TOWARDS AUTOMATED SOFTWARE TESTING

TECHNIQUES, CLASSIFICATIONS AND FRAMEWORKS

Richard Torkar

Blekinge Institute of Technology
Doctoral Dissertation Series No. 2006:04

School of Engineering

Towards Automated Software Testing

Techniques, Classifications and Frameworks

Richard Torkar

Blekinge Institute of Technology Doctoral Dissertation Series
No 2006:04
ISSN 1653-2090
ISBN 91-7295-089-7

Towards Automated Software Testing

Techniques, Classifications and Frameworks

Richard Torkar

Department of Systems and Software Engineering
School of Engineering
Blekinge Institute of Technology
SWEDEN

Department of Systems and Software Engineering
School of Engineering
Publisher: Blekinge Institute of Technology
Printed by Kaserntryckeriet, Karlskrona, Sweden 2006
ISBN 91-7295-089-7

To my father

The most exciting phrase to hear in science, the one that heralds new discoveries, is not "Eureka!" but rather, "Hmm... that's funny..."
Isaac Asimov (1920 - 1992)

ABSTRACT

Software is today used in more and different ways than ever before. From refrigerators and cars to space shuttles and smart cards. As such, most software, usually need to adhere to a specification, i.e. to make sure that it does what is expected.

Normally, a software engineer goes through a certain process to establish that the software follows a given specification. This process, verification and validation (V & V), ensures that the software conforms to its specification and that the customers ultimately receive what they ordered. Software testing is one of the techniques to use during V & V. To be able to use resources in a better way, computers should be able to help out in the "art of software testing" to a higher extent, than is currently the case today. One of the issues here is not to remove human beings from the software testing process altogether—in many ways software development is still an art form and as such pose some problems for computers to participate in—but instead let software engineers focus on problems computers are evidently bad at solving.

This dissertation presents research aimed at examining, classifying and improving the concept of automated software testing and is built upon the assumption that software testing could be automated to a higher extent. Throughout this thesis an emphasis has been put on "real life" applications and the testing of these applications.

One of the contributions in this dissertation is the research aimed at uncovering different issues with respect to automated software testing. The research is performed through a series of case studies and experiments which ultimately also leads to another contribution—a model for expressing, clarifying and classifying software testing and the automated aspects thereof. An additional contribution in this thesis is the development of framework desiderata which in turns acts as a broad substratum for a framework for object message pattern analysis of intermediate code representations.

The results, as presented in this dissertation, shows how software testing can be improved, extended and better classified with respect to automation aspects. The main contribution lays in the investigation of, and the improvement in, issues related to automated software testing.

ACKNOWLEDGMENTS

First of all, I would like to thank my supervisors *Dr. Stefan Mankefors-Christiernin* and *Dr. Robert Feldt*. Their guidance and faith in me, has helped me many times during the last couple of years. I have yet to learn half of what they know.

Secondly, I would like to take the opportunity to thank my examiner *Professor Claes Wohlin* for his counseling and knowledge in the field of software engineering and scientific writing. His feedback has been appreciated and invaluable to me, both in his role as examiner but also, in part, as supervisor. Of course, I am also grateful that *Blekinge Institute of Technology* allowed me to take on a position as a Ph.D. student.

Indeed, several other colleagues have helped me in my work. *Andreas Boklund* provided many interesting discussions and good feedback, while my co-authors in one of the first paper I wrote, *Krister Hansson* and *Andreas Jonsson*, provided much support and hard work.

Several companies, most notably *WM-Data*, *EDS*, *DotNetGuru SARL* (*Thomas Gil, CEO*) and *Ximian Inc.* (now *Novell Inc.*), as well as colleagues in the open source and free software community, have contributed to this thesis. Without them this thesis would have looked completely different.

Financial support when doing research is essential and for that I thank the *European Union* and the *Knowledge Foundation*. Naturally, I am also indebted to the people at *University West* for allowing me to work at the *Department of Computer Science*, while doing my research.

Many Ph.D. students have had a life before their studies—in my case it was the *Swedish Army*. How to receive and give feedback, how to create a plan (and be prepared that it will change at first contact with the enemy) and how to work in a team, are all things that, believe it or not, have been useful to me during my studies.

A Ph.D. student also, usually, has a life besides the studies—in this case family and friends are important. I will always be indebted to my *mother*, who has supported me no matter which roads I have taken in my life. I thank my *sister* for always being there, even though I have not been there for her always, and my *girlfriend* (soon to be wife I

hope), which I know I have been guilty of neglecting all too many times, especially the last couple of months.

Finally, I would like to thank my *father* who passed away many years ago. He showed me the joy of constantly seeking knowledge, no matter what. I have tried to follow his advice as much as possible. This thesis is in most parts his doing.

TABLE OF CONTENTS

Chapter 1

Introduction

1.1 PREAMBLE

Testing software in an automated way has been a goal for researchers and industry during the last few decades. Still, success has not been readily available. Some tools that are partly automated have evolved, while at the same time the methodologies for testing software has improved, thus leading to an overall improvement. Nevertheless, much software testing is still performed manually and thus prone to errors[1], inefficient and costly.

Today several questions still remain to be solved. As an example we have the question of when to stop testing software, i.e. when is it not economically feasible to continue to test software? Clearly, spending too much time and money testing an application which will be used a few times, in a non-critical environment, is probably a waste of resources. At the same time, when software engineers develop software that will be placed in a critical domain and extensively used, an answer to that question needs to be found.

Next, there is the question of resources. Small- and medium-sized companies are today, as always, challenged by their resources, or to be more precise, the lack thereof. Deadlines must be kept at all costs even when, in some cases, the cost turns out to be the actual reliability of their products. Combine this with the fact that software has become more and more complex, and one can see some worrying signs.

The question of complexity, or to be more precise—the fact that software has grown in complexity and size—is very much part of the problem of software testing. Software

[1] The IEEE definition of error, fault and failure is used throughout this thesis.

systems have grown in an amazing pace, much to the frustration of software engineers, thus making it even more crucial trying to at least semi-automate software testing. After all, a human being costs much money to employ while hardware is comparatively cheap...

However, not everything in this particular area can be painted black. Newer programming languages, tools and techniques that provide better possibilities to support testing have been released. In addition to that, development methodologies are being used, that provide a software engineering team ways of integrating software testing into their processes in a more natural and non-intrusive way.

As such, industry uses software testing techniques, tools, frameworks, etc. more and more today. But, unfortunately there are exceptions to this rule. There is a clear distinction between, on the one hand, large, and on the other hand, small- and medium-sized enterprises. Small- and medium-sized enterprises seem to experience a lack of resources to a higher degree than large enterprises and thus reduce and in some cases remove the concept of software testing all together from their software development process. Hence the reasons for introducing and improving automated software testing is even clearer in this case.

The aim of the research presented in this dissertation is to improve software testing by increasing its effectiveness and efficiency. Effectiveness is improved by combining and enhancing testing techniques, while the factor of efficiency is increased mainly by examining how software testing can be automated to a higher extent. By improving effectiveness and efficiency in software testing, time can be saved and thus provide small software development companies the ability to test their software to a higher degree than what is the case today. In addition, to be able to look at automated aspects of software testing, definitions needs to be established as is evident in this dissertation. Consequently, in this dissertation, a model with a number of definitions is presented which (in the end) helps in creating desiderata of how a framework should be constructed to support a high(er) degree of automation.

The main contribution of this dissertation is in improving and classifying software testing and the automated aspects thereof. Several techniques are investigated, combined and in some cases improved for the purpose of reaching a higher effectiveness. While looking at the automated aspects of software testing a model is developed wherein a software engineer, software tester or researcher can classify a certain tool, technique or concept according to their level of automation. The classification of several tools, techniques and concepts, as presented in this dissertation, implicitly provide requirements for a future automated framework with a high degree of automation—a framework which in addition is presented in this dissertation as well.

This thesis consists of research papers which are edited for the purpose of forming chapters in this thesis. The introductory chapter is organized as follows. Section 1.2 in-

troduces the basic notions used and outlines the frame of reference for this thesis, while the last part of the section presents our particular view of software testing. Section 1.3 presents the research questions that were posed during the work on this thesis. Section 1.4 presents the research methodology as used in this thesis. Section 1.5 presents related work with respect to this thesis. Sections 1.6 and 1.7 cover the main contributions and the outline of the whole thesis respectively. Finally, Section 1.8, lists the papers which are part of this dissertation.

The rest of the thesis is constituted by a number of chapters (further presented in Section 1.7) and ends with conclusions (Chapter 11).

1.2 CONTEXT

The research as put forward in this thesis is focused on analyzing, improving and classifying software testing, especially the automated aspects thereof. Software testing is, in our opinion:

> the process of ensuring that a certain piece of software item fulfills its requirements.

Automation aspects in software testing, on the other hand, is focused on keeping human intervention to a minimum. In order to test a software item's requirements a software engineer first needs to understand the basics of software quality, however, thus we turn our attention to some basic concepts before continuing.

1.2.1 SOFTWARE QUALITY

What are the quality aspects a software engineer must adhere to, with respect to software development? It is not an easy question to answer since it varies, depending on what will be tested, i.e. each software is more or less unique, although most software has some common characteristics.

Some quality aspects, which can be found by examining today's literature (see e.g. [140] for a good introduction), are:

- Reliability—the extent with which a software can perform its functions in a satisfactorily manner, e.g. an ATM which gives $20 bills instead of $10 bills is not reliable [46], neither is a space rocket which explodes in mid-air [178].
- Usability—the extent with which a software is practical and appropriate to use, e.g. a word processor where the user needs a thick manual to be able to write a simple note, possesses bad usability.

- Security—the extent with which a software can withstand malicious interference and protect itself against unauthorized access, e.g. a monolithic design can be bad for security.
- Maintainability—the extent with which a software can be updated and thus adhere to new requirements, e.g. a software that is not documented appropriately is hard to maintain.
- Testability—the extent with which a software can be evaluated, e.g. a bloated or complex design leads to bad testability.

Of course there exist more characteristics, e.g. portability, efficiency or completeness, but nevertheless, if a software engineer in a small- or medium-sized project would, at least, take some of the above aspects into account when testing software, many faults would be uncovered. It might be worth mentioning that going through all these attributes, time and again, is a time-consuming task, but when an engineer has gone through this process several times (s)he will eventually gain a certain amount of knowledge, and thus be able to fairly quickly see which attributes are essential for a particular software item.

However, needing to re-invent the wheel, is something that should be avoided. Fortunately, *The International Organization for Standardization* has collected several of these attributes, i.e. quality aspects that a software engineer could test for, in the ISO 9126 standard [140]. Nonetheless, all of these attributes affect each individual software in a unique way, thus still putting demands on a software engineer to have intricate knowledge of many, if not all, characteristics.

1.2.2 SOFTWARE TESTING

Software testing, which is considered to be a part of the verification and validation (V & V) area, has an essential role to play when it comes to ensuring a software's implementation validity to a given specification. One common way to distinguish between verification and validation is to ask two simple questions [267]:

- Verification—are we building the product right?
- Validation—are we building the right product?

First of all, software testing can be used to answer the question of verification, e.g. by ensuring, to a certain degree, that the software is built and tested according to a certain software testing methodology, we can be assured that it has been built correctly. Secondly, by allowing customers and end-users to test the software currently being

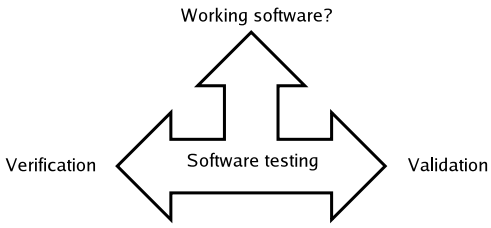

Figure 1.1: Software testing and V & V.

developed, a software development team can ensure that the correct product has been built.

Hence, the way to view software testing would be to picture it being a foundation stone on which V & V is placed upon, while at the same time binding V & V together (Figure 1.1).

Unfortunately, software testing still, by large, inherit a property once formulated by Dijkstra [68] as:

> Program testing can be used to show the presence of bugs, but never to show their absence!

The quote, is often taken quite literarily to be true in all circumstances by software engineering researchers but, obviously, depending on the statistical significance one would want to use, the above quote might very well not be true (in Chapter 4 a different stance regarding this problem is presented).

To sum it up, software testing is the process wherein a software engineer can identify e.g. completeness, correctness and quality of a certain piece of software. In addition, as will be covered next, software testing is traditionally divided into two areas: white box and black box testing. This thesis emphasizes the black box approach, but as the reader will notice, Chapter 10 also touches on the subject of white box testing.

WHITE BOX

White box testing [22, 204], structural testing or glass-box testing as some might prefer calling it, is actually not only a test methodology, but also a name that can be used when describing several testing techniques, i.e. test design methods. The lowest common denominator for these techniques is how an engineer views a certain piece of software.

In white box testing an engineer examines the software, using knowledge concerning the internal structure of the software. Hence, test data is collected and test cases

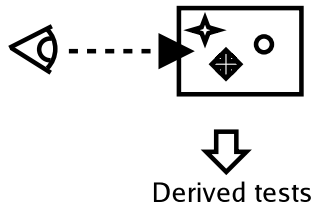

Figure 1.2: The transparent view used in white box testing gives a software engineer knowledge about the internal parts.

are written using this knowledge (Figure 1.2).

Today white box testing has evolved into several sub-categories. All have one thing in common, i.e. how they view the software item. Some of the more widely known white box strategies are coverage testing techniques:

- Branch coverage.
- Path coverage.
- Statement coverage.

Branch coverage [54] is a technique to test each unique branch in a given piece of code. For example, each possible branch at a decision point, e.g. switch/case statement, is executed at least once, thus making sure that all reachable code is executed in that limited context.

Path coverage, on the other hand, deals with complete paths in a given piece of code. In other words, every line of source code should be visited at least once during testing. Unfortunately, as might be suspected, this is very hard to achieve on software that contains many lines of code, thus leading to engineers using this technique mostly in small and well delimited sub-domains e.g. that might be critical to the software's ability to function [120].

Statement coverage's aim is to execute each statement in the software at least once. This technique has reached favorable results [232], hence the previous empirical validation makes this technique fairly popular. It is on the other hand questionable if this particular technique can scale reasonably, thus allowing a software tester to test large(r) software items.

Even though white box testing is considered to be well-proven and empirically

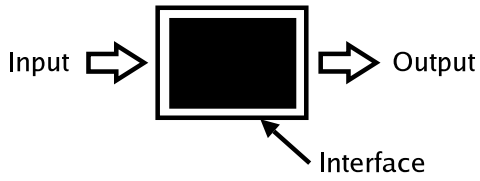

Figure 1.3: In black box testing, the external view is what meets the eye.

validated it still has some drawbacks, i.e. it is not the silver bullet[2] [38] when it comes to testing. Experiments have shown ([16] and later [108]) that static code reading, which is considered to be a rather costly way to test software, is still cheaper than e.g. statement testing, thus making certain researchers question the viability of different coverage testing techniques [102]. Worth pointing out in this case is that a coverage technique does not usually take into account the specification—something an engineer can do during formal inspection [165, 166].

BLACK BOX

Black box testing [23, 204], behavioral testing or functional testing, is another way to look at software testing. In black box testing the software engineer views, not the internals but instead the externals, of a given piece of software. The interface to the black box and what the box returns and how it correlates to what the software engineer expects it to return, is the essential corner stone in this methodology (Figure 1.3).

In the black box family several testing techniques do exist. They all disregard the internal parts of the software and focus on how to pass different values into a black box and check the output accordingly. The black box can, for example, consist of a method, an object or a component. In the case of components, both Commercial-Off-The-Shelf (COTS) [160] and 'standard' [88] components, can be tested with this approach (further clarifications regarding the concept of components are introduced in Subsection 1.2.4).

Since many components and applications are delivered in binary form today, a software engineer does not usually have any choice but to use a black box technique. Looking *into* the box is simply not a viable option, thus making black box testing techniques useful in e.g. component-based development (CBD). In addition to that, CBD's approach regarding the usage of e.g. interfaces as the *only* way to give access to

[2] A silver bullet is according to myth the only way to slay a werewolf and in this case the faulty software is the werewolf.

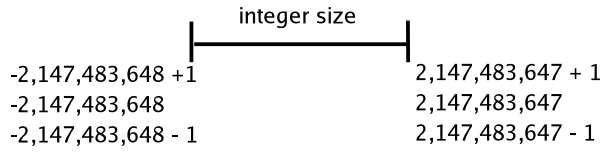

Figure 1.4: Using boundary value analysis to test an integer.

a component, makes it particularly interesting to combine with a black box technique.

Worth mentioning here is the concept of intermediate representations of code (e.g. byte code or the common intermediate language). By compiling code into an intermediate form, certain software testing techniques can be easily applied which formerly were very hard if not impossible to execute on the software's binary manifestation (this concept is further covered in Chapter 10).

As mentioned previously, there exist several techniques within the black box family. An old, but nevertheless still valid, collection of basic black box techniques can be found in [204]. What follows next is a list of the most common techniques and a short explanation of each.

- Boundary value analysis.
- Cause-effect graphing.
- Random testing.
- Partition testing (several sub-techniques exist in this field).

Boundary value analysis (BVA) is built upon the assumption that many, if not most, faults can be found around *boundary* conditions, i.e. boundary values. BVA has showed good results [244] and is today considered to be a straightforward and relatively cheap way to find faults. As this thesis shows (Chapter 2), BVA is among the most commonly utilized black box techniques in industry.

In Figure 1.4 an example is shown on how a simple integer could be tested (note however that these numbers only apply to today's PCs). In other words, by testing the various boundary values a test engineer can uncover many faults that could lead to overflows and usually, in addition to that, incorrect exception handling.

Cause-effect graphing [22, 204], attempts to solve the the problem of multiple faults in software (see multiple fault assumption theory, pp. 97–101 in [146], for an

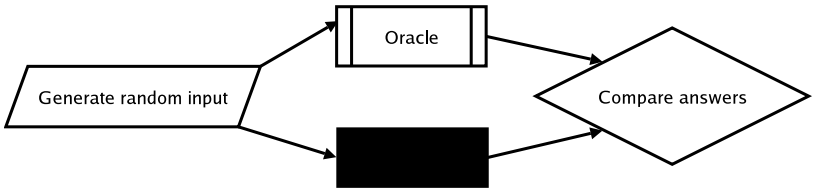

Figure 1.5: Random testing using an oracle.

exhaustive explanation). This theory is built upon the belief that combinations of inputs can cause failures in software, thus the multiple fault assumption uses the Cartesian product to cover all combination of inputs, often leading to a high yield set of test cases.

A software test engineer usually performs a variant of the following steps when creating test cases according to the cause-effect graphing technique [204, 213, 221]:

1. Divide specification into small pieces (also known as the divide and conquer principle).
2. List all causes (input classes) and all effects (output classes).
3. Link causes to effects using a Boolean graph.
4. Describe combinations of causes/effects that are impossible.
5. Convert the graph to a table.
6. Create test cases from the columns in the table.

Random testing [122], is a technique that tests software using random input (Figure 1.5). By using random input (often generating massive number of inputs) and comparing the output with a known correct answer, the software is checked against its specification. This test technique can be used when e.g. the complexity of the software makes it impossible to test every possible combination. Another advantage is that a non-human oracle [81], if available, makes the whole test procedure—creating the test cases, executing them and checking the correct answer—fairly easy to automate. But, as is indicated by this thesis, having a constructed oracle ready to use is seldom an option in most software engineering problems. In one way or another, an oracle needs to be constructed manually or semi-automatically (in Chapter 8 a discussion regarding the generation of oracles, whether manual, automatic or semi-automatic, is to be found).

Partition testing [45, 144, 296], equivalence class testing or equivalence partitioning, is a technique that tries to accomplish mainly two things. To begin with, a test engineer might want to have a sense of complete testing. This is something that partition testing can provide, if one takes the word 'sense' into consideration, by testing part of the input space.

Second, partition testing strives at avoiding redundancy. By simply testing a method with one input instead of millions of inputs a lot of redundancy can be avoided.

The idea behind partition testing is that by dividing the input space into partitions, and then test one value in that partition, it would lead to the same result as testing all values in the partition. In addition to that, test engineers usually make a distinction between single or multiple fault assumptions, i.e. that a combination of inputs can uncover a fault.

But can one really be assured that the partitioning was performed in the right way? As we will see, Chapter 6 touches on this issue.

After have covered white and black box techniques, which software testing traditionally has been divided into, one question still lingers... What about the gray areas and other techniques, tools and frameworks that does not nicely fit into the strict division of white and black boxes?

ON COMBINATIONS, FORMALITY AND INVARIANTS

Strictly categorizing different areas, issues or subjects always leaves room for entities not being covered, in part or in whole, by such a categorization (the difficulty when trying to accomplish a categorization is illustrated in Chapter 8). As such, this section will cover research which is somewhat outside the scope of how software testing is divided into black and white boxes. After all, the traditional view was introduced in the 60's and further enforced in the late 70's by Myers [204], so one would expect things to change.

