.98507
Entry-Exit	0.96403	0.94735	0.96850	0.99444
Exit-Exit	0.92567	0.94574	0.96612	0.98718

state recording locations. The result presented reflect the mean AUC of error detection predicates following predicate refinement.

The results presented in Table 5.11 demonstrate that there is no clear relationship between error detection predicate efficiency and the code locations used in fault injection analysis. As can be seen from Tables 5.7-5.10, the standard deviation in AUC of the generated predicates varies across data sets. Given these observations it is clear that, whilst the results presented suggest that the data mining algorithm plays a significant role in determining the efficiency of the error detection predicates generated, no such relationship is evident regarding fault injection and state recording locations. However, more research is required in this area before general conclusions can be reached.

The application of the proposed approach has illustrated its capability to generate predicates that are efficient by design. Indeed, the levels of efficiency achieved by the derived error detection predicates make them suitable for use in dependable software systems, not least because the approach has been shown to generate predicates that are complete or accurate. The standard deviation associated with the generation of predicates has also been shown to be low, particularly when the evaluation employed used stratified samples in the cross validation process, i.e., having little or no data repetition across the training and test data should hinder the derivation of similar predicates during cross validation. Perhaps most crucially, the case studies presented have served to demonstrate that the proposed approach can be used as a systematic approach to generating error detection predicates for real-world, infinite-state software systems. Indeed, the results presented surpass those achieved where a functional specification and domain knowledge were available, as well as those where repli-

cation and repeated execution were employed to detect and tolerate transient value errors [75] [159].

5.5 Efficient Predicates and Variable Criticality

To this point a systematic approach for the generation of efficient error detection predicates has been proposed, with case studies demonstrating the application of the approach in the context of various algorithms, configurations and software systems. As well as being the first systematic approach to generation of error detection predicates for real-world, infinite-state software systems, the fact that the proposed approach is independent of the metric suite developed in Chapter 4, means that the predicates it generates may serve as a basis for assessing the capability of the importance metric to identify critical variables. In particular, it is possible to determine how critical the program variables identified by the importance metric are by evaluating the levels of efficiency that can be achieved by error detection predicates generated using only these critical variables.

5.5.1 Variable Importance and Error Detection

If efficient error detection predicates can be constructed using only critical variables, then this is a strong validation of the thesis that an efficient EDM consists of a set of critical variables, not least because it would demonstrate that there is at least one set of justifiably critical variables that facilitate efficient error detection for specified locations in a target system. To determine whether efficient error detection predicates can be constructed using the critical variables identified through the application of a threshold to the importance metric, a new set of error detection predicates for each software module are now evaluated.

Table 5.12 shows the evaluation of the error detection predicates that were generated using all variables in each software module, whilst table Table 5.13 shows the evaluation of the error detection predicates generated using only the most important 25% of the variables in each software module, i.e., a threshold

was set based on ranking position. The statistics shown in Table 5.12 are based on the error detection predicates previously generated and optimised under the most effective data mining algorithm, i.e., decision tree induction. These result have been repeated here for convenience. In Tables 5.12 and 5.13 the *FPR* and *TPR* columns give the mean false positive and true positive rates taken across all cross validations. A false positive here corresponds to the situation where a predicate incorrectly detects a state as being failure-inducing, whilst a true positive corresponds to a predicate correctly identifying a failure-inducing state. The *AUC* column shows the area under the ROC curve. To reiterate, the *AUC* column aggregates the performance of the generated predicates with respect to TPR and FPR. Again, a mean AUC close to 1 is desirable in the design of error detection predicates, though may not always be achievable due to theoretical constraints [76], whilst an AUC of 0.5 is indicative of random performance. To provide a measure of how the efficiency of the generated error detection predicates varied during cross validation, the *SD* column gives the standard deviation in AUC for 10-fold cross validation. It would be hoped that no significant increase would be observed in the *SD* column when generating error detection predicates using only important variables.

Observe from Tables 5.12 and 5.13 that the difference in the efficiency of the predicates generated using all variables and those generated using only the most important 25% of variables is small. The largest difference in AUC when comparing these results is associated with data set *7Z1-A*. For this data set the predicates generated using all variables have a mean AUC of 0.97168, whilst those generated using only important variable have a mean AUC of 0.96535, giving a difference of just 0.00633 in this worst case. Observe also that the absolute AUC values for predicates generated using important variables are consistently high, with the maximum and minimum being 0.99924 and 0.89616 respectively. These consistently high AUC values, which are indicative of high true positive and low false positive rates, serve to suggest that error detection predicates generated using important variables can safeguard the functioning of

Table 5.12: Predicate efficiencies achieved using all variables (also Table 5.6)

Data Set	**TPR**	**FPR**	**AUC**	**Var**
7Z1-A	0.94347	0.00012	0.97168	0.01732
7Z1-B	0.96912	0	0.98456	0.00003
7Z1-C	0.96541	0	0.98271	0.00003
7Z2-A	0.99001	0.00048	0.99477	0.00020
7Z2-B	0.98922	0.00201	0.99361	<0.00000
7Z2-C	0.99111	0.00181	0.99465	0.00014
7Z3-A	0.99792	0.00002	0.99895	0.00017
7Z3-B	0.99792	0	0.99896	0.00010
7Z3-C	0.99868	0	0.99934	0.00010
FG1-A	0.79282	0.00011	0.89636	<0.00000
FG1-B	0.95842	0.00001	0.97920	0.00100
FG1-C	0.82232	0.00014	0.91109	0.00024
FG2-A	0.98622	0.00010	0.99306	0.00017
FG2-B	0.99218	0.00021	0.99599	0.00001
FG2-C	0.98108	0	0.99054	0.00008
FG3-A	0.99063	0.00021	0.99521	0.00026
FG3-B	0.98071	0.00317	0.98877	0.00173
FG3-C	0.98780	0.00060	0.99360	0.00173
MG1-A	0.97922	0.00092	0.98915	0.00014
MG1-B	0.98084	0.00010	0.99037	0.00008
MG1-C	0.97990	0.00210	0.98890	<0.00000
MG2-A	0.97404	0	0.98702	<0.00000
MG2-B	0.97404	0	0.98702	<0.00000
MG2-C	0.97280	0	0.98640	<0.00000
MG3-A	0.99381	0	0.99691	0.00003
MG3-B	0.99381	0.00032	0.99675	0.00026
MG3-C	0.99890	0	0.99945	<0.00000
	0.9697	**0.0005**	**0.9846**	**0.0009**

a software system. Further, the fact that standard deviation in AUC remains low, even unchanged in many cases, when only important variables are used in the generation of error detection predicates means that efficient predicates can be consistently generated across separate cross validations. This implies that the proposed approach remains robust when using only important variables, which is particularly important given that using data sets containing fewer variables, in effect, reduces the amount of information available during the construction of error detection predicates for EDMs.

Table 5.13: Predicate efficiencies achieved using important variables

Data Set	TPR	FPR	AUC	SD
7Z1-A	0.93090	0.00020	0.96535	0.05477
7Z1-B	0.95873	0.00005	0.97934	0.00014
7Z1-C	0.96011	0	0.98006	0.00007
7Z2-A	0.98802	0.00238	0.99282	0.03162
7Z2-B	0.98765	0.00355	0.99205	0.00173
7Z2-C	0.98990	0.00202	0.99394	0.00707
7Z3-A	0.99574	0.00477	0.99549	0.00200
7Z3-B	0.99520	0.00011	0.99755	0.00141
7Z3-C	0.99361	0.00006	0.99678	0.00141
FG1-A	0.79232	0.00001	0.89616	<0.00000
FG1-B	0.95708	0.00002	0.97853	0.00100
FG1-C	0.82100	0.00024	0.91038	0.02828
FG2-A	0.98502	0.00042	0.99230	0.04472
FG2-B	0.99196	0.00342	0.99427	0.00100
FG2-C	0.98027	0.00093	0.98967	0.00283
FG3-A	0.98862	0.00003	0.99430	0.00024
FG3-B	0.97994	0.00437	0.98779	0.00141
FG3-C	0.98608	0.00501	0.99054	0.00004
MG1-A	0.97809	0.00111	0.98849	0.01414
MG1-B	0.98004	0.00084	0.98960	0.00173
MG1-C	0.97902	0.00492	0.98705	0.01000
MG2-A	0.97328	0.00003	0.98663	<0.00000
MG2-B	0.97346	0.00028	0.98659	0.00141
MG2-C	0.97261	0.00030	0.98616	0.00141
MG3-A	0.99330	0.00305	0.99513	0.00003
MG3-B	0.99368	0.00086	0.99641	0.00026
MG3-C	0.99847	0	0.99924	<0.00000
	0.9676	**0.0014**	**0.9831**	**0.0077**

5.5.2 Variable Importance and Decision Tree Depths

In order to understand why efficient error detection predicates can be constructed using only critical variables, the predicates structures generated by the decision tree induction algorithm, i.e., the data mining algorithm used to generate these efficient error detection predicates, should be analysed. As described previously, the decision tree induction algorithm performs a greedy search of the space of all possible trees choosing decision node attributes that maximise the reduction in the entropy of the class label. Figure 5.4 shows an example of the type of tree generated by the decision tree induction algorithm in the

context of generating efficient error detection predicates. The tree is based on a predicate generated during the decision tree induction experiments presented preciously. In Figure 5.4, non-leaf nodes are labelled with variables, edges are labelled with potential variable states and leaf nodes are labelled with a failure classification, where true indicates failure and false indicates non-failure. An error detection predicate is derived from the structure shown in Figure 5.4 by interpreting the tree as a conjunction of disjunctions. For example, the program variable $VarOne$ is labelling the root node and each edge emanating from it represents a set of values for $VarOne$, e.g., ≤ 43.32. Following two such these edges from the root results in a conjunctive expression being created, i.e., $(VarOne \leq 43.32) \wedge ((VarTwo > 523))$. If edges are followed from the root node to a leaf node then a complete conjunctive expression will be associated with a class label, failure or non failure in the case of this thesis, and the number of instances captured by the conjunctive expression, e.g., consistently taking the leftmost edges yields the conjunctive expression $(VarOne \leq 43.32) \wedge ((VarTwo > 523)) \wedge (VarFour > 0)$ that captures 126 instances of failure in the data set used in decision tree construction.

The premise of this analysis method is that critical variables will feature near the root, i.e., at a lower depth measured from the root, of a decision tree constructed during decision tree induction because these program variables capture the most information regarding the ultimate success of a software system execution, i.e., critical variables provide the greatest reduction in the entropy of the class label. This validity of this evaluation approach is ensured by (i) the mutual focus on failure-inducing states, (ii) the independence of the error detection predicate generation mechanism from the importance metric and (iii) the manner in which decision trees are constructed in the generation of error detection predicates.

The case studies shown in Section 5.4 used 10-fold cross validation to accurately asses the effectiveness of generated error detection mechanisms. This meant that data set was partitioned into ten stratified samples, then for each

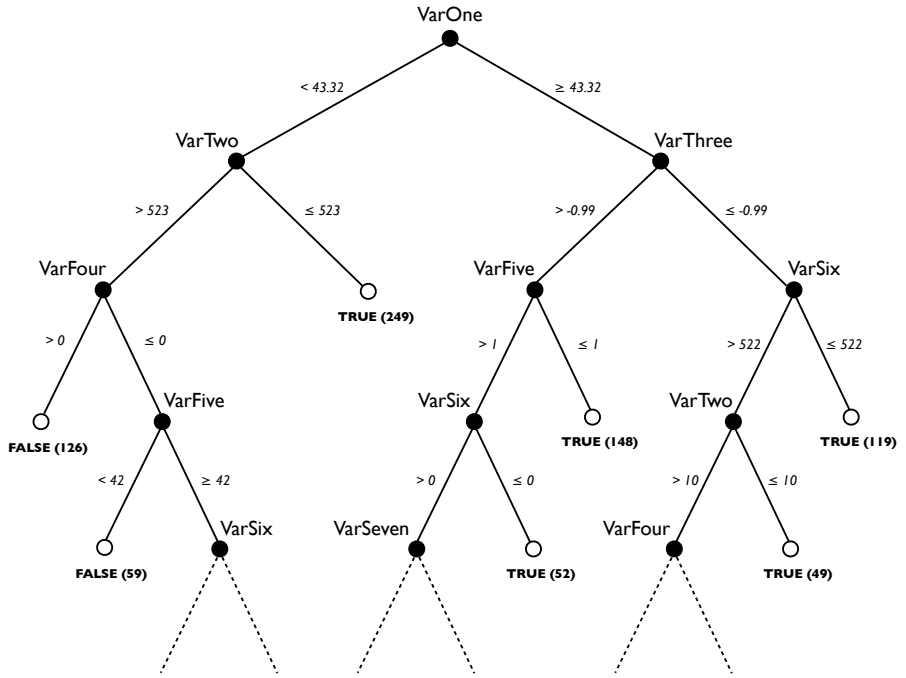

Figure 5.4: An example decision tree constructed during decision tree induction

cross validation run, one of the partitions was used as the test sample, whilst the other nine were used as the training set. This meant that three predicates were generated for each fault injection data set. As 3 fault injection data sets, corresponding to three distinct pairs of injection and sampling locations, were compiled for each software module, a total of 30 separate error detection predicates were generated for each target software module.

