CRANFIELD UNIVERSITY

JEREMY GARDINER

DELAYED FAILURE OF SOFTWARE COMPONENTS USING STOCHASTIC TESTING

CRANFIELD DEFENCE AND SECURITY

PhD THESIS

CRANFIELD UNIVERSITY

CRANFIELD DEFENCE AND SECURITY

DEPARTMENT OF INFORMATICS AND SYSTEMS ENGINEERING

PhD THESIS

Academic Year 2011-2012

Jeremy Gardiner

Delayed Failure of Software Components Using Stochastic Testing

Supervisor: Dr. Graham Fletcher

October 2011

ABSTRACT

The present research investigates the delayed failure of software components and addresses the problem that the conventional approach to software testing is unlikely to reveal this type of failure. Delayed failure is defined as a failure that occurs some time after the condition that causes the failure, and is a consequence of long-latency error propagation. This research seeks to close a perceived gap between academic research into software testing and industrial software testing practice by showing that stochastic testing can reveal delayed failure, and supporting this conclusion by a model of error propagation and failure that has been validated by experiment. The focus of the present research is on software components described by a request-response model. Within this conceptual framework, a Markov chain model of error propagation and failure is used to derive the expected delayed failure behaviour of software components. Results from an experimental study of delayed failure of DBMS software components MySQL and Oracle XE using stochastic testing with random generation of SQL are consistent with expected behaviour based on the Markov chain model. Metrics for failure delay and reliability are shown to depend on the characteristics of the chosen experimental profile. SQL mutation is used to generate negative as well as positive test profiles. There appear to be few systematic studies of delayed failure in the software engineering literature, and no studies of stochastic testing related to delayed failure of software components, or specifically to delayed failure of DBMS. Stochastic testing is shown to be an effective technique for revealing delayed failure of software components, as well as a suitable technique for reliability and robustness testing of software components. These results provide a deeper insight into the testing technique and should lead to further research. Stochastic testing could provide a dependability benchmark for component-based software engineering.

ACKNOWLEDGMENTS

I would like to thank the following people for their support and guidance on the long winding road of my PhD research. My supervisor, Dr. Graham Fletcher. Former supervisors and members of the thesis committee, Dr. Stuart Reid, Ian Whitworth, Ray Burcham, Dr. Lindsey Gillies, Prof. Peter Hill, Dr. Jacob Mulenga, and Dr. John Pryce. Colleagues in the wider world of software engineering, Dr. Steve Counsell, Dr. Cem Kaner, Pat McGee, Dr. Alex Milton, Robin Murison, and Ian Smith. Lastly, I would like to thank my wife, Caroline, without whose support this undertaking would not have been possible.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

GLOSSARY

BNF	Backus-Naur Form
COTS	Commercial Off-The-Shelf
BS	British Standard
DBMS	Data Base Management System
ERR	Extended Random Regression
HVAT	High Volume Automated Testing
IEEE	Institute of Electrical and Electronics Engineers
IPOS	Input, Process, Output, Storage
ISTQB	International Software Testing Qualification Board
MBT	Model-Based Testing
MTBF	Mean Time Between Failures
MTTF	Mean Time To Failure
RAGS	Random Generation of SQL
SQL	Structured Query Language
SUT	Software Under Test

All programs and experimental data for the present research are available at:
http://www.woomerang.com/research/PhD/.

1 INTRODUCTION

1.1 Introduction

"Success is only a delayed failure" – Graham Greene, A Sort of Life (1971)

The subject of this thesis is delayed failure of software components using stochastic testing. Failure of computer software is a serious problem in a society increasingly dependent on software-based systems, for example Lake (2010) describes eleven 'infamous software bugs' affecting space-exploration mission software, business system software, medical and military software. Software failures are caused by errors that arise due to faults in the software that have been introduced during software development (Ammann & Offutt, 2008). A *delayed failure* is defined for the present research as a failure that occurs some time after the condition that causes the failure; this thesis advances the proposition that delayed failure of software components is a consequence of long-latency error propagation and that the conventional approach to software testing is unlikely to detect this type of software failure.

The present research has conducted an experimental study of delayed failure of data base management system (DBMS) software components (MySQL and Oracle XE) using stochastic high volume automated testing (HVAT) with random generation of SQL (structured query language). The scope and context of the present research is limited to software components described by a 'stateless' request-response model, and delayed failures resulting from long-latency error propagation described by a state-transition model of error propagation and failure; these two models together provide the 'conceptual framework' for the research.

This chapter presents an introduction to software component testing and terminology, motivation for the present research, an introduction to delayed failure, the research problem, experimental studies of DBMS, key elements and focus of the research, the contributions to knowledge, and lastly the structure of the thesis.

1.2 Software Component Testing

Software component testing is a relatively new and very active area of research that reflects the increasing use of components in software engineering. The British Computer Society (BCS) Testing Standards Working Party defined BS 7925-2 Standard for Software Component Testing (working draft 3.4) in 2001; Reid (2000) discusses the development of this standard. The software component testing standard BS 7925-2 (2001) defines a software component as "*a minimal software item for which a separate specification is available*".

Software testing has been defined as "*the process of executing software with the intent of detecting faults"* (Myers, et al., 2004). Software testing is concerned with the assessment of software quality factors, such as correctness, performance, reliability, and security (Pan, 1999-b).

In conventional software testing, testers use their skill and judgement to select appropriate conditions (typically, particular combinations of input value and software state) as test cases to be executed against the software under test (SUT). The resulting behaviour of the SUT (typically, a particular output value) is compared with the expected behaviour and if these are different, a failure has occurred and a fault in the software is suspected (Myers, et al., 2004). In general, the output value produced by a software component depends on both the input value and on the current software state; the software 'state' is the aggregate of the values of all program variables held in memory at an instant of time (IEEE, 1991).

Since a software component may have many thousands of possible input values and many thousands of possible states, exhaustive testing of all combinations of input value and state is not usually possible (Kaner, 2003-a). Consequently, faults can often remain undetected after testing and these may lead to failure of the software after it has been released.

This section introduces fundamental approaches to software testing, high volume automated testing, component-based software engineering, software component reliability, reliability measures, software component robustness, and stochastic testing.

1.2.1 Fundamental Approaches to Software Testing

Software testing techniques have been categorised on the basis of their contrasting fundamental approaches: *static* vs. *dynamic* testing, and *black-box* vs. *white-box* testing (Lewis, 2009). In static testing, the software specification and/or program code is examined for faults without executing the program; while in dynamic testing, faults are revealed by running the program on a computer and observing its behaviour. In black-box testing, test cases are based on the functional and other specified requirements of the software without considering the details of the program code; while in white-box testing, test cases are based on knowledge of the program code and design (Beizer, 1990; Utting & Legeard, 2007). These fundamental approaches to software testing suggest categories of software testing techniques, represented as a quadrant diagram in Figure 1.

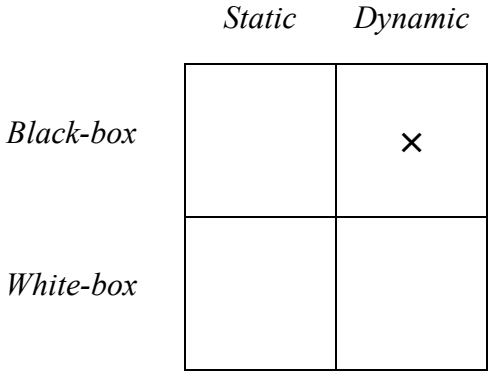


Figure 1: Fundamental approaches to software testing

Each of the four quadrants in Figure 1 may be considered an important area of software testing research in its own right, for example the upper left-hand quadrant in Figure 1 can be characterised as *static black-box testing* in which the functional and other specified requirements of the software are examined without executing the program.

However, the focus of the present research falls within the upper right-hand quadrant marked as an X in Figure 1, characterised as *dynamic black-box testing*. Within the category of dynamic black-box testing, two further contrasting fundamental approaches can be identified: *positive testing* vs. *negative testing*. Positive testing employs only valid input values in order to demonstrate that software conforms to its specification (conformance testing), whereas negative testing employs invalid input values (or a mixture of valid and invalid values) to test software robustness; software testing will generally include both positive and negative testing (Beizer, 1990; Utting & Legeard, 2007).

1.2.2 Component-Based Software Engineering

The goal of component-based software engineering is to build software systems by the composition of pre-existing software components; this is expected to lead to improvements in software reliability and reduced cost, especially with the advent of commercial off-the-shelf (COTS) components (Gao, Tsao & Wu, 2003).

It is usual to test software components both individually before composition and afterwards as an integrated system (Bhor, 2001) because software systems may exhibit emergent behaviour that cannot be predicted by considering the components in isolation (Johnson, 2006).

A necessary corollary of the goal of component-based software engineering is to be able to predict the properties of the assembled system, and for this it is first necessary to characterise the properties of the individual components (Hamlet, Mason & Woit, 1999). Firesmith (2005) gives the following quality factors for software components: 'capacity, correctness, dependability, interoperability, performance, and utility'; dependability is itself composed of the quality factors *availability*, *defensibility*, *reliability*, and *robustness*, while defensibility is in turn composed of the quality factors *safety*, *security*, and *survivability*. Since stochastic testing can be used to characterise the reliability and robustness of software components, these properties are investigated in the present research.

1.2.3 Software Component Reliability

Reliability has been defined as "*the ability of a software component to perform its required functions under stated conditions for a specified period of time, or for a specified number of operations"* (IEEE, 1991).

In principle, software component reliability might be determined by the use of statistical testing, that is random testing with a probability distribution ('operational profile') based on the expected use of the component in operation (Musa, 1993; Koziolek, 2005). In practice, an operational profile is difficult to define for a software component, because all the possible ways in which the component might be used are unknown, and measures of software reliability using operational profiles are open to criticism (Hamlet, 1994). However, components might be compared on the basis of their measured reliability for a specified test profile as one factor of design trade-off in component-based software engineering (Hamlet, Mason & Woit, 1999).

1.2.4 Reliability Measures

Pan (1999-a) has remarked: "*Measuring software reliability remains a difficult problem because we don't have a good understanding of the nature of software. There is no clear definition to what aspects are related to software reliability. We can not find a suitable way to measure software reliability, and most of the aspects related to software reliability.*" A further difficulty is that measures of software reliability depend on the operational profile used, so that testing with a different operational profile is likely to result in a different value for measured reliability (Whittaker & Voas, 2000). Nevertheless, reliability measures originally derived from theories of hardware reliability have been applied to software reliability (Fenton & Pfleeger, 1997).

A commonly used measure of software reliability is the Mean Time To Failure (MTTF); MTTF is a statistical measure defined as "*the expected time that a system will operate before the first failure occurs"* (Storey, 1996).

An equivalent measure of software reliability is failure rate, denoted by λ and defined as "*failures per unit time or per number of transactions"* (IEEE, 1991); for a constant failure rate λ then MTTF = 1/λ (Storey, 1996).

In software testing, an alternative measure to MTTF is the *F-measure* defined as "*the number of test cases required to detect the first program failure"*. The F-measure can be shown to be equivalent to MTTF (Chen, Kuo & Merkel, 2004).

Other measures of software reliability include the Mean Time Between Failures (MTBF) defined as "*the expected or observed time between consecutive failures in a system or component*" and availability *A* defined as "*the degree to which a system or component is operational and accessible when required for use"* (IEEE, 1991).

The Mean Time To Repair (MTTR) is defined as "*the expected or observed time required to recover a system or component and return it to normal operation"* (IEEE, 1991). The MTTR is also known as the Mean Time To Recovery (Kyne, et al. 2010).

By definition, MTBF = MTTF + MTTR and *A* = MTTF / MTBF (Teorey & Ng, 1998).

The reliability measures MTBF and availability are not used in the present research, since these depend on the MTTR and this is not a factor in the experiments.

In the present research, the F-measure during stochastic testing with a 'positive' profile containing only valid input values will be recorded, and it is argued that this provides a comparative measure of software component reliability.

1.2.5 Software Component Robustness

IEEE (1991) defines robustness as "*the degree to which a component or system can function correctly in the presence of invalid inputs or stressful environmental conditions*." From this definition it can be inferred that robustness testing is negative

testing, employing only invalid input values (or a mixture of valid and invalid input values) (Varpiola & Takanen, 2008).

One approach to the generation of negative test cases is by 'mutation' of positive test cases (data mutation) as described by BS 7925-2 (2001) and Shan & Zhu (2006, 2007) and this is the approach taken in the present research.

Robustness testing is generally not concerned with the correctness of the software (comparison with the expected result) but with 'terminal' failures such as a software crash or hang; robustness testing therefore does not require a sophisticated test oracle, beyond the general principle that the software should not crash or hang (Koopman, et al. 2002).

In the present research, the F-measure during stochastic testing with a 'negative' profile containing only invalid input values (or a mixture of valid and invalid values) will be recorded, and it is argued that this provides a comparative measure of software component robustness.

1.2.6 Dependability

Reliability and robustness are subsumed in the more general concept of dependability, as discussed by Avizienis, Laprie & Randell (2004), Firesmith (2005) and Koziolek (2005).

Vieira & Madeira (2009) define a dependability benchmark as "*a specification of a standard procedure to measure both the dependability and performance of computer systems or components"*. Several authors have discussed software dependability benchmarks; Madeira, et al. (2002) present a framework for dependability benchmarks and give examples including general purpose operating systems, on-line transaction processing systems, and embedded systems.

Such dependability benchmarks might be used as a basis for comparison of software components.

1.2.7 Stochastic Testing

Stochastic testing is a software testing technique that employs an extended random sequence of test cases to exercise the software under test; stochastic testing is expected to be an effective testing technique for software reliability and robustness (Mao & Lu, 2005).

Stochastic testing effectively traces a path between random points in the 'test space' of input value and software state; this is illustrated by a two-dimensional scatter diagram of random points with axes of software state and input value in Figure 2. The diagram in Figure 2 was generated using a computer program written by the author.

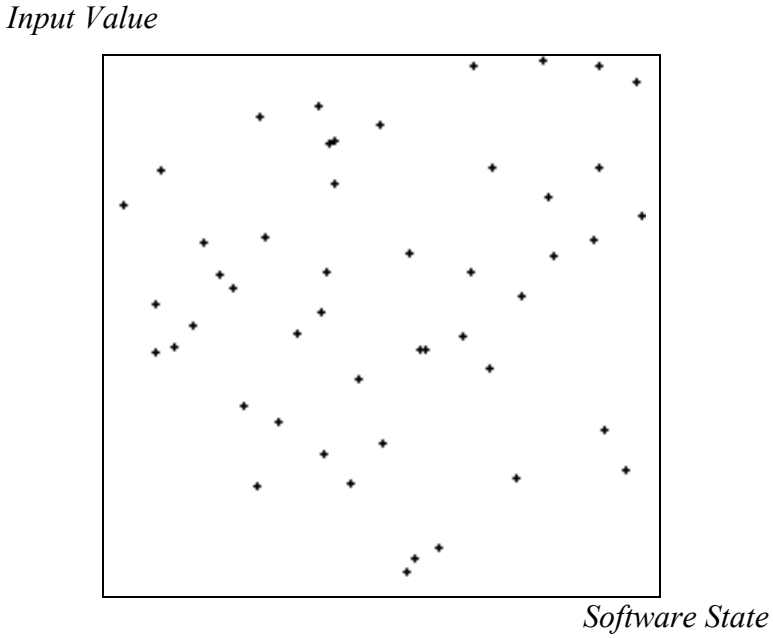


Figure 2: Random data points in the test space

Stochastic testing is important for the present research because an extended sequence of test cases does not continually reset the software state, unlike the conventional approach

to software testing with test 'setup' and 'cleanup' steps. Stochastic testing allows software errors to propagate beyond a single test case and is therefore a good candidate as a testing technique for delayed failure.

1.3 Terminology

This section gives definitions of special technical terms used in this thesis: fault, failure, error, error propagation, and latency. Definitions of general software testing terminology can be found in the following standard references:

BS 7925-1 (2004) Glossary of Software Testing Terms
IEEE (1991) Standard Computer Dictionary 610
ISTQB (2007) Standard Glossary of Terms used in Software Testing.

1.3.1 Fault, Failure, and Error

Prasad, McDermid & Wand (1996) have noted that the terms *fault, failure,* and *error* are not used consistently in the software engineering literature. The following definitions are adapted from software testing terminology in the standard references, and will be used throughout this thesis:

Fault *Deviation of program code from its intended design, or a mistake in the component design*; a fault is also known as a *defect* or a *bug*.

Failure *Deviation of software component behaviour from its specification*, where a specification is defined as "*a description of a component's function in terms of its output values for specified input values under specified preconditions"* (BS 7925-1, 2004).

Error *An incorrect software state; a state that was not intended by the designers of the component*.

1.3.2 Error Propagation and Latency

Yim, Kalbarczyk & Iyer (2009) introduce the following terms in the context of a model of error propagation and latency, as shown in Figure 3.

Activation A fault is *dormant* until it is *activated* and causes an error. Activation may occur for example due to execution of program code that contains a fault.

Propagation An error is *latent* until it causes a failure. An error may be transformed into other errors in a chain of error propagation until a failure occurs.

Latency *Fault activation latency* is the period between the start of software execution and fault activation. *Error propagation latency* is the period between fault activation and failure.

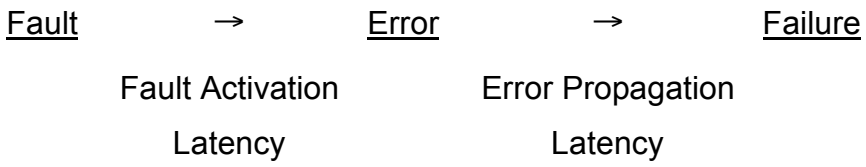


Figure 3: Error propagation and latency

1.4 Motivation

The author worked as a software test engineer in the telecoms industry from 1987 to 2004. The motivation for the present research originated in the author's (unpublished) experience of high-volume automated testing and model-based testing (MBT) between 1997 and 1999.

This section describes a fundamental problem encountered in software testing, how this problem led to the idea of using a stateless request-response model in software testing, and how this model in turn led to the idea of delayed failure of software.

1.4.1 A Fundamental Problem in Software Testing

A fundamental problem encountered in software testing is the 'coverage problem' – the location of software faults in the test space is unknown, and the region of the test space covered in testing may or may not overlap the region of the test space containing faults, as illustrated by Slutz (1998). This problem can be paraphrased as, 'how to do sufficient testing?' given that "*complete testing is impossible*" Kaner (1997).

In the author's experience, manual test case design was found to be too slow to allow a sufficiently large testing space to be covered with reasonable time and effort, and so an attempt was made to address this problem by using high volume automated testing to vastly increase the covered region of the test space.

1.4.2 High Volume Automated Testing

Automated software-testing tools can perform automatic test execution and automated comparison of results, and these tools can increase testing speed and reduce testing errors; however, even with automatic test execution, manual generation of test cases places a serious limitation on the volume of tests that can be executed. High volume automated testing (HVAT) addresses this problem by the automatic generation and execution of very large numbers of test cases (Kaner, Bond & McGee, 2003).

With HVAT, either deterministic or randomly generated test cases are produced automatically from a specification or model of the SUT, by the use of suitable software tools. For automated test execution, automatic comparison of test results with the expected outputs is performed, and so automatic generation of expected outputs by a 'test oracle' is required. Since the output response from the SUT generally depends on both the input request and on the state of the software, the test oracle must include a description of the software state (Nyman, 2000). However, in the author's experience,

the creation and maintenance of a state-based automated test oracle with reasonable time and effort was a problem, and a simpler approach was sought; a 'stateless request-response' model of the SUT was devised in response to this problem.

1.4.3 Stateless Request-Response Model

A common model of communication between software components is the request-response, or client-server model as described by Martin-Flatin (2005) and shown in Figure 4. In this communication model, a client sends a request to a server, and the server sends back a response containing the requested information.

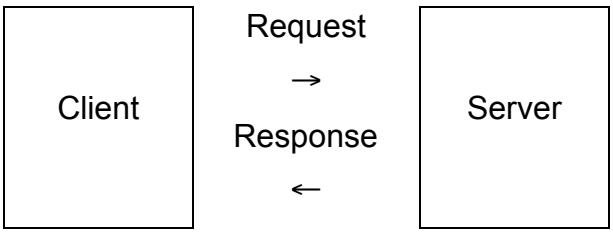


Figure 4: Client-server model

In the stateless request-response model devised by the author, the SUT is regarded as a 'black box' that regardless of its internal state, should either accept a valid input request, or reject an invalid input request. For any given input request, it should be possible to determine from the software specification if this is a valid request that should be accepted, or if this is an invalid request that should be rejected. This stateless request-response model provided the basis of a simplified automatic test oracle. With this oracle, a failure of the SUT should be detected if a valid input request is rejected, or if an invalid input request is accepted (or if no output response is returned at all, as in the case of a crash or hang of the SUT) regardless of the software state.

1.4.4 Delayed Failure

Despite the more limited ability to detect failures, compared to a state-based oracle, in the author's experience the stateless request-response model provided a cost effective

approach, complementary to manual testing; however, this approach raised the question of what was the resulting state of the SUT, if an invalid input request was accepted.

The present research into delayed failure of software components was motivated by the following questions: If the SUT continues to correctly process subsequent input requests after accepting an invalid input request, can it be relied upon to continue to operate correctly? Might it fail unpredictably some time later, due to being left in an incorrect, but non-observable state?

These questions immediately suggested the further idea, that delayed failure might potentially also occur when processing valid input requests, leading to 'invalid' and 'valid' instances of delayed failure, and furthermore it seemed likely from the author's experience that delayed failures might not be detected by the 'conventional' approach to software testing.

1.4.5 The Conventional Approach To Software Testing

The conventional approach to software testing (whether manual or automated) is to include 'setup' and 'cleanup' steps in test cases, so that the test cases are executed independently and with predictable results. This avoids the need for a complex state-based test oracle, and keeps test traces short and hence easier to investigate following a test failure (see 'identifying failing input requests' in Chapter 2); a test trace is a history of requests and responses during the execution of a test case.

Typical steps in a conventional test case are (Haftmann, Kossmann & Lo, 2007):

1. Test setup (initial software state for test)
2. Test execution (request-response sequence)
3. Test cleanup (undo changes to software state)

However, this approach prevents software errors from propagating beyond a single test case (Nyman, 2000) and so this approach is unlikely to reveal delayed failures that

result from long-latency error propagation; faults that cause delayed failures might therefore avoid detection and remain in released software, resulting in lower reliability and robustness.

1.5 Delayed Failure

A delayed failure is a failure that occurs some time after the condition that causes the failure. Delayed failure in a software component can be a result of long-latency error propagation. As well as delayed software failure, the phenomenon of delayed failure is also known from failure delay systems and the delayed failure of brittle materials in materials science.

1.5.1 Failure Delay Systems

The ubiquitous presence of software in the modern world, and the potentially insidious nature of software faults, was recently impressed on the author, on being unable to start his car and discovering the battery to be flat. It turned out that the Multiplex Integrated Control Unit (MICU) of the vehicle was failing to enter a 'sleep mode' after the ignition was switched off, resulting in a parasitic battery drain that would flatten the battery if the car was left standing for a sufficiently long period. According to the manufacturer, the fault was caused by a software error and the MICU was replaced. The operation of the MICU is described in United States Patent Application 20100082198, 'Vehicle Load Control Device'.

An interesting feature of this failure is that, although the MICU entered an incorrect state shortly after the ignition was switched off, the car only failed to start if left standing for several hours. A general class of systems where failure does not occur immediately, but only after a delay are known as failure delay systems. These have been studied by Limnios (1988), who states that a characteristic of these systems is that failure occurs if and only if the conditions for failure are maintained for a sufficient time. Limnios (1988) gives the example of a water reservoir fed by a pump; if the pump stops working, the flow of water from the reservoir does not fail, provided the pump is

restarted before the reservoir is empty. However, if the pump stops working for a sufficiently long time, then the flow of water from the reservoir will fail.

The failure of the vehicle MICU is an example of a failure delay system involving a software component, although not a delayed failure of software, as such.

1.5.2 Delayed Software Failure

Although the existence of delayed failure in software was reported as early as 1971 (Pyle, McLatchie & Grandage) a review of the software engineering literature has found few systematic studies of delayed software failure. There are a small number of studies of error propagation, such as that of Yim, Kalbarczyk & Iyer (2009), however these are not linked to software testing techniques.

There are several possible reasons why delayed failure of software during testing might not have been recognised as important; first, because of the use of 'setup' and 'cleanup' in test cases, conventional software testing is unlikely to detect delayed failures. Second, any delayed failures found in testing could be difficult to recreate for debugging and so might simply be ignored as 'irreproducible'. Third, if the delay is sufficiently long then another type of software failure may occur before the delayed failure, thus 'masking' the delayed failure.

1.5.3 Delayed Failure of Brittle Materials

Delayed failure ('static fatigue') of brittle materials such as ceramics, glass, and steel is well known and has been extensively reported in the literature of materials science, for example: Fischer-Cripps (2007) "*Depending on environmental conditions, brittle solids may exhibit time-delayed failure, where fracture may occur some time after the initial application of load"*. This behaviour has been studied for a considerable time, as noted by Fuller, Luecke & Freiman (2001) "A*s early as the 1920s, the strength of glass and other brittle materials was understood to be limited by the presence of small cracks. Under stress, the small cracks would grow into larger cracks until reaching a critical size, at which point the material would fracture catastrophically.*"

Software systems have, indeed sometimes been called 'brittle' by analogy with the failure behaviour of brittle materials; Forsberg, Mooz & Cotterman (2005) "*The inability of software to deal with unexpected inputs is sometimes referred to as brittleness*" while Bellovin (2006) discusses the 'brittleness' of software in the context of software security.

Bush, Hershey & Vosburgh (2000) have attempted to construct a theory of brittle systems based on this analogy: "*This behaviour is characterized by a sudden and steep decline in performance as the system state changes. This can be due to input parameters which exceed a specified input, or environmental conditions which exceed specified operating boundaries*". A theory of delayed failure of software by analogy with delayed failure of materials, with error propagation in software playing the role of crack propagation in materials, might be interesting, however this is beyond the scope of the present research.

1.6 Research Problem

This section states the research problem and explains why investigation of this problem is worthwhile; and discusses an alternative approach to that taken in the present research, the technique of 'extended random regression' (ERR).

1.6.1 Problem Statement

The research problem: *Faults that cause delayed failures are likely to avoid detection by the conventional approach to software testing and remain in released software, resulting in lower reliability and robustness (than if they had been removed). Little is known from the software engineering literature of the failure delay characteristics and failure delay mechanisms of software components, of effective testing techniques for revealing delayed failure, or of the effect of delayed failure on software component reliability and robustness.*

1.6.2 Why Investigation of this Problem Is Worthwhile

The argument was put forward in the 'Motivation' section of this chapter that the conventional approach to software testing is unlikely to detect faults that cause delayed failures, because this approach prevents software errors from propagating beyond a single test case. If this is correct then these faults are likely to remain in released software after testing, provided no additional testing techniques (such as stochastic testing) are employed. There is some evidence that testing techniques that could reveal this type of fault have only recently begun to be more widely employed, for example, tools supporting random testing of DBMS such as MySQL and Oracle (Stoev, 2009; Naraine, 2009). Indeed, the present research provides examples of faults resulting in delayed failure that have been found in released versions of commercial software components using stochastic testing.

To the extent that delayed failure affects software reliability and robustness, these faults can be expected to result in lower reliability and robustness of released software, than if they had been removed. Limnios (1988) describes reliability modelling of failure delay systems, however, there appear to be few systematic studies of delayed failure in the software engineering literature, and the effect of delayed failure on software component reliability and robustness does not appear to be well understood.

It is known that long test traces present difficulty in debugging (Zeller & Hildebrandt, 2002) and so faults resulting in delayed failure after software release can be expected to be particularly costly to identify and fix. Better understanding of testing techniques that effectively and efficiently reveal delayed failure should contribute to improved reliability and robustness, and reduced cost of software components.

1.6.3 Extended Random Regression

Kaner, Bond & McGee (2003) present a case study of 'extended random regression' (ERR) a HVAT approach that addresses the oracle problem by choosing a large collection of existing tests that the SUT has already passed, and re-executing these in a

random order. Although the tests are designed to be independent of one another, test execution may have unintentional side effects that change the software state; the order of execution of tests affects the outcome of the test run and some tests in the run may fail. The success of extended random regression as a test technique suggests that conventionally written software tests do in fact have an ability to detect some faults that cause delayed failures, provided tests are run multiple times in long random sequences, however this is not the conventional way of executing tests, which are usually executed in a fixed order.

Berndt & Watkins (2005) give examples of complex, distributed system failures in which long sequence testing might have been effective; failures of the Patriot Missile System, London Ambulance Service, Railway Switching in Germany, Kaner's (1997) example of a phone system, and the NASA Mars Rovers.

1.7 Experimental Studies of DBMS

The motivation behind the present research was to study delayed failure of software components in general; however, a concrete example of a software component must be chosen as the focus for experimental study, and DBMS were identified as a suitable choice of subject. The present research therefore includes a contribution to the experimental study of DBMS and DBMS testing techniques. There is a growing literature in this area since the work of Slutz (1998) and Costa & Madeira (1999). The choice of DBMS as the subject for experimental study clearly influences the research methodology experimental procedures, and the domain of validity of the conclusions that can be drawn.

Experimental studies of DBMS fall within the relatively new field of experimental software engineering, exemplified by Basili (1996).

This section discusses the choice of DBMS as a subject of study, then introduces DBMS and SQL and the chosen DBMS components MySQL and Oracle XE, and finally describes the experimental approach adopted in the present research.

1.7.1 Choice of DBMS as a Subject of Study

In considering an experimental investigation of delayed failure of software components, the choice of DBMS as a subject of study was guided by the following criteria:

1. There is a perceived gap between academic research into software testing and industrial software testing practice (Reid, et al., 1999) therefore the chosen component should be a real software product, ideally in widespread commercial and industrial use. DBMS meet this criterion and MySQL and Oracle XE were chosen as being representative examples of commercial, off-the-shelf (COTS) DBMS products.

2. The chosen component should be a 'stable' production software release, having presumably passed the component supplier's quality control and release test process. Although precise details of the suppliers' test process are not known, they may be assumed to represent a baseline 'industry standard' practice. If the research method reveals faults that were not removed during release testing then it might be the case that the 'industry standard' testing process is not as effective at revealing these faults. MySQL and Oracle DBMS have documented release histories extending over several years, and in the case of MySQL there is a public 'bugs database' containing a detailed history of known faults and bug fixes and details of test suites.

3. The chosen component should be freely and easily available to researchers to allow replication of the research. The MySQL and Oracle XE DBMS may be downloaded from the suppliers' websites free of charge under appropriate licensing terms.

4. The specification of the chosen component should be freely and easily available. This is necessary in order to be able to specify test cases for the component. Both MySQL and Oracle XE have extensive documentation, both downloaded from the suppliers' websites and in published textbooks, and there is an international standard for SQL (ISO/IEC 9075, 2008). In the case of MySQL, program code is freely available as Open Source.

5. The chosen component should have an easily accessible and well-documented interface allowing automated testing. The Perl database interface (DBI) is freely available and supports both the MySQL and Oracle DBMS. Furthermore, the author already had some previous experience of using the Perl database interface for MySQL.

6. For the present research, the chosen component must conform to the 'stateless request-response model' of the conceptual framework described in Chapter 3. DBMS architecture typically follows a request-response model (Date, 2004) and any SQL request can be determined to be syntactically valid or invalid, regardless of the database state, by reference to the DBMS specification, such as the MySQL 5.0 Reference Manual (2010) or the Oracle® Database SQL Reference (2005).

7. The chosen component should be the subject of related work in the software engineering literature. Experimental studies of DBMS and DBMS testing are active areas of current research as discussed in Chapter 2; for example Slutz (1998), Haritsa (2006), Tuya, Suárez-Cabal & de la Riva (2006), Finnigan (2009), Garcia (2009), Naraine (2009), Stoev (2009), Khalek & Khurshid (2010).

8. The state of a DBMS is a combination of the software state and the database state. Although the software state of a component cannot usually be observed, in the case of a DBMS the content of database tables may be examined using the SQL statements SHOW TABLES, SHOW COLUMNS *table* and SELECT * FROM *table* thus providing some partial information about the DBMS state.

1.7.2 Introduction to DBMS and SQL

An introduction to DBMS and DBMS architecture can be found in Date (2004); a block diagram of a typical DBMS architecture is shown in Figure 5. Typically this architecture follows the request-response model described in Chapter 1 as shown in Figure 4. The client sends a request to the DBMS, which causes a database operation to be requested. The database returns a response to the DBMS with the result of the operation and the DBMS finally returns a result to the client. The DBMS typically

manages transactions composed of sequences of database operations and provides the capability to 'roll back' transactions in the event of detecting an error (Greenwald, Stackowiak & Stern, 2004; Vaswani 2004).

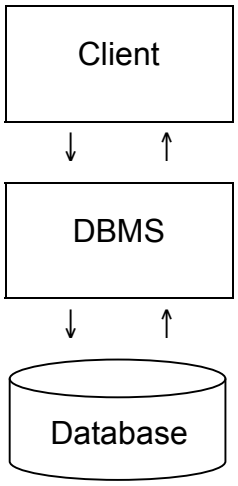


Figure 5: Block diagram of DBMS architecture

The database holds logically organised data; in the case of a relational database, the data is organised as a collection of tables. SQL (structured query language) provides a syntax and semantics for communication between the client and the DBMS and has facilities for creating, reading, updating, and deleting database tables. An introduction to SQL can be found in Date (2004) and in Greenwald, Stackowiak & Stern (2004). The SQL international standard is ISO/IEC 9075 (2008), however both MySQL and Oracle offer a core sub-set of the earlier 2003 version of this standard, along with proprietary extensions.

1.7.3 MySQL

MySQL is a freely available Open Source relational DBMS that provides storage and management of data held in tables using SQL. Oracle Corporation acquired Sun Microsystems in 2010 and so acquired ownership of MySQL.

MySQL 5.0 release history from the MySQL 5.0 Reference Manual (2010) is shown in Table 1. MySQL 5.0.1 was released in 2004 (still with some open critical bugs) however the version of MySQL under test (5.0.22) was released in 2006 as a stable production version.

Version	*Release Date*	*Notes*
MySQL 5.0.0	22 December 2003	Alpha release
MySQL 5.0.1	27 July 2004	Still some open critical bugs
MySQL 5.0.2	01 December 2004	-
MySQL 5.0.22	24 May 2006	-

Table 1: MySQL 5.0 release history

A technical guide to MySQL can be found in Vaswani (2004); MySQL provides a well-documented programming interface through the Perl database interface (DBI) as described in the MySQL Administrator's Guide (2004), the MySQL 5.0 Reference Manual (2010) and by Robert (2010).

The MySQL database server returns either error responses or success responses to the client. The error responses *syntax error* and *empty query* do not depend on the DBMS state. Other error responses and success responses are dependent on the software state or the database state; an SQL statement may be *accepted* (stateless behaviour) even though it does not *succeed* (state-based behaviour). The MySQL client-server protocol is described in the MySQL Internals Manual (2005). Schumacher (2005) describes the handling of invalid inputs in MySQL 5.0.

1.7.4 Oracle XE

Oracle Database 10g Express Edition (Oracle XE) is a free 'starter' edition of Oracle 10g based on Oracle Database 10g Release 2, but limited to running on a single processor with up to 1 GB of RAM and 4 GB of user data as described in the Oracle® Database Express Edition Installation Guide (2007). The beta version was released in

2005. All of the Oracle Database 10g product family are based on the same database engine architecture as described in Greenwald, Stackowiak & Stern (2004).

A programming interface for Oracle Database is available through the Perl database interface (DBI) as described by Robert (2010).

Oracle's programming language extension to SQL is called PL/SQL and is described in the Oracle® Database SQL Reference (2005) and by Murray (2008).

1.7.5 Experimental Approach

The experimental approach taken in the present research investigates the reliability, robustness and delayed failure characteristics of software components by comparing measures made on two DBMS using two stochastic testing techniques. Both testing techniques execute randomly generated SQL against a random database.

Positive tests for reliability comparison are performed using randomly generated syntactically valid SQL in an approach similar to that of Slutz (1998). Negative tests for robustness comparisons are performed using randomly generated invalid SQL produced by SQL mutation. This approach applies for DBMS the concept of testing with 'hostile data streams' as described by Jorgensen (2002). The experimental approach for determining F-measure and failure delay is based on an analysis of the state transition model of error propagation and failure, described in Chapter 3.

The experimental approach takes account of frameworks and guidelines for empirical studies of test techniques, based on a review of the literature of experimental software engineering presented in Chapter 2. Some of the methodological challenges encountered with the experimental approach were differences in SQL syntax between MySQL and Oracle XE, the need for automatically separating valid and invalid SQL statements following SQL mutation, and reproducibility of failures, as described in Chapter 5.

1.8 Key Elements and Focus of the Research

This section describes the key elements, and focus, of the research.

1.8.1 Key Elements of the Research

The key elements of the present research are:

a) Software component testing and delayed failure. The research problem arises from considering software component testing in the context of delayed failure. The investigation of the research problem should demonstrate an effective test method for revealing delayed failure of software components.

b) Reliability and robustness of software components. Reliability and robustness are important quality factors for software components and so these are interesting in the context of delayed failure.

c) Stochastic testing. Stochastic testing is likely to be an effective method for revealing delayed failure as well as a suitable technique for reliability and robustness testing of software components.

d) Experimental studies of DBMS. The MySQL and Oracle XE DBMS were selected as suitable subjects of study for an experimental approach to the research problem.

e) The conceptual framework. The conceptual framework described in Chapter 3 provides a context for the experimental approach and a model of delayed software failure that can be validated by experiment.

1.8.2 Research Focus

A Venn diagram as shown in Figure 6 may be helpful in illustrating the focus of the present research. This diagram shows the focus of the present research marked as an X at the overlap of fours areas: delayed failure, DBMS research, stochastic testing, and experimental software engineering. Venn diagram image is from *Wikimedia Commons*.

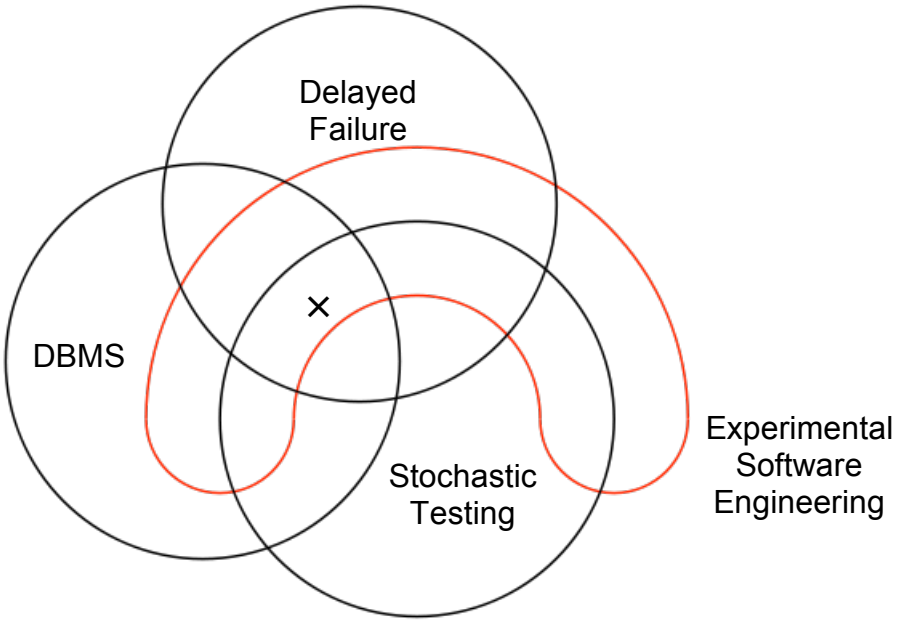


Figure 6: Research focus

1.9 Contributions to Knowledge

This section briefly summarise the contributions to knowledge of the present research. A full summary of contributions appears in Chapter 9.

a) That delayed failure is an important failure mode of software components.
b) That the conventional approach to software testing is unlikely to detect delayed failure.
c) A methodology for benchmarking software component dependability.
d) That stochastic testing is an effective technique for revealing delayed failure.
e) Corroboration of related work in DBMS testing using random generation of SQL.
f) A state transition model of delayed failure in software components.
g) That the research problem can be solved, at least for the components considered in the study.