In this section combinations of white box and black box techniques will be covered (combinations of different testing techniques within one research area, such as black box testing, is partly covered in Chapters 5 and 6). Furthermore, formal methods for software testing will be introduced and an introduction to how test data generation research has evolved lately, will be covered.

First the concept of horizontal and vertical combinations will be introduced. A horizontal combination, in the context of this thesis, is defined as being a combination wherein several techniques act on the same level of scale and granularity. For example, combining several black and/or white box techniques for unit testing, is by us seen as a horizontal combination. On the other hand, combining several techniques on different scale or granularity, e.g. combining a unit testing technique with a system testing

technique (c.f. Figure 1.6 on page 13), is by us considered to be a vertical combination.

Now why is it important to make this distinction? First, in our opinion, combinations will become more common (and has already started to show more in research papers the last decade). While the horizontal approach is partly covered in this thesis (see especially Chapter 6 and in addition e.g. [151]) the vertical approach is not. Different vertical approaches have the last years starting to show up, see e.g. [134, 270, 285], and more research will most likely take place in the future.

Formal methods [1, 5, 35] is not a subject directly covered by this thesis (but formal methods in software testing is on the other hand not easily fit into a black or white box and hence tended for here). Using formal methods an engineer start by, not writing code, but instead coupling logical symbols which represents the systems they want to develop. The collection of symbols can then, with the help of e.g. set theory and predicate logic, be [253]: "checked to verify that they form logically correct statements." In our opinion, formal methods is the most untainted form of Test Driven Development (TDD) [19] and can lead to good results (Praxis High Integrity Systems claim one error in $10,000$ lines of delivered code [253]). On the other hand, there are problems that need to be solved in the case of formal methods. First, does formal methods really scale? In [253] Ross mentions a system containing $200,000$ lines of code, which is not considered to be sufficient for many types of projects. Second, to what extent are formal methods automated? In our opinion, more or less, not at all. The generation of the actual software item is automatic, but the generation needs specifications which are considered to be very cumbersome to write.

Finally, with respect to test data generation, a few new contributions have lately been published which has affected this thesis and most likely can have an impact on software testing of object-oriented systems by large in the future [84, 126, 219, 230] (in addition it is hard to categorize these contributions following a traditional view). Ernst et al. and Lam et al. has lately focused on generating likely invariants in object-oriented systems. A likely invariant is, to quote Ernst et al. [230]:

> ... a program analysis that generalizes over observed values to hypothesize program properties.

In other words, by collecting runtime data, an analysis can be performed where the properties of these values can be calculated to a certain degree (compare this to Claessen's et al. work on QuickCheck [53] where they formally set properties beforehand). By executing a software item, an engineer will be able to, to put it bluntly, generate Δ-values of an existing system being developed.

As an example, suppose a software executes a method twice; with the integer input value 1 the first time, and 5 the second time. Δ in this case (when accounting for the

boundaries) are the values $1, 2, 3, 4, 5$, i.e. we have input values (or likely invariants) for the values 1 and 5. Even though this is a very simple example it might give the reader an idea of the concept (using the definition of Δ, as is done in the example, is not always this straightforward in real life).

The contributions regarding likely invariants by Ernst et al. and Lam et al. is decidedly interesting for two reasons. First, they can generate likely invariants for complex types. Second, by having likely invariants a fully automated approach can be reached (an engineer does not need to formally define properties in advance). In other words, a fully automated approach for testing object-oriented, imperative, software systems might be realizable.

1.2.3 AIMS AND LEVELS OF TESTING

In the previous section several methods and techniques, for testing software, were covered. In this subsection an overview of, and some examples on, the different views of testing are given, thus providing an introduction to and understanding of the various aims that testing can adhere to.

The attentive reader might have noticed that the previous sections did not cover the concepts of *static* [44, 203] and *dynamic* [24] analysis. This is rightly so since in this thesis these concepts are not viewed as testing techniques themselves, but rather as supporting techniques that can be used for testing software. These two techniques approach software analysis in two different ways. In static analysis an engineer does not actually run the program while in dynamic analysis, data regarding the software behavior is collected during run-time. Some of the techniques that can be used in dynamic analysis are profilers, assertion checking and run-time instrumentation (Chapter 10), while static analysis uses tools, such as source code analyzers, for collecting different types of metrics. This data can then be used for e.g. detecting memory leaks [129] or invalid pointers [179].

Apart from the above two concepts (dynamic and static) a system can, in addition, be seen as a hierarchy of parts, e.g. sub-systems/components, objects, functions and a particular line of code. Since a system or a piece of software can be rather large; testing small parts initially and then continuously climb up the pyramid, would make it possible to achieve a reasonably good test coverage on the software as a whole, without getting lost in the complexity that software can exhibit.

In [267] an example of a five-stage testing process is given, which is illustrated in Figure 1.6 (next page). It is important to keep in mind that this is only *one* example and usually a software testing process varies depending on several outside factors. Nevertheless, software testing processes as used today, often follow a bottom-up approach, i.e. starting with the smallest parts and testing larger and larger parts, while the soft-

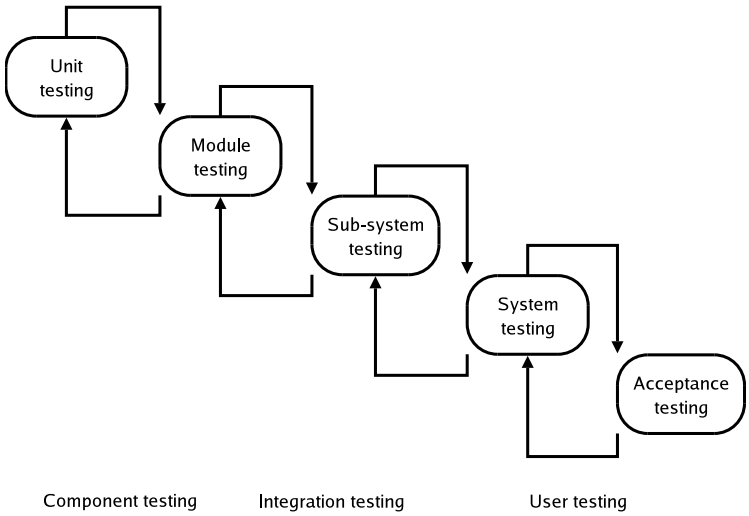

Figure 1.6: Bare-bone example of a testing process taken from [267].

ware evolves accordingly (see [107, 171] for an overview of some of the most common evolutionary software development approaches).

Each and every stage in Figure 1.6 can be further expanded or completely replaced with other test stages, all depending on what the *aim* is in performing the tests. Nevertheless, the overall aim of all test procedures is of course to find run-time failures, faults in the code or, generally speaking, deviations from the specification at hand. However, there exists several test procedures that especially stress a particular feature, functionality or granularity of the software:

- System functional testing.
- Integration testing.
- Regression testing.
- Load testing.
- Performance testing.
- Stress testing.
- Security testing.
- Installation testing.
- Usability testing.
- Stability testing.
- Authorization testing.
- Customer acceptance testing.
- Deployment testing.

In the end, it might be more suitable to use the word 'aspect' when describing the different test procedures. The phrase *aspect-oriented testing* illustrates in a better way, in the author's opinion, that a software test engineer is dealing with different aspects of the same software item.

1.2.4 TESTING IN OUR RESEARCH

This thesis focuses on the black box concept of software testing. Unfortunately the world is neither black nor white—a lot of gray areas do exist in research as they do in real life. To this end, the last part of this thesis takes into account the possibility to look 'inside' a software item. The software item, in this particular case, is represented in an intermediate language and thus suitable for reading and manipulating. Intermediate representations of source code is nowadays wide-spread (the notion of components is supported very much indeed by the the intermediate representations) and used by the Java Virtual Machine [177] and the Common Language Runtime [141].

Indeed, the main reason for taking this approach is the view on software in general. We believe that Component-Based Software Engineering (CBSE) and Component-Based Development (CBD) [88, 160] will increase in usage in the years to come.

Since the word *component* can be used to describe many different entities a clarification might be appropriate (for a view on how completely differently researchers view the concept of components please compare [34, 39, 192, 273]). Components, in the context of this thesis, is [88]:

> [...] a self-describing, reusable program, packaged as a single binary unit, accessible through properties, methods and events.

In the context of this thesis, the word *self-describing* should implicitly mean that the test cases, which a component once has passed, need to accompany the component throughout its distribution life-time (preferably inside the component). This is seldom the case today. The word *binary*, on the other hand, indicates that a white box approach, even though not impossible, would be somewhat cumbersome to use on a shipped component—this is not the case as this thesis will show (in Chapter 10 a technique is introduced wherein a white box approach is applied on already shipped components).

Finally, all software, as used in this thesis, is based on the imperative programming paradigm (whether object-oriented or structured) [235] and can be considered as 'real life' software (and not small and delimited examples constructed beforehand to suite a particular purpose).

1.3 RESEARCH QUESTIONS

During the work on this thesis several research questions were formulated which the research then was based upon. The initial *main* research question that was posed for the complete research in this thesis was:

> *Main Research Question:* How can software testing be performed efficiently and effectively especially in the context of small- and medium-sized enterprises?

To be able to address the main research question several other research questions needed to be answered first (RQ2–RQ10). In Figure 1.7 (next page) the different research questions and how they relate to each other are depicted.

The first question that needed an answer, after the main research question was formulated, was:

> RQ2: What is the current practice concerning software testing and reuse in small- and medium-sized projects?

Simply put, the *main* research question might have been a question of no relevance. Thus, since this thesis is based upon the main research question, it was worthwhile taking the time to examine the current practice in different projects and see how software reuse and, especially, software testing was practiced. The answer to this research question is to be found in Chapter 2 together with an analysis of how software testing is used in different types of projects. To put it short, the answer to RQ2 divided the research, as presented in this thesis, into two areas covering effectiveness in software testing techniques and efficiency in software testing (for a discussion regarding effectiveness and efficiency, in addition to a formal definition of these words, please see Chapter 6, Definitions 6.1–6.2). To begin with, the research aimed at exploring the factor of effectiveness (RQ3–RQ6) while later focusing on efficiency and the automated aspects of software testing (RQ7–RQ10).

In order to examine if the current practice in software development projects was satisfactory for developing software with sufficient quality, RQ3 evolved into:

> RQ3: Is the current practice, within software development projects, sufficient for testing software items?

The answer to RQ3 is to be found in Chapter 3, and provides us with meager reading with respect to the current practice in software projects. Additionally, the answer

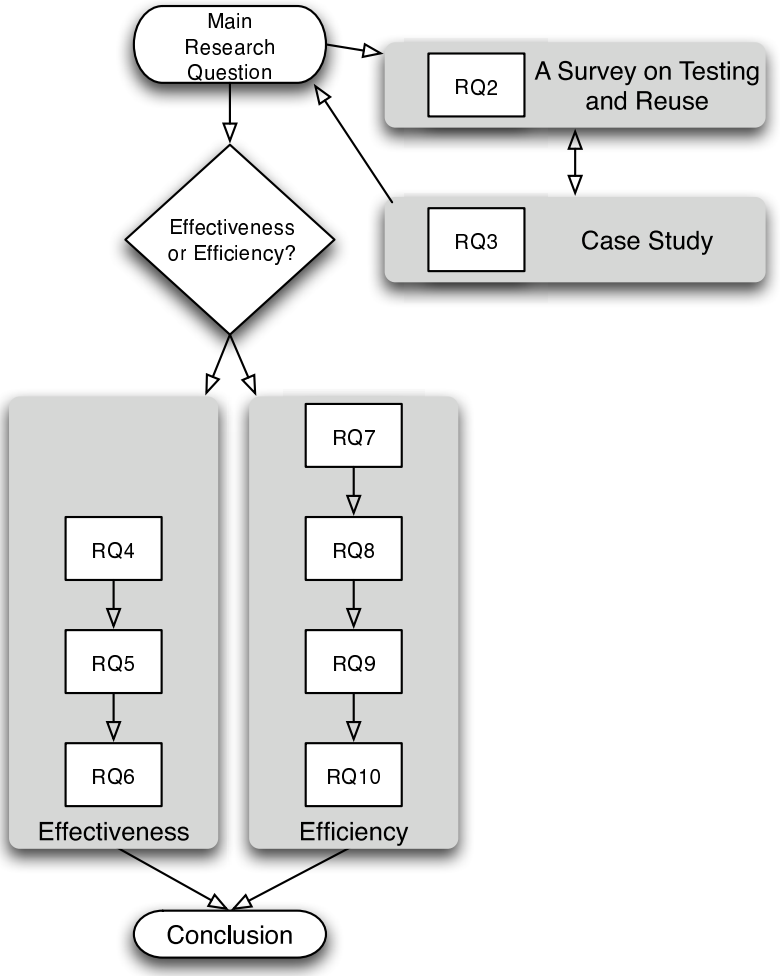

Figure 1.7: Relationship between different research questions in this thesis.

to RQ3 indicated that the answer to RQ2 was correct (regarding the poor status of software testing in many software development projects) and so, in addition, further shows the importance of the main research question.

Since a foundation for further research now had been established several research questions could be posed which in the end would help in answering the main research question.

> RQ4: How can a traditional software testing technique (such as random testing) be improved for the sake of effectiveness?

The answer to RQ4 can be found in Chapter 4 which introduces new kinds of quality estimations for random testing and hence indirectly led to Research Question 5:

> RQ5: How do different traditional software testing techniques compare with respect to effectiveness?

The answer to RQ5 can be found in Chapter 5 which compares different traditional software testing techniques. The comparison in RQ5 eventually led to the question of combining different testing techniques:

> RQ6: What is the potential in combining different software testing techniques with respect to effectiveness (and to some extent efficiency)?

At this stage in this thesis, the focus turns away from the factor of effectiveness and a full emphasis is put on the issue of efficiency. Since RQ2 indicated that there existed a shortage of resources for projects one of the conclusions was that software testing techniques not only need to be *better* at finding faults, but more importantly need to be automated to a higher degree and thus, in the long run, save time for the process' stake-holders.

Thus the following question was posed:

> RQ7: What is the current situation with respect to automated software testing research and development?

The answer to RQ7 gave an approximate view of the status of automated software testing, but nevertheless was hard to formalize in detail due to the complexity of the research area and the share amount of contributions found. To this end, a model was developed which was able to formalize the area of software testing focusing, in the case of this thesis, especially on automated aspects:

RQ8: How can the area of software testing and its aspects be formalized further for the sake of theoretical classification and comparison?

The model with its accompanying definitions (as presented in Chapter 8) which partly is an answer to RQ8, was then used to classify, compare and elaborate on different techniques and tools:

RQ9: How should desiderata of a future framework be expressed to fulfill the aim of automated software testing, and to what degree do techniques, tools and frameworks fulfill desiderata at present?

Finally, the last part of this thesis (Chapter 10) focuses on the last research question:

RQ10: How can desiderata (as presented in RQ9) be implemented?

Chapter 10 provides research results from implementing a framework for automated object message pattern extraction and analysis. Research Question 10, indirectly, provided an opportunity to: a) examine the possible existence of object message patterns in object-oriented software and, b) show how object message pattern analysis (from automatically instrumented applications) can be used for creating test cases.

Before any work on solving a particular research questions starts (a research question is basically a formalization of a particular problem that needs to be solved) a researcher needs to look at *how* the problem should be solved. To be able to do this, one must choose a research methodology.

1.4 RESEARCH METHODOLOGY

First of all, the research presented in this thesis aimed at using examples in the empirical evaluations which were used in industry, especially among small- and medium-sized projects. This, mainly because the research performed will hopefully, in the end, be used in this context. In addition to that, the academic community has endured some criticism for using e.g. simple 'toy' software, when trying to empirically validate theories.

Furthermore, the research as presented in this thesis always tried to have an empirical foundation, even when a theoretical model was developed as in Chapter 8. To focus on the theory only and disregard an empirical evaluation would be, for our purposes, meaningless, especially so when the previous paragraph is taken into consideration. Empirical evaluations, of some sort, were always used as a way to investigate if a certain theory could meet empirical conditions, hence each step in this thesis was always evaluated.

Initially, in this thesis (see Chapter 2), a *qualitative* approach was used with some additional *quantitative* elements (see [14] and [275] respectively). The results from Chapter 2 led us to the conclusion that more must be done by primarily, trying to improve current testing techniques and secondarily, looking at the possibilities at automating one or more of these techniques.

To this end, an *exploratory study* [198] (Chapter 3) was set up where a reusable component was tested in a straightforward way. The aim was to try to show that even basic testing techniques, e.g. unit testing, can uncover faults in software that had been reused and reviewed by developers. At the same time the study gave some indication on the validity of the various aspects of the survey in Chapter 2.

The survey and the exploratory study provided indications that some areas could benefit from some additional research. Thus the following work was conducted:

- An improvement in how a software engineer might use random testing (Chapter 4), hence combining it with other statistical tools.
- A comparative study (Chapter 5) between two black box techniques, i.e. partition and random testing, giving an indication of the pros and cons of the respective techniques.

The research on improvements, with respect to random testing (Chapter 4), was performed using theoretical constructions which later were empirically evaluated, while the comparative study, in Chapter 5, was implemented by empirically evaluating a software item already used by industry. The methodology used in this chapter is common in software engineering research and described in e.g. [229].

Next, in Chapter 6, an empirical evaluation was performed where the following issues were researched:

- Strengths and weaknesses of different testing techniques.
- Combination of different testing techniques.

In the case of Chapter 6, the same type of research methodology was used, as in Chapter 5, but with the addition that the focus was on researching the improved effectiveness of combining several testing techniques.

Chapter 7 is a 'classic' literature study whereas a rigorous methodology is applied for the purpose of outlining automated software testing research. It is appropriate to mention here that the literature study in no way claims to be exhaustive (being exhaustive in this context would most likely require a thesis of its own) but instead attempts to take into account the significant areas of interest.

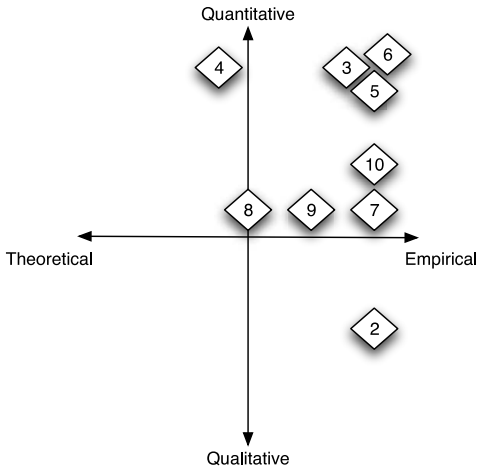

Figure 1.8: Methodologies used in this thesis. The numbers in each diamond corresponds to the chapter numbers in this thesis.

Chapter 8 has a theoretical foundation which is then strengthened via an empirical evaluation. The model, which is introduced in Chapter 8 is used to compare, classify and elaborate on different automated software techniques and then further, empirically, strengthened by the survey in Chapter 9 whereas certain aspects of the model is extensively used.

Finally, in Chapter 10, a case study is conducted where a framework is developed and used on different software items with the aim to extract software testing patterns automatically.

Figure 1.8 provides a rudimentary view of the methodologies as used in this thesis.

1.5 RELATED WORK

Before covering the contributions of this thesis related work will be presented in two parts. To begin with, the most relevant related work covering each individual chapter (Chapters 2–10) will be presented. Finally, related work for this thesis as a whole will be covered.

1.5.1 RELATED WORK FOR INDIVIDUAL CHAPTERS

To begin with, Chapter 2 of this thesis presents a survey. Surveys are performed in software industry on a regular basis (a few on quarterly or annual basis). Related work for this chapter focused on two areas: surveys conducted in industry and open source development. As a consequence two surveys were found to be of particular importance apropos this chapter. First, The Standish Group International's CHAOS reports (all reports can be found here [279]) which focuses on quarterly and annual investigations of the software industry examining trends in e.g. project failures, development costs, project management, outsourcing, etc. and second, the FLOSS reports [139], which solely focus on open source and free software issues. Unfortunately, at the time of writing Chapter 2 (2002), no study could be found regarding software testing in small- and medium-sized enterprises and open source environments, hence leading to the conclusion to conduct a survey on our own.

Related work for Chapter 3 is summed up mainly by three contributions. To begin with, Parnas and Clements' [226] work on stage-wise evaluation of software was considered to be of interest since one of the intentions with the exploratory study, performed in Chapter 3, was to show that the developers relinquished from Parnas and Clements' conclusions, hence indicating that further help was needed in the area of software testing. The second piece of related work, relevant for Chapter 3, was Rosenblum's [252] work on "adequate testing". Chapter 3 clearly indicates that the concept of "adequate testing" is hard to follow and the results can be inauspicious, i.e. the developers did not abide by Rosenblum's conclusion that more should be tested earlier. Finally, this leads inevitably to Boehm's [29] conclusion that the longer a fault stays in a software system the more expensive it is to remove.