Tables 5.14-5.21 show the importance ranking and importance values of the ten most important variables in each instrumented module. These values were calculated using the approach described in Chapter 4 and the experimental conditions described in Chapters 3. Tables 4.1-4.8 can be used to assess the correlation between the importance of a variable and its depth in a decision tree representing a predicate, as they show the minimum tree depth at which a variable was used to label a decision node in any predicate for a given module. The root of a decision tree is assumed to have a depth of 1. The decision was taken to use this as a basis for comparison with the importance ranking because,

Table 5.14: Importance values and minimum decision tree depths for 7Z1

Importance Ranking	**Variable Identifier**	**Importance Metric**	**Min. Tree Depth**
1	processedPosition	0.002473	1
2	remainLen	0.002466	1
3	distance	0.001217	2
4	posState	0.001215	2
5	ttt	0.001213	2
6	matchByte	0.001211	3
7	probLit	0.001210	3
8	dicPos	0.001210	3
9	range	0.001207	4
10	kMatchLen	0.001206	3

at each state of decision tree construction, the decision tree induction algorithm selects a decision node attribute that maximise the reduction in entropy of the class label. In effect, the decision tree induction algorithm selects the variable whose value can be viewed as providing the most information regarding system failure. This is commensurate with the stated aims of the importance metric. To appreciate why the minimum decision tree depth is appropriate, as opposed to a measure such as average tree depth, consider Figure 5.4. $VarTwo$ is used to label decision nodes at depths of 2 and 4. In the former case, this allows 249 instances of failure to be discerned using a simple predicate, i.e., $((VarOne < 43.32) \wedge (VarTwo \leq 523))$, whilst the latter allows only 49 instances of failure to be discerned using a more complex predicate, i.e., $((VarOne \geq 43.32) \wedge (VarThree \leq -0.99) \wedge (VarSix > 522) \wedge (VarTwo \leq 10))$. The selection of a decision node attribute that maximise the reduction in entropy of the class label means that nodes at a lower depth will capture more failure information, hence the pattern in the $VarTwo$ example will always be observed. This reasoning is the basis for the decision to use minimum decision tree depth in this evaluation.

Observe from Tables 5.14-5.22 that there is a pattern between the importance ranking and the minimum tree depth of the variables in each software module. Variables with a higher importance ranking generally feature at the lower levels on the predicate structures generated during decision tree induction. Further,

Table 5.15: Importance values and minimum decision tree depths for 7Z2

Importance Ranking	**Variable Identifier**	**Importance Metric**	**Min. Tree Depth**
1	numberStreams	0.141488	1
2	highPart	0.120285	1
3	unpack	0.050787	1
4	sizeIndex	0.035976	2
5	i_unpack	0.015447	2
6	attribute	0.004906	2
7	numInStreams	0.002448	3
8	numSubstream	0.002447	3
9	unpackSize	0.002447	3
10	nextHeaderOffset	0.002447	4

Table 5.16: Importance values and minimum decision tree depths for 7Z3

Importance Ranking	**Variable Identifier**	**Importance Metric**	**Min. Tree Depth**
1	seekInStreamSint	0.036212	1
2	wMode	0.036173	2
3	res	0.022356	2
4	oSize	0.004825	3
5	moveMethod	0.004824	3
6	CFlp	0.004823	3
7	pos	0.002994	3
8	lengthR	0.001209	2
9	pHandle	0.001205	3
10	cSize	0.001205	3

it is interesting to note the relationship between the value of the importance metric and the minimum decision tree depth observed for the highest ranked variables. For example, the two highest ranked variable in Tables 5.14 and 5.19 have importance metric values that are much greater than the other variables in their respective tables. This is then mirrored in the minimum decision tree depth, which shows that these variables are the only variables that feature at the root of the decision trees generated for their respective software modules. Variables in 5.14-5.22 that do not feature in an error detection predicate, such as *inf* and *done* in Table 5.20, have a lower importance relative to other variables in their respective tables. Indeed, every variable with an importance ranking of 1-5 features in an error detection predicate. This observation demonstrates

Table 5.17: Importance values and minimum decision tree depths for FG1

Importance Ranking	**Variable Identifier**	**Importance Metric**	**Min. Tree Depth**
1	Weight	0.272212	1
2	EmptyWeight	0.256266	1
3	bixx	0.253467	2
4	bixy	0.253467	2
5	bixz	0.253467	2
6	bizz	0.253435	2
7	biyz	0.253435	3
8	biyy	0.253411	2
9	Mass	0.144018	3
10	PmTotalWeight	0.136427	3

Table 5.18: Importance values and minimum decision tree depths for FG2

Importance Ranking	**Variable Identifier**	**Importance Metric**	**Min. Tree Depth**
1	currentThrust	0.265753	1
2	hasInitEngines	0.255719	2
3	numTanks	0.252775	2
4	totalFuelQuantity	0.252658	2
5	firsttime	0.252043	2
6	dt	0.063108	3
7	electricEng	0.051481	3
8	throttleAdd	0.026494	4
9	enme	0.026133	-
10	te	0.023202	4

that variables identified by the importance metric feature in the efficient error detection predicates generated through decision tree induction.

Having applied the proposed approach for the generation of efficient error detection predicates to assess the capability of the importance metric to identify critical variables, it has been shown that efficient error detection predicates can be generated using only critical variables and that program variables identified by the importance metric feature in efficient error detection mechanisms. As well as serving to validate the capability of the importance metric to identify program variables that should be captured by error detection predicates, these results also serve to validate the thesis that an efficient EDM consists of a set of critical variables. Indeed, in this chapter it has been shown that an efficient

Table 5.19: Importance values and minimum decision tree depths for FG3

Importance Ranking	Variable Identifier	Importance Metric	Min. Tree Depth
1	compressLen	0.127795	1
2	groundSpeed	0.046646	1
3	steerAngle	0.000686	2
4	contractType	0.000657	2
5	bDampRebound	0.000464	2
6	eDampType	0.000396	2
7	serviceRe	0.000304	3
8	GearPos	0.000062	4
9	rfrv	0.000038	2
10	retractable	0.000010	4

Table 5.20: Importance values and minimum decision tree depths for MG1

Importance Ranking	Variable Identifier	Importance Metric	Min. Tree Depth
1	selfWrite	0.019506	1
2	bitridx	0.019189	1
3	whiChannel	0.019107	2
4	gainA	0.011914	2
5	curFrame	0.011897	2
6	inf	0.011892	-
7	cuFile	0.002456	3
8	wrdpntr	0.002452	4
9	inbuffer	0.002451	-
10	done	0.002442	-

EDM can consist only of a set of critical variables.

5.6 Implications and Discussion

The case studies presented have demonstrated that the proposed approach is capable of generating predicates for efficient error detection mechanisms. In particular, decision tree induction and rule induction have, even under a baseline configuration, been shown to be effective and consistent methods for generating predicates which exhibit high accuracy and completeness. In the case of decision tree induction and rule induction, generated predicates are represented as a tree structure to be interpreted as a conjunction of disjunctions and directly as a first-order predicate respectively. This reduces the implementation of an EDM

Table 5.21: Importance values and minimum decision tree depths for MG2

Importance Ranking	Variable Identifier	Importance Metric	Min. Tree Depth
1	sampleWin	0.360391	1
2	batchSample	0.236631	2
3	curSamples	0.212292	2
4	first	0.211843	2
5	op	0.100860	3
6	linpre	0.020400	-
7	rinpre	0.019693	-
8	totsamp	0.019625	-
9	cursamples	0.013566	-
10	cursamplepos	0.013462	-

Table 5.22: Importance values and minimum decision tree depths for MG3

Importance Ranking	Variable Identifier	Importance Metric	Min. Tree Depth
1	maxAmpOnly	0.316021	1
2	dSmp	0.115867	2
3	winCont	0.114893	3
4	sum	0.108781	2
5	mSamp	0.101039	2
6	bandPtr	0.077107	2
7	window	0.062254	2
8	windowSL	0.048595	3
9	sBuffs	0.002447	3
10	b0	0.002446	-

based on the representations generated by these algorithms to the process of interpreting a decision tree or first-order predicate.

Despite the presented case studies suggesting that the decision tree induction and rule induction algorithms yield significantly more efficient error detection predicates than naïve Bayes and logistic regression, it is not possible to conclude that these algorithms will consistently outperform other algorithms, including naïve Bayes and logistic regression. As any two classification algorithms can differ only in the class boundary that they define, i.e., the boundary defined to classify system failures and non-failures in the generation of error detection predicates, it is not possible to determine which classification algorithm will define an boundary that is appropriate for a particular data set. Indeed, it is

current practise in data mining approaches to classification problems to seek out an acceptable model through the investigation of many classification algorithms.

As fault injection analysis is commonly used in the validation of dependable software systems, the availability of fault injection data can often be assumed. This means that the main cost of applying the proposed approach is associated with the execution of data mining algorithms, which in-turn means that the cost of generating efficient predicates using our approach is related to data set magnitude, the data mining algorithm applied and the comprehensiveness of the refinement undertaken, i.e., the number of algorithm configurations that are considered in model refinement. It was shown in the cases studies presented in Section 5.4 that using only a baseline configuration of several data mining algorithms can yield highly-efficient error detection predicates and that a naive parameter search, i.e., systematically varying the level of sampling applied to data sets, can allow the efficiency of those predicates to be consistently improved, often to levels that would make the associated EDMs applicable in the design of dependable software systems.

The focus of the analysis presented has been on generating predicates for EDMs that are capable of detecting failure-inducing system states. Hence, the fault injection analysis undertaken focused on recording the system state during varied executions and whether those executions resulted in a system failure. This focus contrasts with existing work in fault injection analysis, which typically adopts the view that an error is any deviation from a fault-free execution, i.e., a golden run. Interestingly, whilst the approach described in this chapter is not directly applicable in this context, a similar approach can be adopted to derive error detection predicates that can identify such deviations from a golden run.

The novelty of the proposed approach for the generation of error detection predicates is in the application of data mining to fault injection data sets in order to obtain predicates for efficient EDMs. The main advantage of this approach is that efficient error detection mechanisms can be obtained by design, rather than a system specification or the experience of software engineers.

5.7 Summary and Conclusion

In this chapter a systematic approach to the design of efficient error detection predicates has been proposed, with a view to providing the first such systematic design approach for real-world, infinite-state software systems and validating the metric suite developed in Chapter 4. The premise of the proposed approach was that, given program locations at which EDMs will be located and for which the associated error detection predicates must be designed, data mining techniques can be applied in the analysis of fault injection data sets to obtain efficient error detection predicates for those EDMs. Following its description, the proposed approach was applied to software modules in three complex software systems, with error detection predicates being generated and evaluated for multiple code locations in each software module. The results presented demonstrated that the proposed approach can be used to generate efficient error detection predicates, i.e., error detection predicates exhibiting high accuracy and completeness. To validate the capability of the importance metric to identify critical variables, the predicate structures generated under decision tree induction, which was shown to be the most effective data mining algorithm under test, were compared with the relative rankings generated by the importance metric. Further, a new set of error detection predicates were generated using data sets containing information regarding only important variables, i.e., variables identified by the importance metric. The comparisons between the generated predicate structures and the relative rankings derived from the importance metric, in addition to the efficiencies of the newly generated error detection predicates, served to validate the capability of the importance metric to identify critical variables.

The validation of the importance metric has demonstrated its capability to identify critical variables and shown that it is possible to generate efficient error detection predicates using only important variables. However, this validation does not demonstrate, in any meaningful way, the dependability enhancements that can be achieved through the protection of critical variables. In order to

demonstrate the existence of a set of critical variables whose correctness is central to the proper functioning of a software system, it must be shown that software system dependability can be achieved through the protection of a small number of program variables. In the next chapter the level of dependability that can be achieved through the identification and protection of critical variables is explored, with a view to further validating the thesis that an efficient EDM consists of a set of critical variables.