1.10 Thesis Structure

This section gives a brief summary of each chapter of the thesis.

Chapter 1. INTRODUCTION
This chapter presents an introduction to software component testing and terminology, the motivation for the present research, an introduction to delayed failure, the research problem, experimental studies of DBMS, key elements and focus of the research, the contributions to knowledge, and lastly the structure of the thesis.

Chapter 2. LITERATURE REVIEW
This chapter presents a review of the software engineering literature in the areas of software component testing techniques, software component reliability, software component robustness, stochastic testing, alternative approaches to stochastic testing, experimental studies of DBMS, and delayed failure.

Chapter 3. CONCEPTUAL FRAMEWORK
This chapter presents the simple stateless request-response model and a refinement of this model developed during the present research – the partitioned request-response model; along with the state transition model of error propagation and failure, the Markov chain model, and the process of delayed failure.

Chapter 4. RESEARCH QUESTION AND HYPOTHESES

This chapter presents a statement of the research question, the research hypotheses H_1 and H_2 together with the associated null hypotheses, calculations of expected values of F-measure and failure delay using the Markov chain model, and why answering this question is worthwhile.

Chapter 5. METHODOLOGY AND EXPERIMENTAL PROCEDURE

This chapter first presents the research goals, a generic procedure for testing the hypotheses, the experimental design, experimental configurations, methods for random generation of SQL and SQL mutation, DBMS initialisation, database creation, and the method for identifying minimal test sequences. This chapter then presents the procedures to be followed in the 1st experiment - MySQL (invalid and valid SQL) and the 2nd experiment - Oracle XE (valid SQL), and discusses methodological challenges, experimental questions, and data analysis.

Chapter 6. RESULTS

This chapter presents results for the 1st experiment using MySQL with invalid and valid SQL, MySQL bugs found during the experiment, and results for the 2nd experiment using Oracle XE with valid SQL.

Chapter 7. ANALYSIS

This chapter presents an analysis of the experimental results. This analysis looks for differences between experimental profiles P_1 and P_2 with the same SUT and for differences between SUT S_1 and S_2 with the same experimental profile, as well as considering the accuracy of the response profile regression analysis and the statistical distribution of the results.

Chapter 8. DISCUSSION

This chapter presents a discussion of the present research within the context of stochastic testing, SQL mutation, experimental studies of DBMS, fault detection effectiveness and efficiency, the conceptual framework, delayed failure, the research problem, and threats to validity.

Chapter 9. CONCLUSION

This chapter presents a summary of the conclusions of the present research, a summary of the contributions to knowledge, and areas for future research.

Chapter 10. REFERENCES

This chapter presents references to the software engineering literature as given in this thesis.

APPENDIX A. DBMS RESPONSE PROFILE RESULTS

This appendix contains the full experimental results for MySQL and Oracle XE DBMS response profiles.

APPENDIX B. TEST MACHINE CONFIGURATIONS

This appendix records the hardware and software configurations for the MySQL and Oracle XE test machines.

APPENDIX C. EXPERIMENTAL RESULTS FOR 1st EXPERIMENT

This appendix describes the data sets and experimental results for the 1st experiment and gives details of the program and data files used to automate the experimental procedure.

APPENDIX D. CONFERENCE PAPER (TAIC PART 2006)

Abstract of conference paper presented at the Testing Academic & Industrial Conference – Practice And Research Techniques, (TAIC PART) 2006.

APPENDIX E. CONFERENCE PAPER (UKSMA 2009)

Abstract of conference paper presented at the 20th Annual Conference of the United Kingdom Software Metrics Association, (UKSMA) 2009.

APPENDIX F. NORMAL PROBABILITY PLOTS

This appendix presents normal probability plots of F-measure results. These results appear to be approximately normally distributed.

1.11 Summary

This chapter presented an introduction to software component testing and terminology, the motivation for the present research, an introduction to delayed failure, the research problem, experimental studies of DBMS, key elements and focus of the research, the contributions to knowledge, and lastly the structure of the thesis.

2 LITERATURE REVIEW

2.1 Introduction

This chapter reviews relevant areas of the software engineering literature and shows how the major topics relate to each other, and how they relate to the rest of the present research. The review covers every technique used in the research, including statistical tools and measures of reliability, and includes a description of the investigated methods, a description of alternative methods, and related work.

Remarkably, there appear to be few systematic studies of delayed failure in the software engineering literature, and in particular, there appear to be no studies of stochastic testing as related to delayed failure, or studies of delayed failure of DBMS. This gap in knowledge provides further justification for the present research.

This chapter presents a review of the software engineering literature in the areas of software component testing techniques, software component reliability, software component robustness, stochastic testing, alternative approaches to stochastic testing, experimental studies of DBMS, and delayed failure.

2.2 Software Component Testing Techniques

Software component testing was introduced in Chapter 1, where testing techniques were categorised on the basis of the fundamental approaches of static, dynamic, black-box, and white-box testing techniques (Lewis, 2009) and the focus of the present research was identified as falling within dynamic black-box testing techniques.

Software component testing techniques must address the problem of the size of the test space and the 'oracle problem' (Kaner, 1997; Kaner & Bach, 2005). One approach to the test space problem is to employ high volume automated testing techniques, for example exhaustive high-volume testing, random function equivalence testing, and extended random regression (Kaner, Bond & McGee, 2003).

Software components are prone to a large number of different types of faults and several 'taxonomies' of software faults have been collected, such as Beizer (1990). Several empirical studies of software component testing techniques have been conducted which compare their fault detection effectiveness and efficiency, such as Reid (1997).

Software component testing techniques are complementary to the process of debugging and removal of faults, and this process begins with identifying failing input requests (Zeller & Hildebrandt, 2002) and this process is informed by models of error propagation and failure such as that of Yim, Kalbarczyk & Iyer (2009).

This section reviews the test space problem, the oracle problem, high volume automated testing, fault taxonomies, empirical studies of test techniques, fault detection effectiveness and efficiency, identifying failing input requests, and models of error propagation and failure.

2.2.1 The Test Space Problem

The size of the test space is a fundamental problem in software testing; Kaner (1997) says that "*complete testing is impossible*", since this would require testing every possible input value with every possible software state. Kaner, Falk & Nguyen (1993) and Myers, et al. (2004) advance similar arguments. Binder (2000) (in the introduction to Chapter 5: Test Models) says: "*Testing can be viewed as a search problem. We are looking for those few input and state combinations that will reach, trigger, and propagate bugs out of trillions and trillions, which will not. Brute force is impotent at this scale. Testing by poking around is a waste of time that leads to unwarranted confidence.*"

The large size of the testing space for software-intensive digital electronic systems is illustrated by the following example given by Kaner, Falk & Nguyen (1993), Kaner (1997, 2000) and Kaner, Bond & McGee (2003). This was a software bug in a phone system, where a specific sequence of events caused a stack overflow resulting in a

system crash. The bug was reported as being very specific and unlikely to be found by a conventional approach to software test design; this bug and several other 'long-sequence' bugs in the phone system were eventually found by using stochastic testing together with diagnostic instrumentation of the program code. The state diagram of the phone system as derived from Kaner (1997) is shown in Figure 7.

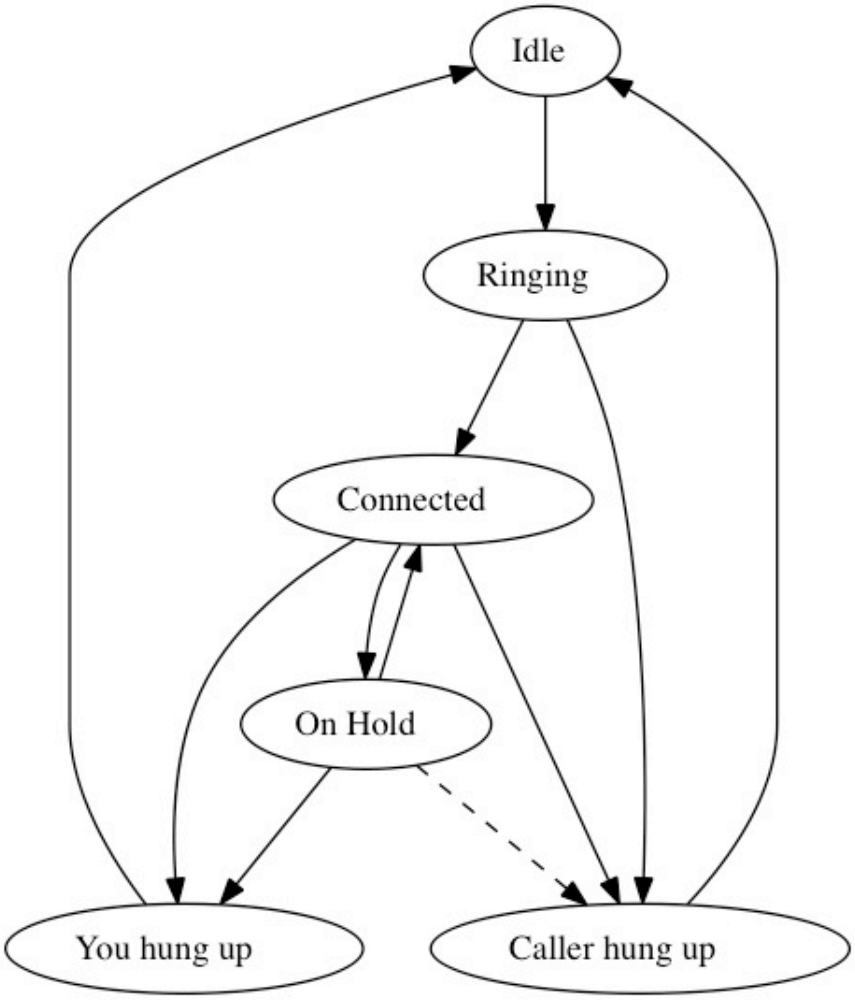


Figure 7: State diagram of phone system

In the phone system, up to ten callers would be placed on a hold queue; however, there was a bug in this system such that if a caller hung up while on hold, the data for that call was not properly removed from the system stack. After twenty such errors had accumulated on the stack, placing another call onto the hold queue would overflow the

stack, causing a system crash. The number of possible call scenarios that would have to be tested to catch this bug is very large, and Kaner considers it unlikely that manual test generation from the model would include the specific crashing bug case. Kaner relates in passing that the system developers originally missed off the transition from 'On Hold' to 'Caller hung up' (dotted line) from the model, showing that even simple software models may contain errors.

2.2.2 The Oracle Problem

Kaner & Bach (2005) summarise the oracle problem with the question "*How will you know whether the program passed or failed the test?*" and note that "*our ability to automate testing is fundamentally constrained by our ability to create and use oracles*".

The output of a software system or component typically depends on both the input and the software state; the input, process, output, storage (IPOS) model shown in Figure 8 provides a general model of a system with this property (Stair & Reynolds, 2008).

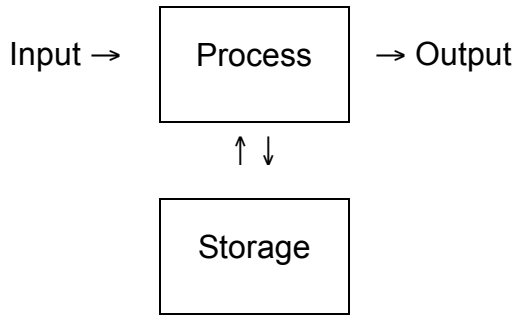


Figure 8: IPOS model

Predicting the outcome of a long sequence of test cases therefore requires a test oracle having a state-based model of the software component. State-based models of complex software-intensive systems can be very large and the construction and maintenance of these models presents a practical challenge, for example Nyman (2000) makes the following observation on the use of state-based models: "*The greatest single cost is generating the model or state table. It's not unusual to need a 50,000-node state table*

for a moderately complex product. Continuing to add new features results in state explosion in which the number of nodes increases geometrically. So creating the model is seldom a one-time cost; for large models or tables, maintenance becomes a major cost factor". However, Nyman adds: "*they may be cost effective for critical parts of a project where the state table can be kept small*".

The effort of building and maintaining a state-based model can be comparable to the effort that would be involved in conventional manual test design; Utting (2007) and Utting & Legeard (2007) report case studies in which test design effort with MBT was 17% to 35% less than manual test design.

2.2.3 High Volume Automated Testing

High volume automated testing (HVAT) was introduced in Chapter 1. HVAT consists of the automatic generation and execution of very large numbers of test cases, and usually employs a very simple test oracle, such as 'did the software crash?' (Nyman, 2000; Kaner, 2003-b).

Kaner, Bond & McGee (2003) have discussed HVAT as a response to the test space problem. Kaner (2003-b) notes that "*although the individual tests are often weak, they make up for low power with massive numbers*"; however, HVAT produces long test traces that require analysis to derive 'minimal' test cases (see 'identifying failing input requests'). McGee & Kaner (2004) list the types of fault they expect HVAT to be able to find: "*buffer overruns and security holes, special cases, timing related errors, corrupt memory or stack, memory or resource leaks, resource exhaustion*".

McGee & Kaner (2004) suggest that HVAT techniques can "*dramatically increase the reliability of software that must run for long periods of time without stopping or rebooting*" and that "[*HVAT] techniques are much more effective on code that is already fairly stable and passes basic functional tests. For such code, we believe it can lead to substantial increases in reliability*." They note "*there is very little empirical research on [HVAT] techniques*".

Kaner (2000) describes the following HVAT techniques, as discussed by several authors (these are discussed in more detail later on in this chapter, in the sections on 'stochastic testing' and 'alternative approaches to stochastic testing'):

a) Exhaustive high-volume testing (Hoffman & Kaner, 2003)
b) Random function equivalence testing (Hoffman & Kaner, 2003)
c) Stochastic testing using 'dumb monkeys' (Nyman, 2000)
d) Stochastic testing with diagnostic instrumentation of program code (this is the phone system example given by Kaner (1997) as discussed in 'the test space problem' and shown in Figure 7)
e) Stochastic testing using a state model (Whittaker, 1997)

Kaner, Bond & McGee (2003) add extended random regression and 'hostile data stream' testing (Jorgensen, 2002) to this list.

2.2.4 Fault Taxonomies

Several authors have developed extensive classification schemes ('taxonomies') of software faults and failures. These are useful to software testing practitioners and researchers in identifying common types of software faults and failures, and the testing techniques that may be able to reveal them.

Beizer (1990) as amended by Vinter (2001) provides an appendix 'Bug Taxonomy and Statistics' that classifies many types of fault and includes statistics of the occurrence of these faults in real software projects. Kaner, Falk & Nguyen (1993) provides an appendix of 'common software errors' that describes over 400 software faults.

A comprehensive classification scheme can be found in the IEEE Computer Society Standard Classification for Software Anomalies (IEEE, 1993). The classification scheme for recognition of software 'anomalies' given in this standard assigns reference codes (prefixed RR for 'recognition') to several 'symptoms' of failure, and a selection of these are shown in Table 2.

Symptom of Failure	*Code*
Operating system crash	RR510
Program hang-up	RR520
Program crash	RR530
Correct input not accepted	RR541
Wrong input accepted	RR542
Wrong output format	RR551
Incorrect output result/data	RR552
Incomplete/missing output	RR553
Failed required performance	RR560
Perceived total product failure	RR570
System error message	RR580

Table 2: Classification of failure symptoms

Siewiorek, et al. (1993) give the following failure classifications for robustness testing with invalid input values: 'no error detected' (considered to be a failure), 'time-out' (late response), 'incorrect output', 'crash', 'crash with error message' ('abort'); correct behaviour is considered to be a response with an error message.

2.2.5 Empirical Studies of Test Techniques

Experimental or empirical software engineering is a relatively new field of research exemplified by Basili (1996): "*Software engineering needs to follow the model of other physical sciences and develop an experimental paradigm for the field*".

Several authors have pointed out the need for a consistent framework for reporting empirical studies in software engineering:

Jedlitschka & Pfahl (2005) say: "*One major problem for integrating study results into a common body of knowledge is the heterogeneity of reporting styles: (1) It is difficult to locate relevant information and (2) important information is often missing. Reporting*

guidelines are expected to support a systematic, standardized presentation of empirical research, thus improving reporting in order to support readers in (1) finding the information they are looking for, (2) understanding how an experiment is conducted, and (3) assessing the validity of its results". They give a set of guidelines for reporting experimental results, and suggest the research community should evaluate these guidelines.

Kitchenham, et al. (2007) critique the guidelines proposed by Jedlitschka & Pfahl (2005) and conclude: "*The current guidelines need to be revised and then subjected to further theoretical and empirical validation*".

Channon & Koch (2007) propose that full details of software engineering experiments should be made available, including raw data, methodology, source code, execution profile, tools, and documentation.

Harman, et al. (1999) and Reid, et al. (1999) point out the need for a consistent framework for reporting empirical studies in software testing; Reid, et al. (1999) propose a framework in which the following experimental factors are to be recorded: subjects, objects, treatments, state variables, and response variables. Briand (2007) notes that: "*Empirical studies are crucial to software testing research in order to compare and improve software testing techniques and practices*".

Several empirical studies of test techniques have been conducted: for example, Reid (1997) reported a comparison of Equivalence Partitioning, Boundary Value Analysis, and random testing which concluded that "*an implementation of BVA was found to be most effective, with neither EP nor random testing half as effective. The random testing results were surprising, requiring just 8 test cases per module to equal the effectiveness of EP, although somewhere in the region of 50,000 random test cases were required to equal the effectiveness of BVA*".

2.2.6 Fault Detection Effectiveness and Efficiency

Reid, et al. (1999) define test effectiveness as "*the ability of a technique to detect faults in software*" and give the mathematical formulation shown in equation (1).

$$\textit{Test effectiveness} = \textit{Number of faults detected} / \textit{Total number of faults} \quad (1)$$

Test effectiveness as defined in equation (1) is difficult to measure in practice since the total number of faults cannot be known with certainty; the existence of faults must be deduced from observed failures and there need not be a one-to-one relationship between fault and failure. Furthermore, faults may lie dormant for some time after software is released into service (Reid, et al., 1999).

Reid, et al. (1999) define test efficiency as "*the number of faults detected divided by the time taken to perform the testing*" and give the mathematical formulation shown in equation (2).

$$\textit{Test efficiency} = \textit{Number of faults detected} / \textit{Total time} \quad (2)$$

The total time (in person-hours) may be taken to include the effort to create the test cases, and any models or oracles, as well as for test execution and analysis of results.

Despite earlier work such as that of Reid (1997) and Reid, et al. (1999), BS 7925-2 (2001) (Annex C - Test technique effectiveness) notes that, "*Research into the relative effectiveness of test case design and measurement techniques has, so far, produced no definitive results*".

Measures of fault detection effectiveness include the E-, F- and P-measures defined in Table 3 as discussed by Chen, Kuo & Merkel (2004), Chen & Merkel (2007), Liu & Zhu (2008) and Parizi, et al. (2009). Liu & Zhu (2008) also define the S-measure as the sample standard deviation of any one of the previous three measures and state that "*the smaller the S-measure, the more reliable the testing method*".

Measure	*Definition from Liu & Zhu (2008)*
E	The expected (average) number of failures detected by the test set
F	The number of test cases required to detect the first failure
P	The probability of detecting at least one failure in a given test set
S	The variation of fault detecting ability

Table 3: Measures of effectiveness

2.2.7 Identifying Failing Input Requests

Identifying failing and 'precondition' requests in long test traces is important to assist debugging and removal of faults (Zeller & Hildebrandt, 2002). This is a particular problem for HVAT, where a test trace might contain many thousands of requests and responses. One approach to this problem is simplification of the test trace by the elimination of 'redundant' test steps to produce a 'minimal' test case; that is, a short sequence of input requests that reproduces the original failure.

Zeller & Hildebrandt (2002) and Cleve & Zeller (2005) describe an automated technique for simplifying test traces known as 'delta debugging' by repeatedly executing altered versions of a failing test sequence until a passing sequence is found; the difference between the two sequences is (presumably) the cause of the failure.

Slutz (1998) describes automatic simplification of randomly generated SQL statements to aid debugging during DBMS testing. The simplified SQL statement produces the same failure behaviour as the original statement, although semantically the two statements may not be equivalent.

2.2.8 Models of Error Propagation and Failure

Two basic types of software failure model are described in the software engineering literature: a linear cause-effect model and a probabilistic state transition model.

The first type of software failure model is a linear cause-effect sequence of error propagation: fault → error → failure where a dormant fault is first activated and causes an error, and then the error propagates as a chain of errors until a failure eventually occurs. However, in the model a dormant fault will not necessarily be activated and an error will not necessarily produce a software failure. This model of error propagation and failure is discussed by: Koopman (1999); Avizienis, et al. (2004); Avizienis, Laprie & Randell (2004); Cleve & Zeller (2005); and Yim, Kalbarczyk & Iyer (2009).

An earlier, yet more sophisticated model was presented by Prasad, McDermid & Wand (1996); this is a state transition model of fault activation, error propagation and failure referred to as the DAP (dormant, active, propagation) model. Although this transition model is not probabilistic, it allows more complex failure behaviour than the strictly linear cause-effect sequence model.

Ammann & Offutt (2008) propose an alternative terminology, 'RIP' ('reachability, infection, propagation') for error propagation and failure. In this model, fault activation is called 'reachability' as program execution reaches a fault in program code, and an incorrect software state (error) is called an 'infection' which propagates by program execution to cause a failure. The three conditions of reachability, infection, and propagation must all be present for software failure to occur.

None of the previous authors explicitly mention latency or delay in the model; however Yim, Kalbarczyk & Iyer (2009) introduce the concepts of fault activation latency and error propagation latency, as shown in Figure 3 (refer to Chapter 1, Terminology).

The second type of software failure model is a probabilistic state transition model (Markov chain model) in which probabilities are assigned to transitions between

software states, including failure states. Transitions between software states take place as the result of the software processing input requests. This model allows a statistical approach and provides a model for software reliability (Whittaker & Thomason, 1994; Thomason & Whittaker, 1999). However, these authors do not explicitly mention error propagation in their failure models, nor do they explicitly mention latency or delay in the models.

A software design approach called 'software rejuvenation' seeks to reduce the frequency of software failure by periodically restarting running software, thus preventing errors from propagating to the point of failure. This approach is described by: Huang, et al. (1995), Bernstein (2002), Bernstein & Kintala (2004), and Trivedi & Vaidyanathan (2008). The approach is described in terms of a probabilistic state transition model containing an initial 'robust' state, a 'failure probable' state, a 'rejuvenation' state, and a 'failure' state; probability distributions are assigned to transitions between the states. Software failure occurs as the result of transitions from the 'robust' state via the 'failure probable' state to the 'failure state'. 'Rejuvenation' occurs when transitions take place from the 'failure probable' state via the 'rejuvenation' state back to the 'robust' state.

Shukla (1994) and Brukman, Dolev & Kolodner (2003) discuss 'self-stabilisation' of software; this process may delay or prevent the ultimate failure of a software component by returning the component to the 'robust' state.

Both rejuvenation and self-stabilisation must be 'fail-stop' to be effective, that is, the error detection latency must be small, otherwise an error state can propagate to the component output before action is taken, perhaps even being written to persistent storage (Chandra & Chen, 1998).

Michael & Jones (1996) present an empirical study into the propagation of errors, which suggests that, "*either all data state errors injected at a given location tend to propagate to the output, or else none of them do*". They call this property "*homogeneous*

propagation" and conclude "*that one can draw conclusions about the behaviour of many data-state errors after examining only a few*".

2.3 Software Component Reliability

Software component reliability was introduced in Chapter 1. This section reviews composition of component reliability, MTTF and F-measure, statistical testing and operational profiles.

2.3.1 Composition of Component Reliability

Hamlet, Mason & Woit (1999) propose a 'foundational theory' of software component reliability; the composition of components would ideally be supported by component 'data sheets' that would provide sufficient technical quality information about components to allow system designers to make reliability predictions for the combined system. They contend that random testing using a specified profile based on formal specification of component behaviour would provide a suitable statistical measure.

Mason (2002-a): "*One of the desirable properties of predictable assembly is reliability. Given reliability and transformation functions for components, it is possible to accurately compose reliabilities. Currently the transformations are limited in their domain of applicability, but we are working to extend their domain.*"

Stafford & McGregor (2002) note several issues in predicting the reliability of composed components, including the choice of operational profile, unit of time, and definition of failure, which may differ for the composed system compared to the conditions of reliability estimation for the component.

2.3.2 MTTF and F-Measure

Reliability measures including MTTF and F-measure were introduced in Chapter 1. Mean Time To Failure (MTTF) is a statistical measure defined as "*the expected time that a system will operate before the first failure occurs"* (Storey, 1996). An equivalent

measure of software reliability is failure rate, denoted by λ and defined as "*failures per unit time or per number of transactions"* (IEEE, 1991). The reliability function $R(t)$ can be defined as the probability that there will be no failure before time t and for a constant failure rate λ then $R(t) = e^{-\lambda t}$ when it can be shown that the MTTF = 1/λ (Storey, 1996).

Chen, et al. (2004, 2007) define the F-measure as "*the number of tests required in a sequence to detect the first program failure"* and show that (for random testing with replacement) if the software under test has a constant failure rate (probability of failure) p then the F-measure is distributed according to the geometric distribution. They show that probability density function is $P(X=n) = q^{(n-1)}p$ where $q=1-p$ and the MTTF is $E(X) = 1/p$ with variance $\text{Var}(X) = q/p^2$. If $p << 1$ then the variance is approximately $1/p^2$ and the standard deviation is approximately $1/p$.

2.3.3 Statistical Testing

Statistical software testing uses an operational profile to reflect the expected actual usage of the software. This addresses the perceived shortcomings of random testing and equivalence partitioning as black-box techniques, in that they provide a uniform coverage of software, unrelated to actual usage (Weber, 2002).

2.3.4 Operational Profiles

As discussed in Chapter 1, measures of software reliability depend on the operational profile chosen, that is, on the frequency of occurrence of each possible input value or choice of software function, as discussed by Musa (1993) and Koziolek (2005). Mason (2002-b) defines an operational profile as "*a statistical description of the environment in which a system is used*".

Binder (2004) considers the case for testing using an operational profile to be 'compelling'. However, the use of operational profiles has some problems: (a) operations that are expected to occur infrequently are not well tested, although these will eventually occur in service (b) if the operational profile changes in the future (or if

the operational profile used in testing was not accurate) then untested functionality is suddenly exposed (Whittaker & Voas, 2000).

Hamlet (1994) critiques reliability measures based on an operational profile especially in the case of reusable software components for which the operational profile is not known.

Whittaker & Thomason (1994) suggest that a single operational profile is insufficient for many types of software, because the probability distribution of input values during software execution is not constant.

2.4 Software Component Robustness

Software component robustness was introduced in Chapter 1. Software component robustness testing (also known as 'fuzz' testing or penetration testing) includes both random testing (Mukherjee & Siewiorek, 1997) and deterministic testing techniques (Varpiola & Takanen, 2008).

Robustness has been defined as "*the degree to which a component or system can function correctly in the presence of invalid inputs or stressful environmental conditions*" (IEEE, 1991) and robustness testing therefore employs negative testing (Utting & Legeard, 2007).

Siewiorek, et al. (1993) give an alternative definition of robustness as "*the ability to identify and handle errors in a consistent and predictable manner*". This definition would presumably include handling errors due to valid inputs, as well as those due to invalid inputs, and so robustness testing by this definition is not restricted only to negative testing. Nevertheless, the authors go on to describe benchmarks of system robustness only in terms of invalid inputs, as do Mukherjee & Siewiorek (1997).

Varpiola & Takanen (2008) suggest that much of software testing is positive (conformance) testing and that insufficient negative testing is performed; they discuss

fuzzing as negative testing, and maintain that systematic grammar based test generation is more effective than random testing for robustness testing, because random testing has a low probability of finding security-related faults.

This section reviews negative testing, fuzz testing, and measures of robustness.

2.4.1 Negative Testing

Robustness testing using negative tests is discussed by Utting & Legeard (2007) who distinguish between 'format testing' that employs invalid input values and 'context testing' that employs valid input values presented to the SUT in an invalid order (such as will happen during stochastic testing).

Siewiorek, et al. (1993) identify four possible combinations of program responses to an invalid input; detect the error (or not) and fail (or not), and use these as a basis for program failure classification during robustness testing (as described earlier in this chapter under 'Fault Taxonomies').

Negative test cases can be produced by mutation of positive test cases (data mutation) as described by Shan & Zhu (2006, 2007). BS 7925-2 (2001) gives guidance on the application of syntax testing with examples of both positive and negative test cases, including rules for generating test cases with invalid syntax using mutation, as discussed in Chapter 5.

2.4.2 Fuzz Testing

Fuzz testing or 'fuzzing' has its origins in a series of reports on the stability and reliability of UNIX utilities, which crashed or hung when fed random input strings, by Miller, Fredriksen & So (1990), Miller, et al. (1995) and Bowers, Lie & Smethells (2001). Fuzz testing was later applied to the problem of security vulnerability testing (testing for software robustness to malicious invalid inputs) especially for network protocol software. The classic type of software robustness failure is buffer overflow (Jorgensen, 2002).

Examples of automated tools employing fuzzing include PROTOS (Kaksonen, Laakso & Takenen, 2000); Ballista (Koopman, et al. 2002); SPIKE (Aitel, 2002) and Sulley (Sutton, Greene & Amini, 2007). 'Fuzzing' is defined by Sutton, Greene & Amini (2007) as "*the process of sending intentionally invalid data to a product in the hopes of triggering an error condition or fault*". They add, "*These error conditions can lead to exploitable vulnerabilities*".

More recently, fuzz testing has been applied to robustness testing of DBMS, for example Garcia (2009) reports on the use of SQL Fuzz testing at Microsoft Corporation.

2.4.3 Measures of Robustness

Siewiorek, et al. (1993) and Mukherjee & Siewiorek (1997) have developed robustness benchmarks for UNIX systems that provide an indication of robustness for some areas of software system functionality.

Varpiola & Takanen (2008) point out that negative testing increases code coverage beyond what is possible with positive tests, and this might provide a metric for robustness testing; another metric for robustness testing might be input space coverage, however finding good metrics for robustness testing remains a challenge. Garcia (2009) observes that there is a need for further research into robustness testing metrics, such as code coverage; such metrics can help determine "*when we can stop fuzzing*" (Sutton, Greene & Amini, 2007).

2.5 Stochastic Testing

Stochastic testing was introduced in Chapter 1. This section presents an overview of stochastic testing, and considers stochastic testing as an example of a Monte Carlo method, and the relationship between run length and test effectiveness; before reviewing random search of software states and limitations of undirected random testing.

2.5.1 Overview of Stochastic Testing

In classical random testing, as described for example in BS 7925-2 (2001), the values used in each test case are selected at random, but the test cases are independent of one another. In stochastic testing, as described by Kaner (2000) and by Kaner, Bond & McGee (2003), a randomly chosen sequence of test cases is executed, where the result of each test case depends also on the history (order) of previous test cases. Both classical random testing and stochastic testing should be less prone than are conventional testing techniques to the so-called 'pesticide paradox' described by Beizer (1990) in which the rate of detection of faults decreases with repeated execution of the same tests; this is because random testing effectively produces a 'new' test case on each test run (Robinson, 1999).

Whittaker (1997) introduces stochastic testing as a formal approach to software testing in which testers create stochastic models (Markov chains) of software behaviour (rather than individual test cases) and test cases are automatically generated from the stochastic models. Stochastic testing is sometimes known as 'monkey testing', however despite this pejorative term, Nyman (2000) reports that a 'dumb monkey' that can choose randomly from a large range of input values and finds very obvious bugs like crashes and hangs can nevertheless be highly effective. Kaner & Bach (2004) present an overview and analysis of stochastic testing, alongside other random testing techniques.

2.5.2 Monte Carlo Methods

Stochastic testing can be viewed as an example of a Monte Carlo method; the term *Monte Carlo* is often used to describe any method that employs repeated random sampling to estimate a value. Korver (1994) shows how Monte Carlo analysis can be applied to the estimation of software reliability; an advantage of this approach is that it can be applied without any knowledge of the structure or complexity of the SUT. However, as Korver points out, statistically a large number of random samples are required for reliable estimation, because the standard error decreases with the square root of the sample size and so the sample size must be increased by a factor k^2 to reduce

the standard error by a factor k. The standard error is defined as the square root of (s^2/n) where s is the sample standard deviation and n is the sample size.

2.5.3 Test Run Length and Test Effectiveness

The relationship between test run length and test effectiveness for stochastic testing is not well understood. Slutz (1998) observes that: "*if the distribution is adequate, stochastic testing has the advantage that the quality of the tests improves as the test size increases*". Andrews, et al. (2008) conclude based on case studies that "*the choice of test length dramatically impacts the effectiveness of random testing*". They report that a small number of long runs may be more effective than a large number of short runs, although long runs produce long test traces that are difficult to debug (see 'identifying failing input requests').

2.5.4 Random Search of Software States

Monte Carlo methods generally have a slow rate of convergence as discussed by Korver (1994). However, Menzies, Owen & Cukic (2002) and Menzies, Owen & Richardson (2007) report a saturation effect in the random search of state models: "*random search of finite state machines exhibits an early saturation effect [and] quickly yields all that can be found, even after a much longer search*". This effect is called '*clumping*' and these authors make the hypothesis that "*the effective state space of a program is relatively small when compared to all reachable states*". This finding suggests stochastic testing may be more efficient than might otherwise be expected, in achieving effective software state coverage.

2.5.5 Limitations of Undirected Random Testing

Undirected random testing (such as stochastic testing) leads to inefficient coverage of the test space, because the same state transitions may be executed many times over (Robinson, 1999). Furthermore, coverage of the test space is non-uniform, with clusters of data points in some areas and relatively light coverage in others, as illustrated by Figure 2 in Chapter 1 and discussed by Chen & Merkel (2007) and Chen, et al. (2007).

2.6 Alternative Approaches to Stochastic Testing

In order to address the limitations of undirected random testing, several approaches for improving coverage of the test space have been proposed. These include deterministic approaches such as exhaustive high-volume testing, quasi-random testing, adaptive testing, directed random testing, and genetic algorithms.

2.6.1 Exhaustive High-Volume Testing

Exhaustive high-volume testing may sometimes be possible: Hoffman & Kaner (2003) give an example of testing mathematical functions on a massively parallel computer. In this case, the integer square root for all integers on a 32-bit machine was tested in about six minutes; this found two errors that would probably not have been found without exhaustive testing. In this case, exhaustive testing of 2^{32} = 4,294,967,296 integer values was possible, and was justified by the expected use of this machine in life-critical applications. In each case, the calculated value was compared to the value provided by an independent test oracle. The authors contrast this case with a second example of testing 64-bit integer square roots. In this case, exhaustive testing was not possible as there are $2^{32} \times 2^{32} = 2^{64}$ integer values, which at the same rate as in the previous example would have taken approximately 4×10^8 hours (a large random sample of values over a wide range of values was tested instead; the authors refer to this technique as 'random function equivalence testing').

2.6.2 Quasi-Random Testing

Chen & Merkel (2007) discuss the use of 'quasi-random' testing to overcome the tendency in random testing for clustering and under-population in the test space; they present F-measures for testing using quasi-random sequences and show that this approach can be more effective than random testing.

2.6.3 Adaptive Testing

Adaptive testing uses feedback from the results of executing previous test cases to guide the generation of further tests. Adaptive testing has been discussed by Ince (1987) and by Hierons (2009). The 'delta debugging' technique of Cleve & Zeller (2005) for simplifying test traces is a form of adaptive testing.

Adaptive testing of software systems can perhaps be understood in terms of the input, process, output, storage (IPOS) model with feedback as shown in Figure 9. This is a general model of an information system with a feedback loop (Stair & Reynolds, 2008) where a measurement of the output of a system is fed back to the input of the system.

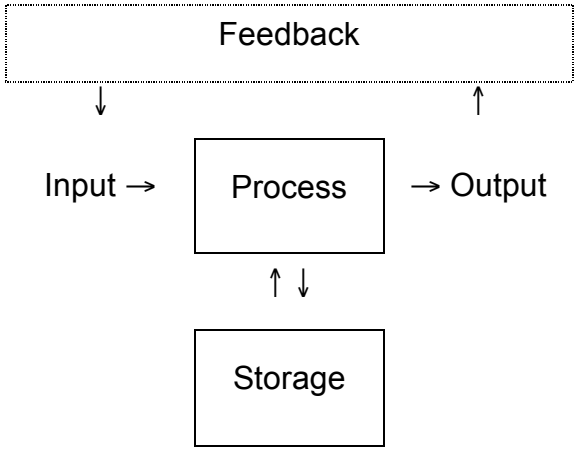


Figure 9: IPOS model with feedback

2.6.4 Directed Random Testing

Directed random testing is an adaptive testing technique based on random testing. Some reports suggest directed random testing is both more efficient and more effective than either deterministic testing or undirected random testing, for example see Chen, et al. (2007), Liu & Zhu (2008), Parizi, et al. (2009). However, Arcuri, Iqbal & Briand (2010) compared random testing, adaptive random testing, and search-based testing using genetic algorithms and in this study, no one of the three techniques was found to be

better in general than the others. There are several different directed random testing techniques - two examples are given here:

Godefroid, Klarlund & Sen (2005) present a technique called Directed Automated Random Testing (DART) in which feedback is provided through instrumented program code. The DART approach also derives the interface specification from analysis of the source code. Because DART has access to the interface specification of the SUT it is able to avoid generating random values that would simply be rejected by any input value 'filtering' code. In an experimental evaluation, DART found program faults in a few seconds or less, whereas an undirected random search did not find the faults even after many hours of searching.

Pacheco, et al. (2007) present a technique called Random tester for Object-Oriented Programs (RANDOOP) that uses feedback from the results of execution of program methods to build sequences of test cases for unit testing of classes. They report that feedback-directed random test generation achieves better code coverage and error detection in experimental evaluations than either deterministic test generation or undirected random test generation.

2.6.5 Genetic Algorithms

One promising variation on the theme of directed random testing is the use of genetic algorithms. Genetic algorithms are a class of techniques that search for a best or optimal solution to a problem in a manner that has been compared to the mechanism of biological evolution. A population of candidate solutions to a problem are evaluated using a 'fitness function' and the properties of the fittest solutions (which may not yet be optimal solutions) are propagated into a new generation of candidate solutions by a process of mixing ('crossover') and random mutation. The procedure is repeated over many generations until an optimal solution is eventually reached (Coley, 1999).

Godefroid & Khurshid (2004) describe the use of genetic algorithms for exploring very large state spaces in an attempt to avoid the problem of 'state space explosion'.

A genetic algorithm was used to choose the next state transition at each step of the state space exploration. In an experimental comparison of this approach to both deterministic search and undirected random search of the state space for two example programs, the genetic algorithm approach discovered program faults after searching for about one or two hours, however the other approaches did not discover the faults even after eight hours of search.

Berndt & Watkins (2005) advocate the use of genetic algorithms for high volume automated test generation for long sequence testing alongside other approaches such as the extended random regression (ERR) approach described by Kaner, Bond & McGee (2003).

Bati, et al. (2007) has extended the work of Slutz (1998) on the testing of DBMS, with the use of genetic algorithms guided by the results of query execution, for test generation. Undirected random generation of SQL results in many database queries that do not return results, and so do not test the underlying DBMS very effectively. This approach was reported to find ten times more defects than the original undirected approach.

2.7 Experimental Studies of DBMS

Experimental studies of DBMS were introduced in Chapter 1. This section reviews DBMS testing, methods of random generation of SQL, SQL mutation, and database creation.

2.7.1 DBMS Testing

Slutz (1998) performed a study of syntax testing of DBMS using grammar-based random generation of SQL (described in the next section) and reported that around 14% of executed random SQL statements resulted in a failure or incorrect result.

Costa & Madeira (1999) performed a study of software fault injection on the Oracle DBMS and reported that 18% of injected faults resulted in an aborted transaction, while

5% of injected faults resulted in the DBMS hanging. However, none of the injected faults affected the integrity of the database. Costa, Rilho & Madeira (2000) performed a similar study and reported that around 50% of injected software faults resulted in DBMS failures.

Willmor & Embury (2005) give criteria for adequacy of testing of database systems and observe that DBMS faults may involve interactions of program state and persistent database state. Haftmann, Kossmann & Kreutz (2005) also make this point.