Next, related work for Chapter 4 is covered by several contributions dealing with the issue of to what extent a test can be trusted. Frankl's et al. contribution on evaluating testing techniques (directly connected to the issue of reliability) [103] and Williams, Mercer, Mucha and Kapur's work on examining code coverage issues [299], albeit theoretical, should be considered related work to this chapter. In addition Hamlet's many contributions on random testing, dependability and reliability [120, 121, 122, 123, 124] is of interest to this chapter due to its close correspondence to the subject studied.

Chapter 5, presents a comparative study of different black box techniques and examines work done by Ntafos [211] (a theoretical comparison of partition and random testing) and Reid's contribution [244] (an empirical study comparing several different test methodologies). Related work for Chapter 6, which presents research aimed at examining effectiveness issues in combining testing techniques, can equally be summed up by the previously mentioned contributions, but in addition contributions by Boland

et al. [32] and Gutjahr [119] is of interest (they conclude that the picture regarding combinations of testing techniques is substantially complex).

In Chapter 7 a literature study is presented embodying many references in the field of software testing and the automated aspects thereof. Since no such literature study was known to the author (disregarding the automated aspects a few surveys can be found in the area of test data and test case generation, e.g. see [78] and [234] respectively) the focus on related work can in this aspect be summed up, to some extent, by contributions covering the aspects on *how* to perform said studies. In this respect Kitchenham, Dybå and Jørgensen's paper on *Evidence-Based Software Engineering* is relevant since they detail precisely *how* such a study should be performed (e.g. justifying the method of search while discussing the risks associated with the search method).

Chapters 8 and 9 are closely connected to the question of classifying and comparing different software testing techniques, tools and methods. Related work for these chapters is the ISO 9126 standard [140] which covers, in some aspects rigorously, certain quality aspects of software. In addition Parasuraman's and Sheridan's contributions on automatization aspects in human-system interaction—albeit not with a software testing focus—is very much interesting and closely connected to the *geist* of Chapters 8 and 9. In the chapter covering future work (Chapter 11) we once again touch on the issues that Parasuraman and Sheridan have spent time on researching.

Obviously, Fewster and Graham's [93], Kaner, Falk and Nguyen's [150], Poston's [233], Jorgensen's [146] and Sommerville's [267] more traditional and 'simplistic' views on certain aspects of software testing lays as a foundation to Chapters 8 and 9. It goes without saying that the work of these researchers provides a foundation to this thesis.

Finally, Chapter 10, combines related work mainly from three areas. First, the concept of patterns in object-oriented software is covered by e.g. [27, 62, 97, 167]. Nevertheless, as the reader will notice, the word pattern has a slightly different meaning in Chapter 10, compared to how the authors of these publications use it. Second, related work by Ernst et al. [84, 219, 230] and Lam et al. [126] are closely connected to our work. Nevertheless, their work focuses very much on a concept called likely invariants (further discussed below and in Chapters 9–10), while Chapter 10 on the other hand focuses on pattern analysis, which *then* can be used in combination with their contributions. Third, Lewis' work on the omniscient debugger [174] follows the same principle Chapter 10 tries to adhere to, i.e. see-all-hear-all.

In the end, related work for Chapter 10, is favorable represented by Claessen and Hughes' QuickCheck [53], Koopman's et al. work on Gast [157] and Godefroid's et al. paper on DART [111].

The contributions concerning QuickCheck and Gast are interesting in that they accomplish automation in software testing by means of formulating properties to be

obeyed during testing. Unfortunately, both contributions focus very much on declarative languages (i.e. Haskell[3] in this case) and thus leave imperative languages to fend for themselves.

Godefroid's et al. contribution, on the other hand, is interesting in that they focus on static source code parsing of imperative languages (Chapter 10 presents a dynamic approach) and scalar values (Chapter 10 focuses on 'real life' applications with complex data types). In addition, DART is not by us considered to be automatic to the extent which this thesis strives at, thus leading to the next subsection.

1.5.2 RELATED WORK FOR THE THESIS

In this subsection related work covering this thesis as a whole is presented. To begin with, a few books will be presented and the subsection ends by presenting references such as research papers and theses. It goes without saying that finding a thesis which is exactly the same would be pointless—after all a thesis should be unique—nevertheless, in this subsection several references are covered which in part are *aiming* at solving or researching the same issues as this thesis tries to.

There exist many books covering the area of software testing. Unfortunately very few look at *automated* software testing. Despite that, two books are worth mentioning in this context which at least try to look into these issues. First a book by Fewster and Graham titled *Software Test Automation: Effective Use of Test Execution Tools* [94] and second a book by Dustin, Rashka and Paul—*Automated Software Testing: Introduction, Management, and Performance* [73]. Unfortunately, even though these two books are excellent in what they try to accomplish, they still try to accomplish the wrong thing for the purpose of this thesis. Both books cover software testing very much from a project/process perspective (even from the project manager perspective). Alas, leading us to research papers and theses.

Looking at research papers, there are especially three contributions that are closely related to the purpose of this thesis. First, Godefroid's et al. contribution on directed automated random testing [111], second, Yuan and Xie's work on Substra [308], and finally, Kropp's et al. contribution on robustness testing of components [162]. Godefroid's et al. work on DART is examined in Chapter 10, while Kropp's et al. contribution is covered in Chapters 7–9. Yuan and Xie's work on Substra is on the other hand not covered by any chapter in this thesis due to the novelty of their work. Substra is a framework which focuses on automatic integration testing of software (thus falling somewhat outside the scope of this thesis), and have a refreshing view on how one can assemble several testing process steps in one and the same framework (very much in

[3] `http://www.haskell.org`

the same spirit as the framework presented in Chapter 10).

Finally, with respect to theses, especially two are worth mentioning in this context. Xie's thesis titled *Improving Effectiveness of Automated Software Testing in the Absence of Specifications* [303] and Meudec's eminent thesis *Automatic Generation of Software Test Cases from Formal Specifications* [191]. Even though Meudec's thesis has a slightly different focus than this thesis (focusing on formal specifications) the spirit is the same—**automation**. Xie's contribution on the other hand has the same aim as this thesis but unfortunately focuses on generating many test cases and then trying to select the 'right' one from this population—the automatization benefit with this approach is highly questionable since it would imply that a human being must participate actively in the test process on all levels (which is evident by examining [303]).

1.6 CONTRIBUTIONS OF THESIS

First, this thesis presents research on the current usage of different testing techniques in industry today, with a focus on small- and medium-sized enterprises. In addition to that some black box techniques are theoretically extended while at the same time being evaluated empirically. Special consideration is placed upon the future goal of automatically creating and executing different types of software testing techniques.

Second, a formal model with accompanying definitions, for classifying, comparing and elaborating on software testing and the automated aspects thereof is presented and validated in this thesis. The model is used for classifying several techniques and, in addition, desiderata concerning an automated software testing framework is developed with the help of the model. The framework is implemented and validated against several 'real life' software items of considerable size.

Thus, the main contributions of this thesis are (in no particular order):

- A view of the current state of practice in industry.
- A formal model for classifying software testing and the automated aspects thereof.
- A framework for supporting automated object message pattern analysis with the possibility to create test cases.
- Empirical evaluations of several black box techniques.

By looking at the testing techniques in use today, improvements could in the end be made, by:

- Improving different techniques' effectiveness even more.

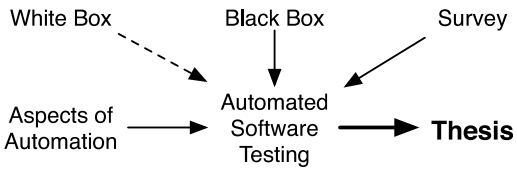

Figure 1.9: Research areas this thesis covers. The dashed line indicates a weak connection.

- Combining several techniques, thus reaching an even higher effectiveness and/or efficiency.
- Creating test cases and test suites automatically and/or semi-automatically by adapting the proposed framework.
- Using the proposed model to classify and compare software testing tools when needed, i.e. in industry to buy the right tool, or for researchers when comparing different tools for the sake of research.

For a general overview of the areas this thesis is assembled around please see Figure 1.9.

1.7 OUTLINE OF THESIS

In Chapter 2, the results from a survey are presented and analyzed. The survey gives the reader an overview concerning the state of practice among software developers and companies. In addition to that, the survey acts as a pointer to different areas and techniques, within the main area of software testing, that are either overlooked or too cumbersome to use today.

Chapter 3 gives an overview on what software testing really means when it comes to software reliability. For example, what types of faults can be found in a component that is already being reused? In addition, the findings in Chapter 2 are found to hold true. Chapters 4 and 5 focus on extending and improving the usage of random testing and partition testing. While Chapter 6 focuses on investigating the combinations of different testing techniques, i.e. partition and random testing.

Chapters 7 and 8 present research covering the development of a formal model with its accompanying definitions. The model is then applied on a number of cases

and desiderata is developed in Chapter 9. Chapter 10 focuses on implementing the desiderata and presents a framework for object message pattern analysis. The last chapter, Chapter 11, contains a summary and conclusion of this thesis as a whole and in addition points out some future research issues which are of interest.

1.8 PAPERS INCLUDED IN THESIS

Chapter 2 is based on an research paper—*A Survey on Testing and Reuse*—published in the proceedings of the *IEEE International Conference on Software—Science, Technology & Engineering (SwSTE'03)*. The survey in Chapter 2 was conducted during the year 2002.

Chapters 3 and 4 were originally published in the proceedings of the *14th International Symposium on Software Reliability Engineering (ISSRE 2003)*. The research presented in these chapters was done in the years 2002 and (early) 2003. The title of the former chapter, when published, was *An Exploratory Study of Component Reliability Using Unit Testing*, while the latter was titled *New Quality Estimations in Random Testing*.

Chapter 5 was originally published in the proceedings of the *Swedish Conference on Software Engineering Research and Practice (SERPS'03)* titled *Fault Finding Effectiveness in Common Black Box Testing Techniques: a Comparative Study*, while Chapter 6 has recently (early 2006) been published in the proceedings of the *The IASTED International Conference on Software Engineering (SE 2006)*.

Chapter 7 was published as a technical report (S-CORE 0501) in 2005 [281]—now pending submission.

Chapters 8 and 9 were written during the latter part of 2004 up until 2005/2006. Chapter 8 has been submitted to the journal *Software Testing, Verification and Reliability* while Chapter 9 has been submitted to *The Journal of Systems and Software*. In addition, Chapter 9 has been rewritten in part to better be in line with Chapter 8 and a revision has been sent to the editors. Chapter 10 has been submitted to the *Automated Software Engineering Journal*.

Several of these chapters have beforehand been published as earlier versions at the *Conference on Software Engineering Research and Practice in Sweden*. In addition, Chapters 2–6 have previously been published in a licentiate thesis titled *Empirical Studies of Software Black Box Testing Techniques: Evaluation and Comparison* [280].

Moreover, an article entitled *Bringing the Monkey to the iPAQ* was published, together with Malte Hildingson, in the August issue (2003) of *.NET Developer's Journal* covering application development on alternative frameworks in handhelds.

Richard Torkar is first author of all chapters except Chapter 4 (a first author has the

main responsibility for the idea, implementation, conclusion and composition of the results). Chapter 4 was written together with Dr. Stefan Mankefors-Christiernin and Andreas Boklund. Richard Torkar was involved in the setup of the experiment as well as investigating and drawing conclusions on the empirical part. He also took part in the theoretical discussions and in writing the paper.

In Chapter 3, Krister Hansson and Andreas Jonsson were co-authors, while Dr. Stefan Mankefors-Christiernin is co-author of all chapters in this thesis except for Chapters 7 and 10. Finally, Dr. Robert Feldt is co-author of Chapters 8 and 9.

Chapter 2

A Survey on Testing and Reuse

Originally published in Proceedings
of the IEEE International Conference
on Software—Science, Technology &
Engineering (SwSTE'03)

R. Torkar &
S. Mankefors-Christiernin

2.1 INTRODUCTION

Software engineering today needs best practices and tools to support developers to develop software that is as fault free as possible. Many tools and methods exist today but the question is if and how they are used and more importantly in which circumstances they are (not) used and why. This survey provides an account of what type of trends exist today in software reuse and testing. The focus was to try to find out how developers use different tools today and what tools are lacking, especially in the field of reuse and testing. The population came from different types of communities and organizations, to better give us a generalized picture of today's developers.

A few surveys in this field, e.g. The CHAOS report by The Standish Group [279] which covers software failures in industry and recently (Aug. 2002) the FLOSS survey by International Institute of Infonomics, University of Maastricht, The Nether-

lands [139], which is a survey/study about developers in the open source [218] and free software [110] world, do exist. But they either focus on a specific population, with its advantages and disadvantages or cover the result of not testing ones software enough.

We believe that there is a need for a more integrated test methodology together with the traditional configuration management process in order to improve the current situation. This chapter covers a survey that took place during late 2002, with the aim to answer some of the questions our research team had with respect to testing and reuse, two areas not usually covered very well in surveys. We wanted to know to what extent reuse was taking place and how frequently (reused) code was being tested.

In our survey we asked software developers from several companies, both national (Swedish and American) and multinational, as well as open source developers from several projects about what type of problems they faced daily in their work. Not surprisingly the answers varied, but many developers gave us the same basic feedback—the systems designed today are complex and the tools for creating these systems are getting more and more complex as well. This indicated that software developers could, among other things, benefit from more integrated and automated testing in today's software development projects.

Yet, other questions in this survey, focused on reuse and testing of re-usable components and code. We wanted to know to what extent reuse was taking place today in projects, how developers test this type of code and if they use some sort of certification. Unfortunately, this [testing reused code] was uncommon among the developers in our population.

All questions discussed in this chapter can be found in Appendix A (pp. 209–216).

2.1.1 BACKGROUND

Many texts today exist concerning the area of testing. *Testing Object-Oriented Systems* by Binder [27], *The Art of Software Testing* by Myers [204] and *How to Break Software* by Whittaker [298] all give a good insight. For more information about testing—especially unit testing—we recommend reading the *IEEE Standard for Software Unit Testing* [138] and *An Exploratory Study of Component Reliability Using Unit Testing* (Chapter 3).

Recently the International Institute of Infonomics at University of Maastricht in the Netherlands [139] published a report (FLOSS) that covered a survey about open source and free software in Europe. This study is interesting in many ways, especially so since some of the questions in their survey touch the areas of software development and software in general. Part IV and V of the FLOSS study [139] is partly being compared to the survey that was carried out and described in this chapter.

2.2 METHODOLOGY

The research upon which this chapter is based on, was done in five separate phases. The first phase was to gather good candidate questions that we considered important. The second phase consisted of selecting the questions that were of most interest (unfortunately very few developers want to answer 350 questions). The third phase consisted of selecting the population to use for the survey, and finally in the two last phases we established how the questions should be asked and answered and put together additional questions that were not asked in the first questioning round.

The research method we followed during this research was a survey approach [14]. We wanted to conduct a survey that would point out areas that software developers found especially weak and in need of attention. We used empirical inquiries from slightly different populations (open source vs. business) to better examine reuse and testing in today's software projects.

One of the disadvantages of a survey is its time factor. It takes time to prepare and it steals time from the population answering the researcher's questions. Gaining access to different company employees, to answer our questions, proved to be the greatest obstacle during this research project.

Another threat to a survey can be the relationship between the questioner and respondent, in our case we estimated this to non-significant as explained later.

Since this survey aimed at explaining to what extent and how, reuse and testing was used today, we chose different organizations and type of developers. The main reason for this was that we wanted to make sure the problems we saw when analyzing the answers were in fact problems that more or less all developers—regardless of company or project—found in their daily work.

Since time was a critical factor it meant that a qualitative approach, e.g. interview, was out of the question. The geographic distribution of the population also indicated that we should not use a qualitative approach, even though telephones etc. can be used. A quantitative approach was also considered to be the best method, in our case, to more easily draw conclusions in a statistical manner. One must, however, add that a qualitative method probably would have given us a richer set of data on which to base our conclusions upon.

By following the advice in [14, 307] concerning pretests, a first testing round was carried out with four developers participating. Thus we could be relatively confident the questions were of the right type and properly formulated.

The survey used self-administered questionnaires [14], with the addition to a web-based approach. This, in combination with our quantitative approach, made us sure that we did not influence our respondents in any way.

The total number of developers contributing to the survey, during late 2002, was

91 (a further four developers were asked but had no time to participate). Of these 91 developers approximately 43% were from the open source and free software development community and 57% from three different companies; one multinational (approx. 100,000 employees), and two national; one Swedish (approx. 20 employees) and one American with approximately 100 employees. All respondents were either business contacts which have been gathered over time or companies participating in adjacent research projects.

When the survey finished, the answers were checked and if any ambiguous answers were found, the survey participant was contacted and additional questions were asked in order to avoid misinterpretations.

It was stressed, at the introduction of the survey, that the respondent should answer all questions with question number 4 in mind (Appendix A, page 209).

2.3 PRESENTATION

The results are presented in three categories which are discussed; one brief section with general questions and two in-depth sections on reuse and testing. As mentioned previously, all the questions relevant to this chapter, are found in Appendix A (page 209).

The general questions cover areas such as which development environments or development kits are being used, and the reuse category covers the area of component and general code reuse with accompanying test procedures. Finally, the test category covers questions that more specifically involve different test methodologies and best practices.

2.4 RESULTS AND ANALYSIS

The results and the corresponding analysis is divided into three parts. First, general questions as posed in the survey are covered. Second, questions regarding reuse in software development are presented. Finally, questions formulated with the intent of uncovering issues regarding software testing are attended to.

2.4.1 GENERAL QUESTIONS

Of the survey participants 55% had an educational level of M.Sc. or higher and only 15% had a high school education or lower. The former number differs from the FLOSS study where only 37% had an educational level of M.Sc. or higher (question 1).

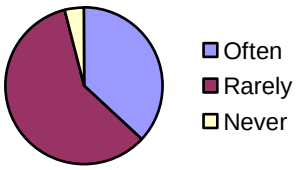

Figure 2.1: Moving deadlines.

The above variance can be explained by two factors; our survey having a larger degree of developers from the business world as opposed to the FLOSS study Part IV [139] which only focused on open source/free software developers and the fact [139] that open source developers are younger in general. This was confirmed in question number three (Appendix A) which indicated that the population we used had a higher average age (96% over the age of 21). These numbers could, simply put, mean that we had a more mature population in the sense of working experience and educational level.

One of the general questions covered the usage of large frameworks—when asked about different development platforms, such as .NET [277], Enterprise Java [95] and CORBA [33], the majority preferred Java and in some cases even CORBA as opposed to .NET. The main reason was that developers felt .NET being too immature at the moment (mid-2002). Even so the usage of .NET and CORBA was now equal with $\sim 25\%$ each. More recent studies show .NET gaining even more momentum [86], and there is a high probability that .NET will be used even more in the future since it is backed by some major interests in industry (question 29).

The question "How often do you move deadlines in projects?" (question 11) clearly showed one of the biggest issues in today's software projects (Figure 2.1). The silver bullet has undoubtedly not been found yet.

With 37% of the developers still moving deadlines often and 59% moving them rarely there is still room for improvement. According to [279], 9% of the projects in larger software companies are on-time and on-budget, in our case we have approximately 4% on-time.

Over 53% of the developers (question 17) claimed that the majority of the projects they took part in encompassed more than $10,000$ lines of code. According to the FLOSS study (Part V) [139] the mean value is $346,403$ bytes [of software source code] in the average project. This indicates that the different communities correlate rather

well.

As we have seen during the analysis of the general questions not much differs, from other studies conducted in the area of software engineering. This could indicate that we had gathered a good sample population that could answer question on reuse and testing (reflecting the average developer), despite us having a smaller population than the FLOSS study. We believe that the validity of the larger FLOSS study with its focus on open source and free software can, in many ways, be generally applicable for the business world.

2.4.2 REUSE

As mentioned previously, the amount of reuse in combination with testing was one of two areas we wanted to focus on since we have not found any surveys covering this area. Nevertheless, some of the surveys that at least touch this subject are [117] and [201], but they either cover success and failure examples of reuse in small and medium size enterprises or a particular feature [software repositories] of reuse. Another paper [249] covers large enterprises which are considered to be successful in the area of software reuse.

The developers were asked several questions with regard to reuse in software engineering. Both component-based reuse and clean code reuse (i.e. cut and paste). Here one could clearly see a distinction between open source developers and developers from the business world. Almost 53% of the developers said that they usually had some element of reuse in their projects (question 31). But sadly only five of these developers were from the business sector. One of the main reasons for this, we found out when asking the respondents, was that consultants usually do not own the code—the customer who pays for the work owns the code. This, naturally, makes it harder to reuse code later on.

Only 36% of the developers actively search for code to be reused (question 15). The low number is not strange when one considers that developers creating components almost never certify them in any way (neither in-house or commercially, e.g. [96]). Only 6% use some sort of certification on a regular basis (question 25).

When it comes to buying components the developers were asked if size or complexity matters the most (questions 32-33), 26% of the developers were of the opinion that size did not matter at all. The complexity aspect of reuse is what makes some developers see a great advantage. Unfortunately, most developers found that components performing complex tasks were hard to find. The reason for this, many developers claimed, is probably that these types of components usually contain business logic made specifically for one company.