CHAPTER 6

A Validation of Critical Variables

To this point it has been shown that it is possible to generate efficient error detection predicates for EDMs based only on critical variables, where these critical variables have been identified through the application of a threshold to the relative rankings generated by the importance metric. However, as the error detection predicates generated were concerned with specified locations in a software module, the program variables captured by these predicates can only be deemed critical with respect to those locations. As the importance metric operates at the level of software modules, i.e., it accounts for all locations where a program variable is used throughout a software module, it is necessary to demonstrate that the critical variables it identifies are critical throughout a software module, rather than just at the locations for which error detection predicates were generated. To determine whether the criticality of the program variables identified by the importance metric extends to wider context of a software module, the impact of protecting these critical variables throughout a software module must be considered. In this chapter an automated wrapper-

based approach for the design of dependable software is proposed, with a view to assessing the impact that the protection of critical variables can have on the dependability of a software system. In contrast to state-of-art approaches, which operate at the level of software modules, the approach developed in this chapter operates based on the replication of critical variables identified by the important metric, i.e., the approach is variable-centric. The results presented demonstrate that the failure rate associated with a software system where critical variables are wrapped, even when this is a small number of critical variables in a single software module, can be several orders of magnitude lower than that of an unwrapped equivalent. These results serve to substantiate the thesis that an efficient EDM consists of a set of critical variables, in that they confirm that the criticality of the program variables identified by the importance metric extends throughout a software module.

6.1 The Wrapping of Critical Variables

It has been argued throughout this thesis that the design of efficient EDMs is an inherently difficult task, with the results presented in Chapter 5 being at the forefront of what can currently be achieved in the generation of efficient error detection predicates for real-world, infinte-state software systems. One approach to overcoming this difficulty is to reuse standard dependability mechanisms that are know to implement efficient error detection predicates, such as majority voting, in the design of EDMs [14]. This can be viewed as akin to the reuse of trusted components in software engineering [54] [90] [116]. However, techniques such as software replication or NVP are expensive, as they typically operate at the level of a software system, e.g., an entire software system or numerous software components may be replicated in some way [14] [15].

To address the expensive nature of replication at a software level, it would be ideal for standard dependability mechanisms, that are known to implement highly-efficient error detection predicates, to be adapted to operate at a finer

granularity. To this end, this chapter proposes an approach to dependability enhancement based on the replication of the critical variables identified by the importance metric. This application of the importance metric serves to reduce the cost associated with replication whilst retaining the potential for dependability enhancement through the reuse of dependability mechanisms that are known to implement efficient error detection predicates. More specifically, the developed approach operates as follows. Firstly, a lookup table, in which the program variables in a software module are ranked according to their importance metric value, is generated. Once this lookup table is obtained, a subset of important variables is duplicated or triplicated, based on the application of a threshold to this lookup table, using software wrappers, i.e., shadow variables are created. Note that these program variables are termed "important", as opposed to "critical", because they are identified on the basis of the importance metric. Then, when an important variable is written during the execution of the software system, the value held by the relevant shadow variables are updated. Similarly, when an important variable is read during the execution of the software system, its value is compared against that of its shadow variables. Subject to the adopted fault model, any discrepancy amongst these values would be an indication of an erroneous state. Depending of the level of replication performed in the application of the approach, i.e., the level of duplication and triplication, an attempt can then be made to recover from this erroneous state through majority voting, forced validity methods or random value selection [14] [59].

Software wrappers have been investigated in many fields of research, such as computer security, software reengineering, database systems and dependable software systems. Although the approaches and techniques applied in these domains have commonalities, the motivation for their usage inevitably varies. To provide context for the described approach, Sections 6.1.1-6.1.3 give an overview of how software wrapper technology is currently used in various application domains, before Section 6.1.4 discusses how the characteristics of the approach developed in this chapter compare with current techniques.

6.1.1 Software Engineering

Software wrappers have been widely applied in software engineering research and practice, usually with a view to overcoming problems associated with the integration of legacy systems [146] [170]. In this context, software wrappers are most typically employed as connector components, usually to allow independently developed software systems to interact or as a means of providing or reconciling supplementary functionality [23] [28] [147]. Indeed, Bartolomei *et al.* focused on how an application programming interface (API) can be wrapped to permit some degree interoperability with alternative APIs, with a particular focus on using wrappers to span software platforms [18]. A key result of this work was the identification of common issues and challenges in software wrapper design, particularly for object-oriented systems. Similarly, Marosi *et al.* used software wrappers to allow legacy applications to execute on desktop grid resources with minimal local modification [107]. This approach was based on the development of a POSIX like shell scripting environment that was used to describe how application software was to be run. Further examples of software wrappers being used in systems integration can be found in the field of database systems, where software wrappers are typically used to encapsulate legacy database systems so that they can be reused or integrated with newly developed systems, often using the same automated wrapper generation techniques that emerged from the application of wrappers in software reengineering [35] [156].

6.1.2 Operating Systems

Software wrapper technology has been extensively investigated in the context of operating systems, where emphasis is often placed on wrapping device drivers and shared libraries [155] [169]. In the context of wrapping shared libraries to enhance robustness, Fetzer *et al.* proposed a highly-automated and adaptable approach for the generation and deployment of wrappers that can prevent the crash, hang and abort failures associated with the use of libraries in

the C programming language [49]. This approach was based on the extraction of type information from header files and manual pages, followed by the generation of bespoke fault-injectors that experimentally established robust argument types for C library function calls. The approach proposed in [49] was evaluated under Ballista tests, which have previously been use to demonstrate that many POSIX C library functions are fragile with respect to invalid arguments [86] [87]. Similarly, research in [154] led to the development of protection wrappers for the enhancement of commercial-off-the-shelf (COTS) components, including shared libraries. In contrast to research focusing on protecting shared resources, Ghosh *et al.* developed software wrappers that allowed applications to gracefully handle operating system failures, such as those induced by device driver failures or system stress [55]. This approach was motivated by previous work in [137], where several operating system calls in the dynamic link libraries (DLLs) of the Win32 API were shown to be fragile and capable of inducing severe consequences when presented with unexpected inputs. Working at a similar level of abstraction, Epstein *et al.* developed a software wrapper-based approach for improving the resiliency of application proxy-based firewalls [43]. Despite the effectiveness of this approach, the fact that it entailed the development of bespoke wrappers for all protocols and protocol variants to be permitted operation through a firewall, means that a significant level of insight and software engineering is required for its adoption.

A consequence of wrapping shared resources and application interfaces is a reduction in error propagation. This is because the behaviours of these components are constrained, thus ensuring that erroneous state are less likely to be entered and propagated. Research in software wrappers for operating systems has focused specifically on addressing the problem of error propagation. For example, Johansson *et al.* proposed approaches for the identification and wrapping of vulnerable locations in operating systems, with a view to directly addressing the error propagation problem [79] [80]. This work was built on the premise that error propagation analysis can reveal the types of errors occurring

in an operating system that will propagate through the operating system and impact applications. Similarly, Fabre *et al.* proposed analysis approaches that can assist in the the design of fault containment wrappers based on the consideration of failure modes [45]. In particular, these approaches made use of the MAFALDA fault injection tool, which was specifically designed as an evaluation tool for microkernel dependability and wrapper design [133].

6.1.3 Dependable Software Systems

With respect to the process of dependability enhancement it is the general intention of a software wrapper to implement a simple, well-understood predicate that constrains the behaviour of a software component, thus limiting the occurrence of an erroneous software state. A central premise of software wrapper technology is that the component being wrapped should be oblivious to the wrapping performed, though in practise this ideal may be violated when a software wrapper imposes a specification constraint that would not have been enforced before wrapping. These notions of simplicity and transparency are embodied by standards for component interoperability such as the Common Object Request Broker Architecture (CORBA), the Distributed Component Object Model (DCOM) and JavaBeans, each of which has been widely applied in the development of dependable software systems [104] [166]. The principles of simplicity and transparency are adhered to in research such as [42], where software wrappers were used to detect contract violations in component-based systems. This work also demonstrated that software wrappers could be used to allow the users of a software system to layer contract-checking components on top of the system without source code access. Moreover, research in [73] and [74] developed approaches for the generation of fault containment software wrappers for the enhancement of component-based systems, with a particular emphasis on maintaining safety properties when components were composed.

Software wrappers have also been used to address the, widely-acknowledged, problem of improving dependability in COTS software [136]. For example, work

in [13] developed an approach for the integration of COTS components to form idealised fault-tolerant COTS components in a dependable software system, essentially providing a mechanism by which a set of fault-intolerant components can be transformed into a functioning dependable software system. A focus on application level issues can also be seen in [47], where software wrappers were combined with checkpointing mechanisms to address the problem of non-atomic exception handling. In particular, this work introduced a notion of failure atomicity in exception handling to capture the semantics of the software wrappers developed and govern the management of checkpointing and recovery. Further, in [48] the authors developed a wrapper-based approach to address the issue of buffer overrun in the C programming language. The approach, which could be adopted without source code access, entailed the transparent interception of C library calls that are known to be unsafe, coupled with argument checking and a subsequent call to the unsafe function with checked arguments.

In the context of computer security, software wrappers have typically been used to encapsulate software systems so that a specific set of security policies can be enforced to protect vulnerable assets, particular when these assets have a public interface. For example, the wrapper-based approach in [32] focused on the transparent protection of a domain name system (DNS) through message inspection. More specifically, a formal system specification was used to characterise DNS clients and name servers with respect to some security objective, before a DNS wrapper that examines the incoming and outgoing DNS messages of a name server was formally specified. This DNS wrapper was designed to detect and drop messages that could cause violations of the defined security objective, thus providing protection again many common DNS attacks, including cache poisoning and a subset of spoofing attacks [5] [57]. As a further example, work in [139] addressed the problem of developing concurrent software systems using the composition of wrapped software components. Building on the process calculus developed in [140] and the causal type to capture permissible information flows developed in [141], this approach proposed to use software wrappers

to encapsulate components and enforce specific security policies. Further to these examples, it has also been shown that security wrappers deployed at the level of an operating system kernel can be used to meet a range of application specific security requirements, with work in [109] and [110] proposing the use of kernel hypervisors to protect against malicious downloadable content and safeguard firewall services. Most significantly, these kernel hypervisors, which were implemented as loadable kernel modules for the Linux operating system, provided an "unbypassable" layer of security within an operating system kernel but did not necessitate kernel modifications.

6.1.4 Evaluation of Existing Software Wrapper Usage

Despite the widespread application of software wrapper technology, not least in the design of dependable software systems, the granularity at which existing approaches have applied serves to differentiate the approach proposed in this chapter. Specifically, the variable-centric nature of the proposed approach, as facilitated by the application of the importance metric, is distinctive when compared to the existing approaches, which are generally concerned with the wrapping of software components such as software modules and entire software systems. A justification for the higher-level focus of existing approaches may relate to the fact that the overheads associated with a software wrapper are typically linked to the number of invocations of that wrapper, meaning that a fine-grained approach, where more software wrappers are to be deployed, will lead to more invocations and, hence, greater overheads. However, the already significant overhead of wrapping software modules, e.g., over 1200% in [47], combined with the focused approach enabled by the importance metric, motivate the consideration of a more selective fine-grained approach to the application of software wrappers. The approach developed in this chapter represents the first variable-centric approach to dependability enhancement through the use of software wrappers and variable replication. It is this variable-centric focus, facilitated by the metric suite developed in Chapter 4, that enables the key ben-

efits of the approach over current state-of-the-art techniques in dependability enhancement. Further, the approach described in this chapter (i) circumvents the need to obtain non-trivial predicates by using standard efficient predicates, e.g., majority voting (ii) circumvents the need to know the optimal location of a given predicate by comparing values on all accesses to critical variables, (iii) allows the efficiency of the associated EDMs to be known a priori, obviating the need for dependability validation [66], (iv) incurs significantly lower overheads than approaches that do not operated with such granularity, and (v) reduces the risk of inserting software bugs during dependability enhancement [47] [142].

6.2 A Wrapper-based Software Design Approach

The approach proposed in this chapter is based on the premise that the replication of critical variables, as identified by the importance metric, can significantly increase software system dependability without incurring significant execution overheads. Thus, the approach serves a mechanism for further validating the thesis that an efficient EDM consists of a set of critical variables. The proposed approach is a three stage process. First, a lookup table ranking variables based on their importance metric value for a given module is generated. Next, all read and write operations on important variables, as defined by the application of a threshold, are identified. These identified operations are known as important actions. Finally, all important actions are protected using software wrappers that implement error detection and correction predicates that are known to be efficient, i.e., comparisons between replicated values and majority voting. An overview of the described approach is shown in Figure 6.1, whilst Sections 6.2.1-6.2.3 provide a description of each stage of the approach.