2.7.2 Random Generation of SQL

Grammar-based random generation of SQL has been successfully employed at Microsoft Corporation for many years in the stochastic testing of DBMS, as reported by Slutz (1998), Bati, et al. (2007) and Garcia (2009).

Slutz (1998) described work on a system for random generation of SQL called RAGS. The RAGS system combined a random testing approach with grammar–based syntax generation to generate very large numbers of long, syntactically valid SQL queries. The complexity of the SQL specification means that it is difficult to predict the result of a sequence of such long, randomly generated SQL queries; to avoid this problem, Slutz compared several different vendors' DBMS products for the same sequence of SQL queries.

Haritsa (2006) has critiqued Slutz (1998), in particular it was not reported what proportion of failures revealed by RAGS, would be caught by existing test libraries; and RAGS was only tested using a very small database (less than 4k bytes). Nevertheless, the work of Slutz has been highly influential and was referenced for example by Hamilton (1998), Bruno, Chaudhuri & Thomas (2002), Zeller & Hildebrandt (2002), Chays, et al. (2004), Perez (2004), Chen & Merkel (2007), Chen, et al. (2007), Tuya, Suárez-Cabal & de la Riva (2007), and Khalek, et al. (2008).

Several DBMS-specific testing tools employing random generation of SQL have become available in recent years, and DBMS for which such tools are available include SQL Server, MySQL, Oracle, and SQLite (2010). Bal, Fan & Lintz (2009) list a number of tools for automated database testing, including RAGS.

Stoev (2009), MySQL Forge (2010), and Khalek & Khurshid (2010) describe the MySQL Random Query Generator (RQG). Like RAGS, the RQC performs grammar-based generation of random SQL queries that can be run against several databases allowing the results to be compared. Khalek & Khurshid (2010) note that RQC can produce invalid queries, and that it is difficult to verify if generated queries are syntactically correct. These authors do not describe any specific faults, however several MySQL faults that were found using the RQG are reported on the MySQL online bug-tracking system at http://bugs.mysql.com/.

Finnigan (2009) and Naraine (2009) describe the Sentrigo 'Fuzzor', an open-source PL/SQL Fuzzer for Oracle available at http://www.sentrigo.com/products/fuzzor/.

SQLite (2010) reports that SQL fuzz testing is performed with approximately 104,000 SQL statements for the SQLite DBMS.

2.7.3 SQL Mutation

SQL mutation is a relatively new area of DBMS testing research; Tuya, Suárez-Cabal & de la Riva (2006, 2007) state that they know of no earlier work on SQL mutation than Chan, Cheung & Tse (2005). More recently, Cabeca, Jino & Leitao-Junior (2009) and Derezińska (2009) also discuss SQL mutation.

In all of the studies mentioned above, mutation operators for SQL are defined in some detail and the results of case studies using automated tools to generate and execute mutated SQL are reported. However, none of these studies appear to have applied SQL mutation directly to the problem of DBMS testing; the use of SQL mutation is restricted to assessing the adequacy of tests designed to distinguish between correct and incorrect

SQL, by comparing the results of executing mutants with the results of executing non-mutated SQL statements for a predefined database. Although some failing responses from the DBMS as a result of executing SQL mutants are mentioned in passing, this was not the focus of the reported studies.

2.7.4 Database Creation

Several predefined databases are widely available for DBMS testing; both MySQL and Oracle provide sample databases, such the Menagerie database described in the MySQL 5.0 Reference Manual (2010) and the Sales History database described in the Oracle® Database SQL Reference (2005).

An alternative approach is the creation of a test database by populating a predefined schema with randomly generated data; such data must conform to the schema integrity constraints. Several tools for creating test databases in this way exist, for example: AGENDA described by Chays, et al. (2004), ADUSA described by Khalek, et al. (2008), and Xplod described by Banahatti, Iyengar & Lodha (2009).

2.7.5 Random Databases

There appears to be relatively little published research into random databases, that is to say, databases with a randomly generated schema, rather than simply populating a predefined schema with random data.

Seleznjev & Thalheim (1998) developed statistical models for the average lengths of keys in random databases.

Korovin & Voronkov (2005) studied some statistical properties of random databases, in particular the existence of ‘threshold’ properties related to the asymptotic behaviour of random graphs; the literature on random graphs is extensive, see for example Janson (1995). Korovin & Voronkov (2005) show that as a random database grows in size, the probability of a particular query returning ‘true’ rapidly converges to either 0 or 1 as given by a threshold function.

Bati, et al. (2007) suggested research into randomly generated schemas as future work.

2.8 Delayed Failure

Delayed failure was introduced in Chapter 1. This section reviews the literature of failure delay systems and of delayed software failure.

2.8.1 Failure Delay Systems

Failure delay systems were introduced in Chapter 1. Limnios (1988) described reliability modelling of failure delay systems and formally defined a failure delay system in which there is a non-negligible delay between the occurrence of a failure condition and the corresponding failure. Examples of such systems include a water reservoir and a two-component stand-by system. In a reservoir in which the flow into the reservoir is interrupted, the flow out of the reservoir is only interrupted after some delay, when the reservoir is emptied. If the flow into the reservoir is restored before the reservoir is emptied, then the flow out of the reservoir remains uninterrupted. In the case of a stand-by system, failure of the first (active) component causes the second (stand-by) component to be activated; if the first component is not repaired, the system will continue to function until failure of the second component. In both examples, the continuous presence of a failure condition for a given time τ (fixed or random) is a condition for failure. Limnios (1988) derived measures of reliability, availability, and maintainability for these types of systems.

Faria (2008) has considered degradation of performance over time in failure delay systems containing several interconnected components by mathematical modelling of failure states in concurrent processes.

2.8.2 Delayed Software Failure

Delayed software failure was introduced in Chapter 1. Pyle, McLatchie & Grandage (1971) described the analysis of a delayed failure in an interactive computer system named 'HUW' on IBM System 360/65. "*Many HUW runs crashed after about one*

hour, with a diagnostic message from OS/360 reporting an illegal Data Control Block". The cause was overwriting of an area of memory; the overwritten area of memory was not used until some time later, when a failure indication was reported. In this case, the EDIT command in HUW had a copying operation that was incorrectly copying a block of data in response to a zero-length input string. The authors concluded that string-handling programs should be designed to deal with zero-length strings, but also hypothesised that "*related bugs are lurking in many other systems*".

The phone system described by Kaner (1997) (state diagram shown in Figure 7) provides an example of delayed software failure. In this case, it was reported that a software bug caused errors to accumulate on a stack, with stack overflow eventually resulting in a system crash.

Thompson (2007) provides an example of delayed failure of a software system in an account of an experience with an in-flight entertainment system on an airplane flight from Las Vegas to Orlando in mid-2005. This system provided a variety of television channels and games through a television monitor and included a phone console with a numeric keypad. A parameter of the game could be incremented on the screen up to a maximum value of 4; but it was found by trial and error that this could also be set to an illegal value of 5 through the phone keypad. The author notes the software was now in an 'unintended state' and that at this point it was possible to continue to increment up to a value of 127. Incrementing again (presumably) resulted in an overview of a variable such that 127 + 1 = −128. The system is then reported to have crashed; all users consoles were disabled until the system was reset. Although the account is uncorroborated and it is not possible to confirm the actual cause of the failure, this example is consistent with error propagation leading to delayed failure; the software continued to operate correctly in the 'unintended state' until the game parameter was incremented beyond 127.

Lu, et al. (2005) developed a benchmark suite for evaluating software bug detection tools and conducted a study of several tools. In this study the authors defined *crash latency* as the latency from the root cause of a bug to the place where the application

finally crashes due to the propagation of the bug. They found crash latency values from zero to 29 million instructions. Software error latency has been studied by Chillarege & Iyer (1987) and by Madeira & Silva (1994). Yim, Kalbarczyk & Iyer (2009) discuss a model of error propagation and latency shown in Figure 3 ('terminology' in Chapter 1).

Tan, Chen & Jakubowski (2006) describe an approach to 'tamper-resistant' software employing a deliberate delayed failure in response to tampering. They observed that alternative techniques deliberately cause a program failure at the place where detection happens, however this allows an adversary to trace back to the tamper detection code, and so they adopted the strategy of introducing a large failure delay. This is achieved by corrupting the software state, in the form of global pointer variables. The authors note that software bugs can cause delayed failure and cite Pyle, McLatchie & Grandage (1971).

2.9 Summary

This chapter presented a review of the software engineering literature in the areas of software component testing techniques, software component reliability, software component robustness, stochastic testing, alternative approaches to stochastic testing, experimental studies of DBMS, and delayed failure.

3 CONCEPTUAL FRAMEWORK

3.1 Introduction

The conceptual framework provides a foundation for the present research and consists of two elements: (a) a 'stateless' request-response model of software component behaviour, and (b) a state transition model of error propagation and failure. The primary research goal (as described in Chapter 5) is to verify by experiment that the state transition model correctly describes the delayed failure behaviour of real software components. The stateless request-response model partitions component behaviour into stateless and state-based elements and defines the domain of the research.

The state-transition model of error propagation and failure defines a two-step 'precondition' and 'trigger' process that characterises delayed failure. The conceptual framework provides a 'bridge' between the more general research problem and the specific experimental hypotheses, methodology and experimental procedures.

This chapter presents the simple stateless request-response model and a refinement of this model developed during the present research – the partitioned request-response model; along with the state transition model of error propagation and failure, the Markov chain model, and the process of delayed failure.

3.2 Stateless Request-Response Model

The stateless request-response model was introduced in Chapter 1. This section describes the simple stateless request-response model and a refinement of this model developed during the present research – the partitioned request-response model.

3.2.1 Simple Request-Response Model

As discussed in Chapters 1 and 2, the output response of a software component generally depends on both the input request and on the software state, as represented in the IPOS model shown in Figure 8 in Chapter 2. Predicting the outcome of a long sequence of test cases consequently requires a test oracle based on a state-based model of the software component. In Chapter 1 it was explained how the need for such an oracle might be avoided by the use of the 'stateless' request-response model which is shown in Figure 10.

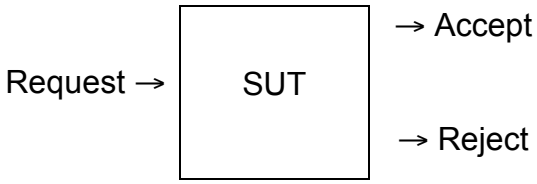


Figure 10: Stateless request-response model

The stateless request-response model ignores the internal state of the SUT; the only knowledge the oracle has, is that the SUT should accept a valid request and reject an invalid request. If the SUT accepts an invalid request, or rejects a valid request, or does not respond (in the case of a hang/crash) then this is identified as a failure of the SUT.

In this model it is assumed that it is always possible to determine if a request is valid or invalid, by reference to the software component specification. This is consistent with BS 7925-2 (2001) where it is stated: "*Given any initial state of the component, in a defined environment, for any fully-defined sequence of inputs and any observed outcome, it shall be possible to establish whether or not the component conforms to the specification*."

Software components that do not conform to the stateless request-response model are excluded from the present research; an example might be a component that simply accepts all requests without validating them, such as the UNIX null device /dev/null.

This device discards all data written to it and produces a 'success' response to the write operation, regardless of the content of the request.

3.2.2 Partitioned Request-Response Model

The stateless request-response model is a 'black-box' model and consequently does not allow more detailed analysis of the process of software component failure; however, the stateless request-response model has been refined in the present research into the 'partitioned' request-response model shown in Figure 11 in which the software component contains both stateless and state-based elements within the component boundary.

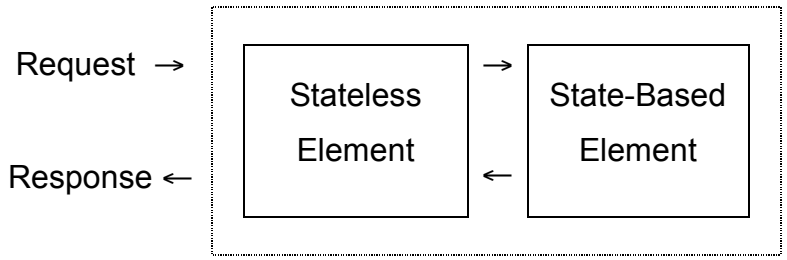


Figure 11: Partitioned request-response model

Note that the use of the partitioned request-response model does not imply or require that the actual implementation of the software component should be partitioned in this way; however, compare Figure 5: Block diagram of DBMS architecture in Chapter 1.

In the partitioned request-response model, requests are validated by the stateless element, and valid requests are passed on to the state-based element; the state-based element produces an output value that is passed back in the 'accept' response. For the case of an invalid request, the stateless element produces a 'reject' response without communicating with the state-based element.

A valid input value should be accepted by the stateless element regardless of the internal state of the state-based element; however, the request may still not succeed if

the state-based element is not in a suitable state to process the request. In this case, the 'accept' response will include an error status. If the request succeeds, the 'accept' response will include a success status as well as any output value passed back from the state-based element. As an example, consider the case of a DBMS component receiving a request to create a database table; if the requested table does not already exist then the request should succeed, however, if the requested table already exists, then an error response should be returned.

The partitioned request-response model retains the simplicity of the stateless model, but allows an analysis of possible mechanisms of software component failure; for example, in the case of a fault, the stateless element may incorrectly reject a valid request, or else may incorrectly accept an invalid request, passing this on to the state-based element. In the latter case this may lead to error propagation and failure within the state-based element.

The following state transition model describes error propagation and failure within a software component (that is, within the state-based element of the partitioned request-response model).

3.3 State Transition Model

This section describes the state transition model of error propagation and failure, the Markov chain model, and the consequent process of delayed failure.

3.3.1 Description of the State Transition Model

Delayed failure of a software component can be understood in terms of a state transition model of error propagation and failure. The model described here is similar to the model previously described by Huang, et al. (1995) and by Bernstein & Kintala (2004). However, the model described by these authors was introduced in the context of what has been called 'software rejuvenation' (the practice of re-initialising a software component before an error has the opportunity to propagate and cause a failure) and did not consider delayed failure. In the present research, delayed failure is described in

terms of long-latency error propagation in a state transition model of error propagation and failure.

In the following account of the state transition model, a subscripted upper case letter such as S_0 denotes a 'macro-state' of the software component, where a macro-state is defined as a collection of software states sharing a common property, as discussed by Datta (2007) and Faria (2008).

The state transition model assumes that the executing software component can be described at all times by one of the following macro-states; either a 'robust' macro-state S_0, a 'precondition' macro-state S_P or a 'failure' macro-state S_F as indicated by circles in the state transition diagram shown in Figure 12. A transition between macro-states is assumed to occur when the software component processes an input request (possibly also producing an output response) and the only possible transitions between macro-states are assumed to be those indicated by the arrows in Figure 12.

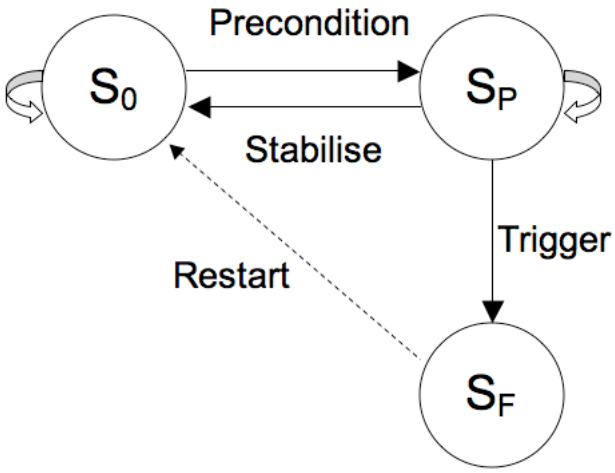


Figure 12: State transition model

Self-loop transitions from S_0 back to S_0 and from S_P back to S_P are assumed to be possible as the component processes input requests, as indicated by the curved block arrows in Figure 12.

The term ‘robust’ is used for the macro-state S_0 because no transition to the ‘failure’ macro-state S_F is possible from S_0 and the term ‘precondition’ is used for the macro-state S_P because any transition to the ‘failure’ macro-state S_F requires a transition to S_P as a precondition to failure. The macro-state S_P is called a ‘failure probable state’ by Huang, et al. (1995) and by Bernstein & Kintala (2004), however the property of being a ‘precondition to failure’ is emphasised in the present research, where latency in the macro-state S_P terminated by a ‘trigger’ event leading to failure is identified as the fundamental mechanism of delayed failure. Software states in the ‘precondition’ macro-state S_P are incorrect software states (errors) that (it is assumed) are not intended by the designers of the component.

This model assumes that the component state at initialisation corresponds to the ‘robust’ macro-state S_0. It is further assumed that the software component processes a sequence of input requests resulting in a sequence of transitions between macro-states, and that this eventually results in a transition to the ‘precondition’ macro-state S_P.

As the software component continues to process input requests, it is assumed that the software eventually either ‘self-stabilises’ through a transition from S_P back to S_0 or else fails through a transition from S_P to the ‘failure’ macro-state S_F. The software might remain indefinitely in the macro-state S_P however it is assumed here that a transition to another macro-state will eventually occur.

This model assumes that failure is a terminal condition, such that the software remains in the ‘failure’ macro-state and does not process further input requests until the component is restarted, so bringing it back once again to the ‘robust’ macro-state S_0 as indicated by the dashed arrow in Figure 12.

Note that the state transition model does not distinguished between a persistent software state and a non-persistent software state; therefore restarting the component is assumed to re-initialise both persistent and non-persistent storage.

The state transition model of error propagation inherently possesses latency that can result in delayed failure, since at least two (and probably more) transitions must occur due to processing input requests before a failure can occur.

Delayed failure is defined for the present research as a failure that occurs some time after the condition that causes the failure; the condition leading to failure in this case is the processing of an input request resulting in the first transition from S_0 to S_P (the 'precondition'). The period of delay is from this input request until the input request resulting in the final transition from S_P to S_F (the 'trigger').

The state transition model provides a description of delayed failure as the result of long-latency error propagation where a 'precondition' input request is followed some time later by a 'trigger' input request.

Note that the precondition and/or trigger events might alternatively be an internal signal within the software component caused, for example, by a timeout or a failure to allocate a resource; however this case falls outside the scope of the present research, which is limited to communication between components in accordance with the request-response model.

Delayed failure is characterized by two latency intervals as shown in Figure 3 in Chapter 1: fault activation latency and error propagation latency. Fault activation latency includes any self-stabilisation transitions from S_P back to S_0. Error propagation latency includes any 'dwell time' in the precondition state due to self-loop transitions from S_P to S_P and the final transition to observable failure.

Either the precondition or the trigger for delayed failure may be either a valid input request, or an invalid input request that has been incorrectly accepted due to a fault. Invalid input values that are accepted by the SUT might be more likely to result in failure than are valid input values, if this case was unanticipated by the designers of the software component.

3.3.2 Markov Chain Model

The adjacency matrix shown in Table 4 is an equivalent representation of the state transition model shown in Figure 12. Here a value of 1 indicates that a transition may take place between the associated macro-states while a value of 0 indicates that no transition is possible.

	S_0	S_P	S_F
S_0	1	1	0
S_P	1	1	1
S_F	0	0	1

Table 4: State transition model adjacency matrix

For the particular case of a random sequence of input values (stochastic testing) the state transition model can be interpreted as an *absorbing Markov chain* by assigning a probability to each of the possible transitions between macro-states. By multiplying each value in the adjacency matrix by the appropriate transition probability the adjacency matrix becomes a transition matrix, as shown in Table 5. Note that each row in the transition matrix sums to 1. In the theory of Markov chains S_F is *absorbing* because no transition from S_F to another macro-state is possible.

	S_0	S_P	S_F
S_0	P_1	P_2	0
S_P	P_3	P_4	P_5
S_F	0	0	1

Table 5: Markov chain transition matrix

Markov chain models of software failure were reviewed in Chapter 2 'Models of Error Propagation and Failure' (Whittaker & Thomason, 1994; Thomason & Whittaker, 1999) however these models do not explicitly model error propagation. The Markov chain model represented by the transition matrix shown in Table 5 provides a model of error propagation through transitions to the 'precondition' macro-state S_P.

Given the Markov chain transition matrix, the behaviour of the absorbing Markov chain model can be analysed using the theory of absorbing Markov chains, as found for example in Grinstead & Snell (1998) (Chapter 11: Markov Chains).

The matrix Q determines the behaviour of the absorbing Markov chain model:

$$Q = \begin{bmatrix} P_1 & P_2 \\ P_3 & P_4 \end{bmatrix}$$

The matrix $N = (I - Q)^{-1}$ is called the fundamental matrix, where I denotes the identity matrix. By Theorem 11.5 of Grinstead & Snell (1998) it can be shown that the expected number of steps before the chain is 'absorbed' for each starting state is given by the sum of the entries in the corresponding row of the fundamental matrix.

The sum of the entries in the first row of the matrix N gives the expected value for F-measure (starting in the macro-state S_0) during stochastic testing and the sum of the entries in the second row of the matrix N (starting in the macro-state S_P) gives the expected value for failure delay. Expected values for F-measure and failure delay under the assumption of particular values of transition probability are calculated in Chapter 4.

The Markov chain model represented by the transition matrix shown in Table 5 assumes a single value for the transition probability between macro-states, that is to say a unimodal probability distribution is assumed for transitions between macro-states.

3.3.3 The Process of Delayed Failure

The simple model of error propagation and failure with latency discussed by Yim, Kalbarczyk & Iyer (2009) implies a two-step process of failure: fault → error → failure however this model does not provide a mechanism that allows analysis of error propagation latency. The state transition model described by Huang, et al. (1995) does provide a mechanism that would allow analysis of error propagation latency, although these authors do not explicitly recognise latency in their model. The state transition model shown in Figure 12 emphasises a 'precondition to failure' where latency in the macro-state S_P terminated by a 'trigger' event leading to failure provides the mechanism for delayed failure.

Latency in the macro-state S_P occurs either as a result of a direct self-loop transition from S_P back to S_P or as a result of an indirect path consisting of a 'stabilising' transition from S_P to S_0 and a further 'precondition' transition from S_0 back to S_P (possibly with one or more intervening self-loop transitions from S_0 back to S_0). This process may continue indefinitely until a failure transition from S_P to S_F occurs; the theory of absorbing Markov chains provides a mathematical analysis of this process in terms of transition probabilities. It should be noted that 'fail-stop' approaches to software fault tolerance such as rejuvenation and self-stabilisation have the potential to prevent delayed failure, provided these mechanisms have sufficiently small error detection latency. However Chandra & Chen (1998) conducted a study of error propagation in the Postgres database system using fault injection and concluded that the fail-stop model was violated in 7% of cases.

3.4 Summary

This chapter presented the simple stateless request-response model and a refinement of this model developed during the present research – the partitioned request-response model, along with the state transition model of error propagation and failure, the Markov chain model, and the process of delayed failure.

4 RESEARCH QUESTION AND HYPOTHESES

4.1 Introduction

The conceptual framework described in Chapter 3 consists of two elements, the stateless request-response model, and the state transition model. The stateless request-response model was shown to apply for the case of DBMS in Chapter 1, and this was a factor in the choice of DBMS as a subject of study.

The question of the validity of the state transition model will now be considered. This question leads to two hypotheses; the first hypothesis H_1 applies in the case of an invalid 'precondition' input request, and the second hypothesis H_2 applies in the case of a valid 'precondition' input request. The corresponding null hypotheses can be tested by experiment. Rejection of the null hypotheses provides an experimental validation of the state transition model.

The order of presentation of the hypotheses, first the 'invalid' case and second the 'valid' case, reflects the order in which these concepts originated in the motivation for the present research, as described in Chapter 1, and this is the order in which the experiments are performed, as described in Chapters 5 and 6.

This chapter presents a statement of the research question, the research hypotheses H_1 and H_2 together with the associated null hypotheses, calculations of expected values of F-measure and failure delay using the Markov chain model, and why answering this question is worthwhile.

4.2 Statement of the Research Question

The research question can be stated simply as: *Is the state transition model valid?*

If the state transition model is valid, it should correctly predict the behaviour of real software components. In Chapter 3, the state transition model was shown to entail a

process of delayed failure. The research question can be restated in the following form: Do real software components exhibit delayed failure, consistent with long-latency error propagation? If the state transition model is valid, delayed failure should be observed in the case of both valid and invalid 'precondition' input requests; the hypotheses H_1 and H_2 correspond to these two cases, respectively. The expected failure behaviour in the case of each hypothesis can be calculated from the Markov chain model as discussed in Chapter 3.

4.3 Hypothesis H_1

This section presents the first hypothesis H_1 the corresponding null hypothesis H_{01} and a calculation of the expected F-measure and failure delay using the Markov chain model.

4.3.1 Hypothesis

Hypothesis H_1: *Delayed failure of a software component may occur due to long-latency error propagation following acceptance of an invalid input request by the SUT.*

The hypothesis H_1 follows from the state transition model; it is expected that an invalid 'precondition' input request being accepted by the SUT (due to a fault) will not lead directly to failure of the software component, but that failure may occur following a subsequent 'trigger' input request some time later.

It is not expected that delayed failure will occur in every stochastic testing experiment. Possible alternatives to delayed failure include the following:

a) The experimental profile might not contain an invalid input request that is accepted by the SUT.

b) The SUT might not have a fault that will cause an invalid input request to be accepted.

c) Invalid input requests that are accepted by the SUT might not cause the SUT to enter a 'precondition' state.

d) The SUT might 'self-stabilise' from the 'precondition' state back to a 'robust' state before a failure can occur.

e) Invalid input requests that are accepted by the SUT might not cause the SUT to fail from the 'precondition' state.

Rather than looking for evidence to support the hypothesis H_1 the experimental method looks for evidence to reject the following null hypothesis H_{01}.

4.3.2 Null Hypothesis

Null hypothesis H_{01}: *Delayed failure of a software component due to long-latency error propagation will not occur following acceptance of an invalid input request by the SUT.*

4.3.3 Expected Values of F-measure and Failure Delay

The expected values of F-measure and failure delay for invalid input requests F_{invalid} and D_{invalid} can be calculated using the Markov chain model as described in Chapter 3.

From the Markov chain transition matrix we have:

$$P_1 + P_2 = 1$$
$$P_3 + P_4 + P_5 = 1$$

Where:

P_1 is the probability of a transition from S_0 to S_0
P_2 is the probability of a transition from S_0 to S_P
P_3 is the probability of a transition from S_P to S_0
P_4 is the probability of a transition from S_P to S_P
P_5 is the probability of a transition from S_P to S_F

Assuming (for example) the following values for state transition probabilities:

$$P_1 = 9/10$$
$$P_2 = 1/10$$
$$P_3 = 1/10{,}000$$
$$P_4 = 9998/10{,}000$$
$$P_5 = 1/10{,}000$$

Here it is assumed that $P_3 = P_5 = 1/10{,}000$ (anticipating order of magnitude of experimental results) and that transitions to S_P due to invalid input values are much more likely say $P_2 = 1000 \times P_5 = 1/10$.

Substituting the values for state transition probability into the matrix Q:

$$Q_{invalid} = \begin{bmatrix} 9/10 & 1/10 \\ 1/10{,}000 & 9998/10{,}000 \end{bmatrix}$$

A computer program written by the author was used to calculate the corresponding fundamental matrix:

$$N_{invalid} = \begin{bmatrix} 20 & 10{,}000 \\ 10 & 10{,}000 \end{bmatrix}$$

The sum of the entries in the first row of the matrix $N_{invalid}$ gives the expected value for F-measure $F_{invalid} = 10{,}020$.

The sum of the entries in the second row of the matrix $N_{invalid}$ gives the expected value for failure delay $D_{invalid} = 10{,}010$.

This analysis assumes that, due to a fault, the SUT will incorrectly accept an invalid input request; however, it is not expected that every invalid input request will be accepted by the SUT. If the probability that an invalid input request is accepted by the

SUT is P_0 then the expected values for F-measure and failure delay derived from the Markov chain model must be divided by P_0 so that (for example) assuming a value of 1/100 for P_0 gives modified values for F-measure F_{invalid} = 1,002,000 and for failure delay D_{invalid} = 1,001,000.

4.4 Hypothesis H_2

This section presents the second hypothesis H_2 the corresponding null hypothesis H_{02} and a calculation of the expected F-measure and failure delay using the Markov chain model.

4.4.1 Hypothesis

Hypothesis H_2: *Delayed failure of a software component due to long-latency error propagation may occur following acceptance of a valid input request by the SUT.*

The hypothesis H_2 follows from the state transition model; it is expected that a valid 'precondition' input request being accepted by the SUT will not lead directly to failure of the software component, but that failure may occur following a subsequent 'trigger' input request some time later.

As for the invalid case, it is not expected that delayed failure will occur in every stochastic testing experiment. Possible alternatives to delayed failure include the following:

a) Valid input requests that are accepted by the SUT might not cause the SUT to enter a 'precondition' state.

b) The SUT might 'self-stabilise' from the 'precondition' state back to a 'robust' state before a failure can occur.

c) Valid input requests that are accepted by the SUT might not cause the SUT to fail from the 'precondition' state.

Rather than looking for evidence to support the hypothesis H_2 the experimental method looks for evidence to reject the following null hypothesis H_{02}.

4.4.2 Null Hypothesis

Null hypothesis H_{02}: *Delayed failure of a software component due to long-latency error propagation will not occur following acceptance of a valid input request by the SUT.*

4.4.3 Expected Values of F-measure and Failure Delay

The expected values of F-measure and failure delay for valid input requests F_{valid} and D_{valid} can be calculated using the Markov chain model as described in Chapter 3.

Assuming (for example) the following values for state transition probabilities:

$$P_1 = 9999/10{,}000$$
$$P_2 = 1/10{,}000$$
$$P_3 = 1/10{,}000$$
$$P_4 = 9998/10{,}000$$
$$P_5 = 1/10{,}000$$

Here it is assumed that $P_2 = P_3 = P_5 = 1/10{,}000$ anticipating order of magnitude of experimental results and assuming that transitions to S_P due to valid input values are much less likely than for the case of invalid values.

Substituting the values for state transition probability into the matrix Q:

$$Q_{valid} = \begin{bmatrix} 9999/10{,}000 & 1/10{,}000 \\ 1/10{,}000 & 9998/10{,}000 \end{bmatrix}$$

The corresponding fundamental matrix was calculated to be:

$$N_{valid} = \begin{bmatrix} 20{,}000 & 10{,}000 \\ 10{,}000 & 10{,}000 \end{bmatrix}$$

The sum of the entries in the first row of the matrix N_{valid} gives the expected value for F-measure $F_{valid} = 30{,}000$.

The sum of the entries in the second row of the matrix N_{valid} gives the expected value for failure delay $D_{valid} = 20{,}000$.

4.5 Why Answering this Question is Worthwhile

As mentioned in Chapter 1, there is a perceived gap between academic research into software testing and industrial software testing practice (Reid, et al., 1999). A software testing practitioner considering employing stochastic testing might take a 'pragmatic' view, that the technique has been shown to reveal faults that alternative software testing techniques did not reveal; for example, see Nyman (2000). However, this argument does not explain the reported success of the technique. Demonstration that stochastic testing can reveal delayed failure due to long-latency error propagation, supported by a model of error propagation and failure which has been validated by experiment, provides a deeper insight into the technique, which has explanatory power and may lead to further research questions. The point here is to fill a 'gap in knowledge' not merely to provide a practical justification of a testing technique.

4.6 Summary

This chapter presented a statement of the research question, the research hypotheses H_1 and H_2 together with the associated null hypotheses, calculations of expected values of F-measure and failure delay using the Markov chain model, and why answering this question is worthwhile.

5 METHODOLOGY AND EXPERIMENTAL PROCEDURE

5.1 Introduction

The research problem introduced in Chapter 1 was investigated through the methods of experimental software engineering: the development of a research question and hypotheses, and an experimental methodology for testing the hypotheses. The experimental methodology is designed firstly to test the hypotheses that were derived from the research question, and in this way to test the validity of the state transition model, and secondly to determine baseline metrics for the components under test.

Before presenting the experimental procedure, this chapter first presents the research goals, a generic procedure for testing the hypotheses, the experimental design, experimental configurations, methods for random generation of SQL and SQL mutation, DBMS initialisation, database creation, and the method for identifying minimal test sequences.

This chapter then presents the procedures to be followed in the 1st experiment - MySQL (invalid and valid SQL) and the 2nd experiment - Oracle XE (valid SQL), and discusses methodological challenges, and experimental questions.

5.2 Research Goals

This section presents the primary and secondary goals of the research.

5.2.1 Primary Research Goal

The primary goal of the present research is to validate the state transition model described in Chapter 3, verifying by experiment if the model correctly describes the delayed failure behaviour of real software components. Stochastic testing with both 'valid' and 'invalid' experimental profiles tests the research hypotheses H_1 and H_2 by rejecting the null hypotheses H_{01} and H_{02} if possible.

5.2.2 Secondary Research Goal

The secondary goal of the present research is to determine baseline metrics for the components under test, including failure delay, reliability, robustness, testing effectiveness and efficiency, and time constant. These metrics should provide the basis for a comparison of different software components and for a comparison of different testing techniques, in corroboration of related work and in future research.

5.3 Generic Procedure for Testing the Hypotheses

Before describing specific experimental procedures, a generic procedure is described. This generic procedure for testing the research hypotheses applies to both hypothesis H_1 and hypothesis H_2 and forms the basis of the experimental procedures for both the 1st and 2nd experiments. The macro-state labelling convention S_0 S_P and S_F are used in the state transition model described in Chapter 3, however the internal state of the SUT is not directly observable and must be inferred from observations of the external behaviour of the SUT.

5.3.1 Generic Procedure Part I

The first part of the procedure finds a 'candidate' software failure:

1. The SUT is initialised into the 'robust' macro-state S_0.
2. The SUT cycles in the 'robust' macro-state S_0 processing requests.
3. A 'precondition' request is accepted by the SUT.
4. The SUT enters the 'precondition' macro-state S_P.
5. The SUT cycles in the 'precondition' macro-state S_P processing requests.
6. A 'trigger' request is accepted by the SUT.
7. The SUT fails (enters the failure macro-state S_F).

5.3.2 Generic Procedure Part II

The second part of the procedure shows that the 'trigger' alone does not result in failure:

1. The SUT is re-initialised into the 'robust' macro-state S_0.
2. The SUT cycles in the 'robust' macro-state S_0 processing requests.
3. The 'precondition' request in Part I is omitted; instead, the same 'trigger' request as in Part I is accepted by the SUT.
4. The SUT does not fail but continues to cycle in the 'robust' macro-state S_0 processing requests.

5.3.3 Generic Procedure Part III

The third part of the procedure shows that the 'precondition' alone does not result in failure:

1. The SUT is re-initialised into the 'robust' macro-state S_0.
2. The SUT cycles in the 'robust' macro-state S_0 processing requests.
3. The same 'precondition' request as in Part I is accepted by the SUT.
4. The SUT enters the 'precondition' macro-state S_P.
5. The SUT cycles in the 'precondition' macro-state S_P processing requests.
6. The 'trigger' request in Part I is omitted; the SUT does not fail but continues to cycle in the 'precondition' macro-state S_P processing requests.

5.3.4 Generic Procedure Part IV

The final part of the procedure shows that the SUT does not fail when both the 'trigger' and the 'precondition' requests are omitted:

1. The SUT is re-initialised into the 'robust' macro-state S_0.
2. The SUT cycles in the 'robust' macro-state S_0 processing requests.
3. Both the 'trigger' and the 'precondition' requests are omitted.

4. The SUT does not fail but continues to cycle in the 'robust' macro-state S_0 processing requests.

The condition for delayed failure as verified by the four parts of the generic procedure is shown in Table 6.

Precondition	*Trigger*	*Failure*
Yes	Yes	Yes
No	Yes	No
Yes	No	No
No	No	No

Table 6: Condition for delayed failure

5.4 Experimental Design

This section first presents the abstract form of the experimental design and then presents the instantiation of the experimental design in the specific form used in the experiments.

5.4.1 Abstract Design

The experiment will compare two different techniques for generating input values for two different software components, therefore a 2 × 2 factorial design having two different SUT S_1 and S_2 and two different experimental profiles P_1 and P_2 is appropriate, as shown in Figure 13. In a factorial design, the effects of more than one independent variable (or factor) are observed simultaneously; this is more efficient than repeating several single factor experiments, and any interactions between factors can be detected (Easton & McColl, 2005).

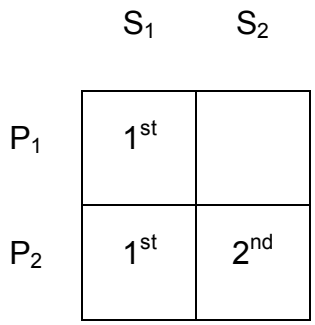


Figure 13: Experimental design

5.4.2 Instantiation of the Design

Each quadrant of the experimental design corresponds to a possible experiment. The software under test S_1 is the MySQL DBMS and S_2 is the Oracle XE DBMS. The profile P_1 contains invalid values as described for the 1st experiment, and corresponds to the hypothesis H_1, while the profile P_2 has only valid values as described for the 1st and 2nd experiments, and corresponds to the hypothesis H_2. The correspondence between the 1st and 2nd experiments and the abstract design is indicated in Figure 13. Each experiment is performed using the same method for applying the experimental profiles to the SUT.

The following metrics are recorded in each of the two experiments: F-measure, failure delay, and time constant. MTTF can be calculated as the average of a number of F-measures for the same software component using stochastic testing with the same experimental profile.

5.5 Experimental Configurations

The experiments using MySQL and Oracle XE were performed using separate personal computers with MySQL running on a Windows machine and Oracle XE running on a Linux machine. The corresponding hardware and software configurations for the test machines are shown in APPENDIX B.

This section describes the program and data file configurations for the MySQL and Oracle XE experiments.

5.5.1 MySQL Data Files

/mysql/data/acc.txt	Accepted statements log
/mysql/data/bnf09.txt	BNF specification 2009 version
/mysql/data/bnf10.txt	BNF specification 2010 version
/mysql/data/err.txt	Failed statements log
/mysql/data/invalid.txt	Invalid SQL from syntax parser
/mysql/data/log.txt	Test trace log
/mysql/data/reset.txt	SQL commands to reset database
/mysql/data/suc.txt	Succeeded statements log
/mysql/data/summary.txt	Summary statistics of run
/mysql/data/valid.txt	Valid SQL from syntax parser
/mysql/data/x.txt	SQL statements to be executed

5.5.2 MySQL Program Files

/mysql/programs/bmut.pl	BNF mutation
/mysql/programs/hgen.pl	Generate SQL from BNF
/mysql/programs/hparse.pl	Parse syntax against BNF
/mysql/programs/init.pl	Initialise log files
/mysql/programs/krun.bat	Batch file for response profile
/mysql/programs/reset.bat	Reset database
/mysql/programs/smut.pl	Syntax mutation
/mysql/programs/xx.pl	Execute SQL statements

5.5.3 Oracle XE Data Files

/oracle/data/acc.txt	Accepted statements log
/oracle/data/cleanup.txt	SQL statements to clean up database
/oracle/data/err.txt	Failed statements log
/oracle/data/log.txt	Test trace log
/oracle/data/oracle_bnf_2.txt	BNF specification
/oracle/data/suc.txt	Succeeded statements log
/oracle/data/summary.txt	Summary statistics of run
/oracle/data/tmp.txt	Temporary work file
/oracle/data/x.txt	SQL statements to be executed

5.5.4 Oracle XE Program Files

/opt/ActivePerl-5.8/bin/perl	Perl
/oracle/programs/cleanup.sh	Clean up database
/oracle/programs/debug.txt	Debug log
/oracle/programs/oracle_gen.pl	Generate SQL from BNF
/oracle/programs/ox.pl	Execute SQL statements
/oracle/programs/reset	Initialise log files
/oracle/programs/run1.sh	Script for F-measure
/oracle/programs/run2.sh	Script for failure delay
/oracle/programs/run3.sh	Script for time constant

5.6 Random Generation of SQL

This section describes the method of automatic generation of SQL, and the 'valid' experimental profiles P_{2a} and P_{2b} used in the 1st experiment with MySQL and P_2 used in the 2nd experiment with Oracle XE.

5.6.1 Automatic Generation of SQL

Automatic generation of random SQL statements is achieved with a grammar-based technique similar to that used by Slutz (1998) as reviewed in Chapter 2 (Random Generation of SQL). The automatic generation of SQL begins with a definition of valid SQL syntax in a machine-readable form.