2.4.3 TESTING

On the question if the developers tested their software, 52% answered yes and another 34% answered that they sometimes test their code (question 19). Unit testing is without a doubt the most used test methodology with 78 out of 91 developers (question 20) using it to some extent. Chapter 3 provides a good indication on the benefits of said testing. Open source developers and developers from the business world test their code equally according to our survey.

When asked about a unit testing example (question 21), which basically consisted of a simple method, the most common approach ($> 40\%$ of the developers) tested extreme values only, i.e. boundary value analysis [204]. A few developers ($\sim 20\%$) tested random values and almost a third of the developers did not test such a method at all ($> 30\%$). The concept of boundary testing seems to be known, both in industry and in the open source world amongst developers, in general. Even though boundary value analysis only catch some of the faults, it is still encouraging to see that at least this basic test technique is being used, to some extent.

Most developers in this survey used some sort of a testing framework which they themselves did not develop (questions 26-27). A majority of the developers testing their code used some of the unit testing frameworks that exist today, most notably some variant of JUnit [20].

As we showed previously only 4% of the projects the developers took part in were on-time. Sadly these respondents usually did not test their software in any way but instead waited for customer feedback as a form of quality assurance. On the other hand 60% of the developers claimed that Verification and Validation (V & V) was the first thing that was neglected (question 36). This was mostly common in the business world, unmistakably so since open source developers usually do not have such strict time frames. Could this lead to higher quality in open source software? Some indications exists that this might be the case [98, 245].

Almost 53% of our respondents stated that they had some element of reuse in their code but only 34% of these 53% claimed that they tested the reused code in any way (c.f. Figure 2.2) (questions 24 and 31).

The reason for this was primarily that they found writing test cases afterwards too tedious. One developer said that while the component was not certified in any way the developers creating it should have tested it. The same developer believed that testing software was more or less unnecessary since customer feedback would give him that advantage anyway.

One thing was consistent with all developers. They all wanted better tools for writing tests cases, especially when the code had been written by someone else. Many of them ($\sim 75\%$) also felt that even though they had written test cases they could still

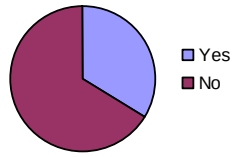

Figure 2.2: Developers with elements of reuse in their projects that also test the reused code.

not be certain that the tests were good enough (question 37). They, simply put, wanted some statistics on how well their tests were written and how well they tested a given method, class or code snippet (e.g. code coverage [299]).

Developers preferring Java had the highest element of testing in their work—this could be explained by developers having the knowledge of JUnit [20] which is considered to be a mature unit testing framework (questions 26-27 and 29). The only discrepancy to this was developers in the open source world, they used several different frameworks [147, 212].

Most developers thought that unit testing was tedious and many times not worth the time spent ("What is the main disadvantage with the test methodology you use?"; question 38). As an alternative test methodology, they focused primarily on testing the software under simulated circumstances as soon as possible e.g. a variation of acceptance testing [266]. We find this to be an alarming sign since unit testing is considered, by many, as being the first vital step in the testing process. By neglecting unit tests many, much harder to find, failures might emerge later on.

If we make a comparison between open source and business, we can see that open source developers in general have a better knowledge of which type of frameworks exist for software testing (question 27); they could, in general, mention several more frameworks they used for covering their needs.

Furthermore, if we combine questions 4, 7-8, 10-11 and 34 it implicates, not surprisingly, that developers in industry, in most cases, have less freedom, higher workload and a lack of time. The lack of testing in combination with reuse could be explained by developers claiming that V & V is the first thing being diminished in a project (question 36).

2.5 DISCUSSION

Before we continue, it might be worth mentioning that, our survey had approximately $\pm 5\%$ of pure statistical errors, while the FLOSS study with its large number of participants, probably ended up with a $\pm 1\%$ error margin. These numbers are to be considered worst case scenarios but, nevertheless, must be taken into consideration when comparing results throughout this chapter. Even so, we find our numbers being more representative for our purposes than the FLOSS study. We needed a broad population with different backgrounds (business and open source), while the FLOSS study concentrated on open source/free software developers only—with no focus on reuse and testing on the whole.

The results from this survey are in line with other surveys conducted before, when comparing general questions. Some discrepancy exist which can largely be explained by other surveys and studies having a larger contribution from the development community in terms of participation, as well as a different mix of business and open source.

Concerning testing we do not have much to compare with, this is one of the few surveys as of now that have such a strong focus on testing and reuse. Some papers cover some aspect of reuse, as already mentioned, while other [99] cover a combination [quality and productivity].

Simple tools for developing software are still widely used. This is explained by the respondents as being simpler to use while at the same time letting the developers keep the control over their code (question 18). This might also indicate why developers find it tedious to learn new test methodologies and tools—Keep It Simple, Stupid (KISS)—is still very much viable in today's projects. It gives us a hint that whatever tools, that are introduced to developers, must be kept simple and easy to learn. The best tool would be one that the developer does not even notice.

With respect to different development platforms that are in use today—Enterprise Java is holding a strong position. This could very well change soon since already 23% of the developers find (in late 2002) .NET being their primary choice. We believe that this number will rise even more and that this will be one of the two target groups [of developers] where simple and automatic tools could come to the rescue. The second group, of course, being the Java developers.

Developers seem to reuse code to a fairly high extent, but unfortunately they do not test the reused code much. In Chapter 3 we show what the effect of such a behavior might be, especially regarding software that is reused a lot.

In this study, we see developers asking for help to test code that is about to be reused. In addition to that, many developers would like to see a tool that could give them an indication on how well their test cases are written (e.g. test coverage)—again KISS.

Developers today need to have a better knowledge on the importance of unit testing. If the foundation which certain software lies upon is not stable, by not using unit tests, then it risks deteriorating everything. Since the workload is high and deadlines creep even closer, developers must be presented with more automated tools for test case creation and test execution. Also tools that give developers an indication on how well the tests cover their software are wanted.

What we found somewhat surprising is the low level of component/library certification taking place. We believed that certification of software had evolved further—beyond academic research [259, 287]. This was not true except for a few cases.

2.6 CONCLUSION

In short, to summarize it, some of the key findings in our survey were;

1. developers reuse code but do not test it to the extent we expected,
2. simple to use tools are lacking when it comes to test creation and test analysis,
3. knowledge on the importance of testing in general and unit testing in particular seem low and,
4. certification of software and/or components seem to be more or less non-existent.

Since 96% percent of the developers are exceeding their deadlines, at least occasionally, one can claim that there is still much room left for improvements.

Chapter 3

An Exploratory Study of Component Reliability Using Unit Testing

Originally published in Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE 2003)

R. Torkar, K. Hansson, A. Jonsson & S. Mankefors-Christiernin

3.1 INTRODUCTION

The use of Commercial-Off-The-Shelf (COTS) software components has increased over the years. The continued success of COTS components, however, is highly dependent on the reliability of the software at hand. In the previous chapter, one of the key findings was that developers reuse components, but they seldom test software before incorporating it in the implementations, especially unit testing is seldom used in this context. At the same time the majority of the developers did not test the compo-

nents during original development either (Chapter 2), hence leading to a paradox of untested software being used again and again.

We do not believe that components and code in general are tested well enough. This makes, in some respect, component-based development (CBD) a potential nightmare. According to Parnas [226], every software product should be evaluated before being used at any later stage in the development process, something that is only partly done. If we are to really succeed in component-based software engineering and reuse in general, we must make sure that developers test [252] their code even more than they currently do. This to ensure that any faults in the product are detected as early as possible, and more importantly, is not 'inherited' with the use of COTS components. Boehm [29] pointed out already 20 years ago that the longer a fault stays in a software system the more expensive it is to remove.

In this chapter we report on an explorative case study on a component already in use in the software community, applying unit testing to it as a third party developer would (should) have, before incorporating it. By doing this, we want to, in a practical case, investigate the reliability of an actual component already in use. In order to try to choose a component that is relatively representative of 'high reliability components' in terms of expected frequent reuse, we tested a core class component (`System.Convert`) in the Mono framework [208]. Since the framework will be a foundation for potentially thousands of applications using it in the open source world, any undetected fault will have very severe repercussions. The component at hand was furthermore already to some extent subsystem and system tested, deemed reliable and reused. No unit tests had to our knowledge been applied, however.

Different persons from the ones actually implementing the class or method, closely mimicking the situation of a developing team testing a COTS component before re-using, wrote all tests. Using a straightforward unit test approach we tested all available methods in the class, finding a total of 25 faults.

We find that even applying a straightforward basic suite of tests to a component before re-using it is of interest to the developers, as well as extra test cases performed after the formal development of the software. The remaining parts of this chapter are devoted to the technical background, results, analysis and the broader scope and impact of our findings.

3.2 BACKGROUND

Software Verification and Validation (V & V) intends to answer two basic questions. Are we building the product right and are we building the right product? In our case: is the product being built, conforming to ECMA International's specifications 334 [76]

and 335 [77]? The former being the C# Language Specification and ECMA-335 being the Common Language Specification, as submitted to the ECMA standardization body by Microsoft, Intel, Hewlett-Packard and Fujitsu Software in December 2001.

These two standards are likely to have a great impact on COTS, CBD and reuse in general the next couple of years. Thus, a need to make sure that the foundation whereas several thousands or even tens of thousands of application will be built upon, is stable. ECMA-334 is further considered to be a standard, which has clear component-based aspects in it and combined with ECMA-335 in conjunction with the framework library, gives the future developer a platform with which (s)he can reuse large parts. The framework is in other words a large collection of components that can and will be reused. Hence, the reliability of these fundamental components must be high.

The component that was tested in this study came from the Mono project [208]. Mono is an open source version of .NET [277], which was hosted by Ximian Incorporation (now Novell Inc.). The goal for Mono is to provide several pieces of components for building new software, most notably a virtual machine, class library and compiler for the C# language.

3.2.1 UNIT TESTING

Unit testing is a well-known technique [38] and has increasingly been used in the last couple of years, especially since the arrival and success of object-oriented languages, such as Java, C++ and more recently C#. Lately also development processes such as XP has made unit testing a closely integrated part of the development. Furthermore, the survey in Chapter 2 shows that unit testing is one of the most common software testing technique used by software developers today.

In unit testing [138] the smallest piece of code (unit) is checked to ensure that it conforms to its requirements. These tests are written once and used many times, during the development cycle, to ensure that code regression is kept at a minimum. Usually the tests are written in the same programming language, which is used to build the software itself. Unit testing should not be used to test relationships or coupling in an object-oriented framework. If that is what one would like to test, then other sub-system testing techniques do exist [290].

3.3 METHODOLOGY

Taking the starting point in the difficulty a developer has in reusing a software component from a third party, we apply a straightforward unit testing scenario. We also assume the software development taking place within the Mono framework, the open

source implementation of .NET, as supposedly being one of the most component- and reuse-oriented platforms today.

As mentioned in the introduction we needed a fairly large class to use in our study. We evaluated several and finally chose the `Convert` [75] class in the `System` namespace. The main reason for choosing this class was its significance and its large number of methods, which would be in need of testing, before using the component in an application. The class provides all standard methods for converting a base data type to another base data type in the framework. This is a typical task delegated to a framework or library in most applications, handling e.g. conversions between hexadecimal and decimal numbers or integers to strings. Hence, possible failures in this class would affect base functionality in a vast number of applications relying on this framework. The namespace `System` also indicates that this class is a core part of the framework.

Assuming the typical limited resources allocated for testing in software development projects (Chapter 2) we chose to only implement a basic suite of test cases. We did not strive, in any way, towards completeness in test coverage, the reason being that we set out to show that even a very basic suite of tests still could find faults in a widely used part of a framework. The basic test cases we are referring to in this case, consisted of testing the boundary conditions and off-nominal cases in which this component should degrade gracefully, without loss of data. Finally, some random input was also carried out on each method being tested.

Since the tests in our case derived from the knowledge of the specification and to some extent, structure of the class(es), a black box approach with some elements of white box testing [149], was used. The tests written had only one objective in mind and that was to find flaws in the implementation according to the specification.

Several tools are available to a developer when performing unit tests of the type mentioned above. Most notable is JUnit [20], which is described by Gamma and Beck as being a regression testing framework and is Open Source [241]. Since JUnit is open source, other developers can port it to different languages. NUnit [212], by Philip Craig, is such a port.

Several programming languages are supported by NUnit, but in our case the C# programming language was the most important. The NUnit framework consists of several classes. These classes contain methods, which the developer uses when constructing test cases.

To compare resulting values, which is very often the case, different `Assert` [182] methods were used in this study. Especially the `AssertEquals` method was used extensively, since if used correctly generated a message that makes it easier for the developer to establish exactly which test case failed.

Listing 3.1: `AssertEquals` example.

```
1 AssertEquals("ID", expectedObject, receivedObject);
```

In Listing 3.1, when the expected object is not the same as the received object, an exception is thrown. The exception includes the value of the expected/received objects and the test ID, so that the developer can easily see where and why it failed (Listing 3.2).

Listing 3.2: Example error message.

```
1 AssertEquals("#A00",(int)2,(short)2);
2 TestChangeType(MonoTests.System.ConvertTest)
3         :#A00 expected:<2> but was:<2>
```

The reason the test failed (Listing 3.2) was that even though the value was equal, the type was not. Notice how the method `ChangeType` is being tested by the method `TestChangeType`. A `Test` prefix is added to a test method so that it will automatically be included into the testing framework when being run the next time.

It is not uncommon to write several tests that manipulate the same or similar objects. To be able to do this in a controlled environment a common base must be established. This base, also known as the fixture, makes sure that the tests are run against a known and well-established foundation (Chapter 8 elaborates on the different entities of a typical software testing technique). The next step is to create a subclass of `TestCase`, add an instance variable for each part of the fixture, override `SetUp()` to initialize the variables and finally use `TearDown()` to release the resources one allocated in `SetUp()` (see Listing 3.3).

Listing 3.3: A simple test case.

```
1 public class ConvertTest : TestCase {
2   bool boolTrue;
3   bool boolFalse;
4   [...]
5   protected override void SetUp() {
6     boolTrue = true;
7     boolFalse = false;
8   [...]}
9 [...]}
```

Once the fixture is in place (Listing 3.3) the developer can write many tests manipulating the same units. If the developer wants to run several tests at the same time, the NUnit framework provides the developer with the class `TestSuite` which can execute any numbers of test cases together.

3.3.1 UNIT TESTING OF SYSTEM.CONVERT

As already mentioned previously, the class `Convert` in the `System` namespace, was selected for a number of reasons. The `System.Convert` class consisted of one public field and 22 public methods, all in all 2,463 lines of code (LOC). Furthermore, each overridden method should be tested to ensure progressive reliability.

The routine for constructing the test method was easily established. First, the specification was read carefully; secondly, boundary, off-by-one and at least one legal input value test was written for each method belonging to `System.Convert`, and finally the tests were run. This process was repeated several times until all methods had tests written for them that covered all contingencies. To ensure all test's integrity we implemented and executed them under the .NET framework [277] before applying the test cases within the Mono framework.

A unit test made for `FromBase64CharArray`, a method which converts the specified subset of an array of Unicode characters consisting of base 64 digits to an equivalent array of 8-bit unsigned integers, will illustrate the principles of the general methodology. The method takes three arguments, the `inArray`, the `offset` (a position within the array) and the `length` (number of elements that should be converted). The array `inArray` is only allowed to consist of the letters A to Z, a to z, numbers 0 to 9 and +,/. The equal sign, =, is used to fill empty space. To make sure that the conversion was correctly made the result and the expected result, both arrays, must be looped through and compared (Listing 3.4).

Listing 3.4: A test comparison.

```
1  for(int i=0; i<result.length; i++)
2          AssertEquals("#U0" + i, expectedByteArr[i], result[i]);
```

The next two examples are test methods for `ToBoolean`. `ToBoolean` is overridden 18 times in `System.Convert`, one for each built-in type, twice for `Object`, twice for `String` and once for `DateTime`. Since the different examples are quite similar only `Int16` and `Char` will be covered. In Listing 3.5, `Int16` is tested; if it is anything but zero it will be converted to true.

Listing 3.5: Testing `Int16`.

```
1  AssertEquals("#D05", true, Convert.ToBoolean(tryInt16));
```

Next the `Char` example, shows that testing exceptions is just as easy. Since a conversion from `char` to `bool` is not allowed an exception should be thrown (Listing 3.6).

Listing 3.6: Catching an exception.

```
1 try {
2         Convert.ToBoolean(tryChar);
3 } catch (Exception e) {
4         AssertEquals("#D20", typeof(InvalidCastException), e.getType());
5 }
```

The test cases written are thus fairly straightforward and test every method's input and output while the specification decides the legality of the outcome of each test.

3.4 RESULTS

By using the described unit testing approach all in all 25 faults were discovered. The test case consisted of 2,734 LOC while the tested class holds 2,463 LOC. This is more or less a 1:1 ratio between class LOC and test LOC, which is considered as being the default in XP [66].

This result in itself clearly indicates the severe problem of reliability in reusable components. That the findings occur in a core class in a framework makes this point even more severe. Virtually any type of fault in such a class could be expected to lead to failures occurring in a wide range of applications. Hence, all the found faults have a very high degree of severability. Because of the nature of the class at hand, i.e. being a core component in a framework in use, the relative reliability is extensively impaired by even a single or a few faults.

Turning to the technical details of the test cases, we cover a few examples fully, before continuing with a summary, in order to keep focus on the general component reliability rather than the individual fault.

Some of the faults detected were clearly the result of a misinterpretation as can be seen in Listing 3.7.

Listing 3.7: A simple fault depicting a misinterpretation of the specification.

```
1 short tryInt16 = 1234;
2 Convert.ToString(tryInt16, 8);
```

In Listing 3.7, the code snippet should, according to the specification, convert the short value 1234 to an octal string, i.e. 2322. What really happened was that the value 1234 got parsed as an octal value and converted to 668. This could easily be proved by changing `tryInt16` to 1239, since the octal number system does not allow the number 9. The result in Mono was now 673, clearly wrong since a `FormatException` should have been thrown. We find it a bit strange that no developer had reported run-time failures of this kind when using the `Convert` class.

Yet another test case discovered a flaw in how hex values were treated (Listing 3.8).

Table 3.1: Overview of results. Faults in italic belong to the misc. exception category.

LOC class	2463
LOC test	2734
Misc. exception failures	15 (1)
Logic fault	*4*
Incorrect control flow	*2*
Signed/Unsigned fault	*6*
Data/Range overflow/underflow	*3*
Misinterpretation	*9*
Unknown	*1*
Σ	25

Listing 3.8: Example of executing an overflow fault.

```
1 Convert.ToByte("3F3", 16);
```

The line in Listing 3.8 should convert 3F3, which is a hex value, to the byte equivalence. Since, in this case, there really is no byte equivalence, 3F3 is 1011 in the decimal number system and the byte type is only allowed to contain values between $0-255$, an `OverFlowException` should be thrown. This was not the case in the current implementation, instead the method returned the value 243. So the converter started over from 0, thus leading to $1011 - 3 \cdot 256 = 243$.

As can be seen from these two simple cases, all the test cases tested a minimum of two things, crossing over the maximum and minimum values for a given method or type, simply by using maxValue$+1$ and minValue-1. This is something that should have been tested during the implementation since it is considered to be one of the standard practices [204].

The above two underlying faults, which were uncovered in the implementation, would naturally lead to strange behavior in an application using `System.Convert`. Probably the only reason why this was not discovered earlier was that the above methods were not exercised in a similar way [as in this survey] by other developers.

As already mentioned, in total 25 faults were found in the `Convert` class. These faults were mainly of two types (Table 3.1) that caused exception failures, e.g. exceptions of type `OverflowException` or `FormatException`, and secondarily, misinterpretation of the specification when the component was created, as we already saw previously, i.e. `Convert.ToString(tryInt16, 8)`.

One unknown failure was found where we could not pinpoint the exact reason. The `Convert.ToString` method (Listing 3.9) should have returned the expected result, but instead it returned -1097262572.

Listing 3.9: An unknown failure which should throw an exception.

```
1 long tryInt64=123456789012;
2 AssertEquals("#O40", "123456789012", Convert.ToString(tryInt64,10));
```

Clearly this is a case of overflow, but no exception was raised, which should have been the case.

What then, could the uncovered faults lead to? In the case of reliability, ISO-9126 [140] mentions maturity, fault tolerance and recoverability. Clearly several of the flaws we found showed relatively immature aspects, e.g. faults that should have been uncovered if the framework had been used more extensively by developers. These faults probably would have been uncovered over time when the framework had been used more. But as we have already mentioned, Boehm [29] has pointed out the need for uncovering faults early in the development process for several reasons.

Fault tolerance implementations in a framework, such as this, should be able to identify a failure, isolate that failure and provide a means of recovery. This was not the case with several of the exception failures we uncovered. Identification and isolation of a failure could in several of these cases be implemented by examining the input for validity, isolate non-valid input and notify the developer of the fault.