6.2.1 Stage 1: Establishing Variable Importance

The first stage of the proposed approach is to establish the criticality of each variable within a target software module. To achieve this the metric suite and

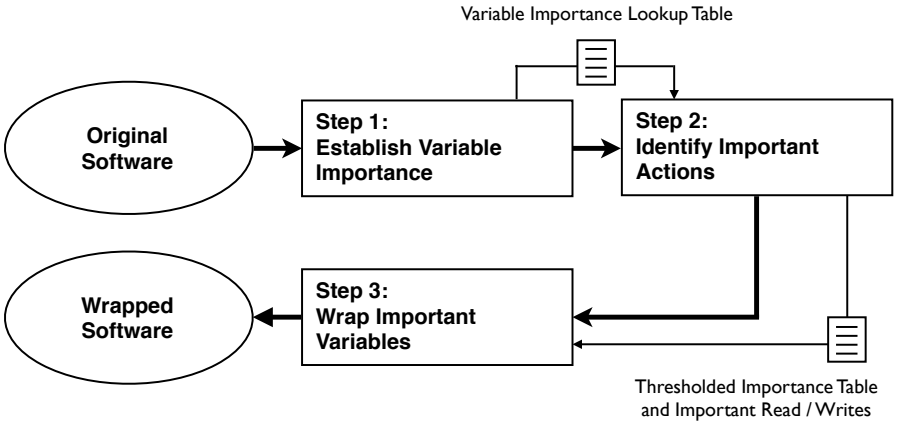

Figure 6.1: An overview of enhancing dependability through the replication of critical variables using software wrappers

fault injection approach developed in Chapter 4 are used to measure variable importance. The importance of all program variables in a target software module can be evaluated automatically in this way. In Chapter 4, as well as [100], fault injection analysis was used to estimate variable importance, though, as stated previously, the metric suite can be evaluated using alternative approaches. Once this first stage of the approach has been completed, a lookup table relating any given variable to its importance metric value can be constructed.

6.2.2 Stage 2: Identifying Important Actions

The second stage of the approach is to identify all read and write actions on important variables. As the replication of a whole software system, or indeed every program variable in a software system, incurs a large overhead, a subset of the most important variables are selected for replication. This is done using thresholds to govern the levels of duplication and triplication performed. Specifically, two thresholds are set to govern the number of duplicated and triplicated program variables; λ_d and λ_t respectively. These thresholds may be defined with respect to importance metric values, though as the absolute important metric values will have little meaning in the context of most software systems, it is reasonable to define thresholds as a proportion of the program variables in a

module. For example, to specify the triplication of the top 10% and duplication of the top 15% of program variables in a target software module, $\lambda_t = 0.10$ and $\lambda_d = 0.15$ would be set. The use of these two thresholds, one for duplication and one for triplication, allows replication overheads to be reduced, as not every program variable must be triplicated in situations where perfect error detection followed by a best-efforts recovery attempt is sufficient. Once threshold values have been set, the program variables to be wrapped can be identified. However, before wrapping can be performed, every possible read and write location on an important variable must be identified. This can be achieved by several means, including system call monitoring and memory management techniques. The only requirement is that all possible read and write actions on important variables must be identified. As will be detailed in Section 6.3, the use of automated source code analysis is advocated as an means for the identification of important actions. The completion of this stage of the approach will mean that all program variables to be wrapped have been identified and a mechanism has been used, or is in place, to identify read and write actions.

6.2.3 Stage 3: Wrapping Important Variables

Having established the importance of the program variables in a target software module, based on the importance metric, and identified all locations where read and write actions could be performed on sufficiently important variables, the final stage of the described approach is to deploy software wrappers in order to protect critical variables in these locations. Two types of software wrapper are employed by the described approach; *read*-wrappers and *write*-wrappers. Description of how these software wrappers operate is given below, whilst pseudocode for the write-wrapper and read-wrapper is shown in Figure 6.2 and 6.3 respectively.

Write-Wrapper: This software wrapper is invoked when an important variable is written. When a variable v is assigned a value $f(\ldots)$, where f is some

Algorithm 1 Write-Wrapper: Writing a variable v

```
v := f(...)
if (rank(v) ≥ λ_t) then
   create(v');
   create(v'');
   v, v', v'' := f(...);
else if (rank(v) ≥ λ_d) then
   create(v');
   v, v' := f(...);
end if
```

Figure 6.2: The algorithm executed by a software wrapper when a write action is performed on a sufficiently critical variable

Algorithm 2 ReadWrapper: Reading a variable v

```
y := g(v, ...);
if (rank(v) ≥ λ_t) then
   y := g(majority(v, v', v''), ...);
else if (rank(v) ≥ λ_d) then
   y := g(random(v, v'), ...);
end if
```

Figure 6.3: The algorithm executed by a software wrapper when a read action is performed on a sufficiently critical variable

function, in the unwrapped module, the ranking of the variable is checked. If the rank of v is in the top λ_t, then two shadow variables, v' and v'', are created. Alternatively, if the rank of v is between λ_t and λ_d, then a shadow variable v' is created. Then, v and all of its shadow variables are updated with $f(\ldots)$.

Read-Wrapper: This software wrapper is invoked when an important variable is read. When a variable y is updated with a function $g(v, \ldots)$ in the unwrapped module, where g is a function and variable v is to be read, the rank of v is checked against λ_t. If the rank of v is greater than λ_t then function g uses the majority of the v, v', v''. If the rank of v is between λ_t and λ_d, then g uses v or v'.

The completion of this third stage will yield a software system where all read

and writes to important variables are protected by an EDM that implements an efficient error detection predicate, i.e. replicated program variable checking. In the case of the most important variables, which have been triplicated, an effective ERM, i.e., majority voting, will also be tightly integrated with the error detection process. On the other hand, those variables that have been duplicated will be given the opportunity to recover based on a random selection between the two values that enabled error detection. In the application of the approach it may be tempting to consistently triplicate important variables, by setting $\lambda_d = \lambda_t$. However, this incurs the maximum level of replication overhead in order to provide an effective ERM, hence in situations where an effective ERM already exists or is not required, i.e., perfect error detection is sufficient and best-efforts recovery is sufficient, this would not be a reasonable approach. Indeed, such a situation may be better served by a configuration where $\lambda_t = 0$.

Each stage of the described approach has been intentionally kept independent of implementation, in that sense that the input and outcomes of each stage are the focus of the description provided, rather than the means by which each stage should be achieved. For example, whilst it is suggested that the generation of the lookup table in the first stage of the approach can be undertaken using the metric suite proposed in Chapter 4, this lookup table can be generated using alternative means, e.g., static analysis or the experience of software engineers. Similarly, the identification of read and write actions on critical variables could be undertaken using dynamic analysis, memory management techniques or system call monitoring, as opposed to the use of automated source code analysis. Despite this implementation independence there are operational dependencies between the stages of the approach. In particular, the means by which software wrappers should be implemented and deployed will be guided by the approach used for the identification of important actions. For example, if memory monitoring techniques were used in the identification of read and write actions on critical variables then it may be difficult to relate these actions to locations in

source code, regardless of how well the target software system is understood. In such situations it would be more appropriate to deploy dynamic software wrappers, rather than relying on source code transformation.

6.3 Case Studies

To demonstrate the dependability enhancement that can be achieved through the protection of a small number of critical variables, the results of applying the described approach to target software modules are shown in Sections 6.3.1-6.3.3.

6.3.1 Stage 1: Establishing Variable Importance

Using the approach developed in Chapter 4 under the experimental conditions detailed in Chapter 3, the spatial and temporal impact of each variable was experimentally estimated. This information, as well as the failure rate for fault injections on each variable, was used to evaluate the importance of the variables in each software system. Note that system failure rate here is, again, assessed on a per variable basis. For example, if a variable is target of 100 fault injection executions and 25 of these result in a system failure, then that variable considered to have a failure rate of 0.25. The entries in the thresholded lookup tables generated by the importance metric represent a subset of the entries that were previously presented in Tables 4.1-4.9. For this reason, the thresholded lookup tables generated during the first stage of the proposed approach are not shown.

To explore multiple applications of the approach described in this chapter, threshold settings of (0.10, 0.15), (0.15, 0.20) and (0.20, 0.20) were applied. In this notation, (0.10, 0.15) denotes $\lambda_t = 0.10$ and $\lambda_d = 0.15$, which indicates that, for each target software module, the top 15% of variables would be wrapped, with the top 10% being triplicated and the next 5% being duplicated.

6.3.2 Stage 2: Identifying Important Actions

Once the importance table for each module had been thresholded, source code analysis was used to identify read and write actions on important variables. The implementation of this source code analyser was based on the premise that reads and writes to important variable are the only operation types that are deemed to be important actions. When adopting a source code analysis approach it must be recognised that any analysis tool must work under the assumption that any unidentifiable operation type could be an action relating to any important variable. This conservative stance ensures that the coverage of the approach is maximised, though unnecessary overheads, with respect to runtime and memory consumption, may be incurred where a location is unnecessarily instrumented.

6.3.3 Stage 3: Wrapping Important Variables

Following the identification of read and write actions on important variables, the software wrappers described in Section 6.2.3 were deployed. As the locations for read-wrapper and write-wrapper deployment were necessarily consistent with the code locations of important read and write actions respectively, information generated during source code analysis was used to drive wrapper deployment.

Figures 6.4 and 6.5 show concrete examples of write-wrapper and read-wrapper deployments respectively. The first line in each example shows the original program statement before wrapping. The second line in each figure illustrates the use of software wrappers. In Figure 6.5 the *dt* variable is being read-wrapped, whilst Figure 6.4 shows the *currentThrust* variable being write-wrapped. Observe that, in both cases, it is necessary to provide the wrapping functions with identifiers for the variable and location. This information is generated, maintained and known only to the wrapping software following the identification of important read and write actions. This means that it has no discernible impact on the execution of the target system, which remains oblivious to the existence of the read-wrapper and write-wrapper deployments.

```
/* currentThrust = Engines[i]->GetThrust();*/
currentThrust = writeWrapper(VARID_17, LOCID_8, Engines[i]->GetThrust());
```

Figure 6.4: A write-wrapper deployment in source code

```
/* tankUPD = calc + (dt * rate); */
tankUPD = calc + (readWrapper(VARID_12, LOCID_4, dt) * rate);
```

Figure 6.5: A read-wrapper deployment in source code

To investigate the dependability enhancement afforded by the described approach, the fault injection experiments described in Chapter 3 were repeated on wrapped target systems, i.e., more than 38 million experiments were performed to measure observed to failure rates with wrapped software systems. A single software module in any target software system has its important variables wrapped at any one time. Although each target system was deterministic, each experiment was triplicated to account for the random choice aspect of the deployed software wrappers. The standard deviation for all sets of experiments was less than 0.00001. Table 6.1 summarises the impact that the described approach had on the dependability of all target software modules. The *Unwrapped Failure Rate* column gives the original system failure rate with respect to all fault injection experiments, i.e., the proportion of system failures of the unwrapped target system when fault injection across all variables in the given software module are considered. The *Wrapped Failure Rate* column gives the same measure for wrapped modules.

Observe from Table 6.1 that the system failure rate of each software module dramatically decreased in all cases under the $(0.15, 0.20)$ threshold setting, thus demonstrating the effectiveness of the approach for dependability enhancement. Further, the decrease in system failure rate of many modules is greater than combined failure rates of the wrapped variables in those modules. For example, module *MG3* had an unwrapped failure rate of 0.002780830, which corresponded

Table 6.1: System failure rates for wrapped and unwrapped modules

Target Module	**Unwrapped Failure Rate**	**Wrapped Failure Rate**		
		(0.10, 0.15)	**(0.15, 0.20)**	**(0.20, 0.20)**
7Z1	0.002407940	0.000017	0.000003	<0.0000001
7Z2	0.007082023	0.000142	0.000014	<0.0000001
7Z3	0.000856604	0.000506300	0	0
FG1	0.001929009	0.000022	0	0
FG2	0.002481621	0.000002047	0.000001	<0.0000001
FG3	0.001471873	0.000135395	0.000002	0
MG1	0.004983750	0.000012083	0.000029	0
MG2	0.007888044	0.000013426	0.0000011	0
MG3	0.002780792	0.000006076	<0.0000001	<0.0000001
	0.00354241	**0.000095147**	**0.00000626**	**<0.0000001**

to 39361 failures. The same module had a wrapped failure rate of 0.000006105, corresponding to 86 failures. This improvement can not be accounted for by the 1142 failures incurred by the three wrapped variables, hence the wrapping performed must account for fault injections elsewhere. The results associated with the $(0.15, 0.20)$ and $(0.20, 0.20)$ threshold settings also show this pattern. In particular, Table 6.1 shows that, when chosen appropriately using the importance metric, safeguarding just 20% of the program variables in a software module can be sufficient to reduce the system failure rate of that software module to 0. Further, in cases where the same threshold was applied but this level of dependability was not reached, the associated software module failure rate was near-zero and dramatically improved over all other threshold settings, thus indicating the near-sufficiency of the identified critical variable with respect to safeguarding the proper functioning of these target software systems.