The syntax of SQL, even for statements with similar functionality, differs between MySQL and Oracle XE. For example, the syntax of the DROP TABLE command is shown in Table 7 for both MySQL and Oracle XE. The notation used in Table 7 is a variant of BNF as used in Chapter 12 of the MySQL 5.0 Reference Manual (2010).

DBMS	*Syntax of DROP TABLE*
MySQL	DROP [TEMPORARY] TABLE [IF EXISTS] *table* [, *table*] ... [RESTRICT | CASCADE] ;
Oracle XE	DROP TABLE *table* [CASCADE CONSTRAINTS] [PURGE] ;

Table 7: Syntax of DROP TABLE SQL statement

While the syntax of SQL statements for MySQL is documented using a human-readable notation based on BNF, the syntax of SQL statements for Oracle XE is documented using Graphic Syntax Diagrams (Oracle® Database SQL Reference, 2005).

Machine-readable BNF specifications were created by hand for a subset of the MySQL SQL syntax and for a subset of the Oracle XE SQL syntax, using a minimalist variant of BNF developed for the purpose as part of the present research which is especially simple to process, illustrated in Figure 14 taking MySQL DROP TABLE as an example. This syntax is readable by the syntax generator, a Perl program used to automatically generate random SQL statements from BNF.

sequence	DROP [s_temporary] \s TABLE [s_if_exists] \s {table_list}
	[s_restrict_cascade] ;
[s_temporary]	s_temporary \0
s_temporary	\s TEMPORARY
[s_if_exists]	s_if_exists \0
s_if_exists	\s IF \s EXISTS
[s_restrict_cascade]	s_restrict s_cascade \0
s_restrict	\s RESTRICT
s_cascade	\s CASCADE
{table_list}	table table_list
table_list	{table} [comma_table_list]
[comma_table_list]	comma_table_list \0
comma_table_list	, {table_list}
{table}	table
table	MYTABLE {digit}
{digit}	0 1 2 3 4 5 6 7 8 9

Figure 14: Specification of MySQL DROP TABLE

The syntax generator considers the BNF specification to consist of a series of 'sequences' and 'choices', each line in the file being a list of tokens that is interpreted either as a sequence or as a choice. A sequence is a token followed by a list of choice tokens or literals, and a choice is a token followed by a list of sequence tokens or literals (the first line in the specification is assumed to be a sequence). By default, a literal is a token that does not occur as the first token on any line in the specification.

Each token or literal in a sequence is processed in turn; when a choice is processed, one of the sequence tokens or literals is chosen at random. Each token encountered is looked for as the first token on a line and recursively processed until a literal is encountered, and this literal is then copied to the output. Special literals are defined for white space \s and null \0.

Examples of possible outputs produced by the syntax generator when processing the BNF specification of MySQL DROP TABLE in Figure 14 are shown below:

```
DROP TABLE IF EXISTS MYTABLE0;
DROP TABLE MYTABLE1;
DROP TEMPORARY TABLE IF EXISTS MYTABLE2;
DROP TEMPORARY TABLE MYTABLE3;
```

5.6.2 'Valid' experimental profiles for MySQL

The 'valid' experimental profile P_{2a} used for determining the response profile and time constant in the 1st experiment consists of randomly generated, syntactically valid SQL statements containing approximately equal numbers of each of CREATE, INSERT and REPLACE SQL statements. The first experimental profile P_{2a} allowed tables to increase arbitrarily in size such that continuing runs to 300,000 executed statements resulted in the message 'table is full' for many tables, and the MySQL DBMS log file reported 'disk is full'. To avoid this, the DELETE statement was added to the profile to produce a second experimental profile P_{2b} and this profile was used for the F-measure experiments. The 'valid' experimental profile P_{2b} for MySQL is shown in Table 8.

SQL Statement	*Number*	*% of Total*
CREATE INDEX	138	13.8
CREATE TABLE	48	4.8
CREATE TEMPORARY TABLE	64	6.4
DELETE	256	25.6
INSERT	254	25.4
REPLACE	240	24.0
Total	**1000**	**100**

Table 8: 'Valid' experimental profile for MySQL

Note that the SELECT statement is implicitly present in the experimental profile in the syntax of CREATE TABLE, as given in the MySQL 5.0 Reference Manual (2010).

> CREATE [TEMPORARY] TABLE [IF NOT EXISTS] *tbl_name* [(*create_definition*,...)] [*table_options*] *select_statement* ;

The 'valid' experimental profile for MySQL can be generated by running the Perl program /mysql/programs/hgen.pl optionally supplying as parameters the number of expansions to be performed and a random number seed value. The 'valid' experimental profile is generated automatically when the batch file /mysql/programs/krun.bat is run.

5.6.3 'Valid' Experimental Profile for Oracle XE

The 'valid' experimental profile P_2 used for determining the response profile and time constant in the 2nd experiment consists of randomly generated, syntactically valid SQL statements containing CREATE, DELETE and INSERT SQL statements as shown in Table 9.

SQL Statement	*Number*	*% of Total*
CREATE TABLE	38	2.52
CREATE GLOBAL TEMPORARY TABLE	32	2.12
DELETE	725	48.05
INSERT	714	47.32
Total	**1509**	**100**

Table 9: 'Valid' experimental profile for Oracle XE

Note that the SELECT statement is implicitly present in the experimental profile in the syntax of CREATE TABLE (refer to the Oracle® Database SQL Reference, 2005).

The Oracle XE experimental profile has a higher proportion of INSERT statements to counterbalance the REPLACE statements that appear in the MySQL experimental profile.

According to Murray (2008) "*The REPLACE statement in MySQL is a dual-purpose statement; it works like the INSERT statement when there is no record in the table that has the same value as the new record for a primary key or a unique index, and otherwise it works like the UPDATE statement. Oracle does not have any built-in SQL statement equivalent to the MySQL REPLACE statement. To convert this statement to Oracle, an emulated function using both the INSERT and UPDATE statements would have to be created: an attempt would first be made to insert data into the table using the INSERT statement and if this failed, the data in the table would then be updated using the UPDATE statement*".

The 'valid' experimental profile for Oracle XE can be generated by running the Perl program /oracle/programs/oracle_gen.pl optionally supplying as parameters the number of expansions to be performed and a random number seed value. The BNF specification for Oracle XE was written in such a way that expansion need not terminate after a finite number of expansions, leaving BNF tokens in some output lines; these lines can be filtered out using the UNIX grep command before executing the SQL as follows:

```
grep -v { ../data/x.txt >../data/tmp.txt
```

For the example shown in Table 9, 3000 generated lines of output were filtered to produce 1509 remaining valid SQL statements. The 'valid' experimental profile is generated automatically when the shell scripts /oracle/programs/run*.sh are run.

5.7 SQL Mutation

SQL mutation was used to generate the 'invalid' experimental profile P_1 used in the 1st experiment. This section describes the approaches to SQL mutation taken in the present

research and the 'invalid' experimental profile used in the 1st experiment, and the syntax parser program used to separate valid SQL from invalid SQL following mutation.

5.7.1 Approaches to SQL Mutation

Two different approaches to SQL mutation were investigated in the course of the present research: (a) mutation of the BNF specification before generation of SQL and (b) mutation of the resulting SQL statements after generation of valid SQL. With either approach, the syntax parser program is used to separate valid SQL from invalid SQL following mutation.

In approach (a) the Perl program /mysql/programs/bmut.pl first separates each BNF statement into tokens and then randomly applies one of the following mutation rules to produce a new BNF statement:

B1. Substitute a fixed value for a token.
B2. Substitute the null string for a token.
B3. Insert an extra fixed value token before a token.

The resulting (invalid) BNF is then used to generate SQL statements, which may happen to be either valid or invalid, but are expected to be more likely to be invalid. In approach (b) the Perl program /mysql/programs/smut.pl first separates each SQL statement into elements and then randomly applies one of the following mutation rules to produce a new SQL statement, which may happen to be either valid or invalid, but is expected to be more likely to be invalid:

M1. Substitute an invalid value for an element.
M2. Substitute an element with another defined element.
M3. Miss out an element.
M4. Add an extra element.
M6. Swap two adjacent elements
M9. Duplicate (repeat) an element

Rules M1 to M4 are syntax mutation rules given in BS 7925-2 (2001), and rules M6 and M9 are additional mutation rules, as discussed in Chapter 8 (Syntax Mutation Extension) 'Example mutation operators'.

5.7.2 'Invalid' Experimental Profile for MySQL

The 'invalid' experimental profile P_1 for MySQL is shown in Table 10 for (a) BNF mutation and (b) syntax mutation. The results of executing both invalid profiles are shown in Table 13: Results for 'invalid' experimental profile, in Chapter 6.

SQL Statement	*(a) Number*	*(a) % of Total*	*(b) Number*	*(b) % of Total*
CREATE	16,164	17.1	12,267	14.0
DELETE	12,037	12.7	19,960	22.7
INSERT	16,616	17.6	19,194	21.8
REPLACE	16,605	17.6	25,156	28.6
Other	33,093	35.0	11,372	12.9
Total	**94,515**	**100**	**87,949**	**100**

Table 10: 'Invalid' experimental profile for MySQL

5.7.3 Syntax Parser

The syntax generator program /mysql/programs/hgen.pl for random generation of SQL, described earlier in this chapter, was adapted as a syntax parser to be used for separating valid SQL statements from invalid SQL statements after mutation. The Perl program /mysql/programs/hparse.pl compares each field in the SQL statement in turn (separated by white space) to literals produced by expanding the BNF specification, until either a match is found or the BNF specification is exhausted. If all fields in the SQL statement are matched, the statement is valid, or if not, the statement is invalid. Valid and invalid statements are copied to separate output files.

5.8 DBMS Initialisation

This section describes the DBMS initialisation procedures to be followed in the 1st and 2nd experiments (for MySQL and Oracle XE). These procedures are intended to ensure that the DBMS is always returned to a known initial state prior to each test run. The initialisation procedures clear the contents of log files and empty the database by dropping any existing tables.

5.8.1 MySQL

1. Following a MySQL crash restart MySQL by entering 'net start mysql5' at the command prompt (this step is not necessary if the PC has been restarted).

2. Verify MySQL is running.

3. Initialise log files by running the Perl program /mysql/programs/init.pl
   This clears the contents of the following files:
   /mysql/data/acc.txt
   /mysql/data/err.txt
   /mysql/data/log.txt
   /mysql/data/suc.txt
   /mysql/data/summary.txt

4. Reset the 'TEST' database by executing the following SQL commands as found in /mysql/data/reset.txt
   DROP DATABASE TEST;
   CREATE DATABASE TEST;
   USE TEST;
5. Check the contents of /mysql/data/log.txt to be sure step 4 was successful.

Steps (3) and (4) are also performed by the batch files /mysql/programs/reset.bat and /mysql/programs/krun.bat.

5.8.2 Oracle XE

1. The PC must be restarted following a crash or hang of Oracle XE.

2. Start Oracle XE by running the following commands:
   . /usr/lib/oracle/xe/app/oracle/product/10.2.0/server/bin/oracle_env.sh
   /etc/init.d/oracle-xe start

3. Verify Oracle XE is running.

4. Initialise log files by running /oracle/programs/reset
   This clears the contents of the following files:
   /oracle/data/acc.txt
   /oracle/data/err.txt
   /oracle/data/log.txt
   /oracle/data/suc.txt
   /oracle/data/summary.txt

5. Reset the 'HR' data base by running /oracle/programs/cleanup.sh
   This executes a DROP TABLE command for each test table (do not drop the database as test tables are created in the 'HR' sample database schema).

6. Check the contents of /mysql/data/log.txt to be sure step 5 was successful.

Steps (4) and (5) are also performed by the shell scripts /oracle/programs/run*.sh

5.9 Database Creation

This section describes the database creation approach used in the experiments and the exponential saturation model of database growth, and presents a justification of the chosen approach.

5.9.1 Database Creation Approach

Following DBMS initialisation, a random database is created; there is no predefined database. Starting from an empty database, database tables are created and populated at random by executing randomly generated SQL statements. This approach to database creation is in contrast to other approaches to DBMS testing that have executed randomly generated SQL queries against a predefined database, for example Slutz (1998), or else have populated a predefined database schema with random data, for example Chays, et al. (2004).

5.9.2 Exponential Saturation Model

The randomly generated database is expected to grow in accordance with an exponential saturation model. Because the probability of success of a random SQL statement should be proportional to the number of database tables, the rate of growth of the database should also be proportional to the number of database tables, up to a maximum number of tables set by the experimental profile.

The exponential saturation model of random database growth is described by equation (1) where y is the number of success responses from the DBMS and x is the number of SQL statements executed. The constants k, A and C are determined by the experimental profile.

$$y(x) = A(1-e^{-kx}) + C = A + C - A(e^{-kx}) \quad (1)$$

The factor $k = 1/T$ for time constant T and the intercept on the y-axis is given by $y = C$.

The function $y(x)$ is shown plotted with arbitrary values of constants and without scale in Figure 15. The diagram was generated using a computer program written by the author.

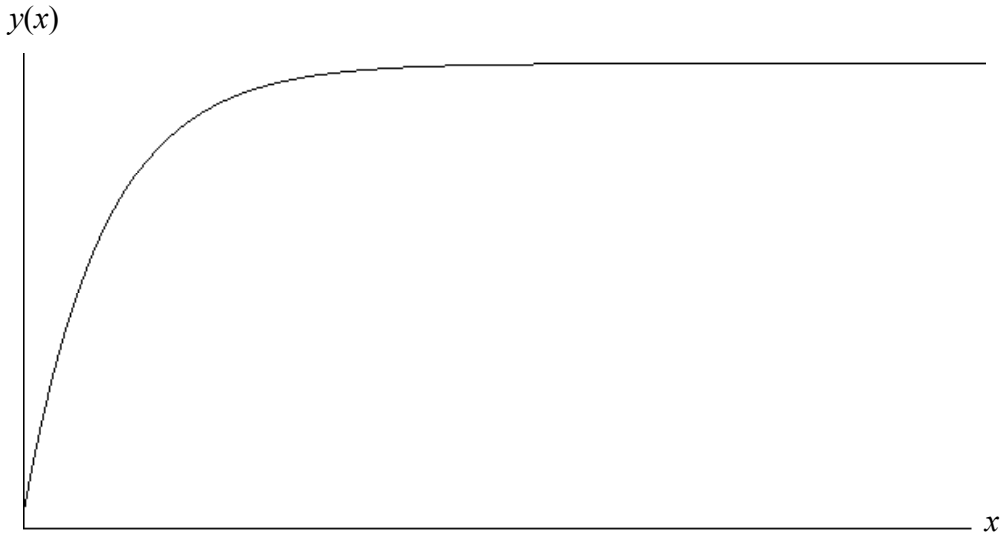


Figure 15: Plot of exponential saturation model

5.9.3 Justification of the Approach

The chosen approach for database creation can be justified, firstly on the basis of avoiding the 'pesticide paradox' described by Beizer (1990) since each test run uses a different database; secondly, testing with a predefined database, presumably intended to reflect the expected actual usage of the DBMS, suffers from the same weaknesses as testing from an operational profile. Database operations that are not expected to occur are unlikely to be tested using a predefined database, and if the DBMS usage profile changes in the future, or if the usage profile chosen for testing is not correct, then untested DBMS functionality may be exposed, leading to activation of dormant faults. The approach may also be justified from a practical viewpoint, on the basis of the simplicity, low cost and convenience, of avoiding the effort of creating a test database.

5.10 Method for Identifying Minimal Test Sequences

This section describes the 'reduction' procedure for identifying minimal test sequences from long test traces (see 'identifying failing input requests' in Chapter 2).

5.10.1 Method for MySQL

For MySQL, the Perl program used to execute SQL statements /mysql/programs/xx.pl copies SQL statements that report 'succeeded' to the file /oracle/data/suc.txt, therefore it is only necessary to copy this file to /oracle/data/x.txt to produce a minimal set of SQL statements to be executed. If executing this file reproduces the failure, the process can be repeated until a minimal test sequence has been found.

5.10.2 Method for Oracle XE

An identical procedure is followed for Oracle XE with the Perl program /oracle/programs/ox.pl and data files /oracle/data/suc.txt and /oracle/data/x.txt.

5.11 1st Experiment - MySQL (Invalid and Valid SQL)

The 1st experiment is performed with MySQL using both invalid and valid SQL. The experimental procedure for part I and part II is performed using invalid SQL while the experimental procedures for F-measure, response profile, and time constant, are performed using valid SQL.

This section first introduces the general procedure used in the 1st experiment and the procedure for generating the 'invalid' experimental profile, and then describes the experimental procedure part I, the experimental procedure part II, possible outcomes of the experiment, and the procedures for F-measure, failure delay, response profile, and time constant.

5.11.1 General Procedure

The experimental procedure for the 1st experiment with invalid SQL consists of two parts. In the first part of the procedure, as shown in Figure 16, the SUT is initialised and a sequence (or 'block') of randomly generated invalid SQL statements is executed, followed by a further blocks of randomly generated valid SQL statements, until the SUT fails. It is assumed that the SUT does fail during the first part of the procedure; otherwise the procedure must be repeated with a different initial seed value until failure occurs.

In the second part of the procedure, as shown in Figure 17, the SUT is re-initialised and only the blocks of valid SQL statements are executed. If a failure of the SUT occurs while executing valid SQL statements in part I of the procedure, but no failure of the SUT occurs while executing valid SQL statements in part II of the procedure, then a delayed failure has been demonstrated. The delayed failure is caused by the SUT accepting a 'precondition' invalid SQL statement followed later by a 'trigger' valid SQL statement.

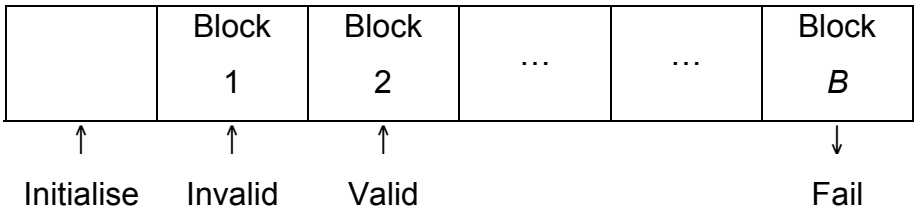


Figure 16: Procedure for 1st experiment part I

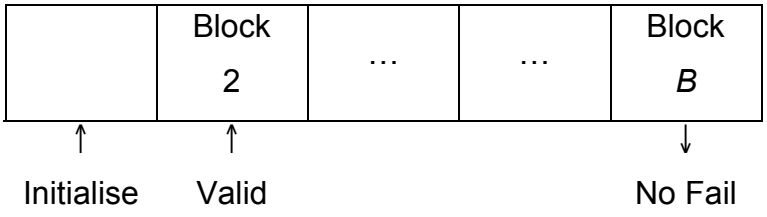


Figure 17: Procedure for 1st experiment part II

The failure delay is then ($B-1$) blocks or the equivalent number of executed SQL statements. The experiment must be repeated several times to allow for random variation. The median value of failure delay is obtained for several runs of the experiment using different initial seed values for generating random blocks of invalid and valid values, and a 95% confidence interval is calculated.

5.11.2 Procedure for 'Invalid' Experimental Profile

The 'invalid' experimental profile P_I for MySQL can be generated using either of the following procedures, corresponding to the two different approaches to SQL mutation.

Procedure (a) for BNF Mutation

Run the following Perl programs:

1. /mysql/programs/bmut.pl BNF mutation
2. /mysql/programs/hgen.pl Generate SQL from BNF
3. /mysql/programs/hparse.pl Parse syntax against BNF

Procedure (b) for Syntax Mutation

Run the following Perl programs:

1. /mysql/programs/hgen.pl Generate SQL from BNF
2. /mysql/programs/smut.pl Syntax mutation
3. /mysql/programs/hparse.pl Parse syntax against BNF

5.11.3 1st Experiment Part I

In the following experimental procedure, a subscripted lower case letter such as s_0 denotes a specific state of the software component, and a subscripted bold upper case

letter such as $\mathbf{S}_1$ denotes a specific sequence of SQL statements. The first part of the experiment consists of four steps, as follows:

Step (1) Clear log files. Initialise the SUT to the start-up state s_0. This step ensures that the experimental procedure is repeatable, as the SUT is always started from the same state. It is assumed that the SUT does not fail immediately on start-up otherwise the experiment is inconclusive. For a DBMS such as MySQL the start-up state s_0 corresponds to an empty database.

Step (2) Execute a randomly generated sequence of valid SQL statements $\mathbf{S}_1$. It is assumed that this causes a sequence of state transitions, leaving the software in an initial state s_1. It is assumed that the software does not fail while executing the sequence of valid SQL statements $\mathbf{S}_1$ otherwise the experiment is inconclusive.

Step (3) Execute a randomly generated sequence of invalid SQL statements $\mathbf{S}_2$ such that at least one of these invalid SQL statements is accepted by the SUT. It is assumed that this causes one or more state transitions leaving the software in a state s_2. It is assumed that the software does not fail while executing the sequence of invalid SQL statements $\mathbf{S}_2$ otherwise the experiment is inconclusive.

Step (4) Execute a randomly generated sequence of valid SQL statements $\mathbf{S}_3$. It is assumed that this causes a sequence of state transitions leaving the software in a state s_3. It is assumed that the software *fails* during this step and so the state s_3 is a failed state, otherwise the experiment is inconclusive. A state diagram for the first part of the procedure is shown in Figure 18.

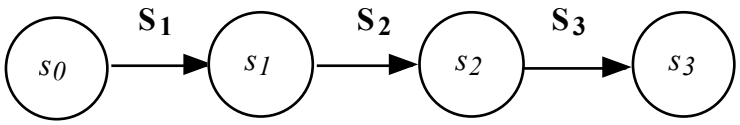


Figure 18: State diagram for 1st experiment part I

5.11.4 1st Experiment Part II

The second part of the experiment consists of three steps, as follows:

Step (5) Re-initialise the SUT to the start-up state s_0.

Step (6) Execute the same sequence of valid SQL statements $\mathbf{S}_1$ as in step (2). This is randomly generated using the same seed value. If the software is deterministic then the SUT will be in the same initial state s_1 as in step (2).

Step (7) Execute the same sequence of valid SQL statements $\mathbf{S}_3$ as in step (2). This is randomly generated using the same seed value. It is assumed that this causes a sequence of state transitions leaving the software in a state s_4. It is assumed that the software does not fail at this step otherwise the experiment is inconclusive. A state diagram for the second part of the procedure is shown in Figure 19.

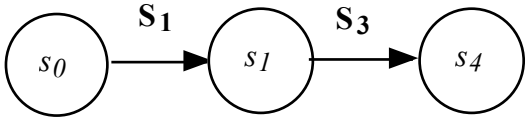


Figure 19: State diagram for 1st experiment part II

5.11.5 Possible Outcomes of the Experiment

The variables in the experiment are the start-up state s_0 (control variable), the sequences $\mathbf{S}_1$, $\mathbf{S}_2$ and $\mathbf{S}_3$ (independent variables) and the states s_1, s_2, s_3 and s_4 (dependent variables).

The state diagram for the complete experimental procedure is shown in Figure 20.

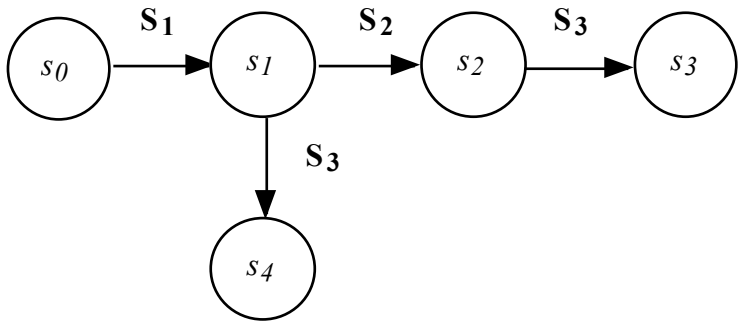


Figure 20: Complete state diagram for 1st experiment

In the first part of the experiment the state s_3 is a failed state. There are two possible outcomes of the second part of the experiment; either the state s_4 is a failed state, or the state s_4 is not a failed state.

If the state s_4 is not a failed state then the failure of the SUT in the first part of the experiment is a consequence of executing the sequence of invalid SQL statements $\mathbf{S}_2$. The state s_2 cannot be the same state as s_1 (assuming the SUT is deterministic) and therefore a transition has occurred between the state s_1 and the state s_2 due to the SUT accepting one or more invalid SQL statements in $\mathbf{S}_2$.

If the state s_4 is a failed state then the failure in the first part of the experiment is a consequence of executing the sequence of valid SQL statements $\mathbf{S}_3$. In this case, no

inference can be drawn regarding the effect of executing the sequence of invalid SQL statements $\mathbf{S}_2$ and the experiment is inconclusive.

All the possible outcomes of the experiment are summarised in Table 11. 'Pass' in a step column indicates that the SUT does not fail. Each of the possible cases leads to an inconclusive outcome except for case (f) and in this case a 'positive outcome' tends to indicate rejection of the null hypothesis for hypothesis H_1.

	Step	*(1)*	*(2)*	*(3)*	*(4)*	*(5)*	*(6)*	*(7)*	*Conclusion*
Case	*(a)*	Fail							Inconclusive
	(b)	Pass	Fail						Inconclusive
	(c)	Pass	Pass	Fail					Inconclusive
	(d)	Pass	Pass	Pass	Pass				Inconclusive
	(e)	Pass	Pass	Pass	Fail	Pass	Pass	Fail	Inconclusive
	(f)	Pass	Pass	Pass	Fail	Pass	Pass	Pass	Positive outcome

Table 11: Possible outcomes of the 1st experiment

5.11.6 Procedure for F-Measure

The procedure for F-measure is as follows:

Step (1) Clear log files. Initialise the SUT to the start-up state s_0.

Step (2) Execute blocks of randomly generated valid SQL statements on the SUT continuing until the SUT fails at some block *B*. Each block is generated using the valid experimental profile P_{2b} with initial seed values incremented by 1 starting from 1.

Step (3) Initialise the SUT to the start-up state s_0.

Step (4) Execute block *B* alone. If this does not result in failure of the SUT then the F-measure is calculated as *B* blocks or the equivalent number of executed SQL statements. If the SUT does fail at this step then the outcome of the experimental run is inconclusive.

The experiment must be repeated several times to allow for random variation. The median value of F-measure is obtained for several runs of the experiment using different initial seed values for generating random blocks of valid values, and a 95% confidence interval is calculated.

5.11.7 Procedure for Response Profile and Time Constant

Measurement of the time constant *T* for the MySQL DBMS response profile is achieved by applying the procedure described below. Procedure steps (1) and (2) are automated using the batch file /mysql/programs/krun.bat as shown in Figure 21. The procedure consists of running up to 100 blocks of SQL statements, where each block contains a configurable number of SQL statements (by default 1000).

Step (1) Clear log files. Initialise the SUT to the start-up state s_0.

Step (2) Execute blocks of randomly generated valid SQL statements on the SUT; until a maximum block number is reached. Each block is generated using the valid experimental profile P_{2b} with initial seed values incremented by 1 starting from 1.

Step (3) Record the number of success responses from the SUT for each block and plot this number against the number of blocks executed.

```
rem created 28/11/2009
rem updated 08/04/2010

rem seed offset parameter %1
set seed=%1
if not defined seed set seed=0
rem statements parameter %2
set n=%2
if not defined n set n=1000

perl init.pl
echo %0 >> ..\data\summary.txt
echo ## Resetting database ## >> ..\data\summary.txt
copy ..\data\reset.txt ..\data\x.txt /Y
perl xx.pl
set /a s=%seed%+1
perl hgen.pl %n% %s% | echo seed = %s% >> ..\data\log.txt
perl xx.pl

set /a s=%seed%+2
perl hgen.pl %n% %s% | echo seed = %s% >> ..\data\log.txt
perl xx.pl

[Intermediate lines not shown…]

set /a s=%seed%+100
perl hgen.pl %n% %s% | echo seed = %s% >> ..\data\log.txt
perl xx.pl
```

Figure 21: Listing of batch file krun.bat

5.12 2nd Experiment - Oracle XE (Valid SQL)

The 2nd experiment is performed with Oracle XE using valid SQL.

This section introduces the procedures used in the 2nd experiment and describes the procedure for F-measure, the procedure for F-measure and failure delay, the procedure for response profile and time constant, and the calculation of time constant according to the exponential saturation model of random database growth.

5.12.1 Introduction to the Procedures

The experimental procedure for the 2nd experiment consists of three elements: a procedure for F-measure, a procedure for failure delay, and a procedure for response profile and time constant. Each element of the experimental procedure involves executing a series of blocks of randomly generated valid SQL statements on the SUT. Each block is generated using the experimental profile P_2 with each block using a different randomising seed value to ensure that the sequence of statements in each block is different; each block contains approximately the same number of statements. In each case, the SUT is initialised before execution of the first block so that the database is empty. Variations on this basic method allow measurement of F-measure, failure delay, and time constant for the SUT. The procedures for failure delay and for time constant are both derived from the basic procedure for F-measure, accordingly the procedure for F-measure is described first.

5.12.2 Procedure for F-Measure

The procedure for F-measure is documented separately here for completeness; however a combined procedure that determines both F-measure and failure delay is given in the next section. Measurement of the F-measure for the SUT may be achieved by applying the method shown in Figure 22. Procedure steps (1) and (2) are automated using the script /oracle/programs/run1.sh shown in Figure 23.

Step (1) Clear log files. Initialise the SUT to the start-up state s_0.

Step (2) Execute blocks of randomly generated valid SQL statements on the SUT continuing until the SUT fails at some block *B*. Each block is generated using the valid experimental profile P_2 with initial seed values incremented by 1 starting from 1.

Step (3) Initialise the SUT to the start-up state s_0.

Step (4) Execute block *B* alone. If this does not result in failure of the SUT then the F-measure is calculated as *B* blocks or the equivalent number of executed SQL statements. If the SUT does fail at this step then the outcome of the experimental run is inconclusive.

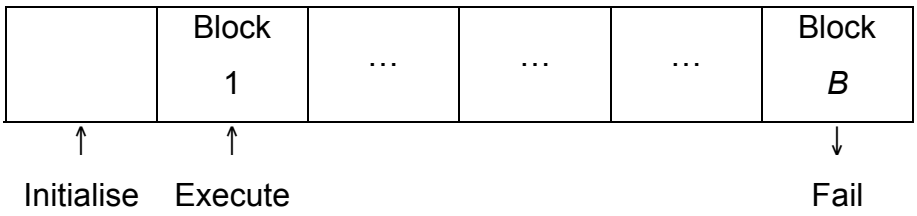


Figure 22: Method for measurement of F-measure

The experiment must be repeated several times to allow for random variation. The median value of F-measure is obtained for several runs of the experiment using different initial seed values for generating random blocks of valid values, and a 95% confidence interval is calculated.

```
# script for F-measure
cd /oracle/programs
. ./reset
cp ../data/cleanup.txt ../data/x.txt
/opt/ActivePerl-5.8/bin/perl ox.pl

s=1618

for ((i = 1; i <= 800; i++))
do
/opt/ActivePerl-5.8/bin/perl oracle_gen.pl 2000 $((s+i))
grep -v { ../data/x.txt >../data/tmp.txt
cp ../data/tmp.txt ../data/x.txt
/opt/ActivePerl-5.8/bin/perl ox.pl
done
```

Figure 23: Listing of script run1.sh

5.12.3 Procedure for F-Measure and Failure Delay

Measurement of both F-measure and failure delay for the SUT is achieved by following the method shown in Figure 24. The experimental procedure for failure delay consists of two parts.

Part I. In the first part of the procedure, the SUT is initialised and a randomly generated sequence of blocks of valid SQL statements is executed; it is assumed that the SUT will fail, otherwise the procedure must be repeated using a different initial seed value.

Part II. In the second part of the procedure, only the ‘precondition’ and ‘trigger’ blocks from the sequence of blocks of valid SQL statements are executed.

If a failure of the SUT occurs in Part I of the procedure, and a failure of the SUT occurs in Part II of the procedure, then a delayed failure has been demonstrated. The delayed failure is caused by the SUT accepting a 'precondition' SQL statement followed later by a 'trigger' SQL statement.

Procedure Part I

Procedure steps (1) and (2) below are the same as procedure steps (1) and (2) as given for the determination of F-measure, and are automated using the script /oracle/programs/run1.sh shown in Figure 23.

Step (1) Clear log files. Initialise the SUT to the start-up state s_0.

Step (2) Execute blocks of randomly generated valid SQL statements on the SUT continuing until the SUT fails at some block *B*. Each block is generated using the valid experimental profile P_2 with initial seed values incremented by 1 starting from 1.

Procedure Part II

Procedure step (3) is automated using the script /oracle/programs/run2.sh shown in Figure 25.

Step (3) The SUT is re-initialised as in step (1) and block 1 is executed followed by block *B* only as shown in Figure 24 (b). If there is no failure then the procedure is repeated, this time executing block 2 followed by block *B* only as shown in Figure 24 (c) (and so on).

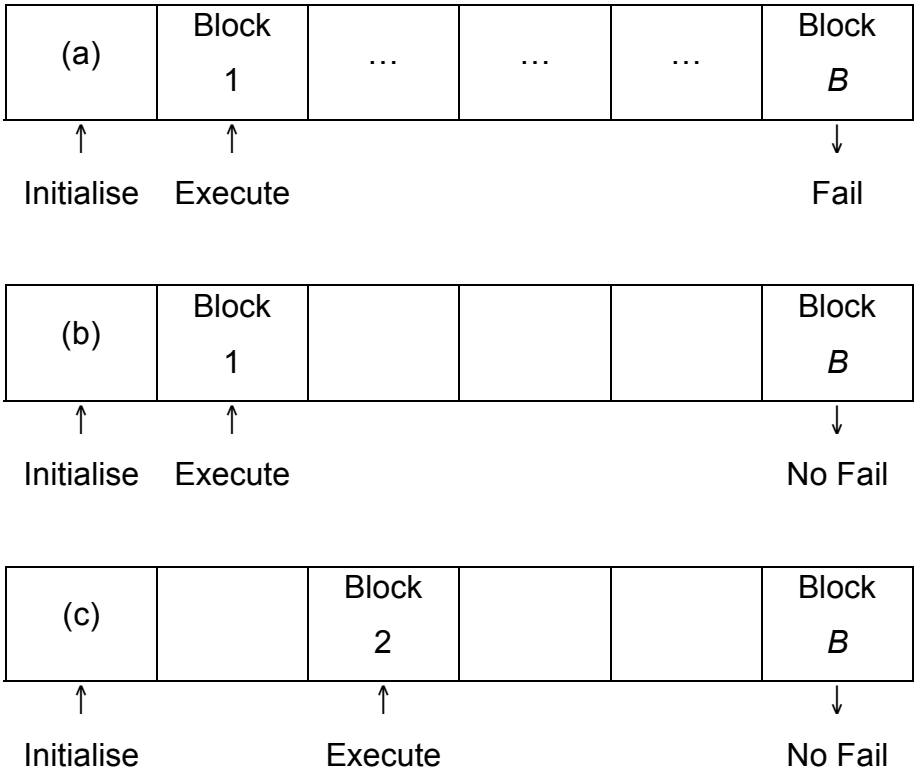
Step (4) When a failure in block *B* eventually occurs, say following execution of block *C* as shown in Figure 24 (d) then the failure delay is given by ($B - C + 1$) blocks or the equivalent number of executed SQL statements.

Step (5) Initialise the SUT to the start-up state s_0.

Step (6) Execute block *B* alone. If this does not result in failure of the SUT then the F-measure is calculated as *B* blocks or the equivalent number of executed SQL statements. If the SUT does fail at this step then the outcome of the experimental run is inconclusive.

The experiment must be repeated several times to allow for random variation. The median values of F-measure and failure delay are obtained for several runs of the experiment using different initial seed values for generating random blocks of valid values, and 95% confidence intervals are calculated.

The experiment is repeated for different numbers of database tables 10, 100, 500, 1000 to account for dependence of F-measure and failure delay on the number of tables.



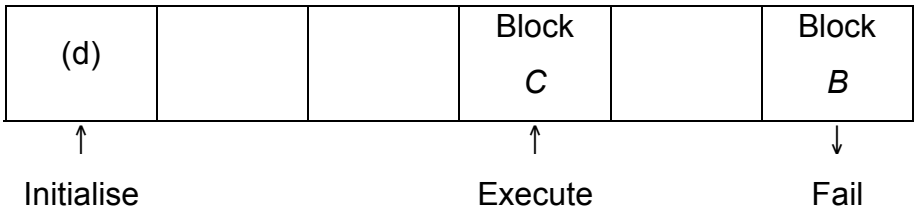


Figure 24: Method for F-measure and failure delay

```
#script for failure delay
cd /oracle/programs
r=1717
s=1722
t=s-r-1

for ((i = 1; i <= t; i++))
do
. ./reset
cp ../data/cleanup.txt ../data/x.txt
/opt/ActivePerl-5.8/bin/perl ox.pl

/opt/ActivePerl-5.8/bin/perl oracle_gen.pl 2000 $((r+i))
grep -v { ../data/x.txt >../data/tmp.txt
cp ../data/tmp.txt ../data/x.txt
/opt/ActivePerl-5.8/bin/perl ox.pl

/opt/ActivePerl-5.8/bin/perl oracle_gen.pl 2000 $((s))
grep -v { ../data/x.txt >../data/tmp.txt
cp ../data/tmp.txt ../data/x.txt
/opt/ActivePerl-5.8/bin/perl ox.pl
done
```

Figure 25: Listing of script run2.sh

5.12.4 Procedure for Response Profile and Time Constant

Measurement of the time constant *T* for the Oracle XE DBMS response profile is achieved by applying the method as shown in Figure 26. Procedure steps (1) and (2) are automated using the script /oracle/programs/run3.sh shown in Figure 27. This is almost identical to /oracle/programs/run1.sh but with the addition of a case statement.

Step (1) Clear log files. Initialise the SUT to the start-up state s_0.

Step (2) Execute blocks of randomly generated valid SQL statements on the SUT until the SUT fails or a maximum block number is reached. Each block is generated using the valid experimental profile P_2 with initial seed values incremented by 1 starting from 1.

Step (3) Record the number of success responses from the SUT for each block and plot this number against the number of blocks executed.

It is desired to achieve a run of *N* failure-free blocks; if a failure (crash or hang) of the SUT occurs during execution of any block in the run, say block number *B* as shown in Figure 26 (a) then the SUT is re-initialised and the run is repeated, but this time omitting execution of the failing block. This is achieved by editing the case statement in script /oracle/programs/run3.sh. After several repetitions of this procedure, execution of a run of *N* blocks without failure is achieved as shown in Figure 26 (b). The number of blocks *N* is chosen to be about 5*T*. This corresponds to approximately 99% of the maximum number of success responses, assuming an exponential saturation model of random database growth as described previously in this chapter by equation (1).

The experiment is repeated for different numbers of database tables 10, 100, 500, 1000 to account for dependence of time constant on the number of tables.

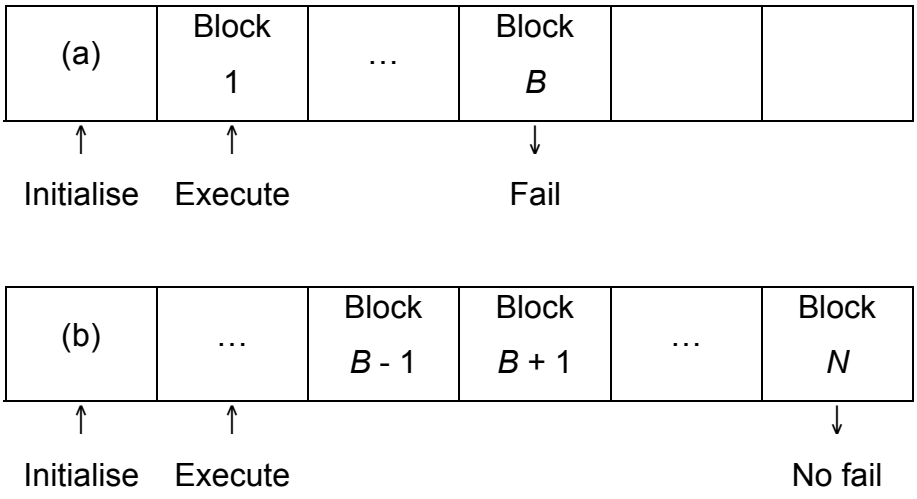


Figure 26: Method for measurement of time constant

```
# script for time constant
cd /oracle/programs
. ./reset
cp ../data/cleanup.txt ../data/x.txt
/opt/ActivePerl-5.8/bin/perl ox.pl

s=1618

for ((i = 1; i <= 800; i++))
do
case "$((s+i))" in 226|488|543) continue ;;
esac
/opt/ActivePerl-5.8/bin/perl oracle_gen.pl 2000 $((s+i))
grep -v { ../data/x.txt >../data/tmp.txt
cp ../data/tmp.txt ../data/x.txt
/opt/ActivePerl-5.8/bin/perl ox.pl
done
```

Figure 27: Listing of script run3.sh

5.12.5 Calculation of Time Constant

The time constant T for both the MySQL and Oracle XE DBMS response profiles is calculated in the same way. The exponential saturation model of random database growth was described earlier in this chapter by $y(x)$ in equation (1) where y is the number of success responses from the SUT and x is the number of blocks of SQL statements executed.