Finally, when an exception is thrown because of a fault in a core component, a developer would have problems recovering, since a stable foundation is expected. On the other hand, an overflow occurring without an exception being thrown would cause a very strange behavior in the application using the method. If it is possible to differ between severability of faults in a core class in a framework such as Mono—all faults being of a very serious nature—a fault that does not cast an exception holds, if possible, an even higher severability than other faults.

3.5 CONCLUSION

CBD is often promoted as one of the great trends within the area of software development. A fundamental problem, however, is the degree of reliability of the individual components, something clearly indicated by our current study.

Mimicking the situation of a third party developer, we chose to apply straightforward unit testing to a core component from the Mono framework, being the open source implementation of .NET. Employing off-by-one, boundary testing and certain legal input for each method we were able to uncover in total 25 faults in the implementation of the class at hand. Although always serious when it comes to a core class like `System.Convert`, some failures did not result in any exception being thrown. A fact that must be—if possible—considered even more severe.

This component had already been subject to certain sub-system and system testing and makes up one of the core parts in the framework. The fact that the component already was in reuse, clearly shows the seriousness of the reliability problem of CBD. Combined with the non-systematic evaluation of components from third parties (evident in Chapter 2) by software developers, i.e. lack of testing before usage as opposed to what was done in this study, the reliability not only of the components but a wide range of resulting applications is jeopardized.

Based on our findings we propose that some sort of low level testing of components should be a foundation for further testing methodologies, more or less without exception. Trusting the foundation, when adding module and sub-system tests, is vital. It is, to put it bluntly, better to add low level testing after implementation or even usage of a piece of software, than not doing it at all. In the specific case at hand a third party developer performing the test cases in this study would have avoided failures late in the development and at the same time aided the CBD community. Sooner of later one will experience failures if testing is not performed properly. The question is; can you afford trusting the origin of a component? Since no de facto certification is widely used today, as could be seen in Chapter 2, we believe the answer is no to that question.

One important thing must be stressed throughout any software project—if a developer finds a fault, they should immediately write a test case for it. That way, the fault will show up again, if the present code deteriorates. This could be considered as a best practice and somehow forced upon the developers during check-in of new or changed source code. If this practice had been followed in the project then some of the faults we found would probably have been found much earlier.

Chapter 4

New Quality Estimations in Random Testing

Originally published in Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE 2003)

S. Mankefors-Christiernin, R. Torkar & A. Boklund

4.1 INTRODUCTION

Testing and evaluation is of major importance for all software and has been the subject of research for some 30 years. Although exhaustive testing in principle can prove software correct, in practice most software has to be tested in some approximate way, trying to uncover most of the faults. Even highly tested systems like the NASA Space Shuttle Avionics software displays a fault density of 0.0001 defects/line of code (at a cost of $1,000$ USD/LOC), which is considered to be close to what is achievable with modern methods [128].

In this respect one of the most difficult questions is the issue of to what extent a test can be trusted, i.e. the quality of a test. The current state of affairs is not entirely satis-

factory since most methods are not possible to quantify in detail at the moment, while others do not compare easily with each other (see [196, 251] for a critical review). Recent theoretical efforts have, however, produced some good results [103, 299] in this field. Taking a more pragmatic point of view, a large number of experimental evaluations have been performed of test methods in order to try to compare the quality of different test approaches, see e.g. [71, 136, 156, 163, 184, 185, 306]. There are also new approaches to improve the choices of test data using data dependency analysis [159] during execution as opposed to a traditional flow analysis.

Although inspiring, more exact results for many methods are lacking. Put in a different way, paraphrasing Dick Hamlet [121], a "success theory is missing"; there are no good ways to determine how good a test that found nothing was. In the area of coverage testing, [71] and [121] do provide theories towards this aim, but since absolute coverage does not in itself constitute fault free software they only provide partial answers.

Given the context, statistical test methods (random testing) are unique in that they do provide a type of answers about possible remaining faults or execution failures [122]. The downside with this information is its probabilistic nature and interchangeability between failure probability and test reliability. In addition to this, random testing suffers from other well known inadequacies, e.g. the inability to detect rare failures. Despite this situation, statistical testing remains one of the major testing approaches, if nothing else as a sort of scientific baseline. Random testing does also provide massive quantitative data, something usually lacking otherwise. Finally random testing may very well in many circumstances outperform other test methods per time unit—if not per test—due to the simpler test data generation (random input). It is thus of significant importance to extend the current work on random testing to include solid quality estimations and eliminate the reliability-probability evaluation problem as far as possible.

In this chapter we present distinct quality estimations and results for random testing, enabling much improved use and interpretation of a random test collection (a collection is by us defined as being a number of test inputs grouped together for the sake of testing on unit, software item or testee). We subsequently make this quality estimation subject of empirical scrutinizing, to ensure the strength of the results using massive simulations encompassing hundreds of billions of inputs. The remaining parts of this chapter are consequently divided into a more general overview of random testing, fundaments for quality estimations and empirical evaluations of the suggested estimation using massive test simulations.

4.2 PROS AND CONS OF RANDOM TESTING

There is no absolute consensus on what statistical testing (or fault detection) is. Partly this is due to the large number of available methods, e.g. modern methods in code inspection [300] are based on a classical statistical approach. To be clear on this point we will from now on strictly refer to test cases, with randomized input and comparison of the output to a well known correct answer, as random testing (this follows e.g. [122]).

Even so, the issue of the exact use of random testing still remains open. Why use random testing at all? Intuitively it is clear that by choosing input on a random basis you always come up short compared to e.g. equivalence partitioning [204] or someone inspecting the code or specification closely and constructing the test based on this information (unless a systematic error is made in the non-random approach). It should also be clear that random testing usually samples only a very small fraction of all possible input. A straightforward `if`-statement that compares two long integers and is executed if found equal (see Listing 4.1), stands a chance of 1 in $4 \cdot 10^9$ to be exercised for a single random test input.

Listing 4.1: The combinatorial aspects of a simple if-statement.

```
1  a=random(1);
2  b=random(2);
3
4  if (a==b) then
5  {
6    //some LOC
7  }
```

On the other hand, similar arguments, although not as strong, can be made about coverage testing. Borrowing the following example from [196] (see Listing 4.2), we note that branch testing with the input value of (3,2) will not result in a failure, while values (2,3) is going to result in an erroneous result.

Listing 4.2: Branch testing.

```
1  int multiplier(int n, int m)
2  {
3    int result=0;
4    while (n>0)
5    {
6      result=result+n;
7      n=n-1;
8    }
9    return result;
10 }
```

The strongest argument in favor of random testing is that mathematical tools provide a possibility to calculate the successes as well as the shortcomings of the method. Blunt or not, random testing possesses a potential industrial robustness. Given a way to fast and easily check if the output result (or operation) is correct—something that usually is referred to as the oracle assumption [122]—random testing is both easy and straightforward to implement. This combined with its random nature (more about this later) allows for extensive, mathematical analysis. Short of AI-like testing approaches, mathematically inspired methods still remain the most powerful tools available in testing. Possibly advances in genetically programmed testing algorithms [190, 194] will render this kind of basic tools superfluous, but at the moment it would be hazardous to draw that conclusion.

An important issue at this stage of reasoning is the actual user's (machine or human) input and the input used in random testing, something that has been improved and well examined in [175] and [202], respectively. Although it is well enough to consider a test collection of $1,000$ inputs with correct behavior as an indication that the method, or similar, usually behave well (i.e. has a low probability of returning the wrong result on a general input), this has not to be true for all input subspaces. Differently put, faults in software are not any more random than, say cards in a game of poker. There is a definitive answer to whether player X has the ace of spades, and it will not change from one time to the next if nothing happens in between in the game (corresponding to no one rewriting the code).

This also means that a method may give completely correct results for all long integer inputs except in the range of 0-100. Given uniformly distributed non-biased input from the user, the method will behave correctly $39,999,999$ times out of 40 millions. In this circumstance—unless the method is executed extremely often—the software at hand can be considered as next to flawless. The mean time to failure (MTTF) will be closing in on one year of continuous operation if the software has an execution time around 1 second. On the other hand random testing will most likely—unless used massively—not uncover the fault at all. In the perspective of uniform input this is correct, i.e. the failure frequency is extremely low (1 in 40 millions).

Unfortunately this line of reasoning may be very misleading in certain circumstances. If the user is a human being, the application happens to be a computer based calculating aid and the user is free to choose input, it is intuitively clear that the probability for the user to choose low numbers as input is extremely much higher in everyday life than to choose high end numbers, e.g. between $1,907,654,1000$ and $1,907,654,1100$ (which has the same input range as 0-100). Hence such software, subject to a well-performed random test collection, with asserted high MTTF based on uniform input, will still be perceived by the user as quite flawed due to the non-uniform nature of the user profile.

It is therefore of outermost importance that the testing conditions, and especially the use of a certain distribution of random input, is clearly defined and declared for any tested software. The necessity of clear definitions and routines is a general truth in all testing activities, but unusually important in random testing because of the complex mathematical nature.

A direct consequence of the poor sampling of rare events is that some extreme values in a method's or object's input/attributes will never, or at least very rarely, be tested using random methods. This is especially true since a Pseudo Random Number Generator (PRNG) usually is used to produce input in random testing. Given the fact that PRNGs produce output in a certain legal range, illegal values are not possible to obtain (unless the legal range is larger than the input, e.g. long integers are used to probe an input domain of short integers). The wisdom to draw from this is that extreme and illegal values should be tested in connection to, but separately from pure random testing of legal input, see e.g. [190, 306] for a practical example. In addition, illegal values are nowadays caught by most modern compilers.

Despite these problems, random testing—given properly handled input distributions—have certain great advantages as pointed out earlier. Random test data is usually very fast to produce due to the automatic and straightforward nature of the PRNGs generating the input. Given a method and corresponding oracle that perform reasonably well and execute in a few seconds, random testing may cover tens of thousand cases in 24 hours. Automatic over-night testing can give tabulated test results 'en masse' for tested software units.

It has to be admitted though that the oracle assumption makes random testing much more difficult above unit-level. At the same time random testing is not, and has never been, suitable for subsystem or system testing; the possible combinations of input is so huge and almost always subject to a specialized user input that the random test collection ends up probing the wrong corner almost regardless of its size, unless it is uniquely tailored for the system.

Random testing hence offers advanced brute force testing of units where straightforward oracles are available. The results have to be carefully used, but as one of very few methods, random testing offers the possibility of genuine mathematical analysis.

4.3 FUNDAMENTS OF QUALITY ESTIMATIONS

Quality estimations of random testing has traditionally been connected to classical statistical instruments, i.e. confidence levels and believed performance.

Given enough time we would find the true failure frequency θ_t for uniformly dis-

tributed random input:

$$\theta_t = \frac{\text{Number of test data causing failures}}{\text{Number of all test data}} \tag{4.1}$$

Normally θ_t is only possible to determine exactly by exhaustive testing, or approximately using massive testing. It should also be noted that we, due to the assumption of uniform distributions, do not care about in which sub-domain of the input the failures take place—all input is equally probable, and hence the related failures too (with a different distribution the different inputs would have to be weighted or measured differently).

In most software it is not possible to even come close to a value of θ_t determined by exhaustive testing. Given a single float number as input, there are $4 \cdot 10^9$ different possible input values (see Section 4.2). By testing the software, using a limited number of randomly chosen inputs it is possible to apply statistical analysis to the results for a quantitatively controlled estimation of θ_t though.

When M failures are found by running N tests, Nelson's probabilistic theory [278] gives that a guessed (estimated on the basis of the tests) failure frequency θ_g has an upper confidence bound α according to:

$$1 - \sum_{j=0}^{M} \left(\frac{N}{j} \right) \theta_g^j (1 - \theta_g)^{N-j} \geq \alpha \tag{4.2}$$

The confidence bound determines the probability of the test suit being a representative one, while θ_g is the software engineer's guess of the true failure frequency. To be close to certain that the tests are 'good' means that the software engineers become restricted in the range of failure intensity they can guess at; a 'squeeze play' [121] takes place between testing and failure probabilities.

Although straightforward, the above theory fails to offer unambiguous quality estimation but leaves extensive parts of the interpretation to the testing crew. Secondly the nature of testing makes a statement like: "Given the failure frequency θ_g the probability for experiencing no failures at runtime ten times in a row is $(1 - \theta_g)^{10}$", highly dubious if there are input domains with very high (or absolute) failure intensity. Strictly speaking the combinatorial term would be domain dependent and from a random input perspective, subject to a non-trivial probability distribution in itself. More extensive reviews of the constant failure frequency assumption and domain-partitioning problem can be found elsewhere, see e.g. [121, 122]. As we will see, there exists an alternative formalism that avoids this kind of reasoning though.

4.3.1 FAILURE FUNCTIONS

Given the nature of software and the problem of the under-decided solutions above, we choose to introduce the concept of *failure functions* as a mathematical tool. Intuitively it is clear that a piece of software either fails or succeeds for a specific set of input, regardless of the input type (numeric, text, logic etc.) and software at hand. This process is non-random, and very much repeatable. It also matches existing mathematical problems, i.e. integrals.

To be able to be more exact in our reasoning we start out by defining the input space $\mathbf{X}$ for a specific software $\mathbf{S}$.

Definition 1 (INPUT SPACE) *A software item's regular input set* $\mathbf{X}$ *is defined to be all combinations of values legal with respect to the input variable types of* $\mathbf{S}$.

The above definition is neither 'white box' nor 'black box', since the regular set (input domain) is defined by the legal input types. This implies, first, that exceptions and overflows are not included in the regular domain, but should be treated separately (see [306] for an example). Second, that the domain could be concluded from the specifications, given a detailed enough level of description (it could equally well be defined by information collected in a white box process).

Definition 2 (FAILURE FUNCTION) *Each software item* $\mathbf{S}$ *has a* failure function F_S *defined on the regular input set* $\mathbf{X}$. $F_S(x) = 0$ *for a given value* $x \in X$ *if no failure occur, and* $F_S(x) = 1$ *if a failure is detected when* $\mathbf{S}$ *is executed.*

The function F_S matches the behavior of the software exactly, although detailed knowledge of the function can only be obtained with exhaustive testing. Assuming that the regular input space is large enough to let us approximate F_S with a continuous function (this rules out e.g. pure logical input domains though) and using integration instead of summation, we find that:

$$\theta_t = \frac{\int_x F_S \mathrm{d}x}{\int_x 1 \mathrm{d}x} = \frac{\int_x F_S \mathrm{d}x}{|\mathbf{X}|} =< F_S > \tag{4.3}$$

The bracketed term implies the mean value of F_S. Finding the true failure frequency, the mean value if exact in Equation 4.3 hence becomes identical to integrating (or sum up, if we let go of the continuous approximation) the failure function and dividing by the size of set $\mathbf{X}$. Although trivial as far as relations go, this enables a new variety of tools. It also allows us later on to go beyond the somewhat limited use of just finding θ_t. It should be noted though that this equivalence is only meaningful for input

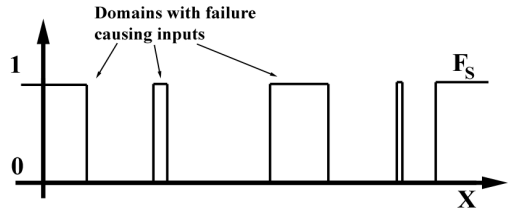

Figure 4.1: The general problem of integration of a failure function F_S and the relation to testing.

with a reasonably large legal span of values (integers, strings, floats etc.) Otherwise the small size of $|\mathbf{X}|$ will render the notion of average and continuity quite meaningless.

Turning to the integration (summation) of F_S we notice that we somehow have to integrate (sum) an unknown function; a problem identical to the evaluation of random testing (since it is just another formulation of it), see Figure 4.1.

Unfortunately the 'unsmooth' nature of the failure function limits the number of available tools for integration. One of the most straightforward methods, Monte Carlo integration [237], applies well, however. The idea is to sample the function (F_S in our case) by random input, calculate the mean and multiply it by the volume the integral is defined on. More strictly we have:

$$\int_x F_S \mathrm{d}x = |\mathbf{X}| < F_S > +\epsilon = |\mathbf{X}| \left(\frac{1}{N} \sum_{i=1}^{N} F_S(x_i) \right) + \epsilon \tag{4.4}$$

The $< F_S >$ term is the mean value of the failure function in the N randomly chosen x_i points in the regular input set $|\mathbf{X}|$. The expression is of course of limited use before the mathematical error terms are estimated. It simply indicates that if one by random choose a large number of points in the legal range of the failure function and take the average, one should come in the vicinity of the absolute average (compare Equation 4.3). Drawing on our established correspondence to random testing, Equation 4.4 just states the fact that an approximation of θ_t is the number of failures found, divided by the number of tests performed.

4.3.2 QUALITY ESTIMATIONS: ELEMENTA

The definitions and equations in the previous subsection, merely cast the problem of random testing in a different form, although it circumvents some of the more traditional

formulations and associated problems. All the more interesting is the estimation of the mathematical error terms in Equation 4.4. As long as each input x_i is chosen statistically independent of each other, the exact nature of how representative the associated $F_S(x_i)$ values are (i.e. the probability distribution of errors) is irrelevant if the number (N) of input values x_i is large enough relative the input space (typically a number of a hundred up to a few thousands inputs).

In this case the central limit theorem guarantees that the resulting probability distribution of errors (mathematical mismatch) will be close to a Gaussian distribution (see e.g. [154] for a proof and explanation of the central limit theorem). Differently stated, random testing will, with its randomized input, ensure that the mathematical error terms in Equation 4.4 approximately follow a Gaussian distribution.

This approximation is valid for all cases when the number of tests is large relative to the input space, i.e. at least decently samples each input variable space. This is an extremely important point to be made since it eliminates any assumption about 'evenly distributed execution failures' or similar which has troubled some of the earlier results in the literature, see e.g. [121, 122] for a discussion.

Although this could have been argued for without using the formalism of integrals, we will retain the form due to reasons revealed later in the text. In accordance with standardized statistical estimations, we identify the mathematical uncertainty (error) in Equation 4.4 with the number (k) of allowed standard deviations (σ):

$$\int_x F_S \mathrm{d}x \approx |\mathbf{X}| < F_S > \left(1 \pm \frac{k\sigma}{N < F_S >}\right) \tag{4.5}$$

Taking all values within one standard deviation (i.e. $k = 1$) from the mean value $< F_S >$ covers 68% of the statistical possibilities, while using two standard deviations ($k = 2$) covers 95%, $k = 3$ covers 99.7% and $k = 4$ approximately 99.99% (see [238] or similar for tables). Although we a priori do not know the variance (σ^2) of the failure function F_S, the sampling of F_S provides a good approximation:

$$\sigma^2 \approx N \left[\left(\frac{1}{N} \sum_{i=1}^{N} F_S^2(x_i) \right) - \left(\frac{1}{N} \sum_{i=1}^{N} F_S(x_i) \right)^2 \right] \tag{4.6}$$

In standard short hand notation this becomes:

$$\sigma^2 \approx N(< F_S^2 > -(< F_S >)^2) \tag{4.7}$$

The average is taken over the N sampled input points. It should be noted that given a very large N approaching the size of the regular input set, this relation becomes exact. Now, using Equation 4.7, Equation 4.5 transforms into:

$$\int_x F_S \mathrm{d}x \approx |\mathbf{X}| \left(< F_S > \pm k \sqrt{\frac{< F_S^2 > - (< F_S >)^2}{N}} \right) \tag{4.8}$$

Since the failure function F_S in our case only takes on the simple values of 1 and 0, $< F_S^2 >$ becomes degenerate with the value of $< F_S >$. Hence for uniform input distribution we have the following result:

Result 1 (TOTAL NUMBER OF FAILURES WITH LARGE ENOUGH INPUT) *If a software* $\mathbf{S}$ *with regular input set* $\mathbf{X}$ *and an associated failure function* F_S *exists and the number of test inputs is large enough, the total number of failures for* $\mathbf{S}$ *on* $\mathbf{X}$ *are given within* k *standard deviations by:*

$$\int_x F_S \mathrm{d}x \approx |\mathbf{X}| \left(< F_S > \pm k \sqrt{\frac{< F_S > - (< F_S >)^2}{N}} \right) \tag{4.9}$$

Knowing the average failure rate in N inputs then immediately returns the quality estimation. As a practical example, using the limit of $k = 4$ standard deviations (returning a coverage certainty of 99.99%), $2,300$ tests and 45 found faults, we get an estimated failure rate of $1.95\% \pm 1.15\%$ (i.e. a maximum of 3.1%). The absolute number of faults increases linearly, however, as does the quality uncertainty with the size of the regular input set.

At this stage it is important to stop to scrutinize the validity of the approximations made so far. The central limit theorem returned the result above under the assumption of a 'large enough number of input values'. Hence the results so far are mathematically valid in all cases where the statistics is good, i.e. much test data is used and a reasonable amount of failures are found. The 'demand' for failures arises from the fact that the number of input values is not only important as an absolute number by itself but also relative the number of observed failures in Equation 4.9 (see the discussion below). In the case of worse numbers, the approximation becomes less precise.