As the focused, fine-grained nature of the replication undertaken by the proposed approach has the potential to permit low executions and memory overheads, it is interesting to consider the performance of the approach with respect to these measures. To this end, Tables 6.2 and 6.3 summarise the worst-case overhead of applying the approach on all target software modules. Table 6.2 gives the peak percentage increase in runtime, whilst Table 6.3 gives the peak percentage increase in memory consumption. These increases reflect peak in-

Table 6.2: Peak increase in execution time incurred by module wrapping

Target Module	**Execution Time (Peak % increase)**		
	(0.10, 0.15)	**(0.15, 0.20)**	**(0.20, 0.20)**
7Z1	26.05%	22.83%	22.91%
7Z2	31.47%	31.55%	31.79%
7Z3	20.36%	20.36%	22.84%
FG1	18.24%	21.07%	24.21%
FG2	35.83%	35.87%	35.83%
FG3	23.53%	30.10%	31.97%
MG1	25.98%	29.44%	31.28%
MG2	28.09%	31.02%	31.08%
MG3	23.17%	27.02%	27.02%
	25.86%	**27.70%**	**28.77%**

Table 6.3: Peak increase in memory usage incurred by module wrapping

Target Module	**Memory Usage (Peak % increase)**		
	(0.10, 0.15)	**(0.15, 0.20)**	**(0.20, 0.20)**
7Z1	07.55%	08.04%	08.82%
7Z2	18.16%	18.92%	20.04%
7Z3	00.94%	00.94%	01.87%
FG1	02.22%	03.28%	03.51%
FG2	03.32%	08.24%	09.04%
FG3	02.03%	03.47%	03.89%
MG1	05.22%	09.22%	10.90%
MG2	04.93%	07.21%	08.84%
MG3	00.58%	04.01%	04.75%
	4.99%	**7.04%**	**7.96%**

creases observed when comparing non-fault injected executions of a unwrapped target software module against non-fault injected executions of a wrapped target software module. The test case executions used for observation were identical to those used in fault injection analysis. All overheads were measured by monitoring target modules in isolation using the Microsoft Visual Studio Profiler running in both sampling and instrumentation modes, with the stated overheads being the maximum observed under either configuration [108].

Observe from Table 6.2 that the minimum and maximum execution overhead, across all replication thresholds, of wrapped modules varies between 18.24% and 35.83%. The worst-case absolute increase in the execution time

of a target module was observed for module *7Z2*, which increased by 31.79% to approximately 28μs. There is a coarse correlation between the increase in execution time and the number of variables in each module, though the frequency with which each variable is used is likely to impact this overhead more directly. Table 6.3 shows that the increases in memory consumption are more varied than increases in execution time. As the memory usage increases shown are the peak observed increases for each target software module, the increase is unlikely to be sustained beyond the execution of a module. Further, the absolute magnitude of these increases may be small. For example, the seemingly large 18.16% increase in memory consumption shown for the *7Z2* target module corresponds to an absolute additional overhead of less than 3 kilobytes.

6.4 Implications and Discussion

Inserting tailored EDMs and ERMs into a software system is likely to result in a low overhead, due to the fact that only a small number of variables and code segments must be added or replicated. However, as argued earlier, this approach necessitates the design of non-trivial predicates, which is known to be difficult [102]. Also, the design and deployment of EDMs often introduces additional bugs into a software system [47]. The approach developed in this chapter circumvents these problems by using standard efficient error detection and recovery predicates, though this comes at the cost of greater execution and memory overheads. This issue can be seen as a tradeoff. Inserting dependability mechanisms directly is difficult and error prone but imposes less overhead, whilst approaches such as the one described in this chapter reuse simpler mechanisms at the expense of greater overheads.

The performance overheads of the proposed approach will vary according to the extent of wrapping performed, i.e., according to λ_d and λ_t. Overhead comparison with similar approaches are desirable but generally invalid due to difference in the extent, intention and focus of the wrapping mechanisms em-

ployed. For example, the results presented in this paper demonstrate that with $\lambda_d = 0.15$ and $\lambda_t = 0.10$, for a single module measured in isolation, our approach introduces a additional runtime overhead of approximately 18%-35% and a memory overhead of approximately 0.5%-20%. In contrast, the approach developed in [47] had a memory overhead for the masking of a fixed-duration function, set at 5μs, of over 1200%. However, the software module-centric focus of the approach in [47], as opposed to the variable-centric focus adopted in this chapter, invalidates such a comparison of overheads.

Given that the developed software wrappers operate by updating replicated variables during writes and selecting the most appropriate value during reads, the described approach will work with variables of different types whenever the notion of equality exists or can be defined for that type. Equality is well-defined for integer, real and boolean types but can also be defined for composite types.

The most significant implication of the approach developed in this chapter can be seen in the results of the case studies presented in Section 6.3. These results demonstrate that adequately protecting a relatively small set of critical variables in a target module can serve to safeguard the proper functioning of a software system. Indeed, the fact that the decrease in system failure rate observed for many target modules was greater than combined failure rates of the wrapped variables in those modules, is a clear indication that the protected variables in each modules are accounting for issues occurring in unwrapped variables, i.e., the propagation of errors is halted by protected variables before a system failure can ensue.

6.5 Summary and Conclusion

In this chapter an automated approach to dependability enhancement based on the replication of the critical variables has been developed, with a view to further validating the criticality of the program variables identified as critical by the importance metric. The developed approach operates based on the reuse

of dependability mechanisms that are known to implement efficient error detection predicates. More specifically, for any target software module, software wrappers are used for the replication of a set of critical variables identified using the metric suite developed in Chapter 4, i.e., shadow variables were created for critical variables. Thus, when an important variable is written during the execution of a target software module, the value held by the relevant shadow variables is updated. Similarly, when an important variable is read during the execution of a target software module, the value of the variable is compared against that of its shadow variables. Any discrepancy amongst these values is an indication of an erroneous state. Depending of the level of replication performed, i.e., duplication or triplication, an attempt can then be made to recover from this erroneous state through majority voting, forced validity methods, or a random value selection. To investigate the dependability enhancement that it can achieve, the developed approach was applied to software modules in all target software systems, with various levels of critical variable duplication and triplication being applied in each case. The results presented demonstrate that the system failure rate associated with a software modules can be dramatically decreased, to 0 in many cases, through the protection of a relatively small set of variables, thus suggesting that a relatively small set of critical variables can be use to capture to correctness of a software system and the thesis that an efficient EDM consists of a set of critical variables.

To this point (i) a metric suite for the identification of critical variables has been developed, (ii) the effectiveness of the metric suite has been validated through the development of a separate approach for the generation of efficient error detection predicates and (iii) the criticality of the program variables identified by the metic suite has been validated with respect to the associated software modules. When considered in combination, these contributions serve as evidence to support the thesis that efficient EDMs consist of a set of critical variables. In the next chapter the contributions made in Chapters 4-6 are analysed, alongside a full discussion of their contribution to supporting this thesis.

CHAPTER 7

Discussion

Chapters 4-6 made novel research contributions in support of the thesis that an efficient EDM consists of a set of critical variables. To this point these contributions have been considered in isolation, with each building on those previous and serving to motivate subsequently presented research. Indeed, the analysis and discussion presented to this point has focused largely on the implications and applications of the contributions made, rather than solely on the implications of these contributions in the context of the stated thesis. In this chapter these contributions are drawn together in support of the thesis that an efficient EDM consists of a set of critical variables. In particular, this chapter summaries and analyses the overarching outcomes of the work presented in each chapter with respect to the thesis that an efficient EDM consists of a set of critical variables, before concluding with a discussion of the potential applications and limitations of the thesis. The discussion presented in this chapter ultimately concludes that the research presented substantiates the thesis that an efficient EDM contains a set of critical variables on the basis that (i) the importance metric is able,

through application of an appropriate threshold, to identify critical variables, (ii) efficient EDMs can be constructed based only on the critical variables identified by the importance metric, and (iii) the criticality of identified variables can be shown to extend across a software module such that an efficient EDM designed for that software module should seek to determine the correctness of the identified variables.

7.1 Summary and Implications

In Chapter 4 a metric suite, alongside a fault injection approach for its evaluation, was proposed for the ranking of the program variables in a software module with respect to a notion of criticality that accounted for the potential impact of those program variables in the spatial and temporal domains. It was shown that the rankings generated by this metric suite, which must be thresholded in order to identify critical variables, were justifiable, robust, and consistent with the intention underpinning its proposal. The development of a mechanism for the identification of critical variables serves as the first step towards supporting the thesis that an efficient EDM consists of a set of critical variables, in that it facilitates the investigation of the critical variables identified with respect to the efficiency of the EDMs that incorporate them.

In Chapter 5 a systematic approach to the generation of highly-efficient error detection predicates for EDMs was proposed based on the application of data mining algorithms to data sets derived from fault injection analysis. More specifically, given a location in a software module, an efficient error detection predicate for an EDM could be generated based on the application of a symbolic pattern learning algorithm to the problem of learning the relationships between program variables that influence the success of a software system execution. The proposed approach enables the design of efficient EDMs with no reliance on the experience of software engineers or a system specification. Crucially, this approach provides a means for assessing the capacity of the importance

metric to identify critical variables, in that it allows the efficiency properties of EDMs generated using all program variable in a software module to be compared against the efficiency properties of EDMs generated using only the critical variables identified by the metric suite proposed in Chapter 4. The comparison demonstrated that EDMs generated using only critical variables were efficient, in that their efficiency properties were equal or only marginally worse than EDMs generated using all the program variables in a software module. This outcome is an indication that a set of critical variables can be used to efficiently detect errors at a specified location in a software module, which serves to support the thesis that an EDM consists of a set of critical variables. That is to say, at a specified location in a software module, a set of critical variables can be used for efficient error detection.

Chapter 6 proposed an approach for dependability enhancement, based on the wrapping of critical variables using software wrappers that implement error detection and correction predicates that are known to be efficient. This approach provided a means for extending the results presented in Chapter 5 to account for the criticality of program variables identified as critical across an entire software module. Put differently, it was shown that, by protecting a relatively small proportion of the identified critical variables in all locations of a software module, the system failure rate associated with that software module can be significantly reduced. The implication of this result is that, in order to design an efficient EDM that is to be located in a particular software module, the values of a set of critical variables must be ensured to be correct, i.e., the EDM must contain such a set of critical variables. Indeed, when taken in conjunction with the overarching outcomes of Chapter 4-5, this result serves to substantiate the thesis that an efficient EDM consists of a set of critical variables.

The research presented in Chapters 4-6 support the thesis that an efficient EDM contains a set of critical variables. This is asserted on the basis that (i) the metric suite proposed in Chapter 4 is able, through application of an appropriate threshold, to identify critical variables, (ii) the approach proposed in

Chapter 5 allows efficient EDMs to be constructed based only on the critical variables identified by the importance metric, and (iii) using the dependability enhancement approach proposed in Chapter 6 the criticality of identified variables can be shown to extend across a software module such that an efficient EDM designed for that software module should seek to ensure the correctness of the identified critical variables.

At this point it should be noted that the stated thesis does not imply that any EDM capturing an appropriate set of critical variables will yield the highest possible efficiency properties. Ideally it would be possible to ensure the correctness of all program variables in a software module for all locations in that software module. However, as discussed in Chapter 1, this is infeasible. Indeed, even if this were feasible, it may not provide perfect error detection capabilities [76]. The use of critical variables in the design of EDMs represents an approach for providing an approximate solution to this problem, whereby far fewer program variables, hence locations, must be considered in the design of efficient, yet possibly imperfect, EDMs. This means that, even when an EDM captures all critical variables, its efficiency properties can still be improved by the suitable incorporation of further program variables. Indeed, the benefit of capturing critical variables, as it has been shown in this thesis, is that these program variables enable high levels of efficiency to be achieved, which means that the subsequent incorporation of further program variables will yield only small improvements in EDM efficiency.

The approach presented in Chapter 6 built on the results of previous chapters to demonstrate that, in order to design an efficient EDM that is to be located in a particular software module, the values of a set of critical variables must be ensured to be correct, i.e., the EDM must contain such a set of critical variables. This approach relied on the importance metric, proposed in Chapter 4, and the application of a suitable threshold for the identification of critical variable variables. However, whilst this approach was fit for this purpose, the existence of a set of critical variables could also have been established by alternative means.