The constants A and C in equation (1) determine the maximum number of success responses, which is equal to $A + C$. The value of $A + C$ is estimated in the experiments as the median of the number of success responses for approximately the last ten blocks. Because the response profile is obtained up to approximately $x = 5T$ at which point the number of success responses $y(x)$ achieves approximately 99% of the maximum value, a margin of 1% is added when estimating the value of $A + C$. Plotting the natural logarithm $log(A + C - y)$ against x should produce a straight line with slope $-k$ where $k = 1/T$ and intercept $log(A)$. As discussed in section 7.4 linear regression analysis of the logarithm of $(A + C - y)$ against x is expected to be accurate provided random variation in the number of success responses $y(x)$ is sufficiently small.

5.13 Methodological Challenges

This section presents methodological challenges associated with BNF specification, separation of valid and invalid values, and reproducibility of failures.

5.13.1 BNF Specification

There are differences in BNF syntax between MySQL and Oracle, and the SQL syntax specifications are not easily machine-readable; therefore machine-readable BNF specifications are created by hand, introducing the possibility of errors in transcribing the specifications. To guard against this possibility, automatically generated SQL is executed against the DBMS and the results are verified against the documented SQL specifications.

Discrepancies may be due to errors in the BNF specifications, that must be corrected, or possibly these may reveal faults in the SUT, or faults in the SQL generator program.

5.13.2 Separation of Valid and Invalid Values

Mutated SQL statements may happen to be either valid or invalid, so mutation produces a mix of valid and invalid SQL statements; these are separated after mutation using the syntax parser program. The syntax parser uses the same BNF specification as the SQL generator program, and so correctness of parsing is subject to possible errors in the BNF specification (as well as possibly faults in the syntax parser program). To guard against this possibility, SQL statements are executed against the DBMS after separation by the parser program, with the expectation that the DBMS should accept valid SQL statements and reject invalid statements; the results are verified against the documented SQL specifications. Again, discrepancies may be due to errors in the BNF specification, or these may reveal faults in the SUT, or in the syntax parser program.

5.13.3 Reproducibility of Failures

Following a crash of MySQL, the PC may be restarted to ‘cold start’ MySQL. Alternatively, a ‘warm start’ may be performed using the command ‘net start mysql5’ and this avoids the delay of restarting the PC. However, during pilot experiments it was found that failures of MySQL following a ‘cold start’ may not be reproducible following a ‘warm start’ which suggests that the start-up state of MySQL may be different in these two cases. A consistent restart procedure must be followed and in the present research a ‘warm start’ of MySQL was always performed following a crash, except at the start of an experimental run when a ‘cold start’ was performed. In the case of Oracle XE, it has been found that this usually hangs the machine on failure, and so the PC is always rebooted, after which a ‘cold start’ of Oracle XE is performed.

5.14 Experimental Questions

A number of experimental questions arise once the details of the experimental methodology have been decided. This section describes experimental questions related to measures of F-measure, failure delay, and time constant.

5.14.1 F-Measure

The following experimental questions are of interest related to F-measure:

a) What is the F-measure for each SUT for each experimental profile?
b) How does the F-measure vary with the number of database tables?
c) How does the F-measure for each SUT change with different profiles?
d) Does the F-measure differ between SUT for the same experimental profile?

5.14.2 Failure Delay

The following experimental questions are of interest related to failure delay:

a) What is the failure delay for each SUT for each experimental profile?
b) How does the magnitude of the failure delay compare with the F-measure?
c) How does the failure delay vary with the number of database tables?
d) How does the failure delay for each SUT change with different profiles?
e) Does failure delay differ between SUT for the same experimental profile?

5.14.3 Time Constant

The following experimental questions are of interest related to time constant:

a) What is the time constant for each SUT for each experimental profile?
b) How does the time constant vary with the number of database tables?
c) How does the time constant for each SUT change with different profiles?
d) How does the time constant compare to failure delay and F-measure?

e) Does time constant differ between SUT for the same experimental profile?

f) Does the response profile conform to the exponential saturation model?

5.15 Data Analysis

This section describes the data analysis to be performed for the experimental results; confidence intervals for the mean, robust statistical estimators, and handling of outliers.

5.15.1 Confidence Intervals for the Mean

Confidence intervals for the experimental results are used, in preference to significance testing, as recommended for example by StatSoft, Inc. (2011). For normally distributed data of sample size n with sample mean μ_s and sample standard deviation σ_s it is usual to calculate a confidence interval for the mean given by $\mu_s \pm t_c\sigma_s/\sqrt{n}$ where t_c is the critical point of the t-distribution for $n-1$ degrees of freedom, and $t_{0.05}$ is often chosen giving a 95% confidence level. However, the mean and standard deviation are very sensitive to the presence of 'outliers' in the data, which 'contaminate' the distribution.

5.15.2 Robust Statistical Estimators

Abu-Shawiesh, Al-Athari & Kittani (2009) have studied the confidence interval for the mean in the presence of outliers. 'Robust' statistical estimators that are relatively unaffected by the presence of outliers include the sample median M and the median absolute deviation about the median (MAD). MAD is defined as the median of $|X_i-M|$ for data values X_i where the index i runs from 1 to n for sample size n. A robust confidence interval for the mean is given by $M \pm 1.253t_cb_n \times 1.4826 \times MAD/\sqrt{n}$ where b_n is a correction factor for small sample size, approximately equal to $n/(n-1)$. A robust estimator for the sample standard deviation σ_s is given by $b_n \times 1.4826 \times MAD$. These estimators are normalised to give correct values for the case of normally distributed data.

5.15.3 Handling of Outliers

Data values X_i for which $|X_i - M| / MAD > 4.5$ are treated as suspected outliers. This approach is independent of the underlying distribution, however for normally distributed data, this is equivalent to the well-known '3 sigma rule' with the probability of X_i falling beyond this range approximately equal to 1/370 = 0.0027 (Maronna, Martin & Yohai, 2006). The median M is recalculated with the outliers removed from the data set for comparison.

5.16 Summary

This chapter first presented the research goals, a generic procedure for testing the hypotheses, the experimental design, experimental configurations, methods for random generation of SQL and SQL mutation, DBMS initialisation, database creation, and the method for identifying minimal test sequences. This chapter then presented the procedures to be followed in the 1st experiment - MySQL (invalid and valid SQL) and the 2nd experiment - Oracle XE (valid SQL) and finally discussed methodological challenges, experimental questions, and data analysis.

6 RESULTS

6.1 Introduction

This chapter presents results for the 1st experiment using MySQL with invalid and valid SQL, MySQL bugs found during the experiment, and results for the 2nd experiment using Oracle XE with valid SQL.

6.2 1st Experiment - MySQL (Invalid and Valid SQL)

This section first presents a summary of the data sets and results for the 1st experiment and then presents results for executing the 'invalid' experimental profiles produced by BNF mutation and syntax mutation, followed by results using the 'valid' experimental profile for F-measure, response profile and time constant, and finally results for MySQL database creation.

6.2.1 Summary of Results for the 1st Experiment

The data sets and experimental results for the 1st experiment using SQL mutation are fully described in APPENDIX C together with details of the program and data files used to automate the experimental procedure.

The following data sets were used in the 1st experiment:

1. Data set A (non-mutated)
2. Data set B (mutated)
3. Data set C (mutated)

Input values used in the experimental runs were drawn randomly from these data sets.

Data set A consisted of non-mutated (valid) SQL statements and contained nearly equal proportions of CREATE (33%) INSERT (34%) and REPLACE (33%) statements.

Data set B consisted of mutated (invalid) SQL statements and contained proportions of CREATE (28%) INSERT (26%) and REPLACE (46%) statements. Data set B was generated by Dr. Jacob Mulenga, using several machines at Cranfield Defence and Security with the same Perl programs as were used to generate the other data sets.

Data set C consisted of mutated (invalid) SQL statements and contained only CREATE statements.

For the 1st experiment there were two experimental runs. The first experimental run used non-mutated (valid) input values from data set A and mutated (invalid) input values from data set B. This run produced 12 inconclusive outcomes and two positive outcomes.

The second experimental run used non-mutated (valid) input values from data set A and mutated (invalid) input values from data set C. This run produced five inconclusive outcomes and two positive outcomes. Overall, four out of 21 experimental outcomes were positive (19%). A summary of results for the 1st experiment is shown in Table 12.

1st experiment	*Delay*	*F-measure*
First run	4195	24,195
	5338	25,338
Second run	**9265**	**29,265**
	5431	25,431
Median M_1	**5384.5**	**25,385**
MAD	**618**	**618**
σ_s	**1248.8**	**1248.8**
Lower 95%	**2894**	**22894.5**
Upper 95%	**7875**	**27875.5**

Table 12: Failure delay results for the 1st experiment

The median value M_1 for failure delay and F-measure for the positive outcomes and the corresponding 95% confidence intervals are shown. The sample standard deviation σ_s is estimated as $b_n \times 1.4826 \times MAD$ and a robust 95% confidence interval for the mean is estimated as $M_1 \pm 1.253 t_{0.05} \sigma_s / \sqrt{n}$ where number of samples $n = 4$ $b_4 = 1.363$ and $t_{0.05} = 3.182$ that is $M_1 \pm 4.03 \times MAD$.

The '3 sigma' outlier test of $M_1 \pm 4.5 \times MAD$ suggests the values of 9265 for failure delay and 29,265 for F-measure in the second run are outliers. After removing the outliers from the data set, the recalculated medians are $M_2 = 5338$ for failure delay and $M_2 = 25{,}338$ for F-measure with revised 95% confidence intervals of 2847.5 to 7828.5 and 22,847.5 to 27,828.5 respectively.

A positive outcome in the 1st experiment tends to indicate rejection of the null hypothesis H_{01} as described in Table 11 ('possible outcomes of the experiment') in Chapter 5.

6.2.2 'Invalid' Experimental Profiles

Two alternative approaches were used to generate the 'invalid' experimental profile used in the 1st experiment, as shown in Table 10: 'Invalid' experimental profile for MySQL, in Chapter 5. The 'invalid' experimental profile was generated from the 2010 version of the BNF specification /mysql/data/bnf10.txt which specifies 260 database table names.

BNF Mutation

For 100,000 generated SQL statements, procedure (a) BNF mutation produced 5485 valid SQL statements (442 CREATE and 5043 DELETE) and 94,515 invalid SQL statements after mutation. The results of executing both the valid and invalid profiles with MySQL are shown in Table 13. Note that it is possible for random BNF mutation to generate a profile containing no valid CREATE statements; however, such a profile cannot be used in the experiments, as no database tables are created.

Syntax Mutation

For 100,000 generated SQL statements, procedure (b) syntax mutation produced 12,051 valid and 87,949 invalid SQL statements after mutation. The results of executing both the valid and invalid profiles with MySQL are shown in Table 13.

Before each run of the profiles (a) and (b) shown in Table 13, 1000 SQL statements were generated using the 'valid' experimental profile with a seed value of 0 and executed with an empty database (as discussed later under the heading of 'MySQL Database Creation').

Profile	*Executed*	*Failed*	*Accepted*	*Succeeded*	*Time (sec)*
(a) Valid	5485	0	5270	215	16
(b) Valid	12,051	5	11,588	458	54
(a) Invalid	94,515	66,105	28,391	19	291
(b) Invalid	87,949	84,593	3339	17	165

Table 13: Results for 'invalid' experimental profile

6.2.3 F-Measure

The F-measure F_{valid} was measured for MySQL with a 'valid' experimental profile. Twenty runs were performed using experimental profile P_{2b} with the database initially empty before each run. Each run was performed with a different random number seed and so each used a different random set of SQL statements. The 'valid' experimental profile P_{2b} was generated from the 2009 version of the BNF specification /mysql/data/bnf09.txt which specifies 260 database table names.

The MySQL DBMS crashed in eight runs out of twenty (40%) with the results shown in Table 14. Experimental runs that did not crash are not shown in the table. Note the number of SQL statements *executed* = *failed* + *accepted* + *succeeded*.

Seed	*Executed*	*Failed*	*Accepted*	*Succeeded*	*Time (sec)*
3	12,884	1	10,939	1944	78
6	22,612	9	19,167	3436	421
10	21,625	6	18,420	3199	403
11	29,057	7	25,841	3209	238
12	45,498	13	39,578	5906	350
16	13,425	2	12,383	1220	237
17	17,352	3	15,914	1435	298
19	30,147	8	26,733	3407	320
M	**22,119**	**7**	**18,794**	**3204**	**309**
MAD	**7,484**	**3**	**6729**	**746**	**71.5**
Lower 95%	**8992**	**1**	**6991**	**1896**	**184**
Upper 95%	**35,245**	**12**	**30,596**	**4512**	**434**

Table 14: F-measure results for MySQL DBMS

The sample median M and median absolute deviation MAD are calculated for each of the response variables in Table 14. For sample size $n = 8$ $\sqrt{n} = 2.828$ $t_{0.05} = 2.365$ and $b_8 = 1.129$ so $(1.253 t_c b_n \times 1.4826)/\sqrt{n} = 1.754$ giving a 95% confidence interval for the mean of $M \pm 1.754 \times MAD$.

The '3 sigma' outlier test of $M_1 \pm 4.5 \times MAD$ suggests that none of the data values in Table 14 are outliers.

6.2.4 Response Profile and Time Constant

The response profile obtained for the MySQL DBMS with 'valid' experimental profile P_{2a} is shown in Figure 28. This is a plot of the number of success responses from the SUT against the number of blocks of SQL statements executed for a database with 260 tables. There were 1482 SQL statements in each block; this was chosen to match the

mean number of SQL statements in each block in the Oracle XE response profile experiment to allow comparison of the results. This plot shows random variation in the number of success responses from the SUT. The experimental results are tabulated in APPENDIX A.

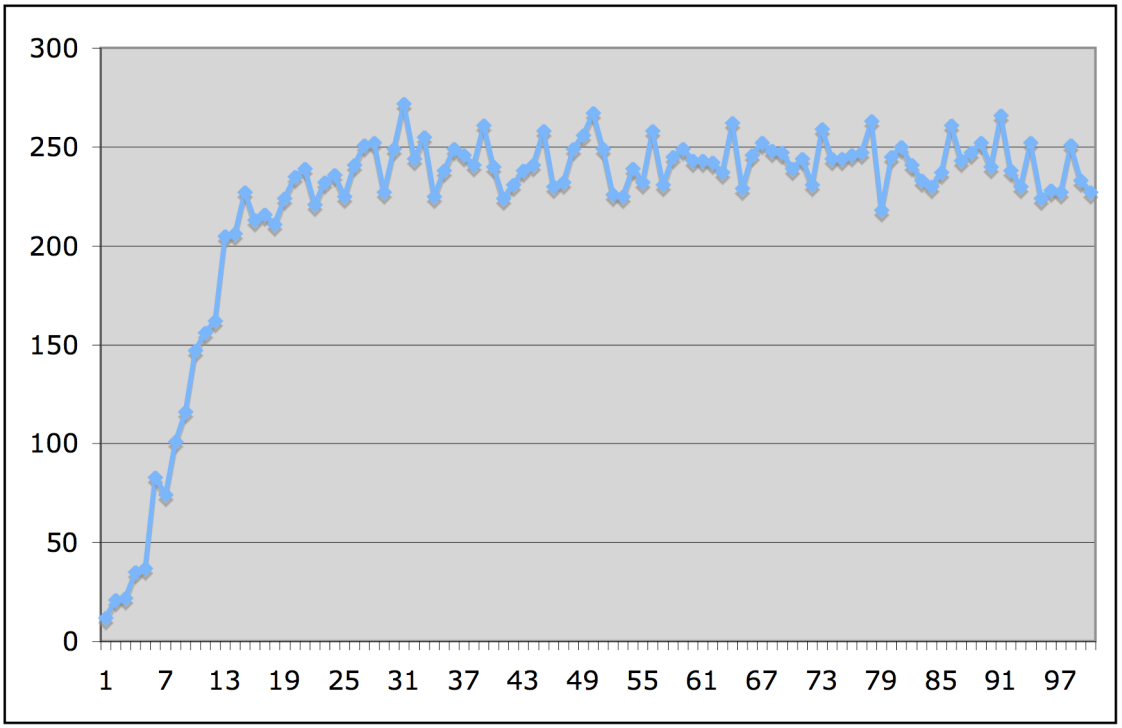


Figure 28: Response profile for MySQL DBMS

Plotting the natural logarithm $log(A + C - y)$ against x should produce a straight line with slope $-k$ and intercept $log(A)$ where $k = 1/T$ for time constant T. Linear regression coefficients and 95% confidence intervals for time constant T were calculated using Microsoft Excel data analysis as shown in Table 15. The value of $(A + C)$ was estimated from the last 10 data points + 1% margin as approximately 234.

Time constant	*Coefficients*	*Lower 95%*	*Upper 95%*
Intercept	2.63	2.44	2.824
Slope	-0.081	-0.097	-0.064
A	13.89	11.49	16.79
C	220.11	222.51	217.21
T	12.31	10.22	15.47

Table 15: Linear regression for time constant

6.2.5 MySQL Database Creation

Using the 'valid' experimental profile P_{2b} 1000 SQL statements were generated with a seed value of 0 and then executed with an empty database. Of these 1000 statements 994 were accepted and 6 succeeded as shown below, creating the six (empty) database tables described in Table 16.

```
CREATE TABLE m2 (u8 YEAR ) MAX_ROWS=36 TYPE=MRG_MYISAM ;
CREATE TABLE y0 (z7 REAL UNSIGNED ) ;
CREATE TABLE g6 (j6 SMALLINT (54) ZEROFILL NOT NULL REFERENCES
m5 (g8) ON UPDATE NO ACTION ON DELETE SET NULL ) CHARACTER SET
HEBREW CONNECTION 'k' ;
CREATE TABLE c6 (k6 REAL ,u5 VARCHAR (9) CHARACTER SET SWE7 KEY
REFERENCES c4 (d1,t1) ON UPDATE RESTRICT ) ;
CREATE TABLE IF NOT EXISTS l8 (b1 LONGTEXT ) ;
CREATE TABLE w0 (z0 SET ('jxf','i') CHARACTER SET CP1250 ) MIN_ROWS 17;
```

Table	*Field*	*Type*	*Null*	*Key*	*Default*
c6	k6	Double	YES		NULL
	u5	varchar(9)	NO	PRI	NULL
g6	j6	smallint(54) unsigned zerofill	NO		NULL
l8	b1	longtext	YES		NULL
m2	u8	year(4)	YES		NULL
w0	z0	set('jxf','i')	YES		NULL
y0	z7	double unsigned	YES		NULL

Table 16: MySQL database tables

The MySQL response for successful statements is ‘0000 SUCCESS’ and the response for failing statements is ‘1064 You have an error in your SQL syntax’ (although no statements in this run failed). Examples of MySQL responses for ‘accepted’ statements from this run are shown below:

1031 Table storage engine for 'c6' doesn't have this option
1051 Unknown table 'a0'
1054 Unknown column 'c3' in 'order clause'
1054 Unknown column 'd4' in 'field list'
1054 Unknown column 'n3' in 'where clause'
1063 Incorrect column specifier for column 's0'
1067 Invalid default value for 'g3'
1072 Key column 'a6' doesn't exist in table
1089 Incorrect sub part key; the used key part isn't a string, the used length is longer than the key part, or the storage engine doesn't support unique sub keys
1136 Column count doesn't match value count at row 1
1146 Table 'test.a0' doesn't exist
1214 The used table type doesn't support FULLTEXT indexes
1253 COLLATION 'latin1_danish_ci' is not valid for CHARACTER SET 'ujis'
1253 COLLATION 'latin1_german2_ci' is not valid for CHARACTER SET 'cp932'

1253 COLLATION 'latin1_german2_ci' is not valid for CHARACTER SET 'latin5'
1391 Key part 'm6' length cannot be 0
1425 Too big scale 319 specified for column 'q4'. Maximum is 30
1427 For float(M,D), double(M,D) or decimal(M,D), M must be >= D (column 'k6')

6.3 MySQL Bugs Reported

This section presents a summary of MySQL bugs found during pilot runs, script debugging, and experimental runs. These were reported using the MySQL online bug-tracking system at http://bugs.mysql.com/

6.3.1 MySQL Bug Report #4046

"*Using column type DECIMAL (0,11) will crash mysql*". This was found to be an existing MySQL bug, so a new bug report was not needed. When creating a table column of type DECIMAL (*M*, *D*) the maximum number of digits (precision) is *M* and the number of digits to the right of the decimal point (range) is *D*. In this case a precision of zero was syntactically valid (although in practice this will not allow any value to be stored) however when the table was later accessed MySQL crashed; this is a delayed failure because the table access may occur an arbitrary time after the CREATE TABLE.

6.3.2 MySQL Bug Report #38696

"*CREATE TABLE ... CHECK ... allows illegal syntax*". The MySQL parser accepts invalid CREATE TABLE syntax and furthermore the database operation is successful. The following is valid syntax according to the MySQL specification:
CREATE TABLE a1 (col_1 INT CHECK (*whatever*));
However the following invalid syntax succeeds:
CREATE TABLE a1 (col_1 INT CHECK *something* (*whatever*));

6.3.3 MySQL Bug Report #39144

"*CREATE TABLE - undocumented behaviour of reference_definition options*". Again, the MySQL parser accepts invalid CREATE TABLE syntax, which succeeds. The parameter *reference_definition* is defined in the MySQL specification as:
REFERENCES ... [MATCH *option*] [ON DELETE *option*] [ON UPDATE *option*]
This has three optional parameters in a given order. However, the actual MySQL behaviour corresponds to:
REFERENCES ... [*choice*] [*choice*] [*choice*] …
Here *choice* is any of MATCH *option* | ON DELETE *option* | ON UPDATE *option* and so allows any number of parameters in any order, including repetition of parameters.

6.3.4 MySQL Bug Report #45639

"*INSERT on MERGE table results in a crash*". The following SQL statements cause MySQL to crash:
CREATE TABLE u3 (e9 INT) TYPE MRG_MYISAM INSERT_METHOD LAST;
INSERT u3 SET e9=42;
Here the optional UNION clause is (perfectly legally) omitted from CREATE for a table of type MERGE and so the reference to the MERGE table is a 'null pointer' and should be read-only. However, a subsequent INSERT request on the table is accepted and this results in a crash because no underlying tables exist for the MERGE reference. This is a delayed failure, because the INSERT may occur an arbitrary time after the CREATE TABLE. The expected response to the INSERT was "*ERROR 1036 Table 'u3' is read only*".

6.4 2nd Experiment - Oracle XE (Valid SQL)

This section presents a summary and detail of results for F-measure and failure delay, and for response profile and time constant, obtained for Oracle XE with valid SQL.

The 'valid' experimental profile was generated from the /oracle/data/oracle_bnf_2.txt BNF specification.

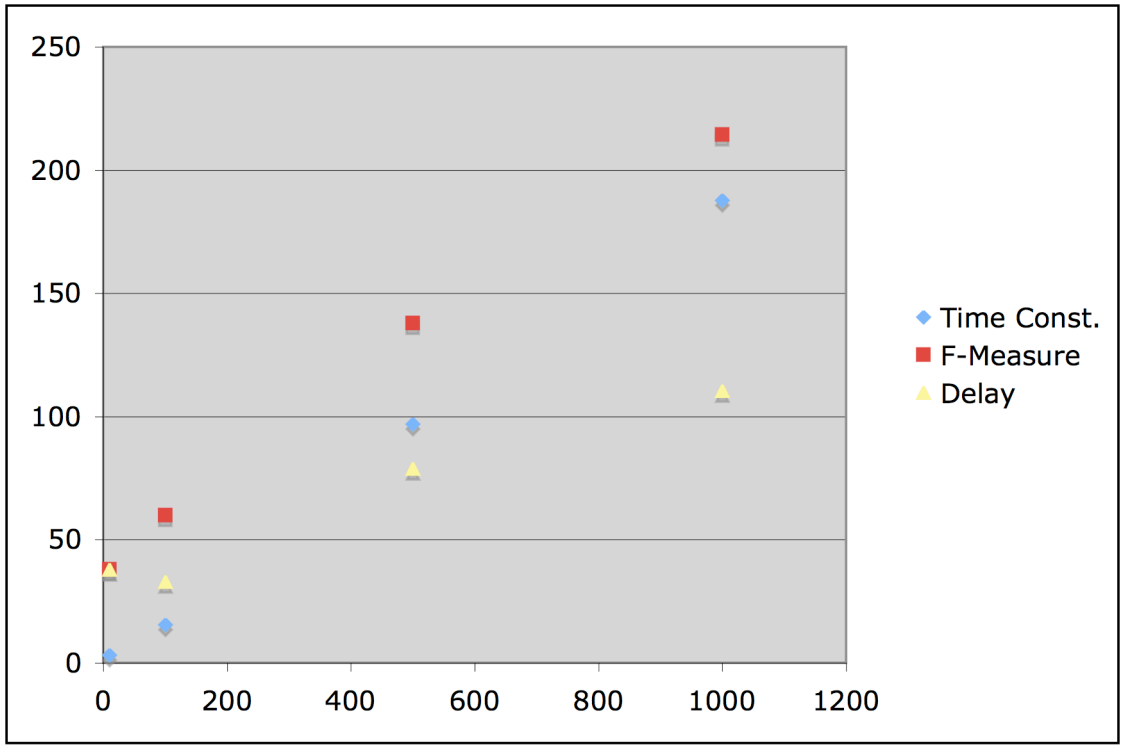
6.4.1 Summary of Results for the 2nd Experiment

A summary of the results obtained with Oracle XE for F-measure, failure delay, and time constant for different numbers of database tables is shown in Table 17, in each case as a number of blocks; there was an average of 1482 SQL statements per block. The summary of results is shown plotted on a graph in Figure 29. The graph was generated using Microsoft Excel. Detailed results for F-measure, failure delay, and time constant appear in the following sections.

Number of tables	*F-measure* F_{valid}	*Delay* D_{valid}	*Time constant T*
10	38	38	3.2
100	60	33	15.6
500	138	79	97
1000	214.5	110.5	187.9

Table 17: Summary of results for Oracle XE

Number of blocks



Number of database tables

Figure 29: Summary of results for Oracle XE

Linear regression coefficients and 95% confidence intervals for F-measure, failure delay, and time constant T were calculated using Microsoft Excel data analysis as shown in Table 18, Table 19, and Table 20.

F-measure	*Coefficients*	*Lower 95%*	*Upper 95%*
Intercept	41.24	18.52	63.95
Slope	0.17	0.13	0.21

Table 18: Linear regression coefficients for F-measure

Delay	*Coefficients*	*Lower 95%*	*Upper 95%*
Intercept	33.02	8.73	57.30
Slope	0.08	0.04	0.12

Table 19: Linear regression coefficients for failure delay

T	*Coefficients*	*Lower 95%*	*Upper 95%*
Intercept	0	N/A	N/A
Slope	0.19	0.18	0.20

Table 20: Linear regression coefficients for time constant

Here it is assumed that the intercept coefficient for time constant is zero. The variation in Delay values for Oracle XE shown in Figure 29 does not appear to be due to the presence of outliers and is probably a straight line after allowing for the large standard deviation of the experimental results.

6.4.2 F-Measure and Failure Delay

Oracle XE results for F-measure and failure delay with 10, 100, 500 and 1000 database tables are shown in Table 21, Table 22, Table 23, and Table 24. Random seed values for block execution in this experiment began at *Offset* +1.

In these tables of results S_1 is the precondition block number and S_2 is the trigger block number. The F-measure F_{valid} was calculated as $S_2 - Offset$ and failure delay D_{valid} was calculated as $S_2 - S_1 + 1$.

The sample standard deviation σ_s was estimated as 1.5 × median absolute deviation (*MAD*) calculated from the sample median M_1 and a revised value for the median M_2 was recalculated after removal of outliers.

Suspected outliers were identified as lying more than 4.5 × *MAD* from the median M_1 and suspected outliers are indicated in the tables as **bold underline**.

Robust 95% confidence intervals for the mean are calculated as $M_1 \pm 1.253 t_{0.05} \sigma_s / \sqrt{n}$ where b_n is taken to be $n/(n-1)$. The experimental differences in median values of F-measure and failure delay for different numbers of tables were not found to be significant at the 95% level.

The following notes appear in the tables of results in the S_1 column for inconclusive runs: (a) no failure occurred during the run, and (b) the SUT failed on a single re-execution of block number S_2. Case (a) occurred in 6 out of 52 runs or 12%. Case (b) occurred in 15 out of 52 runs or 29% and indicates a short latency failure or possibly an immediate failure.

Offset	S_2	S_1	$F_{valid} = S_2 - Offset$	$D_{valid} = S_2 - S_1 + 1$
1	205	(a)	**204**	-
206	623	(b)	-	-
623	647	625	24	23
647	743	648	96	96
743	781	744	38	38
781	933	(b)	-	-
933	960	934	27	27
960	1022	963	62	60
1022	1119	1024	97	96
1119	1144	1120	25	25
1144	1374	1145	**230**	**230**
1374	1400	1377	26	24
1400	1418	1401	18	18
1418	1497	1423	79	75
1497	1605	1498	108	108
1605	1676	1616	71	61
1676	1717	(b)	-	-
1717	1722	1718	5	5
1722	1780	(b)	-	-
1780	1984	(b)	-	-
		M_1	**62**	**49**
		MAD	**36**	**26**
		σ_s	**54**	**39**
		M_2	**38**	**38**
		n	**15**	**14**
		Lower 95%	**21.8**	**18.6**
		Upper 95%	**102.2**	**79.4**

Table 21: Oracle XE results for 10 database tables

Offset	S_2	S_1	$F_{valid} = S_2 - Offset$	$D_{valid} = S_2 - S_1 + 1$
0	38	(b)	-	-
38	98	(a)	60	-
98	146	(b)	-	-
146	219	(b)	-	-
219	693	(a)	**474**	-
693	919	(b)	-	-
919	951	920	32	32
951	1022	989	71	34
1022	1116	(b)	-	-
1116	1148	1142	32	7
1148	1527	(b)	-	-
1527	1618	1530	91	89
1618	1729	(b)	-	-
		M_1	**65.5**	**33**
		MAD	**29.5**	**13.5**
		σ_s	**44.25**	**20.25**
		M_2	**60**	**33**
		n	**6**	**4**
	Lower 95%		**-4.3**	**-20.8**
	Upper 95%		**135.3**	**86.8**

Table 22: Oracle XE results for 100 database tables

Offset	S_2	S_1	$F_{valid} = S_2 - Offset$	$D_{valid} = S_2 - S_1 + 1$
0	226	172	226	55
226	378	279	152	100
378	488	487	110	2
488	543	(b)	-	-
543	902	(a)	**359**	-
902	1001	917	99	85
1001	1455	(a)	**454**	-
1455	1534	1456	79	79
1534	1672	1620	138	53
1672	1837	1734	165	104
		M_1	**152**	**79**
		MAD	**53**	**24**
		σ_s	**79.5**	**36**
		M_2	**138**	**79**
		n	**9**	**7**
		Lower 95%	**65.9**	**30.3**
		Upper 95%	**238.1**	**127.7**

Table 23: Oracle XE results for 500 database tables

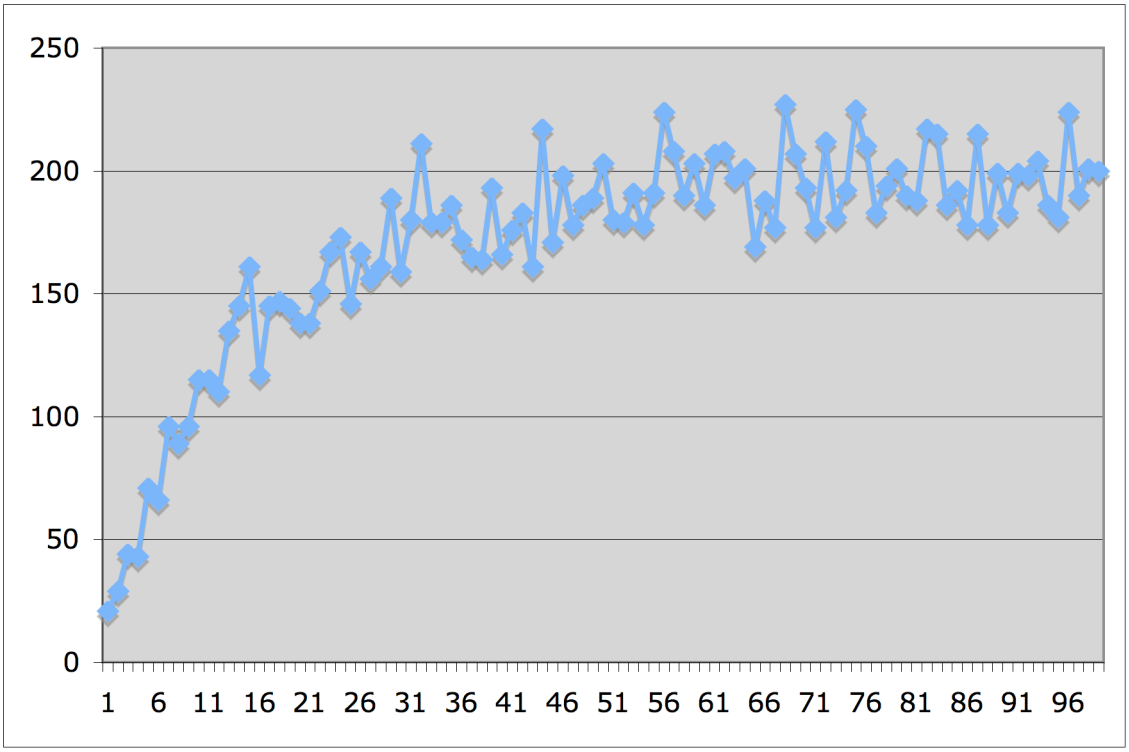
Offset	S_2	S_1	$F_{valid} = S_2 - Offset$	$D_{valid} = S_2 - S_1 + 1$
0	226	172	226	55
226	488	309	262	180
488	543	(b)	-	-
543	(a)	-	-	-
1343	1534	(b)	-	-
1534	1672	1620	138	53
1672	1837	1734	165	104
1837	2047	1885	210	163
2047	2487	2064	**440**	**424**
2487	2706	2590	219	117
		M_1	**219**	**117**
		MAD	**43**	**62**
		σ_s	**64.5**	**93**
		M_2	**214.5**	**110.5**
		n	**7**	**7**
		Lower 95%	**131.8**	**-8.7**
		Upper 95%	**306.2**	**242.7**

Table 24: Oracle XE results for 1000 database tables

6.4.3 Response Profile

The response profile obtained for the Oracle XE DBMS is shown in Figure 30. This is a plot of the number of success responses from the SUT against the number of blocks of SQL statements executed, for a database with 100 tables. There was an average of 1482 SQL statements in each block. This plot shows random variation in the number of success responses from the SUT. The experimental results are tabulated in APPENDIX A.

Number of success responses



Number of blocks

Figure 30: Response profile for Oracle XE DBMS

For the Oracle XE experiments *executed* = *accepted* + *succeeded*, as the test harness program does not detect *failed* responses. The results for seed value 38 are excluded as this caused the DBMS to hang/freeze; an initial pool of 2900 SQL statements was generated for each run producing a mean of 1482 final statements per run.

Taking a moving average of the data set can smooth out the random variation in the number of success responses, as shown in Figure 31. Here a window of size 3 was chosen for the moving average; this was calculated and plotted using Microsoft Excel. This plot is for the same data set as shown in Figure 30.

The plot in Figure 31 suggests the response profile is of the form given by the exponential saturation model of random database growth as described by equation (1).

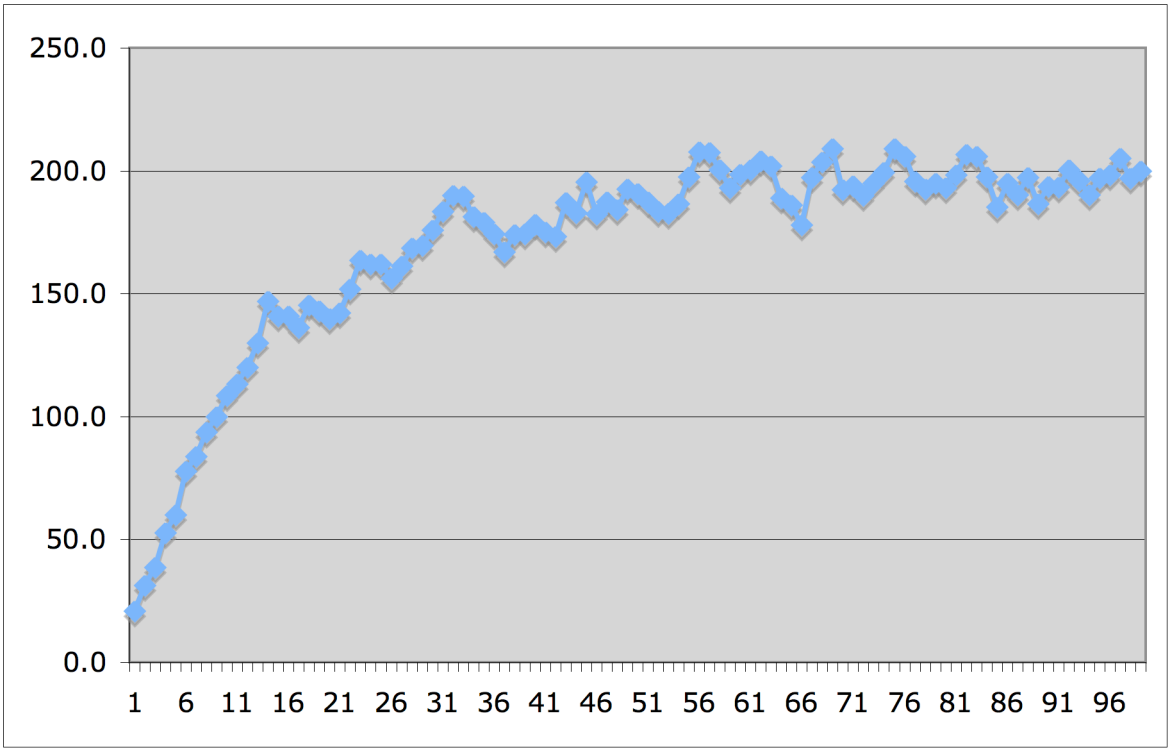


Figure 31: Moving average of Oracle XE response profile

6.4.4 Time Constant

Plotting the natural logarithm $log(A + C - y)$ against x as shown in Figure 32 should produce a straight line with slope $-k$ and intercept $log(A)$ where $k = 1/T$ for time constant T. An initial estimate for $(A + C)$ is found by taking the median of the last 10 data points plus a margin of 1% calculated in this case to be 200.5 as can be confirmed by inspection of Figure 31.