4.3.3 LOWER END QUALITY ESTIMATIONS

The approximate approach to the full-scale probabilistic problem presented here offers a great reduction of complexity in the evaluation of random testing, enabling solid

hands-on quality estimations and calculations easy enough to do on a sheet of paper. It also offers straightforward tabulating of the quality of software units, as well as robustness in performance as the empirical evaluation below shows. The drawback is that the result above does not cover the lower end of the failure frequency in a strict mathematical-statistical sense.

In order to extend the quality estimations to encompass the full range of frequencies we identify the limitations of the approximation and adjust it accordingly below. The necessity of this becomes self-evident considering the lower boundary of the quality estimation in Equation 4.9: given few enough faults the lower quality boundary will become less than zero.

This is a pure complication due to too few input values since σ is divided by the square root of N relative the average of F_S. That is, in the case of few found faults, going from 200 recorded tests to 400 tests will in general do little to the found average, but reduce the boundary term by a factor of $\sqrt{2}$. The normal software engineer could not be expected to extend the number of test cases in order to meet the demands of an academic result though.

Instead we conclude that if the lower boundary given in Equation 4.9 falls below zero, a reasonable quality estimation to make within the current approach is to assume that the entire quality boundary is given by $[0, 2k\sigma]$. This ensures that the total span of the quality estimation remains intact, while the failure frequency might be overestimated, but hardly the other way around. More precisely stated we get:

Result 2 (TOTAL NUMBER OF FAILURES WITH N TEST INPUTS) *If a software item* $\mathbf{S}$ *with regular input set* $\mathbf{X}$ *and an associated failure function* F_S *exists, which has been probed by* N *test inputs, the total number of failures in* $\mathbf{S}$ *on* $\mathbf{X}$ *are given within (at least)* k *standard deviations by:*

$$\begin{aligned} \int_x F_S \mathrm{d}x \approx |\mathbf{X}|(< F_S > \pm C); \sigma = \sqrt{\frac{< F_S > -(< F_S >)^2}{N}} \\ < F_S > -C = \max(0, < F_S > -k\sigma) \\ < F_S > +C = \max(< F_S > +k\sigma, 2k\sigma) \end{aligned} \tag{4.10}$$

For normal fault frequencies Equation 4.10 becomes identical to Equation 4.9, but in the case of very small frequencies it shifts the quality boundary upwards to avoid a spill-over to negative (and hence impossible) values. More strictly put, Equation 4.10 expresses the statistical limits of Equation 4.9 in conjunction with the provided tests in each individual test collection.

This limitation is also directly connected to the quality *demands* raised by the software engineer. Obviously the 'spill over limitation' will be enforced for smaller frequencies if the software engineer chooses a '2 standard deviation quality' boundary, as compared to 4 standard deviations (compare the maximum functions in Equation 4.10). The reason for this seemingly strange behavior is an intrinsic part of the statistical approximation used here. If we want the statistical results to be true for 3 or 4 standard deviations, the demands on the statistical quality, and hence the test collection itself, simply grow essentially larger compared to if we are satisfied with 1 or 2 standard deviations.

Although the results in Equation 4.10 seemingly just provides a crude cut-off in the lower end of the available failure frequency range, it in reality compresses the very same 'cut-off' together with the falling failure frequency, something that is observed empirically to fit data very well (see below). This is so since σ is still estimated trough the observed failures in Equation 4.6 and hence grows smaller together with the failure frequency.

It would now appear that the quality estimation in Equation 4.10—despite corrected lower limit behavior—suffers from the same limitation as other theories. For zero detected failures, Equation 4.10 returns an estimation of exactly zero due to the variance compression discussed above. The inability to prove a test collection successful is a trivial illusion, however, since a reasonably bad scenario is that the very next (not yet executed) test in the collection would have found a fault. Assuming that the true variance (i.e. what would be given by exhaustive testing) corresponds to this scenario we get a possible failure rate of:

$$\int_x F_S \mathrm{d}x \approx |\mathbf{X}| \left(2k \sqrt{\frac{1}{N(N+1)} - \frac{1}{N(N+1)^2}} \right) \leq \frac{2k}{N} \tag{4.11}$$

It should be pointed out at this stage that even if one assumes something along the lines of 'the next two tests would have failed if we did not stop', the quality bound still only increases by $\sqrt{2}$ due to the square root expression. Alternatively, the statistical certainty falls from 4 to slightly less than 3 standard deviations. In addition to this, one should keep in mind that statistics can only properly be used on large collections (which is not always practiced [306]), which in turn ensures highly modest deviations along the lines suggested above. Despite these objections, the simplistic approach used here is sufficient for many purposes and withstand the empirical evaluation quite well as we will see below.

A rightful question to ask now, is to what extent the results in Equations 4.9-4.11 contributes to random testing, apart from an apparent re-formulation of the theory for uniformly distributed inputs?

First of all it is a different approach based on a statistical evaluation instead of Nelson's combinatorial approach and thus *not* a re-formulation. It also offers a great reduction of complexity as noted above—the software engineer can in Equation 4.10 and 4.11, choose the desired quality boundaries as in an ordinary engineering problem, and get a solid estimation. Something practically put to the test in the empirical section of this chapter with very good results.

Furthermore Equation 4.10 represents a definitive simplification—the example of $2,300$ tests and 45 faults found, is far from easily assessed in Equation 4.2. In addition it offers both a simple 'success theory' as well as the access to a vast set of existing advanced numerical methods for Monte Carlo evaluation of integrals [237]. The straightforward method presented here is the simplest possible, but makes a solid ground for further work, one being the possibility to introduce multiple user profiling based on an already executed (used) random test collection.

4.4 EMPIRICAL EVALUATION

No matter how powerful a theoretical result may be, it has to be empirically validated if it is going to be of any practical use to engineers. This is true in all engineering disciplines, including software engineering. We therefore turn our attention to simulated testing, that is, the mimicking of real test situations where controlled software faults are seeded. By looking at the mutation coverage (i.e. how many of the artificial faults that are found) it becomes possible to analyze the effectiveness of a test method, or as in this case, validate the theoretical quality estimations.

In order to use this approach, for verification of the results in the previous section, we have to undertake massive test simulations. Only extensive testing will give the 'true' failure rates and variances. Differently put: performing $100,000$ tests, will the results of Equation 4.10—especially the 'upper quality bound'—hold to be true?

It should be pointed out that it is only by performing this kind of analysis, one really can say anything with relative certainty, about a statistically based theory. Once shown to be reliable on the other hand (which, of course, demands repeated validation beyond this chapter), the theories could readily be used in everyday software engineering in the same way as numerical integration and statistically based methods are used in other engineering disciplines daily.

4.4.1 METHODOLOGICAL FRAMEWORK

Using mutation analysis to estimate the absolute number of faults, failure rates etc., one should seed faults in software that is known to be defect free and well known

in the software community in order to establish a common baseline. This presents a problem since not even extremely reliable software can be said to be absolutely fault free [128] and far from all software is 'well known'. One way to solve this problem is to employ common, well-trusted software for mutation analysis. Some of the more popular benchmark software units are TRIANGLE [64, 146, 306] and FIND [302, 306] which are programs to determine the type of a mathematical triangle and a highly straightforward sorting algorithm respectively.

However, a problem with small and well controlled pieces of software is that they have small bearing on 'real life' software. Furthermore a general criticism exists, concerning random testing, for performing well on small 'toy programs', but considerably worse on software used in real life, see e.g. [306]. The benefit of very small pieces of software on the other hand is the high degree of control in the experiment they allow. It becomes increasingly difficult to tune the failure frequency and behavior of the seeded mutations with the growing complexity of the code.

Trying to resolve this problem and meet the criticism in the field we have employed both a straightforward modulus error in order to allow 'massive testing' as well as extensive material of more realistic cases. A number of well-known numerical methods from [237], some being more than 30 years old in their initial form, are used to host more realistic mutations/faults. The numerical methods at hand have also been continuously and extensively used in real life research and industry for a similar period of time. More extensively corrected, tested and used software is hard to find, possibly short of embedded industrial systems.

Because the methods can be assumed to be defect free, they also provide testing oracles in themselves. That is, we assume that the methods behave properly before mutation and replace the absolute failure frequency with the relative one where comparing the outcome from the original and the seeded code establish the failure. Finally the code is available on CD, which eliminates the human factor in transferring the software to the platform to be used in the simulations (which sometimes renders a defect free software highly dysfunctional).

Turning to the simulations, we have chosen to seed artificial defects with variable failure rates, in each individual software. The combined failure rate is subsequently calculated theoretically or determined by massive testing (tens of millions of inputs or more). In the second phase we perform a vast number of tests for each software and fixed (true) failure rate. The statistically determined variance from the combined number of tests is calculated as well as the theoretical predictions for each individual test collection. A comparison is then made between the different theoretical predictions based on the minor test collections and the actual failure rates. Varying the failure rate, the full experiment is repeated for each rate and software.

4.4.2 TECHNICAL DETAILS

The random input was generated using the PRNG `Ran2` from [237]. All software subject to testing was written in Fortran and thus non-object oriented. The methods in [237] are readily available in the C programming language as well [236], but the Fortran version was chosen out of convenience. In addition to the numerical methods chosen, we implemented a simulated modulus fault for massive testing (see Listing 4.3).

Listing 4.3: Simulated modulus fault.

```
1  i=abs(Mod(testinteger, ErrorFrequency))
2  i=i+ErrorFrequency/2.0
3
4  k1=0
5  if (i.gt.ErrorFrequency) then
6    k1=1
7  endif
8
9  k2=0
10 if (i.ge.ErrorFrequency) then
11   k2=1
12 endif
```

This kind of fault, e.g. by mistake using $>$ instead of $\geq$, typically appears in type conversions, boundary controls, checksum methods or similar. Admittedly it is a type of fault that different kinds of coverage testing techniques normally would find. Still it is a type of defect that is readily found in modern software development (see Chapter 3 for further discussions). It also holds the basic properties of most software errors, being exact (non-random), repeatable and a typical subject of human mistake. Furthermore it is a test with one of the shortest possible execution times, which allows massive testing.

To properly exploit the fast execution of the 'mutation error' we performed tests with the individual batch size of 10, 100 and $1,000$ inputs in each series. The cut-off at $1,000$ inputs was made out of practical considerations (see the total number of tests run below) but could in principle easily be raised to cover extremely long test series (millions of inputs in each run). Although the lower end with 10 inputs hardly state a test series we included these short series for a more complete picture. For each batch size, we varied the error frequency (as per previous code listing) through the values $1/10$ down to $1/1500$ in a 15 steps procedure. In each case, for each failure frequency and batch size, 10 million inputs were used, resulting in a total of 166.5 billion test executions. Hence we utilized the modulus fault to severely challenge the theoretical results in terms of statistical validity.

Table 4.1: Schematic listing of the type of defects seeded in the two numerical methods used for empirical evaluation.

	Type of fault
1.	Parameter error, rare execution
2.	Parameter error, executes more often
3.	Branch-defect, executes moderately often
4.	Combination of 1 and 3
5.	Combination of 2 and 3

Turning to the non-trivial software examples, we have chosen to investigate two real-life methods used in engineering disciplines over the years. This choice has been made in order to test the theoretical approach against not only a laboratory environment, but also realistic problems.

We employed the CISI and SVDCMP methods from [237], seeding well controlled mutations. The former method calculates the cosine and sine integral for a given input value (float) and consists of 70 lines of code. The second method is of a more complicated nature, larger (240 lines) and performs a singular decomposition of any given (mathematical) matrix. The input size of the matrix was chosen to 4x4, with all entries consisting of random float numbers.

In both cases we used two types of seeded defects, typical of human errors: parameters offsets and mutated `if`-statements (conditions shifted). In the latter case the code will end up executing the wrong part of the code, depending on the exact nature of the input. Changing a parameter value will only affect the outcome if it interferes destructively with the execution, e.g. if it is used as a multiplier or is large enough relative a key variable it is added to, in order to tip the balance in an `if`-statement.

Furthermore, these defect types represents two types normally found by different testing methods. While the branch-defect should be detected by a branch-coverage test, a parameter shift could prove intrinsically difficult to find using code coverage. Hence the two fault types not only represents common human mistakes, but also pose a challenge for the current random approach due to the different nature of the two.

To further test the theory at hand we mixed both types of defects, resulting in a more realistic scenario with different levels and numbers of problems (Table 4.1).

As in the case of the previously listed modulus fault, we performed test series of 10, 100 and $1,000$ tests in each. For every type of defect and collection size we performed 10 million test series, resulting in 111 billion tests of the two methods. At each stage the randomized input was fed to the original method to create the answer (as per a traditional oracle approach), which subsequently was compared with the answer from

the seeded methods.

In order to avoid recording round-of errors we controlled each manipulated software method with the seeded defects switched off by running 100 million tests where the answers were compared in the same way as above. No failures were recorded, which strongly suggests the absence of round-of errors or similar. To establish absolute certainty with exhaustive testing was not a viable alternative due to the humongous size of the input space.

4.5 EMPIRICAL RESULTS AND DISCUSSION

Statistical results, especially when it comes to quality estimations or similar, tend to be less hands on than absolute numbers. Neither are the implications always very intuitive or self-explanatory. The results from Section 4.3 are an attempt to solve some of the former ambiguity, and the interpretation of our findings is very straightforward: the probability that the true failure frequency is within the predicted rate ($< F_S > \pm\sigma$) is given by a tabulated number. This on the other hand implies that if the software engineer uses e.g. two standard deviations in her quality estimation in Equation 4.10, the probability of the test collection at hand actually returning an estimation that does not match (within bars) the true failure frequency should be no more than 5%. If one is not satisfied with this (rather high) probability, one should use three or four standard deviations which would give 0.3% and less than 0.01% respectively (see Section 4.3).

If the quality estimations in Section 4.3 are to hold true, the portion of test collections that under- or overestimates the failure frequency may thus not exceed the given numerical limits. More strictly should the resulting distribution of quality estimations comply with the assumed Gaussian approximation when it comes to variance and standard deviations in order to be useful.

Alternatively and more practically formulated, the extremely unlikely 1 in $10,000$ (the case of $k = 4$) test collection which provides an unusually bad sampling of the code at hand should—despite it being a 'pathological' (bad) test collection—still result in a quality estimation that *actually encloses the true failure frequency*. That is, the upper quality boundary should coincide with the true failure frequency for the 1 in $10,000$ ($k = 4$) test collection. Only tests being extremely rare are allowed to fail and actually underestimate the true failure frequency (see above).

4.5.1 MODULUS EVALUATION

We now turn to the modulus fault case described above where we compiled all performed test collections for the different batch sizes and failure frequencies (see Sec-

tion 4.4). For each fixed failure frequency we calculated the total average and variance employing all test cases in all available test collections. In all cases both the average and variance agreed with theory, confirming the theoretically tuned failure frequency.

Continuing with the quality estimations we have plotted the upper quality bound of the pathological test collections (1 in $10,000$ $k = 4$, 1 in 333, $k = 3$ etc., see above) versus the true failure frequency for all investigated collections encompassing $1,000$ tests in Figure 4.2.

An overall very good match is found, with the upper quality bound being very close to the true failure frequency. This directly implies that the quality estimation in Equation 4.10 (Section 4.3 on page 53) does hold the level of certainty introduced there, even when pushed to the limit in massive simulations.

As the number of observed failures goes down to zero, the estimation in Equation 4.11 ensures a constant upper quality bound which properly encompass the true failure frequency. This is illustrated by the leveling out of the failure frequency estimations, turning into constant minimum value ($2k$, see Equation 4.10) plateaus in Figure 4.2. The upper 'success' bound in this case is quite crude, however, even if fully functional as pointed out earlier.

The low frequency result is especially noteworthy since the data points before the minimum value plateaus sets in, corresponds to a single recorded failure in a test collection of $1,000$ tests. What more is, this happens in highly unrepresentative test collections (where the 'proper' number of failures should have been much higher), which represents the case of 1 in $10,000$ (1 in 333 etc.) of all performed test collections. Still the upper bound of the quality estimation virtually coincides with the true failure frequency, something that clearly indicates the strength of the approach and the lower end adjustments in Subsection 4.3.3.

In the case of the test collections with fewer tests (100 and 10 in each collection respectively), the agreement in the 100 case was just as good as in the case of more extensive $1,000$ case. The minimal upper bound (the plateaus) sets in earlier though, due to the lower number of tests in each collection.

Also the 'collections' of 10 test cases did conform to the theory presented in Subsections 4.3.2-4.3.3, but on the other hand does even a single failure imply an estimated upper quality bound of 0.8 failures per run (using $k = 4$), which render most comparison useless. This effectively shows the absolute need for proper statistics using random testing.

4.5.2 CISI AND SVDCMP EVALUATION

Turning to the evaluation of the tests of the mutated numerical methods we notice the same level of agreement. It should be noted though that in this case we do not have

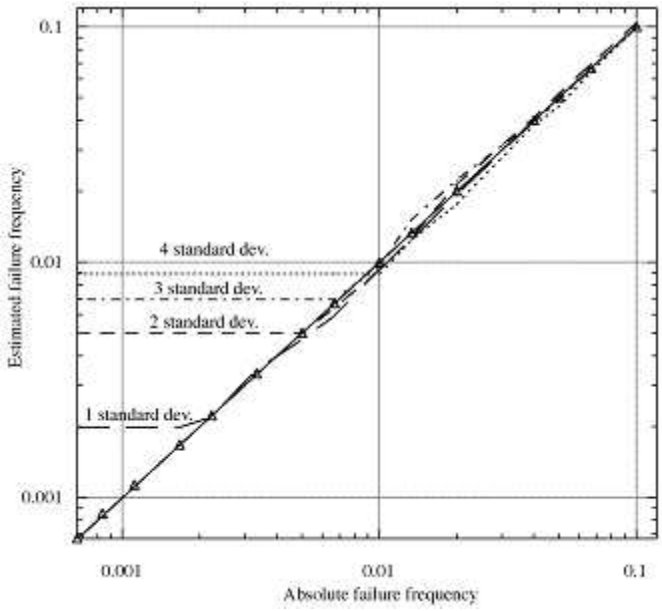

Figure 4.2: The upper boundary of estimated failure frequency in the case of 'unfortunate' test collection (1,000 tests each) corresponding to 1, 2, 3 and 4 standard deviations (in the case of 4 deviations, the figure indicates the upper boundary value in the quality estimation made for the '1 in 10,000' worst test collection). The triangles mark the total average found for each investigated true failure frequency (see the text).

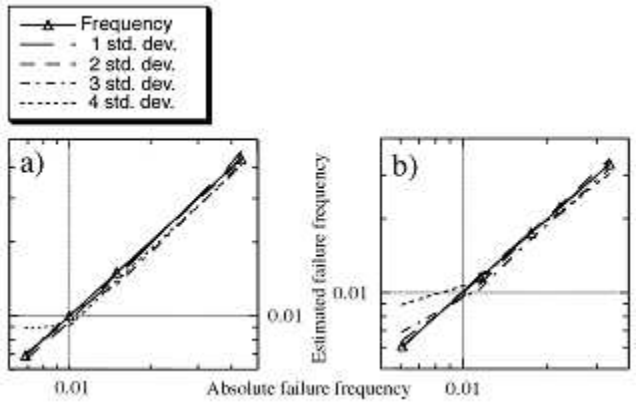

Figure 4.3: Match between the upper quality bound and actual failure frequency for 'worst case' test collections. Triangles mark the failure frequencies for the different defects (see Table 4.1). Panel a) is based on mutation of the CISI method while panel b) is evaluated using SVDCMP. The scale is the same as in Figure 4.2 on page 67.

access to any theoretically calculated failure frequency but simply define the recorded average failure frequency as the 'true' failure frequency. Now, performing the same kind of comparison between the upper quality boundary for the 'pathological' (see above) test collections and the actual failure frequency, we find very good agreement between the theoretical results and the empirical evaluation (Figure 4.3). The 'plateau' phenomenon in the case of four standard deviations (left in both panels) is recognized from the modulus case, but the over all agreement is of the same quality.

The failure frequency is confined to a smaller interval as compared to the modulus fault, but contains on the other hand only five defects in each case due to the considerably longer execution time. Still, there is no difference in agreement between the simple modulus failure and the mutated numerical methods. Nor did we find any differences between agreement for the branch, parameter or mixed defects for the two methods.

Perhaps even more interesting is that the accuracy of the presented method actually is rather independent on the complexity of the input as long as the statistics is good enough for the integral evaluation process underlying our results in Subsections 4.3.2–4.3.3. Otherwise this is a much-debated issue. Hamlet has in several pa-

pers, see e.g. [121], compared the effectiveness of random testing with investigating the fauna of the great lakes by putting in one day's effort of commercial fishing. From this follows the apparent logical conclusion;

> the greater the lake, the poorer a method.

In our current formulation, however, it becomes evident that the input to the failure function F_S is of less interest since we are sampling the failure profile (the integral in Equation 4.3 and 4.4, see also Figure 4.1). A complex input does complicate the use of a more realistic user profile than the uniform distribution used here though, but only in the input generation process itself.