Most notably, the application of feature subset selection techniques would allow the underlying set of features of the data sets collected during fault injection analysis, i.e., a set of critical variables, to be identified for incorporation in error detection predicates of EDM [106] [114] [171]. However, feature subset selection was not used in this thesis because, in addition to providing a mechanism for the identification of critical variables, such techniques would also seek to minimise the set of identified critical variables. As it was the intention of the research presented to support the thesis that an efficient EDM consists of a set of critical variables, the cardinality of the identified set of critical variables, assuming that all program variables in a software module had not been identified as critical, was not a primary concern.

7.2 Applications

There are two main practical applications of the thesis that an EDM consists of a set of critical variables, one of which concerns the design of EDMs, whilst the other concerns the location of EDMs. Firstly, the identification of critical variables serves to simplify the design of efficient EDMs, as the state space that a software engineer must consider in the design of error detection predicates is dramatically reduced. Indeed, the identification of critical variables is complementary to existing experience and specification-based approaches for the design of error detection predicates, in that it can easily be combined with other approaches in an attempt to improve their efficiency or effectiveness through the aforementioned state space reduction. Secondly, following the identification of critical variables, their occurrence in a software module can be used to inform the location of EDMs. Indeed, such an application of the stated thesis is demonstrated by the approach proposed in Chapter 6, where occurrences of critical variables were use to guide the enhancement of software dependability. This potential application can serve to complement existing approaches to the design and location of EDMs, as the locations associated with occurrences of critical

variables may govern the error detection predicates that may be implemented and the efficiency that can be achieved by a proposed EDM.

7.3 Limitations

The thesis that an efficient EDM consists of a set of critical variables has been substantiated by the research presented in Chapters 4-6. However, aside from the issues raised above, there remain limitations of this thesis that must be acknowledged in the context of the work presented. These limitations are discussed below. Note that the limitations of the approaches proposed in Chapters 4-6 are discussed in their respective chapters, whereas the issues raised here focus specifically on the limitations of stated thesis.

The design of efficient EDMs is a central theme of the research presented and the thesis that an efficient EDM consists of a set of critical variables. However, the notion of efficiency applied in this thesis is relatively inexact, in that no threshold was defined, with respect to the adopted efficiency measures, to determine whether a particularly EDM could be viewed as efficient. The motivation of this inexact characterisation of efficiency was based on the fact that, whilst the maximum efficiency that can be achieved by an EDM is well know, i.e., $TPR = 1$, $FPR = 0$ and $AUC = 1$, this maximum level of efficiency may not always be achievable, particularly in situations where read and write constraints are places on program variables [76]. As it is not possible to determine when the maximum achievable efficiency has been reached, this means that a precise characterisation of efficiency, even on a per software system basis, may discount EDMs that are as efficient as they could be for a particular location, thereby violating the intention of the EDM design problem as discussed in Chapter 2.

As mentioned previously, the thesis that an efficient EDM consists of a set of critical variables does not imply that every variable in that set of critical variables must be captured by an EDM in order for it to be efficient or that

doing so would result in the design of an efficient EDM. Indeed, the stated thesis could less succinctly but equivalently be rephrased to state that an efficient EDM consists of a subset of program variables from a software module that can be considered critical. This restatement of the thesis is less succinct but emphasises the point that not every critical variable in a software must be captured in order for an EDM to be efficient at a specific location.

The thesis that an efficient EDM consists of a set of critical variables relies on the existence of some notion of criticality. The research presented in this thesis focused on a specific notion of criticality, as characterised by the measurement of spatial impact, temporal impact and system failure rates. However, whilst demonstrating that a notion of criticality by can be devised to support the stated thesis is sufficient for the validation of the stated thesis, this work does not completely specify the notion, or each and every notion, of criticality that is consistent with the stated thesis. Indeed, an exploration of notion of criticality with respect to program variables and the stated thesis represents an interesting area for future work in EDM design.

Finally, though it is not an inherent limitation of the research presented, the thesis that an EDM consists of a set of critical variables has relatively few implications for the effective composition of error detection predicates. That is, understanding that a set of critical variables must be identified and incorporated by an error detection predicate does not, by itself, enable the design of an efficient EDM. This thesis limitation is somewhat circumvented by the approach proposed in Chapter 5, in that this approach was shown to be capable of generating efficient error detection predicates on the basis of critical critical variables, though it should be noted that the approach provides little specific insight into the general process of composing efficient EDMs and does not provide any guarantees with respect to the generation of a single most efficient error detection predicate for a location.

Having discussed the implications, applications and limitations of the thesis that an efficient EDM consists of a set of critical variables, as well as relating the research presented to this thesis, it remains to conclude with a summary of achievements and a discussion of future work. This concluding summary and discussion is the focus of the final chapter of this thesis.

CHAPTER 8

Conclusion and Future Work

To this point a body of research, analysis and discussion has been presented to substantiate the thesis that an EDM consists of a set of critical variables. Moreover, the implications of the contributions made with regard to the design of efficient EDMs have been drawn together to provide a sound basis to support this notion. In this chapter a summary of these research contributions and a discussion of future work relating to the EDM design problem is provided as a conclusion to this thesis. In particular, the research contributions made throughout Chapters 4-6 are summarised with respect to the stated thesis, whist the discussion of future work focuses on issues such as the inheritance and propagation of criticality among program variables, and the potential for the location of EDMs to be based on achievable efficiency.

8.1 Thesis Summary

In this thesis the notion of program variable criticality has been used to address the problem of EDM design. That is to say, it has been shown that efficient error detection predicates for specified locations, hence EDMs, can be generated based only on critical variables, where this notion of criticality has been shown to extend across software modules associated with these variables. These results were presented in the context of the thesis that:

Where an efficient EDM exists under the defined system model, that efficient EDM consists of a set of critical variables.

The key implications of this thesis are that, if an appropriate set of critical variables can be determined, then (i) the problem of EDM design is dramatically simplified, as the state space that a software engineer must consider in the design of error detection predicates is dramatically reduced, and (ii) the occurrence of critical variables can be used to inform the the location of EDMs.

8.2 Contribution Summary

In support of the stated thesis, the following specific contributions were made to the design of efficient error detection predicates for EDMs:

- A metric suite that generates a relative ranking of variables with respect to the notion of criticality applied throughout this thesis was proposed, as well as a fault injection approach for its evaluation. The metric suite was proposed with a view to facilitating the identification of critical variables. This identification is performed through the application of a threshold to the relative ranking generated, thus allowing a cost-benefit analysis to be undertaken in the design of error detection predicates.
- A systematic approach for the design of efficient error detection predicates was proposed based on the application of data mining techniques to

fault injection data sets. As well as being shown to provide an effective mechanism for the generation of efficient error detection predicates for real-world, infinite-state software systems, this approach was also used to demonstrate that error detection predicates generated using only critical variables achieve similar levels of efficiency as those generated using all program variables. This result is central to the stated thesis, as it implies that the set of critical variables identified by the proposed metric suite can be used to capture aspects of software system correctness.

- A methodology for the design of dependable software systems based on the replication of only critical variables using software wrappers was proposed and applied in order to demonstrate that significant dependability enhancements can be achieved through the protection of a relatively small number of critical variables. This result served to further substantiate the thesis that efficient EDMs consists of a set of critical variables, as the protection of relatively few program variables using error detection predicates that are known to be efficient, e.g., majority voting and variable replication, yields significant improvements in system failure rates.

The contributions detailed above represent novel work in the field of EDM design, where each of these contributions can be drawn together in support of the thesis that an efficient EDM consists of a set of critical variables.

8.3 Future Work

The design of efficient EDMs remains a key challenge in the development of fault tolerant software systems, particularly in the content of real-world, infinite state software systems. Through a focus on the criticality of variables and approaches for the generation of efficient error detection predicates, the research presented in this thesis has addressed this problem from a novel variable-centric perspective, allowing steps to be taken towards the design of efficient EDMs. Despite this progress, there are many areas for future work relating to the EDM

design problem, many of which are related to the notion of program variable criticality and the approach for the generation of efficient error detection mechanisms proposed in this thesis. Several areas for future research relating to the work presented in this thesis are discussed below.

Understanding the Inheritance of Criticality: The notion of program variable criticality established in this thesis is based on the potential worst-case impact of a program variable with respect to corruption in the spatial and temporal domains, as well as the capability of the program variables to induce a system failure when corrupted. Under the adopted system model, actions performed in the execution of a software system will necessarily have an impact on the criticality of a particular variable across different locations in a software module. Developing an understanding of how criticality is inherited through interactions between program variables, i.e., actions involving more than one program variables, would yield several positive results. In particular, such an understanding may facilitate the identification of the smallest subset of critical variables that should be incorporated by the error detection predicate of an EDM. In other words, it would be possible to simplify error detection predicates based on the identification of program variables where criticality originates. This identification may then lead to another potential benefit of understanding the inheritance of criticality among program variables, which is the location of EDMs based on the origins of criticality within a software module. This is subtly different from using occurrences of critical variables to inform the location of EDMs, as the identification of an origin of criticality may not necessarily be an ideal location for an EDM. For example, it is reasonable to speculate that an effective approach to EDM location may be to allow criticality to be inherited by variables to a particular threshold before an EDM must ascertain that the current system state is permissible. Aside from facilitating the simplification of error detection predicates and the location of EDMs, an understanding of how criticality is inherited among program variables could also serve to reduce

the execution overheads associated with the checking of error detection predicates. Indeed, whilst such considerations are beyond the scope of the research presented in this thesis, in order to appreciate this potential benefit it suffices to acknowledge that the cost, in terms of runtime overhead, associated with program variable accesses in non-uniform.

Locating EDMs based on Achievable Efficiency: In this thesis a systematic approach for the generation of efficient error detection predicates has been proposed. The premise of this approach was that, as fault injection analysis captures relationships among the program variables and the success of software system executions, data mining techniques can be applied to learn these relationships and how they impact the success of software executions, with a view to applying the derived relationships as error detection predicates for failure-inducing software system states at a specified location. If the specified location is varied then the levels of efficiency achieved by the associated EDMs may also vary. A possible approach to the EDM location problem would be to base the location of EDMs on the level of efficiency that can be achieved by generated EDMs. That is, following the generation of a set of EDMs for different locations in a software module, the EDMs at the locations that provide the greatest level of efficiency with respect to error detection could be deployed to impart software dependability. This would allow the EDM location problem to be circumvented and ensure that the EDMs deployed in a software system were efficient by design. A caveat on this approach is that locating EDMs based on achievable efficiency presumes that the measured efficiency is accurate, which itself depends on the representativeness of the adopted fault model.

Fault Models for the Design of Efficient EDMs: The fault model adopted during a dependable evaluation expresses the set of faults, and hence potentially erroneous states, that a system must be able to tolerate. The transient data fault model, as adopted in this thesis, is one of many possible fault models.

However, as mentioned previously, there is no guarantee that error detection predicates generated under one fault model will be effective under another fault model. With this in mind, it is important to understand how the selection of a fault model impacts the efficiency of the error detection predicates that can be generated by the approach proposed in this thesis. If such an understanding could be gained then it may be possible to determine the most effective, with respect to the efficiency of the EDMs derived, and representative, with respect to occurrences of faults in real-world software systems, fault models for the generation of error detection predicates. Indeed, it is anticipated that determining the most appropriate fault models for the generation of error detection predicates would facilitate the adoption of the contributions made in this thesis for the design of efficient EDMs.

Bibliography

[1] 7-Zip. http://www.7-zip.org/, September 2011.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353, November 2005.

[3] Y. Afek, H. Attiya, A. Fekete, M. Fischer, N. Lynch, Y. Mansour, D. Wang, and L. Zuck. Reliable communication over unreliable channels. *Journal of the ACM*, 41(6):1267–1297, November 1994.

[4] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. GOOFI: Generic object-oriented fault injection tool. In *Proceedings of the 31st IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 83–88, July 2001.

[5] N. Alexiou, S. Basagiannis, P. Katsaros, T. Dashpande, and S. A. Smolka. Formal analysis of the kaminsky DNS cache-poisoning attack using probabilistic model checking. In *Proceedings of the 12th International Symposium on High-Assurance Systems Engineering*, pages 94–103, November 2010.

[6] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):627–641, June 1999.

[7] T. Anderson and P. A. Lee. *Fault Tolerance - Principles and Practice*. Prentice-Hall, September 1981.

[8] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber. Comparison of physcial and software-implemented fault injection techniques. *IEEE Transactions on Computers*, 52(9):1115–1133, September 2003.