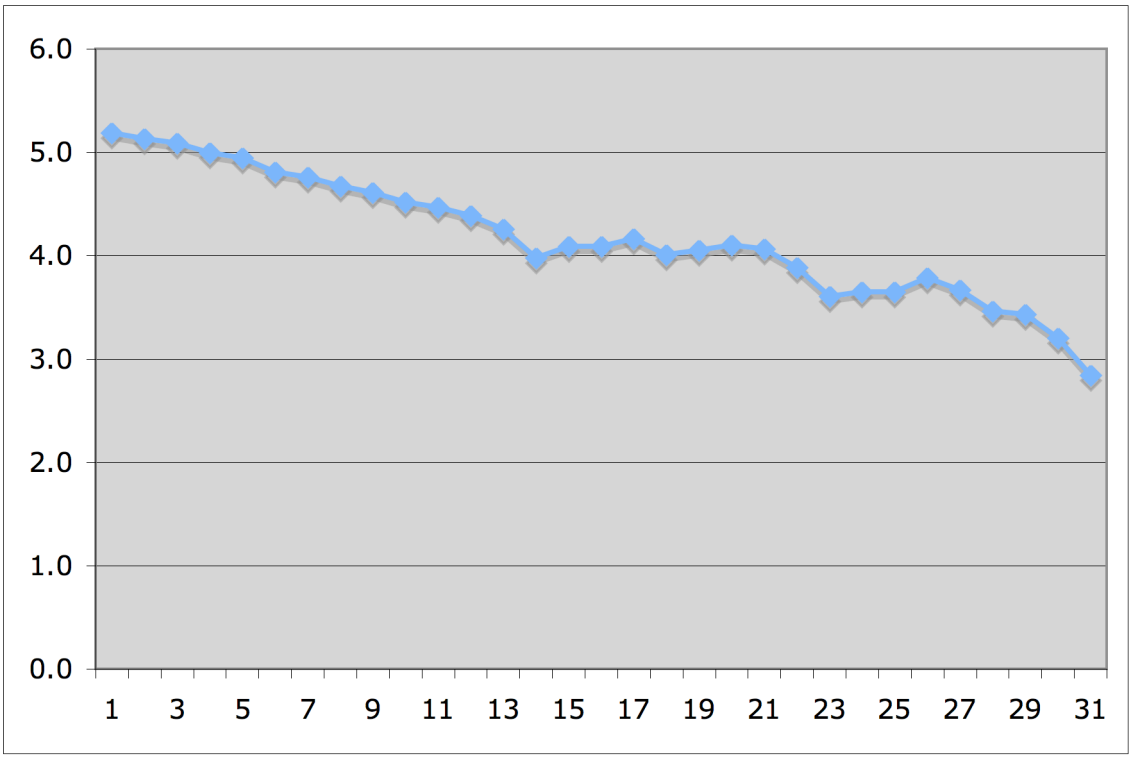


Figure 32: Log plot of response profile for Oracle XE

Linear regression coefficients and 95% confidence intervals for the values of *A, C* and *T* for a database with 100 tables were calculated using Microsoft Excel data analysis as shown in Table 25.

100 Tables	*Coefficients*	*Lower 95%*	*Upper 95%*
Intercept	5.21	5.10	5.31
Slope	–0.064	–0.070	–0.059
A	182.4	164.5	202.1
C	18.1	36	1.6
T	15.6	14.3	17.1

Table 25: Linear regression for time constant (100 tables)

Linear regression coefficients and 95% confidence intervals for the values of *A, C* and *T* for databases with 10, 500 and 1000 tables were calculated using Microsoft Excel data analysis as shown in Table 26, Table 27, and Table 28.

10 Tables	*Coefficients*	*Lower 95%*	*Upper 95%*
Intercept	4.80	4.09	5.52
Slope	−0.31	−0.40	−0.22
A	122.1	59.8	249.1
C	17.0	79.2	−110.1
T	3.2	2.5	4.5

Table 26: Linear regression for time constant (10 tables)

500 Tables	*Coefficients*	*Lower 95%*	*Upper 95%*
Intercept	4.93	4.88	4.99
Slope	−0.010	−0.011	−0.099
A	138.9	131.8	146.5
C	−0.5	6.1	−8.6
T	97.0	93.1	101.3

Table 27: Linear regression for time constant (500 tables)

1000 Tables	*Coefficients*	*Lower 95%*	*Upper 95%*
Intercept	4.89	4.86	4.93
Slope	−0.0053	−0.0055	−0.0052
A	133.6	129.2	138.1
C	0.7	5.1	−3.8
T	187.9	182.5	193.6

Table 28: Linear regression for time constant (1000 tables)

The experimental difference in the values for time constant for 10, 100, 500, and 1000 tables is significant at the 95% level.

6.5 Summary

This chapter presented results for the 1st experiment using MySQL with invalid and valid SQL, MySQL bugs found during the experiment, and results for the 2nd experiment using Oracle XE with valid SQL.

7 ANALYSIS

7.1 Introduction

This chapter presents an analysis of the experimental results. The 1st experiment used MySQL with both invalid and valid SQL while the 2nd experiment used Oracle XE with valid SQL only. Experiments with invalid SQL provided measurements of failure delay only while experiments with valid SQL provided measurements of F-measure, failure delay, and time constant. The experimental design is discussed in Chapter 5 and shown in Figure 13. This analysis looks for differences between experimental profiles P_1 and P_2 with the same SUT and for differences between SUT S_1 and S_2 with the same experimental profile, as well as considering the accuracy of the response profile regression analysis and the statistical distribution of the results.

7.2 Analysis of Results for P_1 and P_2

This section compares the results for F-measure with 'valid' and 'invalid' experimental profiles P_1 and P_2 using MySQL that is SUT S_1.

7.2.1 F-Measure

The median values and 95% confidence intervals for F-measure in the 1st experiment using experimental profile P_1 (after removal of outliers) and experimental profile P_2 are shown in Table 29. The experimental difference in the median values for F-measure for P_1 and P_2 using MySQL S_1 is not significant at the 95% level.

Experimental profile	*F-measure*	*Lower 95%*	*Upper 95%*
P_1	25,338	22,847.5	27,828.5
P_2	22,119	8992	35,245

Table 29: F-measure for P_1 and P_2

7.3 Analysis of Results for S_1 and S_2

This section compares the results for MySQL and Oracle XE that is SUT S_1 and S_2 using the 'valid' experimental profile P_2 for F-measure, failure delay, and time constant.

7.3.1 F-Measure

The median values and 95% confidence intervals for F-measure for S_1 and S_2 are shown in Table 30. F-measure figures given for Oracle XE are the number of blocks. For MySQL the number of SQL statements is given with the equivalent in blocks. The experimental difference in the median values for F-measure for S_1 and S_2 is significant at the 95% level, using interpolated values for Oracle XE with 300 tables (shown in bold), although the 95% confidence intervals for F-measure for Oracle XE with different numbers of tables are overlapping.

Oracle XE	*1482 SQL statements per block (average)*		
Number of tables	*F-measure F_{valid}*	*Lower 95%*	*Upper 95%*
10	38	21.8	102.2
100	60	-4.3	135.3
300	**99**	**30.8**	**186.7**
500	138	65.9	238.1
1000	214.5	131.8	306.2
MySQL			
260	22,119 = 14.9 blocks	8992 = 6.1 blocks	35,245 = 23.8 blocks

Table 30: F-measure for S_1 and S_2

7.3.2 Failure Delay

The median values and 95% confidence intervals for failure delay for S_1 and S_2 are shown in Table 31. Failure delay figures given for Oracle XE are the number of blocks. For MySQL the number of SQL statements is given with the equivalent in blocks; the failure delay figure given for MySQL is the delay in the 1st experiment with profile P_1. The experimental difference in the median values for failure delay for S_1 and S_2 is significant at the 95% level, using interpolated values for Oracle XE with 300 tables (shown in bold), although the 95% confidence intervals for failure delay for Oracle XE with different numbers of tables are overlapping.

Oracle XE	*1482 SQL statements per block (average)*		
Number of tables	*Delay D_{valid}*	*Lower 95%*	*Upper 95%*
10	38	18.6	79.4
100	33	-20.8	86.8
300	**56**	**4.75**	**107.25**
500	79	30.3	127.7
1000	110.5	-8.7	242.7
MySQL			
260	5338 = 3.6 blocks	2847.5 = 1.9 blocks	7828.5 = 5.3 blocks

Table 31: Failure delay for S_1 and S_2

7.3.3 Time Constant

The median values and 95% confidence intervals for time constant for S_1 and S_2 are shown in Table 32. The time constant figures given for both Oracle XE and MySQL are the number of blocks. The experimental difference in the values for time constant for S_1 and S_2 is significant at the 95% level, interpolating values for Oracle XE with 300 tables

(shown in bold). The experimental differences in the values for time constant for Oracle XE with different numbers of tables are significant at the 95% level.

Oracle XE	*1482 SQL statements per block*		
Number of tables	*Time constant T*	*Lower 95%*	*Upper 95%*
10	3.2	2.5	4.5
100	15.6	14.3	17.1
300	**56.3**	**53.7**	**59.2**
500	97	93.1	101.3
1000	187.9	182.5	193.6
MySQL			
260	12.31	10.22	15.47

Table 32: Time constant for S_1 and S_2

7.4 Accuracy of Response Profile Regression Analysis

As discussed in section 5.12.5 (calculation of time constant) plotting the natural logarithm of $f(x) = (A + C - y) = Ae^{-kx}$ against x should produce a straight line. In practice, random variation in the number of success responses $y(x)$ is observed as shown in the response profiles of Figure 28 and Figure 30 and this gives rise to variation in the logarithm plot which is a cause of inaccuracy in linear regression analysis.

The logarithm of $f(x)$ is $logA - kx$, however due to the influence of random variation we actually obtain the logarithm of $R(x)f(x)$, that is $log\ A - kx + log\ R$ where $R(x)$ is a factor due to random variation. If the magnitude of the random variation is small compared to $f(x)$ then $R(x)$ is close to 1 and $log\ R$ is close to zero so the variation is negligible. As shown in Figure 28 and Figure 30 the observed magnitude of variation is approximately $0.1A$ and this is approximately equal to $f(x)$ at $x = 2T$ for time constant T where $k = 1/T$. In this case, regression analysis of log plots should be accurate for $x < 2T$ as shown in Figure 32: Log plot of response profile for Oracle XE.

7.5 Statistical Distribution of Results

Chen, Kuo & Merkel (2004) report that the F-measure for random testing is distributed according to the (discrete) geometric distribution with a mean of approximately $M/log\ 2$ where M is the sample median and $1/log\ 2 = 1.44$ to 2 significant figures. Maronna, Martin & Yohai (2006) show that a robust estimate for the mean of the analogous (continuous) exponential distribution is $M/log\ 2$. However, the results of the present research show a ratio of mean/median ranging from 0.95 to 1.37 for F-measure and from 0.86 to 1.33 for delay, and this suggests the results are not geometrically distributed. This may be a result of the dependence between test cases for stochastic testing of a DBMS, where the result of each test case depends on the DBMS state from the previous test case.

Measures of Kurtosis and Skewness (from Excel data analysis descriptive statistics) suggest the results are very approximately normally distributed: F-measure has a range of Kurtosis from -1.70 to 1.29 and a range of Skewness from 0.23 to 1.10 while delay has a range of Kurtosis from -1.74 to 2.34 and a range of Skewness from -1.13 to 1.22. However the hypothesis of normality cannot be rejected on the basis of these measures because of the small size of the data sets (between 4 and 13 samples). Normal probability plots of F-measure results also suggest these results are approximately normally distributed as shown in APPENDIX F. An assumption of normal distribution when calculating 95% confidence intervals as discussed in section 5.15 is therefore justified.

7.6 Summary

This chapter presented an analysis of the experimental results. This analysis looked for differences between experimental profiles P_1 and P_2 with the same SUT and for differences between SUT S_1 and S_2 with the same experimental profile, and considered the accuracy of the response profile regression analysis and the statistical distribution of the results.

8 DISCUSSION

8.1 Introduction

The present research has investigated the delayed failure of software components and has addressed the problem that the conventional approach to software testing is unlikely to reveal this type of failure. Delayed failure of software components is a consequence of long-latency error propagation.

The focus of this research is on software components described by a request-response model. Within this conceptual framework, a Markov chain model of error propagation and failure has been developed that allows calculation of the delayed failure behaviour of software components based on estimation of state transition probabilities.

Experimental studies of DBMS using MySQL and Oracle XE have been conducted using stochastic testing with random generation of SQL and the results are consistent with expected behaviour based on the Markov chain model. SQL mutation was used to generate negative as well as positive test profiles.

Stochastic testing has been shown to be an effective method for revealing delayed failure, as well as being a suitable technique for reliability and robustness testing of software components. Metrics for failure delay and reliability for these software components have been shown to depend on the characteristics of the chosen experimental profile.

This chapter presents a discussion of the present research within the context of stochastic testing, SQL mutation, experimental studies of DBMS, fault detection effectiveness and efficiency, the conceptual framework, delayed failure, the research problem, and threats to validity.

8.2 Stochastic Testing

This section discusses the present research within the context of stochastic testing. Stochastic testing was introduced in Chapter 1.

Stochastic testing was chosen for the present research because it appeared to be a good candidate as a testing technique for delayed failure and was a suitable technique for a stateless approach. In addition, stochastic testing allows comparative measurement of reliability and robustness of software components with the use of 'valid' and 'invalid' profiles and has a strong link to related work, such as that of Slutz (1998) and Kaner & Bach (2004).

Whittaker (1997) and Nyman (2000) are among the earliest advocates for the effectiveness and efficiency of stochastic testing; yet stochastic testing continues to be an active area of current research in its modern form of 'fuzzing', as described for example by Garcia (2009) and by Khalek & Khurshid (2010).

The BCS Standard for Software Component Testing BS 7925-2 (2001) recognises random testing as a technique, but does not distinguish stochastic testing. Reid (2000) recalls that random testing was only added to the Standard at a late stage, after research comparing random testing to other software testing techniques demonstrated its effectiveness (Reid, 1997); a case could now be made for including stochastic testing in the Standard.

The robustness and reliability of MySQL as an Open Source product was expected to be at least comparable to that of a proprietary product such as Oracle XE. Reasoning, Inc. (2003) give details of faults found in automated source code inspection of MySQL version 4.0.16. This study reported 21 defects resulting in a defect density of 0.09 defects per KLOC, whereas the overall defect density of proprietary database code was reported as 0.58 defects per KLOC (Kilo Lines Of Code). The faults examined included memory leaks, null pointer dereferences, and un-initialised variables. However, for the present research the MTTF (median F-measure) number of SQL statements found in the

experimental study was 24,767 for MySQL and 146,718 for Oracle XE. This apparent difference may be due to differences in the experimental profiles used for MySQL and Oracle XE, that is to say the experimental profile used for MySQL may simply be more effective at revealing failures than the experimental profile used for Oracle XE.

The experimental profiles for the two SUT were created separately because of differences in SQL syntax, and it was expected that a profile based on only the common core SQL syntax would be less effective in revealing faults, compared to profiles incorporating proprietary SQL extensions. It would be interesting to repeat the experiments using a common profile based on the core SQL ISO/IEC 9075 2003 standard.

The order of magnitude of these values also was unexpected, since at a processing rate of 150 statements per second (as found in the response profile experiments for both SUT) 150,000 SQL statements would take 1000 sec to execute, or a little less than 17 minutes. Clearly, commercial DBMS typically operate for very much longer periods than this without failure! According to Hamlet & Voas (1993) desired software reliability is typically of the order of 10^6 to 10^8 executions without failure.

The main reason for the lower than expected MTTF for these DBMS in the experiments is probably that randomly generated SQL statements are more likely to reveal failures than those found in a typical operational profile. Reliability for a typical operational profile might be expected to be high for a 'stable' production software release, firstly because faults that caused failures in typical use have already been reported by users and fixed, and secondly that release testing is probably based on an operational profile, as discussed in Chapter 2.

Randomly generated SQL statements are likely to be very different from typical SQL statements encountered in normal operation; a complex SQL statement such as CREATE TABLE has a large number of optional parameters and so there are a very large number of possible combinations of parameters. Relatively few of the many

possible SQL statements present in a randomly generated profile are likely to be found in a typical operational profile.

Further support to this argument is provided by the provision of a query cache in MySQL. The query cache stores SQL queries, along with their results; the result of a query is available immediately if the query is repeated. Many typical DBMS environments have large numbers of identical queries, and table contents do not change often. The query cache is described in MySQL Administrator's Guide (2004) and in the MySQL 5.0 Reference Manual (2010).

In addition to the unusual syntax of randomly generated SQL statements, stochastic testing also has the feature that valid statements are very likely to be presented to the SUT in an invalid order, a form of negative testing called 'context testing' by Utting & Legeard (2007). Examples include the attempted CREATE of an existing table and attempted INSERT and DELETE operations on non-existing tables, rows, or columns. This accounts for the high proportion of 'accepted' to 'succeeded' SQL statements in the experimental results, and is not expected in a typical operational profile.

For these reasons, stochastic testing cannot be considered a 'pure' reliability testing technique, since it has strong aspects of robustness testing, and should be more correctly regarded as a dependability testing technique. Reliability, robustness, and dependability are discussed in Chapter 1.

8.3 SQL Mutation

This section discusses the present research within the context of SQL mutation; as discussed in Chapter 2, there appear to be few published works on the use of SQL mutation in DBMS testing prior to the present research.

As described in Chapters 5 and 6, two alternative approaches to SQL mutation were used to generate the 'invalid' experimental profile used in the 1st experiment; (a) BNF mutation using the program /mysql/programs/bmut.pl and (b) syntax mutation using the

program /mysql/programs/smut.pl. Refer to Table 10: ‘Invalid’ experimental profile for MySQL and Table 13: Results for ‘invalid’ experimental profile.

It is not evident from the results presented if one approach or the other can be regarded as more effective; this was not the focus of the present research, but might be an interesting topic for further research, as it is believed there has been little related work in this area.

The BCS Standard for Software Component Testing BS 7925-2 (2001) provides the following list of syntax mutation operators, as discussed in Chapter 5:

M1. Introduce an invalid value for an element.
M2. Substitute an element with another defined element.
M3. Miss out a defined element.
M4. Add an extra element.

These operators can be reduced to the repeated application of just two basic operators, ‘cut’ and ‘paste’ as described below. Furthermore, this approach produces several new syntax mutation operators.

The ‘paste’ operator (+) inserts an additional element into the string to the right of the current element, while the ‘cut’ operator (–) deletes the current element from the string: the deleted element is re-used in subsequent paste operations. If there has been no previous cut operation on the string then the paste operator inserts a default element.

As an example, taking the point of operation as the first element of the original string, the string ‘ABCD’ after applying the ‘paste’ operation (+) becomes ‘AXBCD’ then applying the ‘cut’ operation (–) this becomes ‘XBCD’ which is the same as operation M1 as given in BS 7925-2 (2001) ‘introduce an invalid value for an element’.

Further examples of operators produced by this method including M1 to M4 discussed above plus new mutations M5 to M9 are shown in Table 33.

Operator sequence	*Resulting string*	*Equivalent operator*
	ABCD	Original string unchanged
+−	XBCD	M1. Introduce an invalid value for an element
−−+−++−+	CBCD	M2. Substitute with another defined element
−	BCD	M3. Miss out a defined element
+	AXBCD	M4. Add an extra (invalid) element
−−	CD	M5. Miss out two adjacent elements
−+	BACD	M6. Swap two adjacent elements
++	AXXBCD	M7. Add two extra (invalid) elements
−−+	CBD	M3 (miss out) followed by M6 (swap)
+−+	XABCD	M8. Add extra element at the left of the string
+−++−	AABCD	M9. Duplicate (repeat) an element

Table 33: Extended set of mutation operators

Mutation operators M1 to M4 as given in BS 7925-2 (2001) are not sufficient to test the cases of swapped element order (M6) and repeated elements (M9). Testing using the extended set of mutation operators found the MySQL Bug #39144 "*CREATE TABLE - undocumented behaviour of reference_definition options*" as described in Chapter 6. This bug would not have been found by testing using only the mutation operators M1 to M4. The set of mutation operators given in BS 7925-2 (2001) should therefore be extended.

8.4 Experimental Studies of DBMS

This section discusses the present research within the context of experimental studies of DBMS including random database creation. Experimental studies of DBMS were introduced in Chapter 1.

With increasing volumes of data, databases have begun to be used in embedded applications; a survey of DBMS for embedded real-time systems can be found in Tešanovic, et al. (2002).

This ubiquity of DBMS should mean that techniques for testing them, as investigated in the present research, are of increasing importance in the future. Tools and techniques for SQL 'fuzzing' are now beginning to be employed, as reported by Garcia (2009), Naraine (2009) and Stoev (2009).

8.4.1 Database Creation

The test databases generated in the experiments have randomly generated schemas, populated with randomly generated data. This contrasts with the usual approach to DBMS testing as discussed in Chapter 2 and Chapter 5; production tests of the MySQL and Oracle XE DBMS are likely to have been performed with predefined databases, such as one of the example databases, or a copy of a production database.

Testing with a random database is likely to include unusual combinations of data types and values not covered in production testing performed with predefined databases, and so potentially expose dormant faults. This approach to DBMS testing can be viewed as a form of robustness testing of the DBMS, by means of the database interface rather than the SQL interface; refer to Figure 5: 'Block diagram of DBMS architecture' in Chapter 1. This approach to DBMS testing might be an interesting topic for further research, as it is believed there has been little related work in this area.

The databases generated using the experimental procedures described in Chapter 5 should have a random schema that is related to the choice of experimental profile. However, the DBMS imposes integrity constraints on SQL data definition and data manipulation statements, and inserts default values for parameters, and in the case of MySQL, substitutes default values for data types (this feature is optional, see Schumacher, 2005) and this will affect the resulting database schema. Database integrity constraints are discussed in Date (2004), Greenwald, Stackowiak & Stern (2004), the MySQL 5.0 Reference Manual (2010), and the Oracle® Database SQL Reference (2005).

8.5 Fault Detection Effectiveness and Efficiency

This section discusses the present research within the context of fault detection effectiveness and efficiency. Fault detection effectiveness and efficiency were discussed in Chapter 2, in particular with reference to Reid, et al. (1999) and BS 7925-2 (2001).

8.5.1 Fault Detection Effectiveness

It is expected that an effective testing technique should be able to reveal already known faults in the SUT, as well as to reveal new faults previously unknown; the MySQL bugs found during the experiment as discussed in Chapter 6 provide evidence that this is true of the stochastic testing technique investigated in the present research:

Bug Report #4046 was an existing bug.

Bug Report #38696 was similar to existing Bug#35578.

Bug Report #39144 was similar to existing Bug#34455.

Bug Report #45639 was new and repeatable on version 5.0.21 (although not on more recent versions).

This observation is important, because any new testing technique should be at least as effective as alternative techniques if it is to make a useful contribution to software testing practice.

It is interesting to consider if the technique will continue to be effective when faults are removed and testing is repeated; fault detection effectiveness is expected to be less, due to the pesticide paradox of Beizer (1990). However, new versions of software will probably contain new features and hence new defects, provided software development techniques do not improve. Furthermore, random generation of the test profile effectively generates new tests *ad infinitum* (Robinson, 1999) as discussed in Chapter 2, perhaps compensating for the pesticide paradox.

8.5.2 Fault Detection Efficiency

Testing software components with invalid input requests is expected to be less efficient than testing with valid input requests, because there is a low probability of acceptance P_0 as discussed in Chapter 4. Combining the results for both approaches to SQL mutation as shown in Table 13: Results for 'invalid' experimental profile, in chapter 6, gives 160,249 statements failed out of 171,471 executed, that is 93.5% giving a value for P_0 of 0.065.

Automatic generation and execution of test cases, as with other HVAT techniques, greatly increases the efficiency of the technique used in the present research compared to manual or semi-automated testing techniques. The creation of Perl programs to accomplish this was not found to be particularly difficult or time consuming, and once created these offer the promise of re-use with little modification for other varieties of DBMS, or altogether different software components, provided a specification and software interface is available.

Against these advantages must be set the effort required to debug long test traces, although the method for identifying minimal test sequences described in Chapter 5 is helpful. Manual preparation and debugging of SQL specification files in the present research did, however, occupy several weeks of effort on a part-time basis.

The automatic generation of a (random) test database, as opposed to the manual creation of a predefined test database, is another possible efficiency gain, although some time is required in every test run for the database tables to be created.

From the exponential saturation model described in Chapter 5, a period of $3T$ would be expected to be required in each test run for 95% of the database tables to be created, where T is the time constant. However, from the experimental results summarised in Table 34 the median of the ratio of F-measure to time constant F/T is only 1.4 suggesting that failure occurs somewhere between $1T$ and $2T$ that is when between 63% and 87% of database tables have been created (in cases where failure occurs at all).

The data in Table 34 is a summary of data found in Chapter 6 in Table 12, Table 14, Table 15, and Table 17; figures are in multiples of blocks. These results were obtained using the 'valid' experimental profile P_2. No measurement of time constant was obtained for MySQL in run (a) and so the value obtained in run (b) been substituted (*).

DBMS	*Tables*	*Median F-measure F*	*Median time constant T*	*F/T*
MySQL (a)	260	17.13	12.31 (*)	1.39
MySQL (b)	260	14.93	12.31	1.21
Oracle XE	10	38	3.2	11.88
Oracle XE	100	60	15.6	3.85
Oracle XE	500	138	97	1.42
Oracle XE	1000	214.5	187.9	1.14
	Median	**49.0**	**14.0**	**1.4**

Table 34: Ratio of F-measure to time constant

The small ratio *F*/*T* provides support for the hypothesis of a saturation effect, as reported by Menzies, Owen & Cukic (2002) and by Menzies, Owen & Richardson (2007). This effect, possibly due to 'clumping' of software states, would make stochastic testing more efficient than might otherwise be expected, as the smaller effective state space of the SUT can be rapidly covered, as discussed in Chapter 2.

From the point of view of software development testing using this testing technique, new failures might be expected to be found within a period of perhaps 3*T*, or not at all; and after fixing a defect, regression testing might only need be continued to perhaps 3*T*. This observation might begin to determine "*when we can stop fuzzing*" (Sutton, Greene & Amini, 2007).

8.6 The Conceptual Framework

This section discusses the present research within the context of the conceptual framework. The conceptual framework defines the scope and context of the present

research as described in Chapter 3, and consists of a request-response model of software components and a state transition model of error propagation and failure.

8.6.1 Partitioned Request-Response Model

The expected stateless behaviour of a software component is that (a) the component should accept valid requests, and (b) the component should reject invalid requests, as discussed in Chapter 3. The analysis of the partitioned request-response model in Chapter 3 (refer to Figure 11) focussed on two examples of possible failure behaviour:

1. The stateless element might reject a valid request, without passing the request on to the state-based element.

2. The stateless element might accept an invalid request and pass this on to the state-based element.

A more detailed analysis reveals six possible failure behaviours for the stateless element, as shown in Table 35.

Case	*Valid request*	*Request accepted*	*Request passed on*	*Failure*
1.	Yes	Yes	Yes	No
2.	Yes	Yes	No	Yes
3.	Yes	No	Yes	Yes
4.	Yes	No	No	Yes
5.	No	Yes	Yes	Yes
6.	No	Yes	No	Yes
7.	No	No	Yes	Yes
8.	No	No	No	No

Table 35: Possible stateless element behaviours

Only the failure behaviour cases (3), (4), (5), and (6) can be detected from outside the component boundary, and in these cases it is not possible to distinguish whether or not the request was passed on to the state-based element. Failure cases (2) and (7) cannot be distinguished from correct component behaviour.

Another possible failure behaviour of the stateless element is to incorrectly alter (corrupt) a valid request before passing it on to the state-based element; this might apply in cases (1) and (3) in Table 35.

Another possible behaviour of the stateless element might of course be to correct an invalid request before passing it on to the state-based element; if an invalid request is corrected by the stateless element, or if an invalid request is not passed on to the state-based element, then an incorrect software state should not occur as a result.

In addition, it is possible that the state-based element might enter an incorrect state, even if the request is valid; or else might not enter an incorrect state, even when the request is invalid.

This analysis shows that accepting valid requests and rejecting invalid requests does not guarantee the software component state is correct, and that accepting an invalid request does not necessarily mean the software component state is incorrect.

Other possible failure behaviours of the state-based element include accepting a request that conflicts with the current software state, and rejecting a request that is not in conflict with the software state. A stateless test oracle cannot detect these failures.

8.6.2 State Transition Model

The state transition model in Chapter 3 describes the delayed failure behaviour of software components as the result of long-latency error propagation, where a ‘precondition’ input request causes a transition to an incorrect software state, followed some time later by a ‘trigger’ input request that causes a transition to a failure state.

When probabilities are assigned to the transitions between software states, the state transition model becomes an absorbing Markov chain model. Expected values of F-measure and failure delay for valid and invalid input requests are calculated in Chapter 4 using the theory of absorbing Markov chain models, based on rough order of magnitude estimates of transition probabilities.

The expected values of F-measure and failure delay for invalid input requests calculated for hypothesis H_1 were F-measure $F_{invalid}$ = 10,020 statements = 6.76 blocks and failure delay $D_{invalid}$ = 10,010 statements = 6.75 blocks.

The expected values for F-measure and failure delay derived from the Markov chain model must be divided by P_0 and using the value found for P_0 of 0.065 gives modified values for F-measure $F_{invalid}$ = 154,153.8 statements = 104.02 blocks and for failure delay $D_{invalid}$ = 154,000 statements = 103.91 blocks.

No measurements of F-measure and failure delay for purely invalid input requests were obtained in the experiments; the 'invalid' profiles executed as shown in Chapter 6, Table 13: Results for 'invalid' experimental profile, were less than 60 blocks in length.

The expected values of F-measure and failure delay for valid input requests calculated for hypothesis H_2 were F-measure F_{valid} = 30,000 statements = 20.24 blocks and failure delay D_{valid} = 20,000 statements = 13.5 blocks.

Median values for F-measure and failure delay for MySQL as shown in Chapter 6, Table 12: Failure delay results for the 1st experiment, are 17.13 and 3.63 blocks respectively. Median values for F-measure F_{valid} and failure delay D_{valid} shown in Table 17: Summary of results for Oracle XE, are 99 blocks and 60 blocks respectively.

The differences in F-measure and failure delay results for MySQL and Oracle XE may be due to differences in the experimental profiles, as already discussed; however there may also be a possibility that the difference is at least partly due to differences in transition probabilities between macro states P_1 through P_5 as defined in Chapter 3.

8.7 Delayed Failure

This section discusses the present research within the context of failure delay systems and delayed software failure. Delayed failure was introduced in Chapter 1.

8.7.1 Failure Delay Systems

Although there has been work on reliability modelling of failure delay systems by Limnios (1988) and Faria (2008) and failure delay in software has been observed, such as for example crash latency reported by Lu, et al. (2005) these ideas do not appear to have been combined in a theory of reliability modelling of failure delay in software. The Markov chain model described in Chapter 3 and Chapter 4 could provide a reliability model for software components by relating transition probabilities to an operational profile, although there are difficulties in identifying operational profiles for software components, as noted in Chapter 1 and Chapter 2.

The Markov chain model might further provide a basis for a reliability model of software systems, by extending the concept of macro-states to encompass system states; however this lies outside the scope of the present research.

8.7.2 Delayed Software Failure

Although there appears to have been little related work on the delayed failure of software, it seems likely that this type of failure should occur in a wide variety of software components. First, the few examples of reports of delayed software failure, such as Pyle, McLatchie & Grandage (1971), Kaner (1997) and Thompson (2007) are spread across a wide range of software systems; an interactive computer system, a telephone system and an in-flight entertainment system. Second, the underlying mechanism of delayed failure as revealed in studies of software error latency, such as Madeira & Silva (1994) and in the state transition model investigated in the present research, seems to be generally applicable to any state-based software component.

Third, experiments conducted during the present research have revealed delayed failure of both the MySQL and Oracle XE DBMS.

It appears that delayed failure of software has not often been recognised, as discussed in Chapter 1; one important reason may be that, if the failure delay is less than the F-measure for an immediate failure, the delayed failure us likely to be 'masked' by the earlier occurrence of an immediate failure.

8.7.3 Immediate Failure

In contrast to delayed failure, defined as a failure that occurs some time after the condition that causes the failure due to long-latency error propagation, there are two other possible failure behaviours of a software component; 'short latency' failure and 'immediate' failure. A short latency failure occurs a very short time after the condition that causes the failure, due to short-latency error propagation, and differs from delayed failure only in the shortness of the delay; the two-step 'precondition' and 'trigger' process of failure as described in Chapter 3 is still present. An immediate failure, on the other hand, occurs immediately after the condition that causes the failure, in a single step that would be a transition directly from S_0 to S_F in the state transition model (Figure 12).

Immediate failure is not represented in the state transition model as described in Chapter 3, and falls outside the scope of the present research; the research problem focuses on the delayed failure of software components, and the conventional approach to software testing is believed to be effective in revealing both immediate failures and short latency failures.

A short latency failure from the point of view of the experiments conducted in the present research is a failure with a delay less than a single block of SQL statements. Failures having a delay less than a single block cannot be distinguished from an immediate failure with the experimental methodology used, and in such cases the experiment is inconclusive.

The Oracle XE results for F-measure and failure delay with 10, 100, 500 and 1000 database tables shown in Table 21, Table 22, Table 23, and Table 24 have a note (b) in the S_1 column for inconclusive runs where the SUT failed on a single re-execution of block number S_2. This indicates a short latency failure or possibly an immediate failure, and occurred in 15 out of 52 runs or 29%.

The probability of a transition directly from S_0 to S_F would have to be included in a complete reliability model for software components, as F-measure results will include delayed failures, short latency failures, and immediate failures, unless delayed failures are separated using a procedure for identifying 'precondition' requests, as in the experiments performed during the present research. Note that in Chapter 6, Table 14: F-measure results for MySQL DBMS do not distinguish between delayed failures and immediate failures.

Because of random variation introduced by the stochastic testing process, any single experiment might reveal delayed failure, immediate failure, or no failure. The Oracle XE results for F-measure and failure delay with 10, 100, 500 and 1000 database tables shown in Table 21, Table 22, Table 23, and Table 24 have a note (a) in the S_1 column for inconclusive runs where no failure occurred during the run; this occurred in 6 out of 52 runs or 12%.

8.8 Research Problem

This section discusses the present research within the context of the research problem. A statement of the research problem appears in Chapter 1, along with a discussion of why investigation of this problem is worthwhile.

The research problem arises when considering software component testing in the context of delayed failure; faults that cause delayed failures are likely to avoid detection by the conventional approach to software testing and remain in released software.

The present research has shown that stochastic testing is an effective testing technique for revealing delayed failure of software components. The approach taken to the research problem has used the methods of experimental software engineering: the development of a research question and hypotheses, and an experimental methodology for testing the hypotheses.

8.8.1 Research Question and Hypotheses

A statement of the research question appears in Chapter 4, along with the research hypotheses H_1 and H_2 together with the associated null hypotheses, and a discussion of why answering this question is worthwhile.

Delayed failure, consistent with long-latency error propagation, was observed as shown in Table 31: Failure delay for S_1 and S_2 in Chapter 6, for MySQL (invalid precondition) with 95% confidence; and for Oracle XE (valid precondition) with 95% confidence in the case of results for 10 tables and 500 tables, only.

The null hypotheses H_{01} and H_{02} can therefore both be rejected, suggesting that the state transition model is valid and so answering the research question, "*Is the state transition model valid?*"

Experimental demonstration of delayed failure, consistent with long-latency error propagation as described by the state transition model, suggests that the research problem can be solved for the SUT considered in the study, by using stochastic testing with sequences of test cases at least equal to the failure delay.

8.9 Threats to Validity

This section discusses internal, external, and construct threats to the validity of the experimental results.

8.9.1 Threats to Internal Validity

Internal validity pertains to the certainty of the cause and effect relationship between the independent variables and the dependent variables in the experiment, when the control variables remain unchanged during the experiment. The control variables, dependent variables, and independent variables involved in the experiments are listed in Table 36.

Variable type	*Variable*
1. Control	Hardware configuration – see APPENDIX B
2. Control	Software configuration – see APPENDIX B
3. Control	Software under test (Subject)
4. Control	Experimental profile (BNF specification) (Treatment)
5. Control	Initial state of the database
6. Control	Person conducting the experiment (Researcher)
7. Control	Physical environment (location of PC)
8. Control	Settings of DBMS configuration variables
9. Control	Time and date of experiment
10. Control	Versions of Perl programs, batch files and shell scripts
11. Dependent	Execution time (sec)
12. Dependent	Number of statements accepted
13. Dependent	Number of statements failed
14. Dependent	Number of statements rejected
15. Dependent	Number of statements succeeded
16. Independent	Sequence of SQL statements executed
17. Independent	Seed value (initialise random number generation)

Table 36: Variables involved in the experiments

Threats to internal validity arise where the control variables have changed between experimental runs, or where a control variable in the experiment has not been taken into account.

The hardware and software configuration, physical environment, and researcher have not changed over the course of the present research. The reported experiments have been carried out over a period of six years between 2005 and 2011 and some changes to the experimental profiles (BNF specifications) and Perl programs, batch files and shell scripts have occurred over this time. However, these would not have changed during a single repetition of an experiment. Changes to experimental variables were recorded in the author's research logbook, along with the experimental results, for future reference and program files contain version history in comments.

Note that the researcher conducting the experiment, while a control variable, is also a source of (possibly unconscious) bias. Such bias might include the choice of literature sources, the choice of experimental subjects, choice of methodology, choice of metrics, choice of analysis technique, and interpretation of results.

The best defence against these threats to internal validity is probably to replicate the experiments; however this was not possible within the resources and timescales of the present research.

A further internal threat to validity is the power of the experiments, that is, the level of statistical confidence that can be assigned to the results. In the present research, 95% confidence intervals for the mean were estimated for the experimental results and in some instances this showed that the experiment lacked sufficient power to provide confidence at this level. The power of the experiments might be improved by increasing the number of experimental runs, however as noted by Korver (1994) and discussed in Chapter 2, the sample size must be increased by a factor k^2 to reduce the standard error by a factor k.

8.9.2 Threats to External Validity

External validity pertains to the generality of the results; ideally, the results of the present research should generalise to a wide variety of different software components, potentially containing a wide variety of different faults, possibly resulting in a wide

variety of different failure behaviours, in a wide variety of different testing and operating environments.

Threats to external validity arise where the choice of subjects, treatments, or variables involved in the experiments limit the generality of the results.

The focus of the present research is on software components described by a request-response model; however, this is a common model of communication between software components, as described by Martin-Flatin (2005) and should generalise to a fairly wide variety of different software components.

F-measure and delayed failure metrics are based on simple crash/hang failure, because a stateless test oracle was desired, as discussed in Chapter 1. This is a limitation, as correctness of responses is not considered; however a crash, for example due to incorrect exception handling, is a common type of failure, see Mao & Lu (2005).

The behaviour of DBMS may not be representative of software components in general, and the behaviour of MySQL and Oracle XE may not be representative of some other DBMS. However, MySQL and Oracle are both Relational DBMS, probably the most common type of DBMS, and as they are both at least partly written in the C/C++ programming language they are likely to share many of the vulnerabilities and errors common to C/C++ with other software components written in that language.

The SUT studied in the present research run on both the Linux and Microsoft Windows operating systems, which are common system software environments.

The experimental profiles used in the present research are unlikely to be similar to any real operational profiles; as already discussed in this chapter, the experimental profiles may be more likely to reveal software failures than real operational profiles. It may be possible to relate apparent reliability during stochastic testing to actual component reliability, however this is an area for future research. Testing beyond the limits of real operational profiles is an established practice in other engineering disciplines; for

example, Forsberg, Mooz & Cotterman (2005) define 'Qualification' as demonstration that "*the design will perform in the intended environment, with margin*".

The best defence against threats to external validity is probably to replicate the experiments with a different choice of subjects and treatments; this would be an interesting topic for future research.

8.9.3 Threats to Construct Validity

Construct validity pertains to the use of surrogate measures for properties of interest in the research problem or question.

Threats to construct validity arise where the surrogate measure is not truly correlated with the property of interest. Properties of interest in the present research are software component reliability and robustness, failure delay, and fault detection effectiveness; the surrogate measure for these properties in the experiments is the F-measure as discussed by Chen, Kuo & Merkel (2004). Further research into the correlation of the F-measure with these properties is desirable; other measures of fault detection effectiveness have been proposed, including the E- and P-measures defined in Table 3 in Chapter 2, as discussed by Liu & Zhu (2008).

8.10 Summary

This chapter presented a discussion of the present research within the context of stochastic testing, SQL mutation, experimental studies of DBMS, fault detection effectiveness and efficiency, the conceptual framework, delayed failure, the research problem, and threats to validity.

9 CONCLUSION

9.1 Introduction

This chapter presents a summary of the conclusions of the present research, a summary of the contributions to knowledge, and areas for future research.