Finally we note that the test collections with fewer tests conform well to the theoretical results in Equation 4.10, but generally the minimum value plateaus sets in so early that most failure frequencies are covered by them in the same way as with the modulus fault.

4.6 CONCLUSION

In this chapter we have presented a short exposé of arguments against and in favor of random testing. Although outperformed per test, random testing may still perform better per time unit than other test techniques and offers a set of mathematical tools that is otherwise lacking in much testing, especially considering early phases of testing. The drawbacks have so far been evident despite this: ambiguous evaluations and no unique benefit over other approaches.

We have shown, however, that it is possible to transform the issue of random testing into an equivalent problem, formulated using integrals. By applying Monte Carlo evaluation techniques in conjunction with the central limit theorem, it becomes possible to evaluate the software reliability using a given test collection in a very precise manner. Upper bounds on the true failure frequency (as given by exhaustive testing) are easily established within this framework. Furthermore our results allows us to establish an upper bound on the quality of a test collection returning zero failures, i.e. it provides a limited but working success theory.

Massive empirical evaluations with simulated testing scenarios encompassing hundreds of billions of tests have been used to verify the current theory successfully not only for small artificial failures but also seeded faults in two extensively used methods. Taken together this present a relative advancement of the state of random testing and ads a special feature in the case of the success theory. This clearly indicates that there is still room for random testing in today's testing practice.

Chapter 5

Fault Finding Effectiveness in Common Black Box Testing Techniques: A Comparative Study

Originally published in Proceedings of the 3rd Conference on Software Engineering Research and Practise in Sweden (SERPS'03)

R. Torkar &
S. Mankefors-Christiernin

5.1 INTRODUCTION

To test software is probably the most common way to verify a certain degree of reliability. Today's methodologies have all, more or less, evolved around two basic principles [149] white box and black box testing. White box testing, or structural testing,

relies on the tester having knowledge of the logic of the program, while black box testing treats the software as a black box e.g. comparing output and requirements. Even though improvements have been made, the key issue today is still the same as it was two decades ago [204]:

> What subset of all possible test cases has the highest probability of detecting the most errors?[1]

Despite that computer hardware has been following Moore's law [199], it still is impossible to test everything (i.e. exhaustive testing) due to similar exponential growth in available software (see Chapter 4 for an elaborate discussion concerning this matter), thus making developers focusing on quality instead of quantity [301]. Code inspection [264] is one way to ensure quality, i.e. faults being found earlier in the development process, but the question is if this is a viable alternative for small- and medium-sized enterprises, especially as the number of code inspectors are not growing as fast as the size of software. Anyway, using code inspection *only*, is not a viable alternative in any case [165].

Since testing is considered by some developers as being more or less forced upon them [152], the need for automated test tools is in demand (see Chapter 2). Automated testing can contribute to increased productivity and be used for repeated regression testing, among other things. Creating random test input automatically is not something new [143], and the concept of creating stubs has already been covered by others in, e.g. [180]. As component-based (CB) software engineering grows, Korel [160] has already shown the need for better black box testing techniques for CB and commercial-off-the-shelf systems.

The main question this chapter tries to answer is therefore:

> What candidates [black box methodologies] are suitable and efficient in a potential robust and straightforward implementation of automatic test creation on unit level?

By using the word suitable we imply, e.g. a method that has a high(er) probability of finding faults which might cause high-severity failures and/or can easily be automated. Of course, simply finding a lot of faults makes a particular method a good candidate as well. Previous work in this field is e.g. Ntafos' [211] theoretical comparison of partition and random testing, and Rothermel's et al. [82] discussion on how to best prioritize test cases for regression testing, even though their reason for prioritizing test cases [test suites running for several weeks] is not valid in our case. Finally, Reid's

[1] The IEEE definitions of error, fault and failure is used throughout this thesis and hence this should in our context read faults.

contribution [244] is interesting (an empirical study comparing several different test methodologies).

The results of the comparison, which this chapter covers, were derived from injecting faults in a software, a Just-In-Time (JIT) compiler [114], already in commercial use, thus giving us real-life conditions and furthermore software that has already been tested extensively by both developers and users. By applying code inspection [87], unit tests [138] and system testing [267] on a sub-domain [the Input/Output layer], before seeding 240 faults of different severity categories, the software was deemed as being as fault-free as could possible be the case with the limited resources at hand. All tests that were executed on the software used Check [186], a unit test framework for the C programming language [153], since the software was written in the C programming language.

5.2 METHODOLOGY

As mentioned previously, one of the arguments for choosing an already shipped, commercial software, was its maturity and the fact that it was not academic research software. The academic community has faced criticism before, for evaluating software with no real bearing on the reality [306]. Commercial software usually is a bit more complex though, lacking the type of clear boundaries academic research software has, e.g. having a small footprint in terms of lines of code. It also poses some initial difficulty to understand, since a researcher usually is completely new to the design and architecture, affecting the time it takes to really start using the software for her purposes. Furthermore, using commercial software in research can lead to other problems; commercial interests could influence the researcher and not all documentation is always willingly available. At the end of the day, the importance of realistic software examples out-weights the difficulties, however.

Usually, a developer creates test cases in the same language as the software is written in, thus a unit test framework for the C programming language was needed. In particular two frameworks were evaluated; Check and cUnit [7]. Check was deemed as being the more mature of the two, especially so since Check allowed a test to fail without tearing down the whole test suite.

Since the complete JIT compiler ($171,000$ lines of source code [297]) was too large for our purposes a delimitation was needed. The I/O-layer of the JIT compiler was regarded as being a preferable sub-domain to test. This layer is responsible for all input and output the JIT is performing during execution. All in all, approximately $12,000$ lines of code were being tested directly and furthermore an additional $35,000$ lines of code was indirectly affected by our test cases.

Table 5.1: Classification of faults according to severity level.

Failure class	**♯ of seeded functions**
Transient/Recoverable	85
Permanent/Recoverable	25
Transient/Unrecoverable	85
Permanent/Unrecoverable	25

Table 5.2: Seeded faults.

Total ♯ functions tested	**220**
Functions with seeded single faults	200
Functions with seeded double faults	20

When applying the test methodologies previously mentioned, eight faults were uncovered. Only one of these faults could, in the long run, have affected the JIT compiler in a serious way, i.e. introducing an unrecoverable runtime failure.

Next, several types of faults, with different severity levels [267], were seeded into the software (Table 5.1).

As shown in Table 5.2, one fault per chosen method was seeded and an additional 20 faults were added, thus having 20 functions encountering a 'two fault'-scenario for which we employed a straightforward homogeneous distribution. Furthermore we used an 80:20 ratio between the transient and permanent failure severity types for the functions containing single faults, since we especially wanted to challenge the test methodologies concerning severe failure types. Thus a major part of the seeded functions included faults of a type that typically does not result in runtime failure for a homogenous input, but still are vital for the overall functionality, e.g. failure resulting from a null argument. This is especially important when discussing possible automatization, since a common critique of e.g. random testing is that it only finds 'simple' faults.

Included in the 240 faults, were 24 boundary type faults (10%) while 40 of the faults were placed in 20 functions resulting in a 10:1 ratio between functions with single and double faults respectively. To summarize it, there were 200 single faults seeded in different functions (SFF). Furthermore, an additional 40 faults were added in 20 functions, thus simulating a double-fault scenario (DFF).

Finally, several candidates were collected that could be used as black box techniques in our evaluation. By using [204] as a foundation certain 'prime suspects' were gathered. Figure 5.1, (c.f.) depicts the methodology used in this experiment. Worth noticing here, is that code review was used only in preparing the test item, since it is considered to be a white box technique, and this study deals with black box techniques.

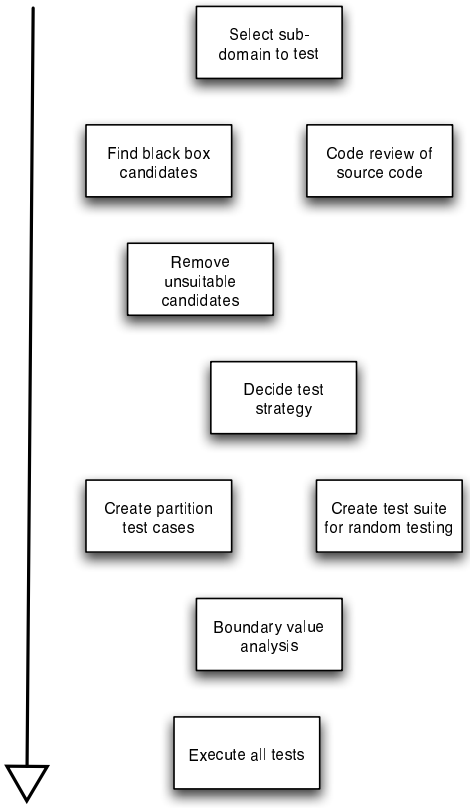

Figure 5.1: Principal overview of the methodology used in this experiment.

5.3 RISKS

Before continuing, some factors that could have affected this study were identified, and are worth pointing out:

- Is seeding faults a viable alternative or are only real faults of interest?
- Is the selection of the sub-domain and faults relevant for our purposes?
- Do any potential bias, among the developers, affect the results?
- Is a case study on a single domain a validity threat?

We believe the first question to be somewhat irrelevant, in our case, since we originally found few real faults (only eight). Furthermore, this question has been tended to by implementing different kind of faults e.g. single/double faults and boundary faults, which are common in today's software development. Yet, it is worthwhile having knowledge regarding some of the problems fault injection can contribute with [183].

The second question is harder to answer. A different sub-domain would naturally let us find other faults. Our selection of faults and domain, however, have been based upon some of the more typical problems found in today's software [144, 149, 183]. Thus, we believe that the findings in the current chapter, concerning the efficiency of the investigated test methodologies, only to a minor extent are affected by the current choices. Furthermore, we have had the support in reviewing test cases, by one of the senior developers responsible for designing the original software. Taken together, this should imply the relevance of the test cases created.

The developers testing the application were never personally involved in designing and implementing the software which should indicate a low probability of bias.

Overall, in our opinion this study design prevented any serious validity threats to the results. Only the last question, might be considered to be a serious validity threat. But extending our study by using other applications in combination with the answers to the second bullet will, in the future, clarify this matter.

5.4 CANDIDATE EVOLUTION

As mentioned previously, several candidates were collected that could be used as black box techniques in our evaluation. Today's literature and common practice was studied and in the next few subsections each of the found candidates are examined.

5.4.1 PARTITION TESTING

This test technique, has been covered both theoretically [45, 124, 144, 296] and in empirical studies [16]. The references show that partition testing has reached favorable results.

Input data normally falls into different categories [partitions] and software normally behaves in the same [equivalent] way for all members of a particular category [22]. By partitioning the input domain, according to some rules [267], a developer can test very large parts of it without needing to traverse the whole input domain. The difficulty in this specific case, would be to automate the partitioning of input domains in a satisfactory way. Nevertheless, partition testing is considered as being a prime suspect in our case, since it holds a promise of both good results and a high degree of robustness in a potential implementation.

Test cases were implemented by dividing the input domain in partitions, following the specification, source code and the developer's previous experience regarding which input values are likely to produce failures.

5.4.2 BOUNDARY VALUE ANALYSIS

Since boundary value analysis (BVA) is a variant of partition testing, the same reasons [as for partition testing] apply for including BVA in the comparison.

Test cases were implemented by having every boundary tested on all functions. This involved testing the actual boundary value and the values directly below and above the boundary value.

5.4.3 CAUSE-EFFECT GRAPHING

One weakness with partitioning and BVA is their inability to examine the compound of input situations. The complication when dealing with this argument is more or less the same as with exhaustive testing—the number of combinations are astronomical. Cause-effect graphing [204] helps the developer to select the right combinations, in some cases by directly adhering to the specification [221]. By creating Boolean graphs, subsequently showing cause vs. effects, test cases can be created.

However, since one of the incentives for this chapter was automatization, we believe cause-effect graphing to initially be somewhat difficult to implement in a straightforward and robust way due to the sheer complexity involved.

5.4.4 ERROR GUESSING

Error[2] guessing is not as some might think black magic. A somewhat experienced developer can use error guessing, i.e. guessing were the fault might be, as a complement to other testing strategies, something Myers [204] once described as "smelling out" errors.

Since error guessing is highly intuitive and typically a human process, we regard it unsuitable for automatization. Only some, more advanced, types of artificial intelligence would make this a viable alternative. The implementation of such a tool in commercial software development seems highly unlikely in the near future.

5.4.5 RANDOM TESTING

The derivative anti-random [184] and especially random [71] testing have been used as statistical test methods, and are unique in that they provide answers about possible remaining faults (see Chapter 4). Test case creation for random testing is also possible to automate to some extent [143], however the problem with needing an oracle [giving correct answers] is likely to show up sooner or later when automatization is at stake. Anyhow, random testing will be included in this comparison due to its possibility to be automated and it showing good results in previous empirical comparisons [244].

Additionally, in this context, it is especially important to notice random testing's good ability to cover input ranges having a large volume, establishing a limit on the failure frequency, even if not necessarily finding all faults. Random testing can thus be used as a technique to estimate the reliability of certain software aspects (see Chapter 4), hence having the potential to work as an automated quality estimation.

All 196 functions (within Faults in Legal Range scope, FLR) were tested with $1, 10, 10^2, 10^3, 10^4, 10^5$ and 10^6 different input values. The input values were allowed to be randomly picked from all legal values, e.g. integer, and tested on the functions containing faults in legal range, i.e. the *Func. with FLR* column in Table 1 (Appendix B, page 217).

5.4.6 EXHAUSTIVE TESTING

As has already been mentioned, the combination of all possible inputs, makes exhaustive testing a mission impossible. But it would be careless to disqualify exhaustive testing for all occasions. Exhaustive testing could be used in extremely small sub-

[2]The IEEE definitions of error, fault and failure is used throughout this thesis and hence this should in our context read fault or failure guessing depending on the aim.

domains, e.g. boolean input or testing some of the most critical functions in a software. This is, of course, also the case when it comes to random testing.

Extensive random testing, instead of exhaustive testing, could instead be a viable option on extremely small sub-domains covering critical parts of the software. In this comparison, exhaustive testing was not used.

5.4.7 NOMINAL AND ABNORMAL CASE TESTING

Both nominal case testing, i.e. test cases using 'correct data' as input, and abnormal case testing, i.e. the opposite, were used during this comparison. A successful test is one that detects a failure when a method, seeded with faults, is executed. This can be achieved both with nominal and abnormal case testing.

5.5 RESULTS

As mentioned earlier, there were eight (8) faults uncovered during the preparation of this comparison. One fault was considered more serious, being of a transient and unrecoverable type, creating a run-time failure without possibility to recover. The initial EP tests did uncover this serious fault and six other faults. One real fault was not uncovered with partition testing, but instead was found when applying random testing. The reason partition testing did not uncover this fault was because errors were made in the partition identification. This particular problem will be discussed more in the next section.

Table 1 (Appendix B, page 217), is a compilation of all the faults that were seeded in the different functions and the detection rates of the different failing functions. The original faults are included in the total amount for each test technique.

The functions, in Table 1, are categorized into functions containing Boundary Value Faults (BVF) and failures occurring with input in the legal range (Faults in Legal Range, FLR). The number of detected failures due to seeded functions are listed for the EP, random and BVA testing techniques (number of detected failing functions containing double faults within parenthesis).

There were 24 (10%) boundary value faults (BVF) included in the total 240 faults. When analyzing the result of random test cases, one must take this into account. The probability for a random test case to hit a BVF is extremely low, consequently there is a need to subtract these types of faults from the random test case category. In Table 1 this is illustrated by the *Func. with FLR* column, i.e. this is the *actual* fault quantity that random test cases should be compared to and not the total number. Obviously, this

is also the case when looking at the Equivalence Partitioning (EP) column, since BVA is illustrated by itself.

For the sake of clarity one example can be drawn from Table 1. The first row covers Transient/Recoverable (T/R) failure types. All in all, 80 single fault functions (SFF) and 5 double fault functions (DFF) were of this type. Included in these 85 faults were six functions of BVF type (always in the SFF category). In total there were 79 functions with Faults in Legal Range (FLR). Using EP, 64 SFF and 4 DFF (presented in Table 1 as 4D) were uncovered out of the total 79 FLR. On the other hand, random testing uncovered 7 SFF and 2 DFF out of the total 79 FLR. Finally, all the seeded boundary value faults (6) were found using BVA.

5.6 ANALYSIS AND DISCUSSION

As can be seen from Table 1, Equivalence Partitioning (EP) is by far the most effective methodology when it comes to uncover functions containing faults. In the Transient/Recoverable category, EP found approximately 86% $((64 + 4)/79)$ of the total number of seeded functions with Faults in Legal Range (FLR), while at the same time discovering 4 out of 5 functions containing double faults (DFF). Similarly, in the Permanent/Recoverable category, 89% FLR and 80% (4/5) of the functions with double faults were found.

When looking at the unrecoverable category, e.g. run-time execution failure, the results are similar—if somewhat lower. The Transient/Unrecoverable severity category, while using EP, had a 72% discover ratio for FLR, while at the same time discovering 80% of the double fault functions. Finally, in the Permanent/Unrecoverable category, EP found 68% FLR and 80% of the DFF. Figure 5.2 (next page), gives a combined picture of how well each test methodology performed in finding the total number of faults.

By applying stricter partition identification techniques, combined with further research in this area, we believe that these numbers can improve even more. On the other hand, it is worth noticing that in this particular case, EP had a high discover ratio when it came to finding functions with double faults.

Turning to random testing, it is important to compare the number of found functions with faults and the number of random tests actually executed, since it is a statistically based methodology. Consequently $1, 10, 100, \ldots$ etc. random tests per method were executed, with the result of no additional detected failures after 10^4 tests (per function), to an upper level of 10^6 tests (c.f. Figure 5.3). All in all, random testing found close to a total of 12% of the functions containing FLR (23/196), while at the same time discovering 25% of the double seeded functions. As Figure 5.3 illustrates, already 100

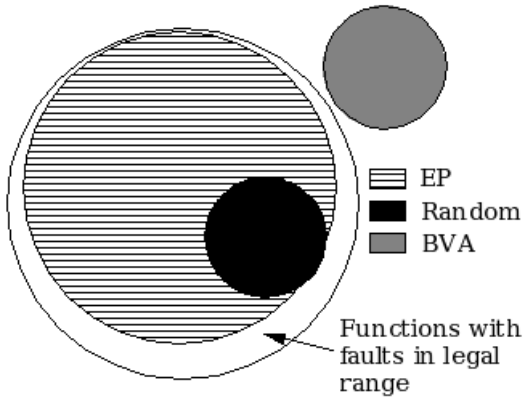

Figure 5.2: Fault discovery distribution. Random testing found one fault which equivalence partitioning did not find.

Figure 5.3: Efficiency of random testing applying $1, 10, 100, \ldots$ test cases per method in the legal range input. No new additional faults were found beyond 10^4 random test cases per function.

tests per method discovered more than 6% of the faults (50% of the total number of faults that random testing found).

As expected, we found that random testing initially discovered 'large-partition-faults', i.e. faults that could be easily found by giving a method nearly any value, since the value would fall within a fault's scope. After a certain amount of random tests, these 'large-partition-faults' were discovered, and the rate of new detected faults per random test case decreased rapidly. This should not be seen as a failure on the behalf of random testing though. Comparing the input range connected to the seeded faults (fault input range, FIR) with the total legal input range, random testing detected all faults with a FIR/FLR of more than 10^{-7}. The remaining seeded faults made up less than a 10^{-6} fraction of the total legal input range of the entire software investigated. Hence, random testing did perform flawlessly in its own context and established a limit on the failure frequency well in accordance with theory. However, the rare failure types, resulting from e.g. null input, should clearly be handled by other means such as e.g. partition testing.

When comparing effectiveness of different test methodologies and their potential use in an automatization process, it is not only important to apply test cases to real-life code, but also test real-life (similar) faults. The research in this chapter, is mainly built on the seeding of simulated faults in a 'cleaned' real-life code. However, looking only at the original uncorrected faults, although making up a statistically very poor example, we found that 7 out of 8 original faults (87%) were uncovered by EP, while random testing uncovered the 8th fault (there were no BVF in the original code). This indicates a reasonable agreement between the findings regarding seeded faults and real-life faults, although further studies are needed since the conclusion can not be said to have a statistical significance.

On the other hand, it is important to realize that there is no silver bullet when it comes to testing, each test technique has its strengths and weaknesses. By applying stringent rules when identifying partitions and combining several test methodologies there is a higher probability of detecting more faults.

Regarding severity, a straightforward model was employed based on a linear scale classification of the seeded functions. Although somewhat crude, this gives a reasonable tool to judge the efficiency of the methodologies in finding faults which could lead to severe failures. Unrecoverable failure types, for obvious reasons, were considered more serious than recoverable. In a similar manner, transient failure types were deemed more severe than permanent, since they are generally more difficulty to find due to their inconsistent behavior, thus leading to Table 5.3 (next page).

By multiplying the effectiveness (%) in finding functions containing faults (in each category) with the weight (c.f. Table 5.3), it was found that partition testing had a severity coverage of nearly 75.5% (c.f. Figure 5.4). The only minor weakness the in-

Table 5.3: Weight classification of the severity associated with different failure types.