[9] J. Arlat, Y. Crouzet, and J.-C. Laprie. Fault injection for dependability evaluation of fault tolerant computing systems. In *Proceedings of the 19th International Symposium on Fault-Tolerant Computing*, pages 348–355, June 1989.

[10] A. Arora and M. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, September 1994.

[11] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

[12] A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, pages 436–443, May 1998.

[13] P. Asterio de C Guerra, C. M. F. Rubira, A. Romanovsky, and R. de Lemos. Integrating COTS software components into dependable software architectures. In *Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 139–142, May 2003.

[14] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transaction on Software Engineering*, 11(12):1491–1501, December 1985.

[15] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *Proceedings of the 1st IEEE-CS International Computer Software Applications Conference*, pages 149–155, November 1977.

[16] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January 2004.

[17] D. Avresky, J. Arlat, J.-C. Laprie, and Y. Crouzet. Fault injection for formal testing of fault tolerance. *IEEE Transactions on Reliability*, 45(3):443–455, September 1996.

[18] T. T. Bartolomei, K. Czarnecki, and R. Lammel. Swing to SWT and back: Patterns for API migration by wrapping. In *Proceedings of the 26th IEEE Conference on Software Maintenance*, pages 1–10, September 2010.

[19] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, November 2005.

[20] A. Bouti and D. A. Kadi. A state-of-the-art review of FMEA/FMECA. *International Journal of Reliability, Quality and Safety Engineering*, 1(4):515–543, January 1994.

[21] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Chapman and Hall/ CRC, January 1984.

[22] C. Brunk and M. J. Pazzani. An investigation of noise-tolerant relational concept learning algorithms. In *Proceedings of the 8th International Conference on Machine Learning*, pages 389–393, June 1991.

[23] G. Canfora, A. R. Fasolino, G. Frattolillo, and P. Tramontana. A flexible wrapper for the migration of interactive legacy system to web services. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, pages 343–344, March 2006.

[24] J. Carreira, H. Madeira, and J. Silva. Xception: a technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, February 1998.

[25] R. Chandra, R. M. Lefever, M. Cukier, and W. H. Sanders. Loki: A state-driven fault injector for distributed systems. In *Proceedings of the 30th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 237–242, June 2000.

[26] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16(1):321–357, May 2002.

[27] N. V. Chawla, D. A. Cieslak, L. O. Hall, and A. Joshi. Automatically countering imbalance and its empirical relationship to cost. *Journal of Data Minining and Knowledge Discovery*, 17(2):225–252, February 2008.

[28] N. Cheng, A. Berzins, Luqi, and S. Bhattacharya. Interoperability with distributed objects through java wrapper. In *Proceedings of the 24th Annual International Computer Software and Applications Conference*, pages 479–485, October 2000.

[29] B. Chess and J. West. *Secure Programming with Static Analysis: Getting Software Security Right with Static Analysis (1st Edition)*. Addison-Wesley, June 2007.

[30] A. P. Chester, M. Leeke, M. Al-Ghamdi, S. A. Jarvis, and A. Jhumka. A modular failure-aware resource allocation architecture for cloud computing. In *Proceedings of the 27th UK Performance Engineering Workshop*, pages 274–290, July 2011.

[31] A. P. Chester, M. Leeke, M. Al-Ghamdi, A. Jhumka, and S. A. Jarvis. A framework for data center scale dynamic resource allocation algorithms. In *Proceedings of the 10th IEEE International Conference on Scalable Computing and Communications*, pages 67–74, August 2011.

[32] S. Cheung and K. N. Levitt. A formal-specification based approch for protecting the domain name system. In *Proceedings of the 30th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 641–651, June 2000.

[33] J. Christmansson, M. Hiller, and M. Rimen. An experimental comparison of fault and error injection. In *Proceedings of the 9th IEEE International Symposium on Software Reliability Engineering*, pages 369–378, November 1998.

[34] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, January 2000.

[35] A. Cleve. Automating program conversion in database reengineering: A wrapper-based approach. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, pages 323–326, March 2006.

[36] W. W. Cohen. Fast effective rule induction. In *Proceedings of the 12th International Conference on Machine Learning*, pages 115–123, July 1995.

[37] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings 6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[38] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[39] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard. Inference and enforcement of data structure consistency spec-

ifications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 233–244, May 2006.

[40] P. Domingos. A general method for making classifiers cost-sensitive. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 155–164, July 1999.

[41] C. W. Ebeling. *An Introduction to Reliability and Maintainability Engineering*. McGraw-Hill, November 1996.

[42] S. H. Edwards, M. Sitaraman, and B. W. Weide. Contract-checking wrappers for C++ classes. *IEEE Transactions on Software Engineering*, 30(11):794–810, November 2004.

[43] J. Epstein, L. Thomas, and E. Monteith. Using operating system wrappers to increase the resiliency of commercial firewalls. In *Proceedings of the 16th Annual Conference on Computer Security Applications*, pages 236–245, December 2000.

[44] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[45] J.-C. Fabre, M. Rodriguez, J. Arlat, and J.-M. Sizun. Building dependable COTS microkernel-based systems using MAFALDA. In *Proceedings of the 7th Pacific Rim International Symposium on Dependalbe Computing*, pages 85–92, August 2000.

[46] W. Fan, S. J. Stolfo, J. Zhang, and P. K. Chan. Misclassification cost-sensitive boosting. In *Proceedings of the 16th International Conference on Machine Learning*, pages 97–105, June 1999.

[47] C. Fetzer, P. Felber, and K. Hogstedt. Automatic detection and masking of nonatomic exception handling. *IEEE Transactions on Software Engineering*, 30(8):547–560, August 2004.

[48] C. Fetzer and Z. Xiao. Detecting heap smashing attacks through fault containment wrappers. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, pages 80–89, August 2001.

[49] C. Fetzer and Z. Xiao. An automated approach to increasing the robustness of C libraries. In *Proceedings of the 32nd IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 155–164, December 2002.

[50] FlightGear. http://www.flightgear.org/, September 2011.

[51] FMEA/FMECA. http://www.fmea-fmeca.com/, September 2011.

[52] J. Furnkranz and G. Widmer. Incremental reduced error pruning. In *Proceedings of the 11th International Conference on Machine Learning*, pages 70–77, July 1994.

[53] F. Gartner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, March 1999.

[54] C. J. M. Geisterfer and S. Ghosh. Software component specification: A study in perspective of component selection and reuse. In *Proceedings of the 5th International Conference on Commercial-off-the-Shelf-Based Software Systems*, pages 9–18, February 2006.

[55] A. K. Ghosh, M. Schmid, and F. Hill. Wrapping windows NT software for robustness. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, pages 344–347, June 1999.

[56] J. H. Graham. FMECA control for software development. In *Proceedings of the 29th International Computer Software and Applications Conference*, volume 1, pages 93–96, July 2005.

[57] F. Guo, J. Chen, and T. Chiueh. Spoof detection for preventing dos attacks against dns servers. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, pages 37–44, July 2006.

[58] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explorations*, 11(1):10–18, June 2009.

[59] M. Hiller. Error recovery using forced validity assisted by executable assertions for error detection: An experimental evaluation. In *Proceedings of the 25th EUROMICRO Conference*, pages 105–112, August 1999.

[60] M. Hiller. Executable assertions for detecting data errors in embedded control systems. In *Proceedings of the 30th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 24–33, June 2000.

[61] M. Hiller, A. Jhumka, and N. Suri. An approach for analysing the propagation of data errors in software. In *Proceedings of the 31st IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 161–172, July 2001.

[62] M. Hiller, A. Jhumka, and N. Suri. On the placement of software mechanisms for detection of data errors. In *Proceedings of the 32nd IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 135–144, June 2002.

[63] M. Hiller, A. Jhumka, and N. Suri. PROPANE: An environment for examining the propagation of errors in software. In *Proceedings of the 11th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 81–85, July 2002.

[64] M. Hiller, A. Jhumka, and N. Suri. EPIC: Profiling the propagation and effect of data errors in software. *IEEE Transactions on Computers*, 53(3):512–530, May 2004.

[65] G. Hoffmann and M. Malek. Call availability prediction in a telecommunication system: A data driven empirical approach. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pages 83–95, October 2006.

[66] M. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, April 1997.

[67] IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance. http://www.dependability.org/wg10.4/, September 2011.

[68] N. Japkowicz. The class imbalance problem: Significance and strategies. In *Proceedings of the 2nd International Conference on Artificial Intelligence*, pages 111–117, June 2000.

[69] S. A. Jarvis, S. D. Hammond, and M. Leeke, editors. *Proceedings of the 26th UK Performance Engineering Workshop*, University of Warwick, Coventry, UK, July 2010.

[70] A. Jhumka, F. Freiling, C. Fetzer, and N. Suri. An approach to synthesise safe systems. *International Journal of Security and Networks*, 1(1):62–74, September 2006.

[71] A. Jhumka and M. Hiller. Putting detectors in their place. In *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods*, pages 33–42, September 2005.

[72] A. Jhumka, M. Hiller, and N. Suri. Assessing inter-modular error propagation in distributed software. In *Proceedings of the 20th IEEE International Symposium on Reliable Distributed Systems*, pages 152–161, January 2001.

[73] A. Jhumka, M. Hiller, and N. Suri. An approach to specify and test component-based dependable software. In *Proceedings of the 7th IEEE International Symposium on High Assurance Systems Engineering*, pages 211–218, October 2002.

[74] A. Jhumka, M. Hiller, and N. Suri. Component-based synthesis of dependable embedded software. *Lecture Notes in Computer Science*, 2469/2002:111–128, January 2002.

[75] A. Jhumka, M. Hiller, and N. Suri. An approach for designing and assessing detectors for dependable component-based systems. In *Proceedings of the 8th IEEE International Symposium on High Assurance Systems Engineering*, pages 69–78, March 2004.

[76] A. Jhumka and M. Leeke. Issues on the design of efficient fail-safe fault tolerance. In *Proceedings of the 20th IEEE International Symposium on Software Reliability Engineering*, pages 155–164, November 2009.

[77] A. Jhumka and M. Leeke. Early identification of locations for dependability components in dependable software. In *Proceedings of the 22nd IEEE International Symposium on Software Reliability Engineering*, pages 25–36, November 2011.

[78] A. Jhumka, M. Leeke, and S. Shrestha. On the use of fake sources for source location privacy: Trade-offs between energy and privacy. *The Computer Journal*, 54(6):860–874, June 2011.

[79] A. Johansson, A. Sarbu, A. Jhumka, and N. Suri. On enhancing the robustness of commercial operating systems. *Lecture Notes in Computer Science*, 3335/2005:148–159, January 2005.

[80] A. Johansson and N. Suri. Error propagation profiling of operating systems. In *Proceedings of the 35th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 86–95, June 2005.

[81] G. H. John and P. Langley. Estimating continuous distributions in bayesian classifiers. In *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence*, pages 338–345, May 1995.

[82] B. W. Johnson. *Design and Analysis of Fault-Tolerant Digital Systems*. Addison-Wesley Series in Electrical and Computer Engineering. Addison-Wesley, January 1989.

[83] E. Jonsson. Towards an integrated conceptual model of security and dependability. In *Proceedings of the 1st International Conference on Availability, Reliability and Security*, pages 646–653, April 2006.

[84] T. M. Khoshgoftaar, E. B. Allen, W. H. Tang, C. C. Michael, and J. M. Voas. Identifying modules which do not propagate errors. In *Proceedings of the IEEE Symposium on Application-Specific Systems and Software Engineering and Technology*, pages 185–193, March 1999.

[85] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transaction on Software Engineering*, 13(1):23–31, January 1987.

[86] P. J. Koopman and J. De Vale. The exception handling effectiveness of posix operating systems. *IEEE Transactions on Software Engineering*, 26(9):837–848, September 2000.

[87] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing*, pages 30–37, June 1998.

[88] M. Kubat, R. C. Holte, and S. Matwin. Machine learning for the detection of oil spills in satellite radar images. *Machine Learning*, 30(2-3):195–215, 1998.

[89] M. Kubat and S. Matwin. Addressing the curse of imbalanced training sets: One-sided selection. In *Proceedings of the 14th International Conference on Machine Learning*, pages 179–186, January 1997.

[90] C. Kuhnel, A. Bauer, and M. Tautschnig. Compatibility and reuse in component-based systems via type and unit inference. In *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 101–108, August 2007.

[91] S. S. Kulkarni and A. Ebnenasir. Complexity of adding failsafe fault-tolerance. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems*, pages 337–344, July 2002.

[92] J.-C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Proceedings of the 15th IEEE International Symposium on Fault-Tolerant Computing*, pages 2–11, June 1985.

[93] J.-C. Laprie. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, December 1992.