9.2 Conclusions

This section presents the conclusions of the present research under the following headings: software component testing, software component reliability and robustness, stochastic testing, experimental studies of DBMS, delayed failure, the conceptual framework, and the research problem.

9.2.1 Software Component Testing

a) The conventional approach to software testing is unlikely to detect delayed failures of software components, and alternative approaches should be employed to detect this type of failure. Delayed failures appear to have avoided detection in release testing for both MySQL and Oracle XE, so that faults leading to delayed failure have remained in the production release of software; the testing technique investigated in the present research was able to reveal these failures, and also revealed some known faults in the MySQL DBMS.

b) High-volume stochastic testing using random generation of valid SQL is an effective DBMS testing technique, able to quickly crash or hang the DBMS server, and does not necessarily require the additional effort of generating invalid SQL statements; this was not expected at the outset of the present research.

c) The MySQL DBMS accepts invalid SQL statements, and SQL mutation is an effective test technique for revealing this type of failure, although acceptance of

invalid SQL statements does not appear to have a significantly higher probability of failure (resulting in a crash) than valid statements. The assertion $P_{2\text{invalid}} > P_{2\text{valid}}$ that is, the assertion that in the state transition model, the probability P_2 of a transition from S_0 to S_P is greater for an invalid input than for a valid input, does not appear to be supported by the experimental results.

d) The mutation operators M1 to M4 as given in BS 7925-2 (2001) are not sufficient to test the cases of swapped element order and repeated elements. The set of mutation operators given in BS 7925-2 (2001) should be extended.

9.2.2 Software Component Reliability and Robustness

a) The present research provides a practical methodology for benchmarking software component dependability (reliability and robustness) and benchmark measures for the MySQL and Oracle XE DBMS have been determined. Metrics for failure delay and reliability for the software components investigated in the study depend on the characteristics of the chosen experimental profile.

b) The MTTF (median F-measure) found in the experimental study was significantly different for MySQL and Oracle XE and this result was unexpected. F-measure values found in the experimental study for both MySQL and Oracle XE were an order of magnitude less than expected.

9.2.3 Stochastic Testing

a) Stochastic testing is an effective method for revealing delayed failure, and is a suitable technique for comparative characterisation of the reliability and robustness of software components; although it is not suited for directly measuring component reliability.

b) The present research provides support for the hypothesis of a saturation effect, as reported by Menzies, Owen & Cukic (2002) and by Menzies, Owen &

Richardson (2007). This effect would make stochastic testing more efficient than might otherwise be expected.

9.2.4 Experimental Studies of DBMS

a) The present research provides experimental results that broadly corroborate related work into testing of DBMS using random generation of SQL, for example as reported by Slutz (1998).

b) Random database creation, in combination with stochastic testing, appears to be a powerful DBMS testing technique. The median of the ratio of F-measure to time constant *F/T* in cases where failures occurred suggests that failure occurs somewhere between 1*T* and 2*T* that is, when between 63% and 87% of database tables have been created.

9.2.5 Delayed Failure

a) Delayed failure is an important failure mode of software components and has been demonstrated at least for the components studied in the present research. There appear to be few systematic studies of delayed failure in the software engineering literature.

b) Experimental demonstration of delayed failure for both ‘valid’ and ‘invalid’ experimental profiles at a 95% confidence level means that the null hypotheses can be rejected, answering the research question in the affirmative by suggesting that the state transition model is valid.

c) Median values for F-measure and failure delay for MySQL were 17.13 and 3.63 blocks respectively. Median values for F-measure and failure delay for Oracle XE were 99 blocks and 60 blocks respectively.

9.2.6 The Conceptual Framework

a) The stateless request-response model provides a useful conceptual framework for black-box testing of software components. Within this conceptual framework, a Markov chain model of error propagation and failure allows calculation of the delayed failure behaviour of software components based on estimation of state transition probabilities.

b) Results of experimental studies of software components (MySQL and Oracle XE DBMS) using stochastic testing are consistent with expected behaviour based on the Markov chain model.

9.2.7 The Research Problem

Experimental demonstration of delayed failure, consistent with long-latency error propagation as described by the state transition model, suggests that the research problem can be solved at least for the DBMS components considered in the study, by using stochastic testing with sequences of test cases at least equal to the measured value of failure delay.

9.3 Summary of Contributions

This section presents the contributions to knowledge of the present research in the following areas: Software component testing, software component reliability and robustness, stochastic testing, experimental studies of DBMS, delayed failure, the conceptual framework, and the research problem.

9.3.1 Software Component Testing

Contributions to knowledge of the present research in the area of software component testing are: (a) The proposition, with supporting evidence, that the conventional approach to software testing is unlikely to detect delayed failures of software components which occur due to long-latency error propagation. (b) Experimental results

demonstrating delayed failure in two examples of production software that have already passed a quality control and release testing process. (c) The identification of a delayed failure mechanism in examples of reported software bugs.

9.3.2 Software Component Reliability and Robustness

Contributions to knowledge of the present research in the area of software component reliability and robustness are: (a) A methodology for benchmarking software component dependability (reliability and robustness). (b) Benchmark dependability metrics for the MySQL and Oracle XE DBMS using the technique.

9.3.3 Stochastic Testing

Contributions to knowledge of the present research in the area of stochastic testing are: (a) The application of stochastic testing to the delayed failure of software components, and evidence that this is an effective technique for revealing delayed failure. (b) Evidence that stochastic testing is a suitable technique for comparative characterisation of the reliability and robustness of software components. (c) Support for the hypothesis of a saturation effect, as reported by Menzies, Owen & Cukic (2002) and by Menzies, Owen & Richardson (2007).

9.3.4 Experimental Studies of DBMS

Contributions to knowledge of the present research in the area of experimental studies of DBMS are: (a) Corroboration of related work in DBMS testing using random generation of SQL. (b) A testing technique for DBMS that appears to be both efficient and effective, using random generation of SQL to create a random database.

9.3.5 Delayed Failure

Contributions to knowledge of the present research in the area of delayed failure are: (a) Evidence that delayed failure is an important failure mode of software components.

(b) A methodology for the measurement of failure delay in software components using both 'valid' and 'invalid' experimental profiles.

9.3.6 The Conceptual Framework

Contributions to knowledge of the present research in the area of the conceptual framework are: (a) An analysis of software component failure behaviour based on the partitioned request-response model. (b) A state transition model of delayed failure in software components and a method for calculation of the delayed failure behaviour of software components using a Markov chain model of error propagation and failure. (c) Experimental validation of the state transition model of delayed software failure with both 'valid' and 'invalid' experimental profiles.

9.3.7 The Research Problem

Contributions to knowledge of the present research in the area of the research problem are: Evidence that the research problem can be solved, at least for the DBMS components considered in the study, by using stochastic testing with sequences of test cases at least equal to the failure delay.

9.4 Future Research

The present research has identified a number of areas where future research could make an additional contribution to knowledge. This section describes potential future research topics in the areas of software component testing, software component reliability and robustness, stochastic testing, experimental studies of DBMS, delayed failure, the conceptual framework, and the research problem.

9.4.1 Software Component Testing

(a) Compare alternative approaches with stochastic testing from the point of view of their efficiency and effectiveness in revealing delayed failure, as discussed in the TAIC PART 2006 conference paper (APPENDIX D). Such approaches

might include adaptive testing, deterministic HVAT, directed random testing, and genetic algorithms.

(b) Replicate the experiments with a different choice of subjects and treatments and compare the results obtained for different software components and for different testing techniques or different experimental profiles.

9.4.2 Software Component Reliability and Robustness

(a) Perform further benchmarking of software components for reliability and robustness (dependability) and failure delay using the methodology of the present research, as discussed in the UKSMA 2009 conference paper (APPENDIX E).

(b) Determine appropriate surrogate measures for reliability, robustness and failure delay in software component testing, in comparison with the F-measure.

(c) Investigate the relationship between measures of reliability and robustness of software components during stochastic testing and actual component reliability and robustness in operational use.

9.4.3 Stochastic Testing

(a) Investigate the relationship between test run length and test effectiveness for stochastic testing. See Andrews, et al. (2008).

(b) Determine criteria for selecting appropriate experimental profiles for reliability and robustness benchmarking of software components such as DBMS using stochastic testing.

9.4.4 Experimental Studies of DBMS

(a) Perform further comparative experimental study of delayed failure in DBMS using a single profile based on the core SQL ISO/IEC 9075 2003 standard.

(b) Perform further research into the use of random databases and randomly generated schemas in DBMS testing, as proposed by Bati, et al. (2007).

9.4.5 Delayed Failure

Relate the failure and fault taxonomies in the literature, as discussed in Chapter 2, to faults found in delayed failure testing, and determine what class of faults are associated with delayed failure, and how frequently these lead to failure in an operational setting. See for example IEEE (1993) in Table 2.

9.4.6 The Conceptual Framework

Develop a reliability model for software components that accounts for both delayed failure and immediate failure, and relates transition probabilities in the Markov chain model to an operational profile.

9.4.7 The Research Problem

Investigate alternative solutions to the research problem such as extended random regression (ERR) as described by Kaner, Bond & McGee (2003) and compare the effectiveness of these approaches in revealing delayed failures to that of stochastic testing.

9.5 Summary

This chapter presented a summary of the conclusions of the research, a summary of the contributions to knowledge, and areas for future research.

10 REFERENCES

Abu-Shawiesh, Al-Athari & Kittani (2009)	Abu-Shawiesh, M.O, Al-Athari F.M, Kittani, H.F, "Confidence Interval for the Mean of a Contaminated Normal Distribution", *Journal of Applied Science*, Volume 9, 2835-2840, 2009.
Aitel (2002)	Aitel, D, "An Introduction to SPIKE, the Fuzzer Creation Kit", *Immunity*, 2002.
Ammann & Offutt (2008)	Ammann, P, Offutt, J, *Introduction to Software Testing*, Cambridge University Press, 2008.
Andrews, et al. (2008)	Andrews, J.H, Groce, A, Weston, M, Xu, R.G, "Random Test Run Length and Effectiveness", *IEEE/ACM Conference on Automated Software Engineering (ASE)*, 2008.
Arcuri, Iqbal & Briand (2010)	Arcuri, A, Iqbal, M.Z, Briand, L, "Black-Box System Testing of Real-Time Embedded Systems using Random and Search-Based Testing", *Testing Software and Systems*, Lecture Notes in Computer Science, Volume 6435, 2010.
Avizienis, et al. (2004)	Avizienis, A, Laprie, J.C, Randell, B, Landwehr, C, "Basic Concepts and Taxonomy of Dependable and Secure Computing", *IEEE Transactions on Dependable and Secure Computing*, Volume 1, Number 1, 2004.
Avizienis, Laprie & Randell (2004)	Avizienis, A, Laprie, J.C, Randell, B, "Dependability and its Threats: A Taxonomy", *Building the Information*

	Society Proc IFIP 18th World Computer Congress, Kluwer Academic Publishers, 2004.
Bal, Fan & Lintz (2009)	Bal, J, Fan, J, Lintz, J, "Automatic Test Case Generation for SQLUnit Testing Framework", Project Report, CS527: Advanced Topics in Software Engineering, University of Illinois at Urbana-Champaign, 2009.
Banahatti, Iyengar & Lodha (2009)	Banahatti, V, Iyengar, S, Lodha, S, "High Utility Data Generation Using DataXplod", Tata Consultancy Services Limited, 2009.
Basili (1996)	Basili, V.R, "The Role of Experimentation in Software Engineering: Past, Current, and Future", *18th International Conference on Software Engineering*, IEEE, 1996.
Bati, et al. (2007)	Bati, H, Giakoumakis, L, Herbert, S, Surna, A, "A Genetic Approach for Random Testing of Database Systems", Microsoft Corporation, 2007.
Beizer (1990)	Beizer, B, *Software Testing Techniques, 2nd edition*, 1990.
Bellovin (2006)	Bellovin, S.M, "On the Brittleness of Software and the Infeasibility of Security Metrics", *Security & Privacy*, Volume 4, Issue 4, IEEE, 2006.
Berndt & Watkins (2005)	Berndt, D, Watkins, A, "High Volume Software Testing using Genetic Algorithms", *Proceedings of the 38th Hawaii International Conference on System Sciences*, 2005.

Bernstein & Kintala (2004)	Bernstein, L, Kintala, C, "Software Rejuvenation", *CrossTalk - The Journal of Defense Software Engineering*, 2004.
Bernstein (2002)	Bernstein, L, "A Stitch in Time", *DoD Software Tech News*, Volume 5, Number 3, 2002.
Bhor (2001)	Bhor, A, "Software Component Testing Strategies", Technical Report, University of California Department of Information and Computer Science UCI-ICS-02-06, 2001.
Binder (2000)	Binder, R.V, *Testing Object Oriented Systems: Models Patterns and Tools*, Addison-Wesley, 2000.
Binder (2004)	Binder, R.V, "Automated Testing with an Operational Profile", *DoD Software Tech News*, Volume 8, Number 1, 2004.
Bowers, Lie & Smethells (2001)	Bowers, B, Lie, K, Smethells, G, "An Inquiry into the Stability and Reliability of UNIX Utilities", University of Wisconsin - Madison Computer Sciences Department, 2001.
Briand (2007)	Briand, L.C, "A Critical Analysis of Empirical Research in Software Testing", *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2007.
Brukman, Dolev & Kolodner (2003)	Brukman, O, Dolev, S, Kolodner, E.K, "Self-Stabilizing Autonomic Recoverer for Eventual Byzantine

	Software", *Proceedings of the International Conference on Software: Science, Technology and Engineering SwSTE '03*, IEEE, 2003.
Bruno, Chaudhuri & Thomas (2002)	Bruno, N, Chaudhuri, S, Thomas, D, "Generating Queries with Cardinality Constraints for DBMS Testing", *IEEE Transactions on Knowledge and Data Engineering*, 2002.
BS 7925-1 (2004)	British Computer Society (BCS), *BS 7925-1 Glossary of Software Testing Terms*, Working draft for the BCS Specialist Interest Group in Software Testing, Version 6.3, 2004.
BS 7925-2 (2001)	British Computer Society (BCS), *BS 7925-2 Standard for Software Component Testing*, BCS Testing Standards Working Party, Working draft 3.4, 2001.
Bush, Hershey & Vosburgh (2000)	Bush, S.F, Hershey, J, Vosburgh, K, "Brittle System Analysis", *Proceedings of VWSIM Virtual Worlds and Simulation Conference*, 2000.
Cabeca, Jino & Leitao-Junior (2009)	Cabeca, A.G, Jino, M, Leitao-Junior, P.S, "Mutation Analysis for SQL Database Applications", *Fourth International Conference on Software Engineering Advances*, 2009.
Chan, Cheung & Tse (2005)	Chan, W.K, Cheung, S.C, Tse, T.H, "Fault-Based Testing of Database Application Programs with Conceptual Data Model", *Proceedings of the fifth International Conference on Quality Software*, IEEE Computer Society Press, 2005.

Chandra & Chen (1998)	Chandra, S, Chen, P.M. "How Fail-Stop are Faulty Programs?", *Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, 1998.
Channon & Koch (2007)	Channon, D, Koch, D, "Experimental Methodology: Issues and Practice", University of Newcastle, 2007.
Chays, et al. (2004)	Chays, D, Deng, Y, Frankl, P, Dan, S, Vokolos, F, Weyuker, E, "An AGENDA for Testing Relational Database Applications", *Software Testing, Verification and Reliability* Volume 14, Issue 1, John Wiley & Sons, 2004.
Chen & Merkel (2007)	Chen, T.Y, Merkel, R, "Quasi-Random Testing", *IEEE Transactions on Reliability*, Volume 56, Number 3, 2007.
Chen, et al. (2007)	Chen, T.Y, Huang, D.H, Tse, T.H, Yang, Z, "An Innovative Approach to Tackling the Boundary Effect in Adaptive Random Testing", *40th Annual Hawaii International Conference on System Sciences* (HICSS), IEEE, 2007.
Chen, Kuo & Merkel (2004)	Chen, T, Kuo, F, Merkel, R, "On the Statistical Properties of the F-measure", *Proceedings of the 4th International Conference on Quality Software*, 2004.
Chillarege & Iyer (1987)	Chillarege, R, Iyer, R.K. "Measurement-Based Analysis of Error Latency", *IEEE Transactions on Computers*, Volume 36, Number 5, 1987.

Cleve & Zeller (2005)	Cleve, H, Zeller, A, "Locating Causes of Program Failures", *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, 2005.
Coley (1999)	Coley, D, *An Introduction to Genetic Algorithms for Scientists and Engineers*, World Scientific Publishing, 1999.
Costa & Madeira (1999)	Costa, D, Madeira, H, "Experimental Assessment of COTS DBMS Robustness Under Transient Faults", *Proceedings of the 1999 Pacific Rim International Symposium on Dependable Computing*, 1999.
Costa, Rilho & Madeira (2000)	Costa, D, Rilho, T, Madeira, H, "Joint Evaluation of Performance and Robustness of a COTS DBMS through Fault-Injection", *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2000.
Date (2004)	Date, C. J, *An Introduction to Database Systems, 8th edition*, Addison-Wesley, 2004.
Datta (2007)	Datta, S, *Metrics-Driven Enterprise Software Development*, J. Ross Publishing, 2007.
Derezińska (2009)	Derezińska, A, "An Experimental Case Study to Applying Mutation Analysis for SQL Queries", *Proceedings of the International Multi-conference on Computer Science and Information Technology*, 2009.
Easton & McColl (2005)	Easton, V, McColl, J, "Statistics Glossary - Design of Experiments", Centre for Applied Statistics, Lancaster

	University, web pages, 2005.
Faria (2008)	Faria, J.A, "Reliability Evaluation of Failure Delayed Engineering Systems", *IFIP International Federation for Information Processing, Volume 266, Innovation in Manufacturing Networks*, Springer, 2008.
Fenton & Pfleeger (1997)	Fenton, N, Pfleeger, S, *Software Metrics: A Rigorous & Practical Approach, 2nd edition*, PWS Publishing Company, 1997.
Finnigan (2009)	Finnigan, P, "A PL/SQL Fuzzer / Fuzzor", Oracle Security, web pages, 2009.
Firesmith (2005)	Firesmith, D, "Achieving Quality Requirements With Reused Software Components: Challenges to Successful Reuse", *Second International Workshop on Models and Processes for the Evaluation of off-the-shelf Components* (MPEC), Carnegie Mellon Software Engineering Institute, 2005.
Fischer-Cripps (2007)	Fischer-Cripps, A, *Introduction to Contact Mechanics, 2nd Revised edition*, Springer-Verlag New York Inc., 2007.
Forsberg, Mooz & Cotterman (2005)	Forsberg, K, Mooz, H, Cotterman, H, *Visualizing Project Management: Models and Frameworks for Mastering Complex Systems, 3rd edition*, Wiley, 2005.
Fuller, Luecke & Freiman (2001)	Fuller, E, Luecke, W, Freiman, S, "Influence of Water on Crack Growth in Glass", National Institute of Standards and Technology Virtual Library, 2001.

Gao, Tsao & Wu (2003)	Gao, J, Tsao, H, Wu, Y, *Testing and Quality Assurance for Component-Based Software*, Artech House, 2003.
Garcia (2009)	Garcia, R, "Case study: Experiences on SQL language fuzz testing", *Proceedings of the 2nd International Workshop on Testing Database Systems*, 2009.
Godefroid & Khurshid (2004)	Godefroid, P, Khurshid, S, "Exploring very large state spaces using genetic algorithms", *International Journal on Software Tools for Technology Transfer (STTT)*, 2004.
Godefroid, Klarlund & Sen (2005)	Godefroid, P, Klarlund, N, Sen, K, "DART: Directed Automated Random Testing", *Programming Language Design and Implementation PLDI*, ACM, 2005.
Greenwald, Stackowiak & Stern (2004)	Greenwald, R, Stackowiak, R, Stern, J, *Oracle Essentials: Oracle Database 10g, 3rd edition*, O'Reilly, 2004.
Grinstead & Snell (1998)	Grinstead, C.M, Snell, J.L, *Introduction to Probability, 2nd edition*, American Mathematical Society, 1998.
Haftmann, Kossmann & Kreutz (2005)	Haftmann, F, Kossmann, D, Kreutz, A, "Efficient Regression Tests for Database Applications", *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2005.
Haftmann, Kossmann & Lo (2007)	Haftmann, F, Kossmann, D, Lo, E, "A Framework for Efficient Regression Tests on Database Applications", *The VLDB Journal - The International Journal on Very*

	Large Data Bases, Volume 16, Issue 1, 2007.
Hamilton (1998)	Hamilton, J, "Are the Real Tough Problems Interesting?", Microsoft SQL Server Development Team, *Los Gatos NSF Industry/Academic Conference*, 1998.
Hamlet & Voas (1993)	Hamlet, D, Voas, J, "Faults on Its Sleeve: Amplifying Software Reliability Testing", *International Symposium on Software Testing and Analysis*, 1993.
Hamlet (1994)	Hamlet, R, "Random Testing", in *Encyclopedia of Software Engineering*, Wiley, 1994.
Hamlet, Mason & Woit (1999)	Hamlet, D, Mason, D, Woit, "Foundational Theory of Software Component Reliability", *10th International Symposium on Software Reliability Engineering*, 1999.
Haritsa (2006)	Haritsa, J, "SQL TESTING", Teaching Course E0 361 Database Management Systems, Computer Science and Automation, Indian Institute of Science, 2006.
Harman, et al. (1999)	Harman, M, Hierons, R, Holcombe, M, Jones, B, Reid, S, Roper, M, Woodward, M, "Towards a Maturity Model for Empirical Studies of Software Testing", *5th Workshop on Empirical Studies of Software Maintenance*, 1999.
Hierons (2009)	Hierons, R, "Adaptive Testing and Ordering Adaptive Test Cases", Brunel University, web pages, 2009.
Hoffman & Kaner (2003)	Hoffman, D, Kaner, C, "A Course on Software Test

	Automation Design", Center for Software Testing Education & Research, 2003.
Huang, et al. (1995)	Huang, Y, Kintala, C.M.R, Kolettis, N, Fulton, N.D, "Software Rejuvenation: Analysis, Module and Applications", *Proceedings of the 1995 International Symposium Fault-Tolerant Computing*, pages 381–390, 1995.
IEEE (1991)	IEEE Computer Society, *IEEE Standard Computer Dictionary 610*, 1991.
IEEE (1993)	IEEE Computer Society, *Standard Classification for Software Anomalies*, 1993.
Ince (1987)	Ince, D.C., "The Automatic Generation of Test Data", *The Computer Journal*, Volume 30, Number 1, 1987.
ISO/IEC 9075 (2008)	ISO International Organization for Standardization, *ISO/IEC 9075 Information Technology - Database Languages – SQL*, *3rd edition*, 2008.
ISTQB (2007)	Glossary Working Party of the International Software Testing Qualification Board (ISTQB), *Standard Glossary of Terms Used in Software Testing*, Version 2.0, 2007, Editor: Erik van Veenendaal (The Netherlands)
Janson (1995)	Janson, S, "Random Graphs", Uppsala University, 1995.
Jedlitschka & Pfahl (2005)	Jedlitschka, A, Pfahl, D, "Reporting Guidelines for Controlled Experiments in Software Engineering",

	International Symposium on Empirical Software Engineering, 2005.
Johnson (2006)	Johnson, C.W, "What are Emergent Properties and How Do They Affect the Engineering of Complex Systems?", *Reliability Engineering and System Safety*, Volume 91, Number 12, Elsevier, 2006.
Jorgensen (2002)	Jorgensen, A, "Testing with Hostile Data Streams", Florida Institute of Technology Computer Science Department, 2002.
Kaksonen, Laakso & Takenen (2000)	Kaksonen, R, Laakso, L, Takenen, A, "Vulnerability Analysis of Software through Syntax Testing", Technical Report, University of Oulu, Finland, 2000.
Kaner & Bach (2004)	Kaner, C, Bach, J, *Black Box Software Testing BBST*, Part 14 - Stochastic Testing, Online Course Notes, Center for Software Testing Education & Research, 2004.
Kaner & Bach (2005)	Kaner, C, Bach, J, *Black Box Software Testing BBST*, Overview - Part 3 (Test Oracles), Online Course Notes, Center for Software Testing Education & Research, 2005.
Kaner (1997)	Kaner, C, "The Impossibility of Complete Testing", *Software QA* Magazine, Volume 4, 1997.
Kaner (2000)	Kaner, C, "Architectures of Test Automation", *Software Testing, Analysis & Review Conference (Star) West, San Jose, CA*, 2000.

Kaner (2003-a)	Kaner, C, "Fundamental Challenges in Software Testing", Colloquium Presentation at Butler University, 2003.
Kaner (2003-b)	Kaner, C, "What is a Good Test Case?", *Software Testing Analysis & Review* (STAR East), 2003.
Kaner, Bond & McGee (2003)	Kaner, C, Bond, P, McGee, P, "High Volume Test Automation", Florida Institute of Technology Computer Science Department, 2003.
Kaner, Falk & Nguyen (1993)	Kaner, C, Falk, J, Nguyen, H, *Testing Computer Software*, International Thompson Computer Press, 1993.
Khalek & Khurshid (2010)	Khalek, S, Khurshid, S, "Automated SQL Query Generation for Systematic Testing of Database Engines", *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ACM digital library, 2010.
Khalek, et al. (2008)	Khalek, A, Elkarablieh, B, Laleye, Y, Khurshid, S, "Query-Aware Test Generation Using A Relational Constraint Solver", *ASE 08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.
Kitchenham, et al. (2007)	Kitchenham, B.A, Al-Khilidar, H, Babar, M.A, Berry, M, Cox, K, Keung, J, Kurniawati, F, Staples, M, Zhang, H, Zhu, L, "Evaluating Guidelines for Reporting Empirical Software Engineering Studies", *Empirical*

	Software Engineering, Springer Science & Business Media, LLC, 2007.
Koopman (1999)	Koopman, P, *Topics in Dependable Embedded Systems: Why Things Break*, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1999.
Koopman, et al. (2002)	Koopman, P, Siewiorek, D, DeVale, K, DeVale, J, Fernsler, K, Guttendorf, D, Kropp, N, Pan, J, Shelton, C, Shi, Y, "The Ballista Project: COTS Software Robustness Testing", Carnegie Mellon University, 2002.
Korovin & Voronkov (2005)	Korovin, K, Voronkov, A, "Random Databases and Threshold for Monotone Non-Recursive Datalog", *Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, Volume 3618, 2005.
Korver (1994)	Korver, B, "The Monte Carlo Method and Software Reliability Theory", Portland State University Computer Science Department, 1994.
Koziolek (2005)	Koziolek, H, "Operational Profiles for Software Reliability", Hasselbring, W. & Giesecke, S. (Ed.), *Dependability Engineering*, Volume 2, Chapter 6, GITO-Verlag, Berlin, 2005.
Kyne, et al. (2010)	Kyne, F, Bank, J, Sanders, D, Todd, M, Viguers, D, Watson, C, Yang, S, "System z Mean Time To Recovery Best Practices", IBM International Technical Support Organization, IBM Redbooks, SG24-7816-00,

2010.

Lake (2010) — Lake, M, “Epic Failures: 11 Infamous Software Bugs”, *Computerworld*, 2010.

Lewis (2009) — Lewis, W, *Software Testing and Continuous Quality Improvement, 3rd edition*, Auerbach Publications, 2009.

Limnios (1988) — Limnios, N, “Failure Delay Systems Reliability Modelling”, *Systems Reliability Assessment: Proceedings of the Ispra Course held at the Escuela Tecnica Superior de Ingenieros Navales, Madrid, Spain, September 19-23, 1988 in collaboration with Universidad Politecnica de Madrid*, Colombo, A.G, Saiz de Bustamante, A, (Ed.), published by Springer, 1990.

Liu & Zhu (2008) — Liu, Y, Zhu, H, “An Experimental Evaluation of the Reliability of Adaptive Random Testing Methods”, *The Second International Conference on Secure System Integration and Reliability Improvement* (SSIRI), 2008.

Lu, et al. (2005) — Lu, S, Li, Z, Qin, F, Tan, L, Zhou, P, Zhou, Y, “BugBench: Benchmarks for Evaluating Bug Detection Tools”, *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

Madeira & Silva (1994) — Madeira, H, Silva, J.G, “Experimental Evaluation of the Fail-Silent Behaviour in Computers Without Error Masking”, *Twenty-Fourth International Symposium on Fault-Tolerant Computing FTCS-24*, 1994.

Madeira, et al. (2002)	Madeira, H, Kanoun, K, Arlat, J, Costa, D, Crouzet, Y, Dal Cin, M, Gil, P, Suri, N, "Towards a Framework for Dependability Benchmarking", Submitted to the 4th European Dependable Computing Conference (EDCC4), 2002.
Mao & Lu (2005)	Mao, C.Y, Lu, Y.S, "Improving the Robustness and Reliability of Object-Oriented Programs Through Exception Analysis and Testing", *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems ICECCS*, 2005.
Maronna, Martin & Yohai (2006)	Maronna, R.A, Martin, D.R, Yohai, V.J, *Robust Statistics: Theory and Methods*, Wiley, 2006.
Martin-Flatin (2005)	Martin-Flatin, J.P, *Conception of Information Systems* Lecture 6: Transaction Monitors, Distributed Information Systems Laboratory, 2005.
Mason (2002-a)	Mason, D, "Probabilistic Analysis for Component Reliability Composition", *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly*, 2002.
Mason (2002-b)	Mason, D, "Probabilistic Program Analysis for Software Component Reliability", Doctoral Thesis, University of Waterloo, Canada, 2002.
McGee & Kaner (2004)	McGee, P, Kaner, C, "Experiments with High Volume Test Automation", Workshop on Empirical Research in Software Testing, International Symposium on Software Testing and Analysis, 2004.

Menzies, Owen & Cukic (2002)	Menzies, T, Owen, D, Cukic, B, "Saturation Effects in Testing of Formal Models", *Proceedings of the 13th International Symposium on Software Reliability Engineering*, 2002.
Menzies, Owen & Richardson (2007)	Menzies, T, Owen, D, Richardson, J, "The Strangest Thing About Software", *Computer*, Volume 40, Number 1, 2007.
Michael & Jones (1996)	Michael, C.C, Jones, R.C, "On the Uniformity of Error Propagation in Software", Technical Report RSTR-96-003-4, RST Corporation, Version 1.1, 1996. Also in *Proceedings of the 12th Annual Conference on Computer Assurance* (*COMPASS '97*).
Miller, et al. (1995)	Miller, B, Koski, D, Lee, C, Maganty, V, Murthy, R, Natarajan, A, Steidl, J, "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services", University of Wisconsin Computer Sciences Department, 1995.
Miller, Fredriksen & So (1990)	Miller, B, Fredriksen, L, So, B, "An Empirical Study of the Reliability of UNIX Utilities", *Communications of the ACM*, Volume 33, Issue 12 1990.
Mukherjee & Siewiorek (1997)	Mukherjee, A, Siewiorek, D, "Measuring Software Dependability by Robustness Benchmarking", *IEEE Transactions on Software Engineering*, 1997.
Murray (2008)	Murray, C, *Oracle® Database SQL Developer Supplementary Information for MySQL Migrations*,

	Release 1.5, E12155-01, 2008.
Musa (1993)	Musa, J.D, "Operational Profiles In Software Reliability Engineering", *IEEE Software*, Volume 10, Number 2, 1993.
Myers, et al. (2004)	Myers, J, *The Art of Software Testing, 2nd edition*, Revised and updated by Badgett, T, Thomas, T.M, Sandler, C, Wiley, 2004.
MySQL 5.0 Reference Manual (2010)	Oracle, *MySQL 5.0 Reference Manual*, 2010.
MySQL Administrator's Guide (2004)	MySQL AB, *MySQL® Administrator's Guide*, MySQL Press, 2004.
MySQL Forge (2010)	MySQL Forge, "Random Query Generator", web pages, 2010.
MySQL Internals Manual (2005)	MySQL AB, *MySQL Internals Manual*, Revision 472, 2005.
Naraine (2009)	Naraine, R, "Fuzzing for Oracle Database Vulnerabilities", ZDNet, web pages, 2009.
Nyman (2000)	Nyman, N, "Using Monkey Test Tools: How to find bugs cost-effectively through random testing", *Software Testing & Quality Engineering (STQE)*, 2000.
Oracle® Database Express Edition Installation Guide (2007)	*Oracle® Database Express Edition Installation Guide 10g Release 2 (10.2) for Linux* B25144-03, 2007.
Oracle® Database SQL	Lorentz, D, *Oracle® Database SQL Reference 10g*

Reference (2005)	*Release 2 (10.2)* B14200-02, 2005.
Pacheco, et al. (2007)	Pacheco, C, Lahiri, S.K, Ernst, M.D, Ball, T, "Feedback-directed Random Test Generation", *Proceedings of the 29th International Conference on Software Engineering*, 2007.
Pan (1999-a)	Pan, J, "Software Reliability", 18-849b *Dependable Embedded Systems*, Carnegie Mellon University, 1999.
Pan (1999-b)	Pan, J, "Software Testing", 18-849b *Dependable Embedded Systems*, Carnegie Mellon University, 1999.
Parizi, et al. (2009)	Parizi, R. M, Ghani, A.A.A, Abdullah, R, Atan, R, "On the Applicability of Random Testing for Aspect-Oriented Programs", *International Journal of Software Engineering and its Applications (IJSEIA)*, Volume 3, Number 4, 2009.
Prasad, McDermid & Wand (1996)	Prasad, D, McDermid, J, Wand, I, "Dependability Terminology: Similarities and Differences", *IEEE Aerospace and Electronic Systems Magazine*, Volume 11, Issue 1, 1996.
Pyle, McLatchie & Grandage (1971)	Pyle, I, McLatchie, R, Grandage, B, "A second-order bug with delayed effect", *Software – Practice and Experience*, 1(3): 231–233, 1971.
Reasoning, Inc. (2003)	Reasoning, Inc., "How Open-Source and Commercial Software Compare: A Quantitative Analysis of Database Implementations in Commercial Software and in MySQL 4.0.16", Technical White Paper, 2003.

Reid (1997)	Reid, S, "An Empirical Analysis of Equivalence Partitioning, Boundary Value Analysis and Random Testing", *Proceedings of the 4th International Software Metrics Symposium*, 1997.
Reid (2000)	Reid, S, "BS 7925-2: The Software Component Testing Standard", *The First Asia-Pacific Conference on Quality Software (APAQS)*, 2000.
Reid, et al. (1999)	Reid, S, Harman, M, Holcombe, M, Jones, B, Roper, M, Woodward, M, "A Framework for Measurement in Software Testing", *7th European International Conference Software Testing Analysis & Review (EuroSTAR)*, 1999.
Robert (2010)	Robert, K, "A Brief Introduction and Overview of Perl", Perl version 5.12.2 documentation, 2010.
Robinson (1999)	Robinson, H, "Graph Theory Techniques in Model-Based Testing", *International Conference on Testing Computer Software*, 1999.
Schumacher (2005)	Schumacher, R, "Guaranteeing Data Integrity with MySQL 5.0", MySQL AB, 2005.
Seleznjev & Thalheim (1998)	Seleznjev, O, Thalheim, B, "Random Databases and Keys", *International Conference of the Chilean Computer Science Society*, 1998.
Shan & Zhu (2006)	Shan, L, Zhu, H, "Testing Software Modelling Tools Using Data Mutation", *Proceedings of the 2006*

	International Workshop on Automation of Software Test, 2006.
Shan & Zhu (2007)	Shan, L, Zhu, H, "Generating Structurally Complex Test Cases by Data Mutation: A Case Study of Testing an Automated Modelling Tool", *The Computer Journal*, Published by Oxford University Press on behalf of the British Computer Society, 2007.
Shukla (1994)	Shukla, S, "Self Stability, Mutual Exclusion, and other Paradigms. A few relevant Definitions", Department of Computer Science, State University of New York at Albany, 1994.
Siewiorek, et al. (1993)	Siewiorek, D.P, Hudak, J.J, Suh, B.H, Segal, Z, "Development of a Benchmark to Measure System Robustness", *Third International Symposium on Fault-Tolerant Computing FTCS-23*, 1993.
Slutz (1998)	Slutz, D, "Massive Stochastic Testing of SQL", *VLDB'98 Proceedings of 24th International Conference on Very Large Data Bases*, 1998.
SQLite (2010)	SQLite, "How SQLite Is Tested", web pages, 2010.
Stafford & McGregor (2002)	Stafford, J, McGregor, J.D, "Issues in Predicting the Reliability of Composed Components", *5th ICSE Workshop on Component-Based Software Engineering*, 2002.
Stair & Reynolds (2008)	Stair, R, Reynolds, G, *Fundamentals of Information Systems, 5th edition*, Course Technology, 2008.

StatSoft, Inc. (2011)	StatSoft, Inc., *Electronic Statistics Textbook*, web pages, 2011.
Stoev (2009)	Stoev, P, "If You Love It, Break It: Testing MySQL with the Random Query Generator", *MySQL Conference & Expo*, 2009.
Storey (1996)	Storey, N, *Safety-Critical Computer Systems*, Addison Wesley Longman, 1996.
Sutton, Greene & Amini (2007)	Sutton, M, Greene, A, Amini, P, *Fuzzing: Brute Force Vulnerability Discovery*, Addison-Wesley Professional, 2007.
Tan, Chen & Jakubowski (2006)	Tan, G, Chen, Y, Jakubowski, M, "Delayed and Controlled Failures in Tamper-Resistant Software", *Proceedings of the 8th Information Hiding Workshop*, 2006.
Teorey & Ng (1998)	Teorey, T.J, Ng, W.T, "Dependability and Performance Measures for the Database Practitioner", *IEEE Transactions on Knowledge and Data Engineering*, 1998.
Tešanovic, et al. (2002)	Tešanovic, A, Nyström, D, Hansson, J, Norström, C, "Embedded Databases for Embedded Real-Time Systems: A Component-Based Approach", Technical report, Department of Computer Science, Linköping University and Department of Computer Engineering, Mälardalen University, 2002.

Thomason & Whittaker (1999)	Thomason, M.G, Whittaker, J.A, "Rare Failure-State in a Markov Chain Model for Software Reliability", *Proceedings of the 10th International Symposium on Software Reliability Engineering*, 1999.
Thompson (2007)	Thompson, H, "How to Crash an In-Flight Entertainment System", CXO Media Inc., web pages, 2007.
Trivedi & Vaidyanathan (2008)	Trivedi, K, Vaidyanathan, K, "Software Aging and Rejuvenation", *Wiley Encyclopedia of Computer Science and Engineering*, John Wiley & Sons, Inc., 2008.
Tuya, Suárez-Cabal & de la Riva (2006)	Tuya, J, Suárez-Cabal, M.J., de la Riva, C, "SQLMutation: a Tool to Generate Mutants of SQL Database Queries", *Second Workshop on Mutation Analysis*, 2006.
Tuya, Suárez-Cabal & de la Riva (2007)	Tuya, J, Suárez-Cabal, M.J., de la Riva, C, "Mutating Database Queries", *Information and Software Technology* 49(4) 398-417, 2007.
Utting & Legeard (2007)	Utting, M, Legeard, B, *Practical Model-Based Testing: A Tools Approach*, Morgan Kauffmann, 2007.
Utting (2007)	Utting, M, "Model-Based Testing: Black or White?", *Google Tech Talks*, 2007.
Varpiola & Takanen (2008)	Varpiola, M, Takanen, A, "How to Deploy Robustness Testing", *HAKIN9*, 2008.

Vaswani (2004)	Vaswani, V, "A Technical Tour of MySQL", *MySQL: The Complete Reference*, Chapter 2, McGraw-Hill/Osborne, 2004.
Vieira & Madeira (2009)	Vieira, M, Madeira, H, "From Performance To Dependability Benchmarking: A Mandatory Path", *Performance Evaluation and Benchmarking: Transaction Processing Performance Council Technology Conference (TPCTC)*, Lecture Notes in Computer Science, Springer, 2009.
Vinter (2001)	Beizer, B, Vinter, O, *Bug Taxonomy and Statistics*, 2001.
Weber (2002)	Weber, R, "Extending the Reach of Statistical Software Testing", Doctoral Thesis, Department of Computer Science and Engineering, University of Minnesota, 2002.
Whittaker & Thomason (1994)	Whittaker, J.A, Thomason, M.G, "A Markov Chain Model for Statistical Software Testing", *IEEE Transactions on Software Engineering* 20(10), 1994.
Whittaker & Voas (2000)	Whittaker, J.A, Voas, J, "Toward a More Reliable Theory of Software Reliability", *Computer*, Volume 33, Issue 12, 2000.
Whittaker (1997)	Whittaker, J, "Stochastic Software Testing", *Annals of Software Engineering*, Volume 4, 1997.
Willmor & Embury (2005)	Willmor, D, Embury, S, "Exploring Test Adequacy For Database Systems", *Proceedings of the 3rd UK Software*

	Testing Research Workshop (UK Test), 2005.
Yim, Kalbarczyk & Iyer (2009)	Yim, K.S, Kalbarczyk, Z.T, Iyer, R.K, "Quantitative Analysis of Long Latency Failures in System Software", *Procedings of the IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2009.
Zeller & Hildebrandt (2002)	Zeller, A, Hildebrandt, R, "Simplifying and Isolating Failure-Inducing Input", *IEEE Transactions on Software Engineering*, 28(2) 183–200, 2002.

APPENDIX A. DBMS RESPONSE PROFILE RESULTS

The full experimental results for MySQL and Oracle XE DBMS response profiles are shown in Table 37 and Table 38 respectively. For Oracle XE the results for *Seed* = 38 are excluded because this caused the DBMS to hang/freeze.

Note the number of SQL statements *executed* = *failed* + *accepted* + *succeeded* for MySQL experiments and the number of SQL statements *executed* = *accepted* + *succeeded* for Oracle XE experiments.

Run	*Executed*	*Failed*	*Accepted*	*Succeeded*	*Time (sec)*
1	1482	0	1470	12	6
2	1482	0	1461	21	5
3	1482	1	1459	22	6
4	1482	0	1447	35	7
5	1482	1	1444	37	6
6	1482	0	1399	83	7
7	1482	0	1408	74	8
8	1482	0	1381	101	9
9	1482	1	1365	116	9
10	1482	0	1335	147	9
11	1482	0	1326	156	10
12	1482	0	1320	162	10
13	1482	0	1277	205	12
14	1482	0	1276	206	11
15	1482	0	1255	227	11
16	1482	0	1269	213	11
17	1482	1	1265	216	9
18	1482	1	1270	211	9
19	1482	0	1258	224	10

20	1482	0	1247	235	11
21	1482	2	1241	239	9
22	1482	0	1261	221	8
23	1482	0	1250	232	11
24	1482	0	1246	236	9
25	1482	1	1256	225	9
26	1482	1	1240	241	10
27	1482	0	1231	251	9
28	1482	0	1230	252	11
29	1482	0	1255	227	11
30	1482	0	1233	249	11
31	1482	0	1210	272	11
32	1482	0	1238	244	10
33	1482	0	1227	255	11
34	1482	1	1256	225	10
35	1482	0	1244	238	9
36	1482	0	1233	249	10
37	1482	1	1235	246	10
38	1482	1	1240	241	11
39	1482	0	1221	261	12
40	1482	1	1241	240	12
41	1482	1	1257	224	10
42	1482	1	1250	231	9
43	1482	0	1244	238	11
44	1482	0	1241	241	9
45	1482	1	1223	258	10
46	1482	1	1251	230	11
47	1482	1	1249	232	11
48	1482	1	1232	249	11
49	1482	0	1226	256	10
50	1482	0	1215	267	10

51	1482	0	1233	249	10
52	1482	0	1256	226	12
53	1482	0	1257	225	12
54	1482	1	1242	239	10
55	1482	1	1249	232	10
56	1482	1	1223	258	8
57	1482	0	1251	231	9
58	1482	2	1235	245	9
59	1482	0	1233	249	10
60	1482	0	1239	243	10
61	1482	0	1239	243	10
62	1482	0	1240	242	9
63	1482	2	1243	237	10
64	1482	0	1220	262	10
65	1482	0	1253	229	10
66	1482	0	1236	246	9
67	1482	1	1229	252	9
68	1482	0	1234	248	10
69	1482	0	1235	247	10
70	1482	0	1243	239	9
71	1482	1	1237	244	10
72	1482	1	1250	231	9
73	1482	0	1223	259	10
74	1482	0	1238	244	10
75	1482	0	1238	244	10
76	1482	1	1235	246	9
77	1482	0	1235	247	9
78	1482	0	1219	263	10
79	1482	0	1264	218	10
80	1482	0	1237	245	11
81	1482	0	1232	250	10

82	1482	0	1241	241	10
83	1482	0	1249	233	9
84	1482	0	1252	230	8
85	1482	1	1244	237	8
86	1482	0	1221	261	10
87	1482	0	1239	243	9
88	1482	0	1235	247	10
89	1482	0	1230	252	10
90	1482	0	1242	240	9
91	1482	0	1216	266	11
92	1482	0	1244	238	10
93	1482	0	1252	230	9
94	1482	0	1230	252	9
95	1482	0	1258	224	8
96	1482	0	1254	228	10
97	1482	1	1254	227	9
98	1482	1	1230	251	9
99	1482	0	1249	233	10
100	1482	2	1253	227	9
Total	148,200	35	126,099	22,066	963

Table 37: Results for MySQL response profile

Seed	*Executed*	*Accepted*	*Succeeded*	*Time (sec)*
1	1468	1447	21	9
2	1510	1481	29	7
3	1516	1472	44	15
4	1494	1451	43	7
5	1499	1428	71	6
6	1436	1370	66	10
7	1488	1392	96	8
8	1500	1411	89	11
9	1473	1377	96	10
10	1439	1324	115	7
11	1488	1373	115	7
12	1510	1400	110	13
13	1465	1330	135	9
14	1512	1367	145	6
15	1525	1364	161	12
16	1464	1347	117	11
17	1497	1352	145	8
18	1495	1348	147	9
19	1501	1357	144	7
20	1502	1364	138	10
21	1489	1351	138	21
22	1493	1342	151	12
23	1477	1310	167	12
24	1519	1346	173	7
25	1479	1333	146	6
26	1454	1287	167	8
27	1462	1306	156	16
28	1463	1302	161	8
29	1514	1325	189	6

30	1438	1279	159	8
31	1512	1332	180	10
32	1506	1295	211	17
33	1441	1262	179	7
34	1457	1278	179	6
35	1457	1271	186	12
36	1510	1338	172	22
37	1428	1263	165	6
39	1466	1302	164	15
40	1506	1313	193	15
41	1454	1288	166	7
42	1479	1303	176	8
43	1494	1311	183	7
44	1467	1306	161	12
45	1510	1293	217	14
46	1415	1244	171	9
47	1504	1306	198	6
48	1540	1362	178	12
49	1470	1284	186	10
50	1498	1309	189	7
51	1520	1317	203	7
52	1474	1294	180	19
53	1453	1274	179	7
54	1441	1250	191	7
55	1470	1292	178	8
56	1450	1259	191	14
57	1470	1246	224	6
58	1501	1293	208	11
59	1498	1308	190	12
60	1476	1273	203	6
61	1479	1293	186	9

62	1460	1253	207	13
63	1508	1300	208	7
64	1472	1275	197	15
65	1496	1295	201	8
66	1523	1354	169	8
67	1458	1270	188	6
68	1505	1328	177	16
69	1507	1280	227	6
70	1500	1293	207	9
71	1478	1285	193	7
72	1487	1310	177	7
73	1472	1260	212	7
74	1489	1308	181	7
75	1470	1278	192	8
76	1542	1317	225	15
77	1493	1283	210	7
78	1477	1294	183	7
79	1445	1251	194	8
80	1467	1266	201	7
81	1459	1269	190	6
82	1449	1261	188	10
83	1501	1284	217	7
84	1486	1271	215	6
85	1463	1277	186	29
86	1431	1239	192	6
87	1480	1302	178	6
88	1452	1237	215	13
89	1497	1319	178	7
90	1495	1296	199	9
91	1495	1312	183	6
92	1467	1268	199	8

93	1497	1299	198	9
94	1469	1265	204	15
95	1458	1272	186	10
96	1490	1309	181	8
97	1507	1283	224	10
98	1492	1302	190	7
99	1447	1246	201	6
100	1492	1292	200	6
Total	146,692	129,798	16,894	946
Mean	**1481.7**	-	-	-

Table 38: Results for Oracle XE response profile

APPENDIX B. TEST MACHINE CONFIGURATIONS

The hardware and software configurations for the MySQL and Oracle XE test machines are shown in Table 39 and Table 40 respectively.

Machine Details	
Evesham Vale Computer	
Asset Number RA001128	
S/N WO00711296	
Software Configuration	
MySQL Ver 14.12 Distrib 5.0.22, for Win32 (ia32)	
This is perl, v5.8.2 built for MSWin32-x86-multi-thread	
(with 25 registered patches, see perl -V for more detail)	
Copyright 1987-2003, Larry Wall	
Binary build 808 provided by ActiveState Corp. http://www.ActiveState.com	
ActiveState is a division of Sophos.	
Built Dec 9 2003 10:19:40	
System Information	
OS Name	Microsoft Windows 2000 Professional
Version	5.0.2195 Build 2195
OS Manufacturer	Microsoft Corporation
System Name	MRM-EVESHAM
System Manufacturer	VIA Technologies, Inc.
System Model	VT82C692BX
System Type	X86-based PC
Processor	x86 Family 6 Model 8 Stepping 1 GenuineIntel ~600 Mhz
BIOS Version	Award Modular BIOS v4.51PG
Windows Directory	C:\WINNT
System Directory	C:\WINNT\System32
Boot Device	\Device\Harddisk0\Partition1
Locale	United States

User Name	MRM-EVESHAM\Michael Moulding
Time Zone	GMT Standard Time
Total Physical Memory	130,544 KB
Available Physical Memory	34,016 KB
Total Virtual Memory	635,456 KB
Available Virtual Memory	445,668 KB
Page File Space	504,912 KB
Page File	C:\pagefile.sys

Table 39: MySQL test machine configuration

Machine Details	
Acer Aspire One Notebook Computer	
Model Number ZG5	
S/N LUS030A096829238062500	
Software Configuration	
Oracle Database 10g Express Edition Release 10.2.0.1.0 – Production	
This is perl, v5.8.9 built for i686-linux-thread-multi	
Binary build 825 [288577] provided by ActiveState http://www.ActiveState.com	
Built Dec 15 2008 03:04:17	
System Information	
Model name:	Aspire one
Operating system:	Linpus Linux Lite vl.0.6.E
CPU:	Intel(R) Atom(TM) CPU N270 @ 1.60GH
System memory:	512 MB
Hard drive:	8 GB
Battery:	Li-ion 2200 mAh
BIOS version:	v0.3109

Table 40: Oracle XE test machine configuration

APPENDIX C. RESULTS FOR 1ST EXPERIMENT

This appendix describes the data sets and experimental results for the 1st experiment and gives details of the program and data files used to automate the experimental procedure.

C.1 Data Set A (Non-Mutated)

This section describes experimental data set A. Before starting the experimental runs, a data set of 50,000 non-mutated (valid) SQL statements was randomly generated by executing batch file build1.bat with a random seed value of 0 as shown below. The valid input values used in both the first and second experimental runs were drawn randomly from this set (data set A, file valid.txt). All statements in this set were unique.

> build1 50000 0
> Sun Jul 15 2007
> 215 non-duplicate lines
> Random generator seed value = 0
> 50000 statements generated
> 871 seconds elapsed

The profile of input values in the non-mutated (valid) data set A is shown in Table 41.

SQL Statement	*Number*	*% of Total*
"CREATE "	16,640	33
"INSERT "	16,782	34
"REPLACE "	17,964	36
" REPLACE"	(1,386)	(3)
Total	50,000	**100**

Table 41: Profile for data set A (non-mutated)

C.2 First Experimental Run

This section describes experimental data set B and the results of the first experimental run. Before starting the first experimental run, eight sets of 10,000,000 mutated input values were randomly generated using different random seed values by Dr. Jacob Mulenga using several machines at Shrivenham. The input values accepted by MySQL were collated and duplicates were removed from the set. A total number of 329,035 accepted mutated (invalid) input values were collated, from which 238,205 unique input values were collected. The invalid input values used in the first experimental run were drawn randomly from this set (data set B, file uniq.txt). The profile of input values in the mutated (invalid) data set B is shown in Table 42.

SQL Statement	*Number*	*% of Total*
"CREATE "	67,233	28
"INSERT "	63,052	26
"REPLACE "	116,220	49
" REPLACE"	(8,300)	(3)
Total	238,205	**100**

Table 42: Profile for data set B (mutated)

The experimental procedure was automated using batch files run1c.bat and run2b.bat and the results of executing the procedure are summarised below:

Sun Jul 15 2007	
run1c.bat 10000 0	No failure (inconclusive)
run1c.bat 10000 1	No failure (inconclusive)
run1c.bat 10000 2	Lost connection in step 3
	Restart MySQL
run2b.bat 10000 2	No failure (positive outcome)
run1c.bat 10000 3	No failure (inconclusive)

run1c.bat 10000 4	Lost connection in step 3 Restart MySQL
run2b.bat 10000 4	No failure (positive outcome)
run1c.bat 10000 5	No failure (inconclusive)
run1c.bat 10000 6	No failure (inconclusive)
run1c.bat 10000 7	Lost connection in step 2 (inconclusive)
run1c.bat 10000 8	No failure (inconclusive)
run1c.bat 10000 9	No failure (inconclusive)
run1c.bat 10000 10	Lost connection in step 2 (inconclusive)
run1c.bat 10000 11	No failure (inconclusive)
run1c.bat 10000 12	No failure (inconclusive)
run1c.bat 10000 13	No failure (inconclusive)

Detailed results for positive outcomes in the first experimental run are shown in Table 43. The step numbers correspond to the experimental procedure as described in Chapter 5, 1st Experiment Part I.

Step	*Seed*	*Executed*	*Failed*	*Accepted*	*Succeeded*	*Time (sec)*
(2)	2	10,000	4	9940	56	36
(3)	2	10,000	0	9975	25	31
(4)	2	4195 (*)	0	4109	86	26
(2)	4	10,000	5	9910	85	37
(3)	4	10,000	0	9950	50	32
(4)	4	5338 (*)	2	5154	182	36

(*) Lost connection to MySQL server during query; see Table 44.

Table 43: Positive outcomes in the first experimental run

The last logged statement for each positive outcome was as shown in Table 44.

Seed	*Last logged statement*
2	INSERT HIGH_PRIORITY INTO o4 SET col5 = DEFAULT /* 42911 */;
4	REPLACE DELAYED m2 SET col6 = DEFAULT ,col0 = DEFAULT ,col9 = 'v' /* 16018 */;

Table 44: Last logged statements for positive outcomes

C.3 Data Set C (Mutated)

This section describes experimental data set C. Before starting the second experimental run, a second initial data set of 14,844 accepted mutated (invalid) input values was prepared by executing batch file build2.bat. Of the 14,845 statements accepted, one was a single ";" and so this was deleted from the set. The invalid input values used in the second experimental run were drawn randomly from this set (data set C). All statements in this set were unique; however the set consisted only of CREATE statements.

> build2 3000000 0
init.pl
Sun Aug 5 2007
inv2.pl
Random generator seed value = 0
454 lines read
31 lines modified
10% mutated
expand2.pl
216 non-duplicate lines
0 warnings
Random generator seed value = 0
3000000 statements generated
48961 seconds elapsed

xx.pl

Mon Aug 6 2007

2813219 statements executed

2798346 failed

14845 accepted

28 succeeded

5325 seconds elapsed

C.4 Second Experimental Run

This section describes the results of the second experimental run. The experimental procedure was automated using batch files run1b.bat and run2b.bat and the results of executing the procedure are summarised below.

Mon Aug 6 2007	
run1b.bat 10000 0	Lost connection in step 3
	Restart MySQL
run2b.bat 10000 0	No failure (positive outcome)
run1b.bat 10000 1	No failure (inconclusive)
run1b.bat 10000 2	No failure (inconclusive)
run1b.bat 10000 3	Lost connection in step 3
	Restart MySQL
run2b.bat 10000 3	No failure (positive outcome)
run1b.bat 10000 4	No failure (inconclusive)
run1b.bat 10000 5	No failure (inconclusive)
run1b.bat 10000 6	No failure (inconclusive)

Detailed results for positive outcomes in the second experimental run are shown in Table 45. The step numbers correspond to the experimental procedure as described in Chapter 5, 1st Experiment Part I.

Step	*Seed*	*Executed*	*Failed*	*Accepted*	*Succeeded*	*Time (sec)*
(2)	0	10,000	4	9952	44	33
(3)	0	10,000	0	9998	2	16
(4)	0	9265 (*)	3	9000	262	65
(2)	3	10,000	5	9965	30	32
(3)	3	10,000	0	10,000	0	15
(4)	3	5431 (*)	3	5345	83	31

(*) Lost connection to MySQL server during query; see Table 46.

Table 45: Positive outcomes in the second experimental run

The last logged statement for each positive outcome was as shown in Table 46.

Seed	*Last logged statement*
0	CREATE TABLE IF NOT EXISTS a1 INSERT_METHOD FIRST CHECKSUM = 0 MIN_ROWS 0 ENGINE = MRG_MYISAM IGNORE SELECT * FROM u0 /* 28724 */;
3	REPLACE DELAYED INTO h4 SET col6 = DEFAULT /* 40055 */;

Table 46: Last logged statements for positive outcomes

C.5 Experimental Configurations

This section describes the program and data files used to automate the experimental procedure for the 1st experiment using MySQL.

C.5.1 MySQL Data Files

/mysql/data/bnf5.txt	BNF specification for MySQL version 5
/mysql/data/inv.txt	Output file for inv2.pl
/mysql/Results/uniq.txt	Output from uniq.pl

C.5.2 MySQL Program Files

/mysql/programs/build1.bat	Batch program to build valid data
/mysql/programs/build2.bat	Batch program to build invalid data
/mysql/programs/expand2.pl	Expand BNF specification
/mysql/programs/inv2.pl	Generate invalid BNF by mutation
/mysql/programs/run1c.bat	Batch program to run experiment part 1
/mysql/programs/run2a.bat	Batch program to run experiment part 2
/mysql/programs/run2b.bat	Batch program to run experiment part 2
/mysql/programs/rx.pl	Execute SQL statements in random order
/mysql/programs/uniq.pl	Make sorted lines unique

C.5.3 Batch File Listings

build1.bat

```
rem batch program to build valid data
rem July 2007
rem statements parameter %1
rem seed value parameter %2
perl init.pl
echo %0 >> ..\data\summary.txt
echo ## building valid data ## >> ..\data\summary.txt
echo ## valid (non-mutated) ## >> ..\data\summary.txt
copy ..\data\bnf5.txt ..\data\inv.txt /Y
perl expand2.pl %1 %2
copy ..\data\x.txt ..\data\valid.txt /Y
copy ..\data\summary.txt ..\data\blog1.txt /Y
rem end of file
```

build2.bat

```
rem batch program to build invalid data
rem August 2007
rem statements parameter %1
rem seed value parameter %2
perl init.pl
echo %0 >> ..\data\summary.txt
echo ## building invalid data ## >> ..\data\summary.txt
echo ## invalid (mutated)     ## >> ..\data\summary.txt
perl inv2.pl %1 %2
copy ..\data\inv.txt ..\data\inv2.txt /Y
perl expand2.pl %1 %2
find "?" ..\data\x.txt >..\data\tmp.txt
echo ## Resetting database ## >> ..\data\summary.txt
copy ..\data\reset.txt ..\data\x.txt /Y
perl xx.pl
echo ## Setting preconditions ## >> ..\data\summary.txt
copy ..\data\bnf5.txt ..\data\inv.txt /Y
perl expand2.pl 10000 0
perl xx.pl
copy ..\data\summary.txt ..\data\blog2.txt /Y
perl init.pl
echo ## Collecting acc and suc ## >> ..\data\summary.txt
copy ..\data\tmp.txt ..\data\x.txt
perl xx.pl
copy ..\data\acc.txt ..\data\invalid.txt
type ..\data\suc.txt >> ..\data\invalid.txt
type ..\data\summary.txt >> ..\data\blog2.txt
rem end of file
```

run1c.bat

```
rem batch program to run experiment part 1(b)
rem modified August 2007
rem uses invalid data from ..\Results\uniq.txt
rem uses rx.pl
rem statements parameter %1
rem seed value parameter %2
perl init.pl
echo %0 >> ..\data\summary.txt
echo ## Resetting database ## >> ..\data\summary.txt
echo ## Resetting database ## >> ..\data\acc.txt
echo ## Resetting database ## >> ..\data\err.txt
echo ## Resetting database ## >> ..\data\log.txt
echo ## Resetting database ## >> ..\data\suc.txt
type ..\data\reset.txt >> ..\data\summary.txt
echo . >> ..\data\summary.txt
copy ..\data\reset.txt ..\data\x.txt /Y
perl xx.pl
echo ## First run (non-mutated) ## >> ..\data\summary.txt
echo ## First run (non-mutated) ## >> ..\data\acc.txt
echo ## First run (non-mutated) ## >> ..\data\err.txt
echo ## First run (non-mutated) ## >> ..\data\log.txt
echo ## First run (non-mutated) ## >> ..\data\suc.txt
copy ..\data\valid.txt ..\data\x.txt /Y
perl rx.pl %1 %2
echo ## second run (mutated) ## >> ..\data\summary.txt
echo ## second run (mutated) ## >> ..\data\acc.txt
echo ## second run (mutated) ## >> ..\data\err.txt
echo ## second run (mutated) ## >> ..\data\log.txt
echo ## second run (mutated) ## >> ..\data\suc.txt
copy ..\Results\uniq.txt ..\data\x.txt /Y
```

```
perl rx.pl %1 %2
echo ## third run (non-mutated) ## >> ..\data\summary.txt
echo ## third run (non-mutated) ## >> ..\data\acc.txt
echo ## third run (non-mutated) ## >> ..\data\err.txt
echo ## third run (non-mutated) ## >> ..\data\log.txt
echo ## third run (non-mutated) ## >> ..\data\suc.txt
copy ..\data\valid.txt ..\data\x.txt /Y
perl rx.pl %1 %2
rem end of file
```

run2a.bat

```
rem batch program to run experiment part 2
rem Single run
rem December 2007
rem no parameters required
perl init.pl
copy ..\data\reset.txt ..\data\x.txt /Y
echo %0 >> ..\data\summary.txt
echo # Resetting database # >> ..\data\summary.txt
echo # Resetting database # >> ..\data\acc.txt
echo # Resetting database # >> ..\data\err.txt
echo # Resetting database # >> ..\data\log.txt
echo # Resetting database # >> ..\data\suc.txt
type ..\data\reset.txt >> ..\data\summary.txt
echo . >> ..\data\summary.txt
perl xx.pl
echo # Single run (non-mutated) # >> ..\data\summary.txt
echo # Single run (non-mutated) # >> ..\data\acc.txt
echo # Single run (non-mutated) # >> ..\data\err.txt
echo # Single run (non-mutated) # >> ..\data\log.txt
echo # Single run (non-mutated) # >> ..\data\suc.txt
copy ..\data\valid.txt ..\data\x.txt /Y
perl xx.pl
rem end of file
```

run2b.bat

```
rem batch program to run experiment part 2
rem July 2007
rem statements parameter %1
rem seed value parameter %2
perl init.pl
copy ..\data\reset.txt ..\data\x.txt /Y
echo %0 >> ..\data\summary.txt
echo # Resetting database # >> ..\data\summary.txt
echo # Resetting database # >> ..\data\acc.txt
echo # Resetting database # >> ..\data\err.txt
echo # Resetting database # >> ..\data\log.txt
echo # Resetting database # >> ..\data\suc.txt
type ..\data\reset.txt >> ..\data\summary.txt
echo . >> ..\data\summary.txt
perl xx.pl
echo # First run (non-mutated) # >> ..\data\summary.txt
echo # First run (non-mutated) # >> ..\data\acc.txt
echo # First run (non-mutated) # >> ..\data\err.txt
echo # First run (non-mutated) # >> ..\data\log.txt
echo # First run (non-mutated) # >> ..\data\suc.txt
copy ..\data\valid.txt ..\data\x.txt /Y
perl rx.pl %1 %2
echo # second run (non-mutated) # >> ..\data\summary.txt
echo # second run (non-mutated) # >> ..\data\acc.txt
echo # second run (non-mutated) # >> ..\data\err.txt
echo # second run (non-mutated) # >> ..\data\log.txt
echo # second run (non-mutated) # >> ..\data\suc.txt
perl rx.pl %1 %2
rem end of file
```

APPENDIX D. CONFERENCE PAPER (TAIC PART 2006)

Conference paper presented at the Testing Academic & Industrial Conference – Practice And Research Techniques (TAIC PART) 2006.

Delayed Failures in Software
Using High Volume Automated Testing

http://www.woomerang.com/research/PhD/jgardiner-hvat.pdf

ABSTRACT

The research described studies delayed failures in software using high volume automated testing (HVAT) and investigates the effectiveness of different HVAT techniques; such techniques include genetic algorithms, model-based testing, penetration testing, robustness testing, and random (stochastic) testing. A delayed failure is a failure that occurs some time after the conditions that lead to the failure are applied. There appear to be no studies of delayed failures of software in the literature and no comparative studies of the effectiveness of different HVAT techniques; therefore research in this area can make an important contribution. Delayed failures in software are unlikely to be revealed by conventional testing techniques; a HVAT technique that systematically reveals delayed failures could lead to improved reliability of software and reduced costs. Experimental work is in progress using the MySQL database server as the software under test.

Keywords: Automated testing, database testing, delayed failures, high volume, HVAT, software testing, testing techniques.

APPENDIX E. CONFERENCE PAPER (UKSMA 2009)

Conference paper presented at the 20th Annual Conference of the United Kingdom Software Metrics Association (UKSMA) 2009.

Benchmarking Software Components
Using High Volume Automated Testing Techniques

http://www.woomerang.com/research/PhD/jgardiner-uksma09-v2d.pdf

ABSTRACT

Component-based software engineering (CBSE) seeks to build software systems by composition of pre-existing software components. Two related problems are how to predict the properties of the assembled system given the properties of individual components, and how to guarantee that one component can be substituted for another without changing the properties of the overall system. A starting point is to establish metrics that characterise a given type of software component and to use these metrics to benchmark individual components. Two important properties of software components are reliability and robustness. Metrics for reliability and robustness are discussed in the context of empirical evaluation of software components using high volume automated testing (HVAT). The particular type of software component considered is a database management system (DBMS) and benchmark results obtained for the MySQL DBMS are presented. It is concluded that random testing is potentially useful for reliability and robustness benchmarking of software components. The benchmark input profile can be generated and executed automatically, provided a suitable component specification is available. Future work will examine criteria for selecting an appropriate input profile and will compare and contrast measurements obtained with another DBMS component.

Keywords: Database management system, metrics, random testing, reliability, robustness, software engineering.

APPENDIX F. NORMAL PROBABILITY PLOTS

This appendix presents normal probability plots of F-measure results. These results appear to be approximately normally distributed (falling approximately on a straight line).

Normal order statistic values N_i for $i = 1$ to n are approximately $0.5 \times log(m + x / m - x)$ where $x = i - m$ and m is the mean value of i for n data values (taking the natural logarithm).

Normal probability plots of F-measure results for MySQL (Table 14) are shown in Figure 33.

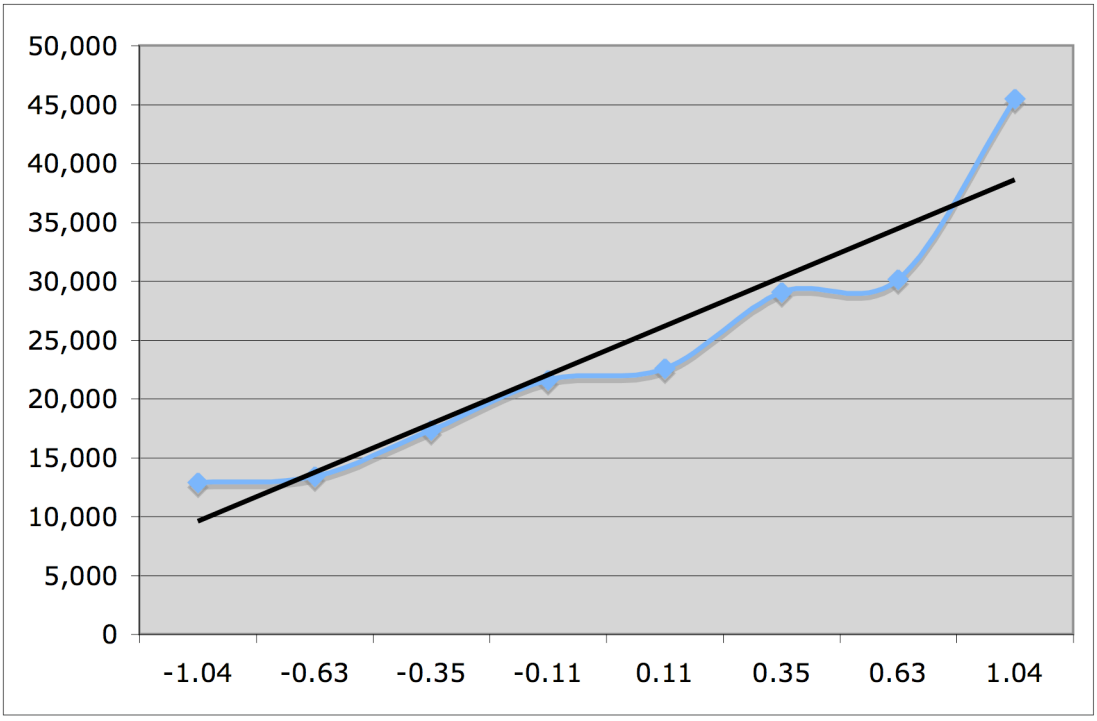


Figure 33: Normal probability plot for MySQL

Normal probability plots of F-measure results for Oracle XE 10 tables (Table 21) are shown in Figure 34.

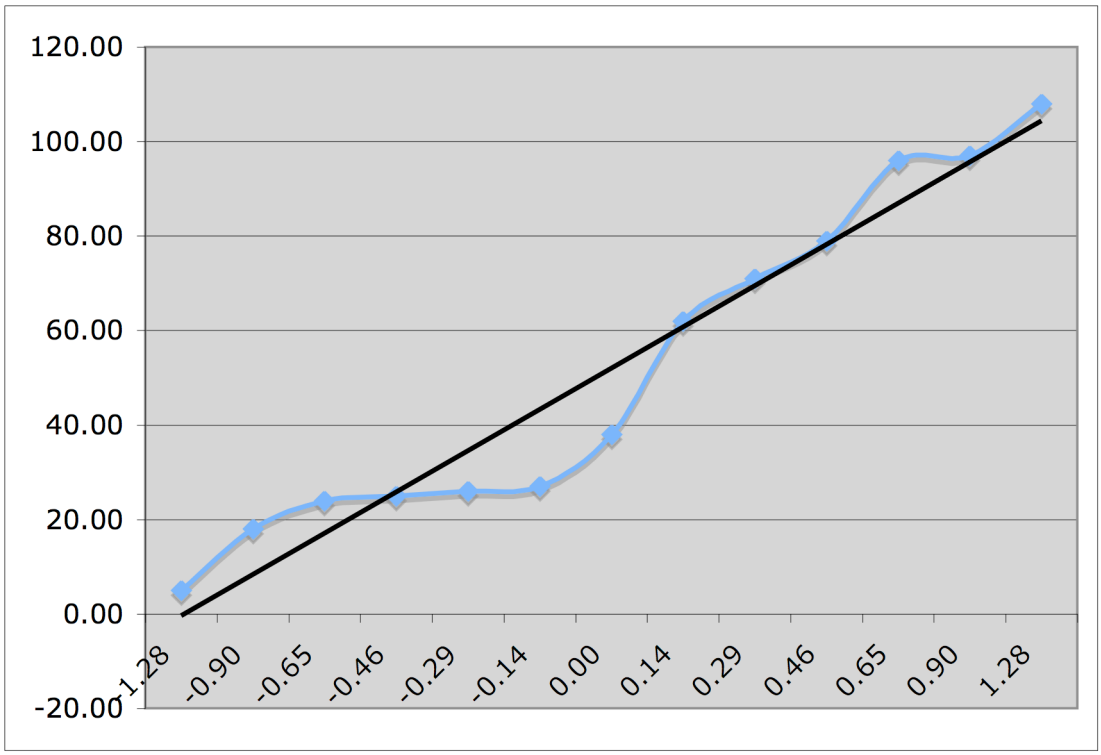


Figure 34: Normal probability plot for Oracle XE 10 tables

Normal probability plots of F-measure results for Oracle XE 100 tables (Table 22) are shown in Figure 35.

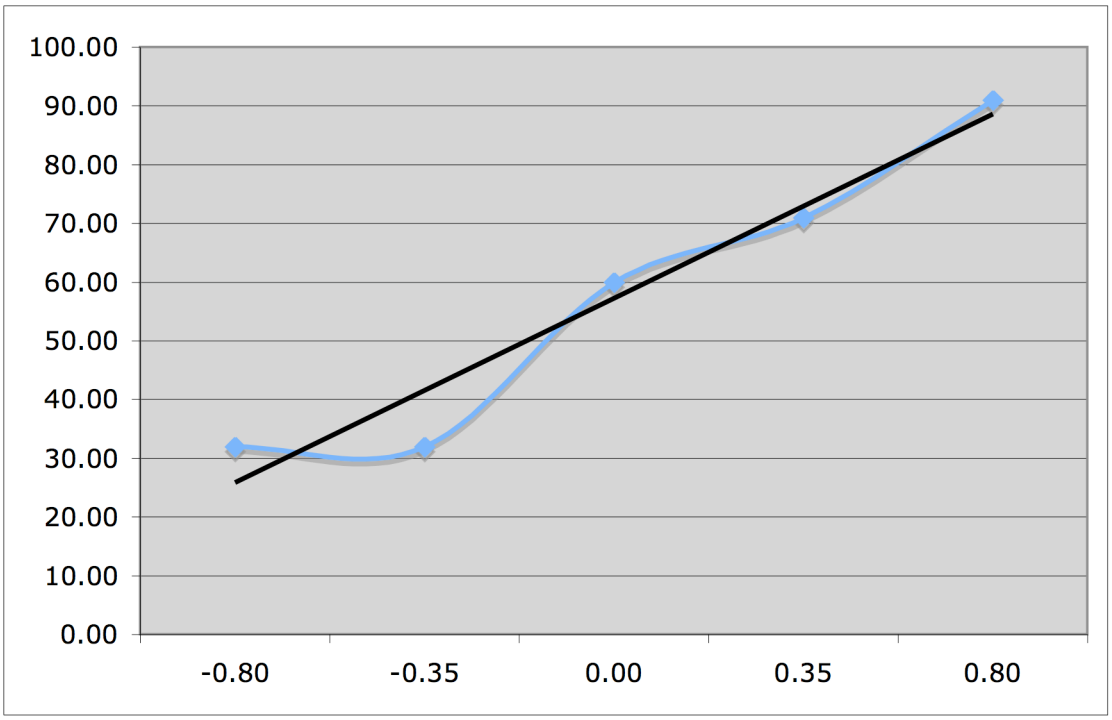


Figure 35: Normal probability plot for Oracle XE 100 tables

Normal probability plots of F-measure results for Oracle XE 500 tables (Table 23) are shown in Figure 36.

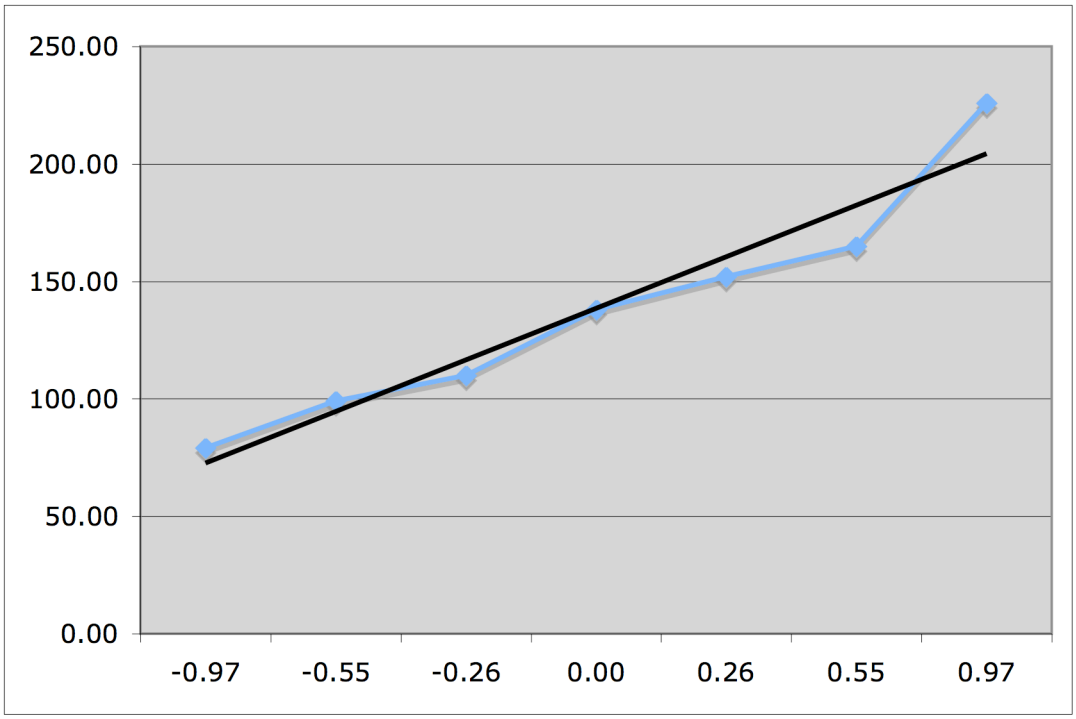


Figure 36: Normal probability plot for Oracle XE 500 tables

Normal probability plots of F-measure results for Oracle XE 1000 tables (Table 24) are shown in Figure 37.

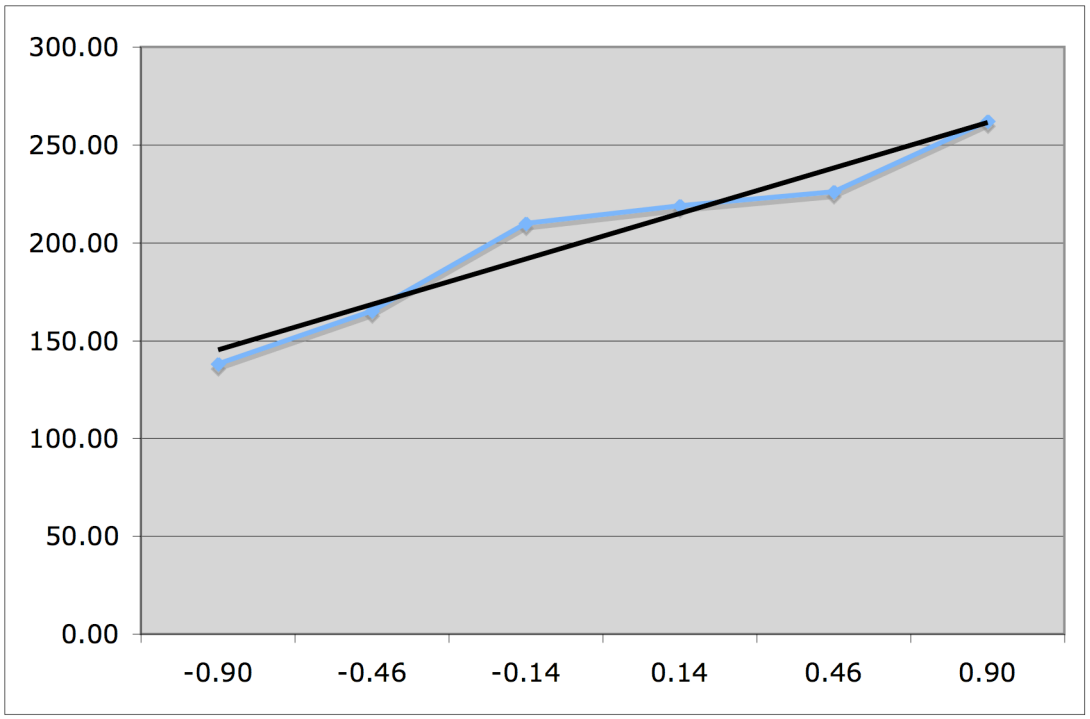


Figure 37: Normal probability plot for Oracle XE 1000 tables