[94] J.-C. Laprie. Dependability of computer systems: Concepts, limits, improvements. In *Proceedings of the 6th International Symposium on Software Reliability Engineering*, pages 2–11, October 1995.

[95] S. Le Cessie and J. C. Van Houwelingen. Ridge estimators in logistic regression. *Applied Statistics*, 41(1):191–201, January 1992.

[96] M. Leeke, S. Arif, A. Jhumka, and S. S. Anand. A methodology for the generation of efficient error detection mechanisms. In *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 25–36, June 2011.

[97] M. Leeke and A. Jhumka. Beyond the golden run: Evaluating the use of reference run models in fault injection analysis. In *Proceedings of the 25th UK Performance Engineering Workshop*, pages 61–74, July 2009.

[98] M. Leeke and A. Jhumka. Evaluating the use of reference run models in fault injection analysis. In *Proceedings of the 15th Pacific Rim Interna-*

tional Symposium on Dependable Computing, pages 121–124, November 2009.

[99] M. Leeke and A. Jhumka. On the tradeoff between privacy and energy in wireless sensor networks. In *Proceedings of the 26th UK Performance Engineering Workshop*, pages 103–110, July 2010.

[100] M. Leeke and A. Jhumka. Towards understanding the importance of variables in dependable software. In *Proceedings of the 8th European Dependable Computing Conference*, pages 85–94, April 2010.

[101] M. Leeke and A. Jhumka. An automated wrapper-based approach to the design of dependable software. In *Proceedings of the 4th International Conference on Dependability*, pages 43–50, August 2011.

[102] N. G. Leveson, S. S. Cha, J. C. Knight, and T. J. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering*, 16(4):432–443, April 1990.

[103] D. D. Lewis and J. Catlett. Heterogeneous uncertainty sampling for supervised learning. In *Proceedings of the 11th International Conference on Machine Learning*, pages 148–156, June 1994.

[104] T. Lewis. Will tiny beans conquer the world again? *Computer*, 29(9):13–15, September 1996.

[105] C.-T. Lu, A. P. Boedihardjo, and P. Manalwar. Exploiting efficient data mining techniques to enhance intrusion detection systems. In *Proceedings of the 4th IEEE International Conference on Information Reuse and Integration*, pages 512–517, September 2005.

[106] K. Z. Mao. Fast orthogonal forward selection algorithm for feature subset selection. *IEEE Transactions on Neural Networks*, 13(5):1218–1224, September 2002.

[107] A. C. Marosi, Z. Balaton, and P. Kacsuk. Genwrapper: A generic wrapper for running legacy applications on desktop grids. In *Proceedings of the 23rd International Symposium on Parallel and Distributed Computing*, pages 1–6, May 2009.

[108] Microsoft - MSDN. http://msdn.microsoft.com/library/ms242709.aspx, September 2011.

[109] T. Mitchem, R. Lu, and R. O'Brien. Using kernel hypervisors to secure applications. In *Proceedings of the 13th Annual Confernece on Computer Security Applications*, pages 175–181, December 1997.

[110] T. Mitchem, R. Lu, R. O'Brien, and R. Larson. Linux kernel loadable wrappers. In *Proceedings of the 1st DARPA Information Survivability Conference*, pages 296–307, January 2000.

[111] L. Morell, B. Murrill, and R. Rand. Peturbation analysis of computer programs. In *Proceedings of the 12th Annual Conference on Computer Assurance*, pages 77–87, June 1997.

[112] Mp3Gain. http://mp3gain.sourceforge.net/, September 2011.

[113] N. Nagappan and T. Ball. Static analysis tools and early indicators of pre-release defect density. In *Proceedings of the 27th ACM/IEEE International Conference on Software Engineering*, pages 580–586, May 2005.

[114] P. M. Narendra and K. Fukunaga. A branch and bound algorithm for feature subset selection. *IEEE Transactions on Computers*, C-26(9):917–922, September 1977.

[115] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(2):111–122, March 2002.

[116] M. Ortega, A. Griman, M. Perez, and L. E. Mendoza. Reuse strategy based on quality certification of reusable components. In *Proceedings of the*

8th IEEE International Conference on Information Reuse and Integration, pages 140–145, August 2007.

[117] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Application-based metrics for strategic placement of detectors. In *Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing*, pages 75–82, December 2005.

[118] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Automated derivation of application-aware error detectors using static analysis. In *Proceedings of the 13th IEEE International On-Line Testing Symposium*, pages 211–216, July 2007.

[119] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Automated derivation of application-aware error detectors using static analysis: The trusted illiac approach. *IEEE Transactions on Dependable and Secure Computing*, 8(1):44–57, January 2011.

[120] K. Pattabiraman, G. P. Saggese, D. Chen, Z. Kalbarczyk, and R. Iyer. Dynamic derivation of application-specific error detectors and their implementation in hardware. In *Proceedings of the 6th Dependable Computing Conference*, pages 97–106, October 2006.

[121] K. Pattabiraman, G. P. Saggese, D. Chen, Z. Kalbarczyk, and R. K. Iyer. Automated derivation of application-specific error detectors using dynamic analysis. *IEEE Transactions on Dependable and Secure Computing*, 99:44–57, May 2010.

[122] M. J. Pazzani, C. Merz, P. Murphy, K. Ali, T. Hume, and C. Brunk. Reducing misclassification costs. In *Proceedings of the 11th International Conference on Machine Learning*, pages 217–225, July 1994.

[123] W. W. Peterson and E. J. Weldon Jr. *Error-Correcting Codes (2nd Revised Edition)*. MIT Press, January 1972.

[124] G. Pinter, H. Madeira, M. Vieira, A. Pataricza, and I. Majzik. A data mining approach to identify key factors in dependability experiments in dependable computing. In *Proceedings of the 5th European Dependable Computing Conference*, pages 263–280, March 2005.

[125] D. Powell. Failure model assumptions and assumption coverage. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 386–395, July 1992.

[126] D. Powell, E. Martins, J. Arlat, and Y. Crouzet. Estimators for fault tolerance coverage evaluation. *IEEE Transactions on Computers*, 44(2):261–274, June 1995.

[127] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, August 1990.

[128] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, October 1992.

[129] J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In *Proceedings of the 6th European Conference on Machine Learning*, pages 3–20, April 1993.

[130] C. Rabejac, J.-P. Blanquart, and J.-P. Queille. Executable assertions and timed traces for on-line software error detection. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing*, pages 138–147, June 1996.

[131] B. Randell. System structure for software fault tolerance. *IEEE Transaction on Software Engineering*, 1(2):221–232, June 1975.

[132] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th ACM/IEEE International Conference on Software Engineering*, pages 105–118, July 1992.

[133] M. Rodriguez, F. Salles, J.-C. Fabre, and J. Arlat. MAFALDA: Microkemel assessment by fault injection and design aid. In *Proceedings of the 3rd European Dependable Computing Conference*, pages 143–160, September 1999.

[134] J. Rushby. Critical systems properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219, June 1994.

[135] S. K. Sahoo, M. Li, P. Ramachandran, S. V. Adve, V. S. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. In *Proceedings of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 70–79, June 2008.

[136] F. Salles, M. Rodriguez, J.-C. Fabre, and J. Arlat. Metakernels and fault containment wrappers. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, pages 22–29, November 1999.

[137] M. Schmid and F. Hill. Data generation techniques for automated software robustness testing. In *Proceedings of the 16th International Conference on Testing Computer Software*, pages 14–18, April 1999.

[138] D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 38–48, January 1998.

[139] P. Sewell. Secure composition of untrusted code: Box pi, wrappers and causality types. *Journal of Computer Security*, 11(2):135–187, May 2003.

[140] P. Sewell and J. Vitek. Secure composition of untrusted code. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 136–150, June 1999.

[141] P. Sewell and J. Vitek. Secure composition of untrusted code: Wrappers and causality types. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 269–284, July 2000.

[142] H. Shah, C. Gorg, and M. J. Harrold. Understanding exception handling: Viewpoints of novices and experts. *IEEE Transaction on Software Engineering*, 36(2):150–161, March 2010.

[143] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(523-656):379–423, July 1948.

[144] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation (3rd Edition)*. A K Peters / CRC Press, December 1998.

[145] I. Sommerville. *Software Engineering*. Addison-Wesley, 8 edition, June 2006.

[146] T. Souder and S. Mancoridis. A tool for securely integrating legacy system into a distributed environment. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 47–55, October 1999.

[147] B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In *Proceedings of the 25th ACM/IEEE International Conference on Software Engineering*, pages 374–384, May 2003.

[148] A. Steininger and C. Scherrer. On finding optimal combinations of error detection mechanisms based on results of fault injection experiments. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, pages 238–247, June 1997.

[149] W. Stevens, G. Myers, and L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, June 1974.

[150] P. Stocks and D. Carrington. A framework for specification based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

[151] N. Storey. *Safety-Critical Computer Systems*. Prentice-Hall, July 1996.

[152] D. T. Stott, B. Floering, Z. Kalbarczyk, and R. K. Iyer. NFTAPE: A framework for assessing dependability in distributed systems with

lightweight fault injectors. In *Proceedings of the 4th International Symposium on Computer Performance and Dependability*, pages 91–100, June 2000.

[153] N. Suri, S. Ghosh, and T. Marlowe. A framework for dependability driven software integration. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS)*, pages 406–415, May 1998.

[154] M. Susskraut and C. Fetzer. Robustness and security hardening of COTS software libraries. In *Proceedings of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 61–71, June 2007.

[155] A. S. Tanenbaum, J. N. Herder, and H. Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, May 2006.

[156] P. Thiran, J. Hainaut, and G. Houben. Database wrappers development: Towards automatic generation. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering*, pages 207–216, March 2005.

[157] K. M. Ting. An instance-weighting method to induce cost-sensitive trees. *IEEE Transactions on Knowledge and Data Engineering*, 14(3):659–665, May 2002.

[158] R. Vemu and J. A. Abraham. CEDA: Control-flow error detection through assertions. In *Proceedings of the 12th IEEE International On-Line Testing Symposium*, pages 6–11, July 2006.

[159] J. Vinter, J. Aidemark, P. Folkesson, and J. Karlsson. Reducing critical failures for control algorithms using executable assertions and best effort recovery. In *Proceedings of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 347–356, July 2001.

[160] J. Voas and L. Morell. Propagation and infection analysis (pia) for debugging software. In *Proceedings of the IEEE Southeastcon*, volume 2, pages 379–383, April 1990.

[161] J. M. Voas. PIE: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, August 1992.

[162] J. M. Voas. Building software recovery assertions from a fault injection-based propagation analysis. In *Proceedings of the 21st International Computer Software and Applications Conference*, pages 505–510, August 1997.

[163] J. M. Voas, F. Charron, and L. Beltracchi. Error propagation analysis studies in a nuclear research code. In *Proceedings of the IEEE Aerospace Conference*, volume 4, pages 115–121, March 1998.

[164] J. M. Voas and K. W. Miller. Putting assertions in their place. In *Proceedings of the 5th International Symposium on Software Reliability Engineering*, pages 152–157, November 1994.

[165] H. Volzer. Verifying fault tolerance of distributed algorithms formally - an example. In *Proceedings of the 1st International Conference on the Application of Concurreny to System Design*, pages 187–197, March 1998.

[166] D. Watkins. CORBA and DCOM: Architectures for distributed computing. In *Proceedings of the 29th Technology of Object-Oriented Languages and Systems*, pages 401–406, June 1999.

[167] S. J. Wenke Lee Stolfo and K. W. Mok. A data mining framework for building intrusion detection models. In *Proceedings of the 20th IEEE Symposium on Security and Privacy*, pages 120–132, May 1999.

[168] K. Wilken and J. P. Shen. Continuous signature monitoring: Low-cost concurrent detection of processor control errors. *IEEE Transactions on Computer-Aided Design*, 9(6):629–641, June 1990.

[169] H. Winroth. A scripting language interface to C++ libraries. In *Proceedings of the 23rd Technology of Object-Oriented Languages and Systems Conference*, pages 247–259, August 1997.

[170] E. Wohlstadter, S. Jackson, and P. Devanbu. Generating wrappers for command line programs: The cal-aggie wrap-o-matic project. In *Proceedings of the 23rd ACM/IEEE International Conference on Software Engineering*, pages 243–252, May 2001.

[171] J. Yang and V. Honavar. Feature subset selection using a genetic algorithm. *IEEE Intelligent Systems and their Applications*, 13(2):44–49, March 1998.

[172] B. Zadrozny, J. Langford, and N. Abe. Cost-sensitive learning by cost-proportionate example weighting. In *Proceedings of the 3rd IEEE International Conference on Data Mining*, pages 435–442, July, 2003.

[173] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, April 2006.