Severity	Weight
Transient/Recoverable	2
Permanent/Recoverable	1
Transient/Unrecoverable	4
Permanent/Unrecoverable	3

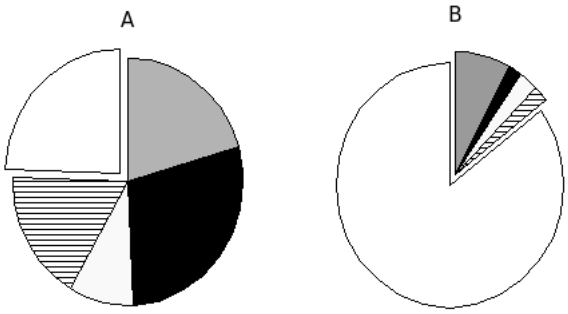

Figure 5.4: Chart A highlights the EP severity coverage, while B shows random testing severity coverage (both with respect to FLR). The white pie by itself symbolizes faults not found of varying severity. Clockwise from the white pie we have the different severity categories; P/U, T/U, P/R and T/R, as explained in Table 1 (Appendix B, page 217).

vestigation of partition testing exposed, was its somewhat lower effectiveness at finding faults in the Transient/Unrecoverable failure category (72%). This, however, should be considered being a rather good number, since it is the most serious type of failures (by weight). Random testing, however, demonstrated a severity coverage of just slightly more than 14%.

Taking the number of faults uncovered, and the severity coverage of the different methodologies, we find EP to be the most efficient and promising test methodology investigated. Combining two or more methods, including EP, in an automatization scheme appears to have a definitive potential to uncover a majority of faults at the unit level. Furthermore, improving the partition identification technique, could give even better results.

5.7 CONCLUSION

Using commercial software and seeding faults, we investigated the efficiency with respect to finding (severe) faults for three different black box test methodologies. Equivalence partitioning (EP), random testing and boundary value analysis (BVA) were chosen for their possible inclusion, in robust and straightforward implementations, of automatic test creation on unit level.

Seeding a total of 240 faults, in four different severity categories, in a 'clean' version of the software, we found EP to uncover more than 79% of the total number of functions containing faults in legal range. Classifying the severity of the different categories, using a linear weight scale, we discovered that EP addressed 75.5% of the introduced fault severity. 72% of the faults resulting in Transient/Unrecoverable failures (deemed the most severe) were uncovered, possibly indicating a lower efficiency when using partition testing for more severe type of faults.

Random testing detected all faults with a connected input range fraction larger than 10^{-7} of the total legal input range, encompassing 23 functions containing faults with one single fault method not detected by EP or BVA. To ensure that we used a high enough number of random tests per method, we employed $1, 10, 100, \ldots$ etc. random test input(s) per method. No new failures were detected beyond the rate of 10^4 random tests per function. Running EP and random tests on the original faults only, EP found seven out of eight faults, while random testing found the 8th.

Boundary value analysis, on the other hand, proved to have a high discover ratio (100%) concerning seeded boundary value faults. Naturally, no other types of faults were detected by BVA.

We conclude that the single most effective test methodology, of those investigated, concerning uncovering faults leading to severe failures, is equivalence partitioning. By combining EP with its 'relative' BVA, we consider it [the combination] to be of vital interest in an automatization scheme. On the other hand, if random testing is allowed to be a 'turbo' option for especially interesting (most used or critical) sub-domains of a particular software, it could provide an additional edge in testing. This could be implemented by using different profiling schemes combined with coverage analysis.

In addition to that, the characteristics of random testing, with respect to quality estimations (as could be seen in Chapter 4), is yet another reason to not disqualify it.

Chapter 6

Evaluating the Potential in Combining Partition and Random Testing

Originally published in Proceedings
of the 24th IASTED International
Conference on Software Engineering
(SE 2006)

R. Torkar &
S. Mankefors-Christiernin

6.1 INTRODUCTION

Computers are becoming faster, while software is growing rapidly in complexity, thus the need for and possibilities in using automated testing will most likely increase in the future. This research subject holds many interesting aspects and challenges, one of them being automatic creation of test cases and the combination of different testing techniques in an automatization scheme.

According to Frankl et al. [101] there are two distinct types of testing methodologies: statistical testing and defect testing. The former being a way to test the software under operational conditions, e.g. performance and reliability, and the latter a way to find deviations between the software and its specification. In this chapter we focus on the latter.

Defect testing can be further divided into black box and white box testing techniques based on how the test engineer views a certain piece of software. Black box testing or functional testing is used when a software is treated as a black box and the engineer studies the input and corresponding output, while white box or structural testing allows an engineer to create test cases derived from the knowledge of the software's internal structure. In this chapter we focus on the former.

Random testing, is one of the more common black box techniques. An important strength of random testing is its ability to indicate the quality of a test [122] (in addition see Chapter 4), i.e. to give a statistical indication of the extent to which a test can be trusted. If an engineer can get numbers such as; '*there is a probability of less than 4% that there exist faults in component X*', it is ultimately better than the current state of the art, i.e. '*component X has been tested for Y hours using methodology Z*'.

However, random testing has been subject to severe criticism, especially concerning the need of an oracle to evaluate the black box answer and the difficulty to find hard found faults, i.e. faults displaying a very low failure frequency (see Chapter 5). Nevertheless, random testing offers a fairly easy path towards automatization, given an existing oracle—a situation common when e.g. porting code. It furthermore offers a mathematical toolbox and robustness that many other approaches lack. In the specific case of partition testing any potential mismatch between specification-derived partitions and actual partitions might prove hard to find.

Partition testing has unfortunately, despite rather good results concerning test effectiveness, not made any great leaps in small- to medium-sized enterprises. The lack of automated tools is certainly one reason. Hence, software engineers [not test engineers] usually restrict themselves to only use simple boundary value analysis [204], while at the same time check one valid and one invalid input (Chapter 2).

Striving for improvements in automatization of testing, partition and random testing present themselves as good candidates for a robust implementation, as Chapter 5 showed. To evaluate the potential increase in effectiveness, when combining these two techniques in a future automatization scheme, thus becomes highly interesting.

Hence, the question we are trying to answer is:

> Aiming for a future automatization; what is the potential, with regards to effectiveness, in combining partition and random testing?

In this chapter we present an evaluation of combining partition and random testing, using a real-life well established software method. Employing 40 separate clones, i.e. copies of the same software, using the single fault assumption, we inserted one fault in each clone and performed subsequent partition and random testing.

This evaluation clearly indicates an increase in effectiveness when adding random testing to partition testing. At the same time, the increase in efficiency inevitably fades because of the added number of random test inputs being used. Nevertheless, the potential in an automatization scheme for creating and executing test cases is visible, thus indicating that a future implementation combining the best of two worlds, i.e. partition and random testing should be pursued.

The remainder of this chapter is organized as follows. Next, a background is given regarding related work. The experimental setup is covered in Section 6.3 and the following subsections, while the results are presented in Section 6.4. In Section 6.5 there is a discussion of the results; in particular on how the results affect our research question and the future goal of automatization. We then conclude this chapter in Section 6.6.

6.2 BACKGROUND

Others have looked at the effectiveness when combining different testing techniques before. Chen and Yu's paper [48] on the relationship between partition and random testing gives an insight on the differences and similarities of the two testing techniques. Boland's et al. work on comparing these testing techniques via majorization and Schur functions [32], provides us with an empirical validation using different, but nevertheless, valid tools. Recent investigations have compared the effectiveness of random testing and partition testing [119, 211], finding a substantially complex picture concerning the effectiveness of the different test approaches. In addition to that, we can see a need for continuously researching this subject as partition and random testing techniques constantly improves.

Software testing and especially the automated aspects thereof is an interesting research topic where there is still room for improvements. Nevertheless, Korel's [159] and Meudec's [191] papers on automatic test data generation are interesting even though they focus on dynamic data flow analysis during software execution or extracting test cases from formal specifications. Using formal methods for creating partition tests has been covered by e.g. Jeff [8]. However, this approach is not pursued in this chapter. In a similar area we can find Yin's et al. [306] work regarding automatization aspects appealing. However, they have a slightly different focus and use anti-random testing, which is not the case in this study.

When looking at the research in the area of testing techniques, especially parti-

tion and random testing, several accomplishments are worth taking into consideration. Equivalence class testing, equivalence partitioning or simply, partition testing, has been covered theoretically to a large extent the last decades, e.g. [45, 124, 144]. Empirical studies made, e.g. [16, 244] and Chapter 5, give an indication on the difficulty of empirically validating the theoretical work made and examining the possibilities to automate the execution and creation of partition tests. This leads to, in some cases, having massive number of test cases and thus the need to prioritize them increases [254].

Boztas' work [37] on trying to improve partition testing, especially when compared to random testing, is also an interesting follow-up on some of the weaknesses of partition testing.

Apart from partition testing, random testing, i.e. the technique of randomizing input for a given method and then comparing the output with a known correct answer [122], has been used extensively in academic research. However, the last three words, "[...] *known correct answer*" usually provides a test engineer with enough problems. To be able to verify the correctness of a given piece of software the correct answer must be known. This is usually done by creating and using an oracle of some sort [81].

One additional matter is worth mentioning concerning random testing, i.e. the question of profiling [175, 202]. Machines might be pseudo-random, but users are anything but random. A user, executing a software, usually behaves in a reasonably predictive way, at least compared to true random input.

6.3 EXPERIMENTAL SETUP

Combining testing techniques will always ensure a fault finding effectiveness equal or greater to the most effective technique of the ingoing approaches. While this truth is self-evident, the exact nature of the improvement in effectiveness is less obvious.

Since we are focusing on automatization and the potential gains in combining partition and random testing, we need to clarify the assumptions made. One of the central issues is the definition of effectiveness:

Definition 3 (EFFECTIVENESS) *Effectiveness in a test methodology is measured as the number of faults (or associated failures) detected for a given piece of software.*

We furthermore define efficiency to be the effectiveness measured per resource unit. In this respect it is important to point out that the relevant resource is computational power, since time spent on testing in a future fully automatic implementation will solely depend on the available computer. Thus we reach the following definition:

Definition 4 (EFFICIENCY) *Efficiency of a test methodology or combinations thereof, in the context of automatization, equals the effectiveness divided by the number of CPU-cycles necessary to reach the current number of detected faults for a given piece of software.*

Many types of faults will be easily revealed with a relatively small numerical effort spent, displaying a high degree of efficiency. However, faults difficult to find, e.g. only triggered within very small input domains not agreeing with straightforward partition schemes, partly challenge the strive for maximum efficiency. These faults are often difficult to find manually as well, which still makes the automatic approach preferable, especially when counting the development in hardware, which economically will improve the benefit of 'low efficiency solutions'. Thus, the effectiveness in detecting these types of faults is of greater importance for a potentially automatic testing scheme to be considered useful, than the immediate efficiency.

Any test methodology (or combination thereof) deemed suitable for automatization should display a high effectiveness in finding faults with small input domains triggering the associated failure, or otherwise being difficult to detect.

The effectiveness/efficiency ratio should not bee too large, though, rendering a solution unpractical for decades to come. Hence, the current investigation assume large, but not infinite resources. The trade-off and the impact on the potential automatization are discussed in detail later.

6.3.1 OBJECT EVALUATION

In this evaluation two categories of seeded fault types are used (both being 'hard to detect') to evaluate the effectiveness of the combination of partition and random testing with respect to automatization. We furthermore make use of the single fault assumption, seeding only one fault per piece of software.

After choosing a suitable method (see below), multiple *identical* copies of the software at hand (clones), were made and subsequently seeded by hand with individual faults. This resulted in 40 different fault seeded software clones (FSSC) which were subsequently tested separately. The original non-permutated method was used as an oracle.

The CUnit [7] test framework was used to run the test suites. The main reason for using CUnit was that the chosen object was written in ANSI C [257] (see next subsection). Hence, this lead to all test cases being written in ANSI C as well, since using the same language as the tested object is usually preferable in unit testing frameworks.

All data gathering was performed by redirecting output to files and then using standard UNIX tools for extracting significant data, e.g. if a fault was found or not.

6.3.2 CHOICE OF METHOD

Different types of software units, i.e. methods, were examined, looking for a number of characteristics. First and foremost, the unit had to be as fault free as possible and preferably a real-life example. Secondly it needed to have a certain size. Testing a method consisting of only 20 lines of code would probably prove to be quite hard when injecting faults.

In the end, the SVDCMP method found in [236], was considered to be a good choice. Consisting of 184 lines of code written in ANSI C it allowed faults to be injected in a straightforward way. A supporting method, `pythag`, which computed $(a^2 + b^2)^2$ without a destructive underflow or overflow, was left intact, while only consisting of five lines of code.

The SVDCMP method constructs a singular value decomposition of any matrix and is declared in Listing 6.1:

Listing 6.1: The SVDCMP method definition.

```
1 void svdcmp (float **a, int m, int n, float w[], float **v)
```

In [236] one can read: "Given a matrix $a[1..m][1..n]$, this routine computes its singular value decomposition, $A = U \cdot W \cdot V^T$. The matrix U replaces a on output. The diagonal matrix of singular values W is output as a vector $w[1..n]$. The matrix V (not the transpose V^T) is output as $v[1..n][1..n]$". For further reading concerning the underlying algorithms please see [112, 271].

This method has, basically, three input variables that are of concern to us. First, we have the actual matrix that should be computed (`float **a`), and then there are the variables `m` and `n` which contain the number of rows and columns the given matrix has. The correctness of the method can be verified by checking the variables `w` and `v` respectively.

6.3.3 FAULT INJECTION

What types of faults, and what number of faults should be injected into a software when testing for effectiveness? Selecting the right faults for injection is not easy and is in itself an elaborate research subject [25, 286].

To properly challenge the possible benefits from using random testing, on top of partition testing, two different fault types are chosen, with 20 faults of each type, using all in all 40 FSSC. The first category (type A in this evaluation) consists of faults where lesser/greater than ($<$, $>$) control statements have been changed to the opposite, or the 'lesser than' statement to 'lesser than or equal to' ($<=$) etc. By changing the flow of execution, even a—in theory—perfect partition scheme would come to divert from the

'true' partitions and thus stand a substantial risk of not detecting all faults. It should be pointed out that this fault type is readily detected using e.g. code inspection [217], but this evaluation is done under a complete black box assumption.

The second type (B-type) of faults injected into the different software clones consists of different parameter faults, such as 'off-by-one' and bitwise faults where values are changed to simulate human mistakes. These kind of faults can be extremely hard to detect even in a white box approach since the correct parameter value has to be known and be compared with. It is also a fault type that sometimes is very hard to detect using partition testing since the detection of run-time failures depends on the exact interaction between input values and parameters; a major shift in parameter value might block the entire method from executing properly, while a small shift might only interact with very specific input.

While both types of faults typically provides a challenge for partition testing, they are also chosen to challenge random testing—some of the type A faults (e.g. the exchange of $<$ to $<=$) holds extremely small fault volumes, while the parameter faults are tunable in failure frequency. Generally lower end failure frequencies have been chosen to properly exercise the challenge that no rare faults are found by random testing.

6.3.4 PARTITION SCHEMES

Several techniques, for creating partition tests, have evolved during the years. In [146], four basic equivalence class testing cases are covered, that might serve as a good baseline:

- Weak normal.
- Strong normal.
- Weak robust.
- Strong robust.

The last two class test cases, i.e. robust, are usually not necessary to cover with today's programming languages. These test cases deal with, e.g. invalid input. These types of faults will usually produce a compiler error or in some cases a compiler warning in most, if not all, strongly typed programming languages. Even though the use of dynamically typed languages has increased in recent years, statically typed ones are probably still the most used and thus what we focus on here

The first two *normal* class test cases, on the other hand, are still very valid when testing software today. *Weak normal* equivalence class testing is built upon the *single*

fault assumption, implying that only well defined input in specific partitions is assumed to trigger any failures at run-time.

Strong normal equivalence class testing, as opposite to weak, tries to cover all possible combination of inputs. The Cartesian product (also known as the cross product) guarantees that we cover the equivalence classes.

Turning to the method at hand we notice that numerical functions such as SVDCMP tend to converge slower for large input. Furthermore, it is generally numerically disadvantageous to use large numbers due to round-off errors. Taken together, this often leads to an increased usage of smaller (normalized) numbers in the input. Hence, we have chosen a restriction on the float input range. The exact choice here is unimportant, but to investigate defects sensitive only to high end numbers would be unrealistic and of little interest in this case.

Although the usage of small numbers in the input has a relatively general validity in this case, different end-users are still going to formulate different additional requirements and hence lead to discrepancies in the specification. Using a black box approach, such discrepancies in specifications or requirements would potentially lead to different partitions in a real-life situation, despite that the singular value decomposition is clearly defined for all 'valid' matrices.

In order to acknowledge these problems, to at least some degree, we have chosen to utilize three different partition schemes (3PS, 4PS, 10PS). If, for example, the method will be used in calculating small numbers, but there exists no special preference, it is reasonable to partition the input range into positive and negative numbers, and also differ between 'high' and 'low' numbers. Hence a reasonably valid partitioning to use, for the entries (x) in the input matrix is:

- $-2.0 \leq x \leq -1.0$
- $-1.0 \leq x \leq 0.0$
- $0.0 \leq x \leq 1.0$
- $1.0 \leq x \leq 2.0$

The second partition scheme (3PS) is based on the imaginary requirement that the 'high-end' partitions have different boundaries, and all 'low-end' numbers are equivalent, regardless of sign. This leads to the partitions $[-1.8, -1.0]$, $[-1.0, 1.0]$ and $[1.0, 2.2]$.

Finally we have also utilized a third 'fine-meshed' partition approach of brute force type, i.e. ten equally spaced partitions over the interval $[-2, 2]$. It should be noted though, that the testing scenarios at hand, are of course neither unique nor self-evident.

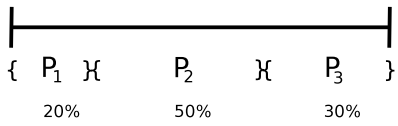

Figure 6.1: Variable I partitioned into three partitions.

The inclusion, however, of different partition schemes here, only intend to uncover the potential sensitivity of the partitioning done in a process such as this. If a partition scheme is implemented in a non-effective way, then other testing techniques will hopefully uncover these flaws.

6.3.5 TESTING SCENARIOS

The two most straightforward testing scenarios, in choosing the effort of the random testing for each partition, are to either use proportional distribution, based on the direct size of each sub-domain, or to assume an equal importance of all partitions. While the former is close to 'normal' random testing—on average random testing with uniform input would result in a similar distribution, but with different variance—the latter focuses on the concept of equivalent partitions. In a genuine partition testing approach there is no obvious way to determine how one equivalent partition would outweigh another.

Taking the above discussion into account, and at the same time remembering that one of the future main goals was automatic execution of tests and setup of test cases, the following two scenarios were chosen:

S1 Generate randomized input, with the size of 10^6, on each of the partitions in an input variable.

S2 Generate randomized input, corresponding to the size of the partition. Also known as proportional partition testing [211].

An explanation of one of the scenarios is appropriate. In S2, if a method has one variable as input, while $3 \cdot 10^6$ random tests are going to be performed and the partitioning is analyzed to be as per Figure 6.1, then $\frac{3}{5} \cdot 10^6$ random input should be tested on partition P_1, while partition P_2 and P_3 should be tested with $1\frac{1}{2} \cdot 10^6$ and $\frac{9}{10} \cdot 10^6$ respectively.

Using small matrices (i.e. 2x2) for convenience, we apply the two testing scenarios, S1 and S2 (Table 6.1). First by executing one value in each partition corresponding

Table 6.1: Partition schemes and maximum number of random test data *per partition* used. 4PS, 3PS and 10PS is short for four-, three- and ten-partition scheme respectively.

Scenario	**Partition schemes**		
	$\mathbf{0.25 \cdot 4}$ (4PS)	$\mathbf{0.2 + 0.5 + 0.3}$ (3PS)	$\mathbf{0.1 \cdot 10}$ (10PS)
S1	10^6	10^6	10^6
S2	$(4 \cdot 10^6) \cdot \frac{1}{4}$	$(10^6 \cdot \frac{3}{5}), (10^6 \cdot 1\frac{1}{2}), (10^6 \cdot \frac{9}{10})$	$(10 \cdot 10^6) \cdot \frac{1}{10}$

to traditional partition testing, then performing random testing for each partition using $10, 100, 10^3, 10^4, \ldots$ etc. tests within the stipulated range. All results are averaged over the total number of available tests, e.g. the number of faults for 10^4 tests is averaged over $100 \cdot 10^4$ tests. Utilizing strategy S1, the random test series continues up to 10^6 for each partition, while in the proportional case (S2), tests will be distributed over the available partitions in direct relation to their size.

The partitioning of the method and distribution of random tests are summarized in Table 6.1.

As a side note it is worth mentioning that the method, in addition to the above schemes, should be tested with variables `m` and `n` being incorrect with respect to the actual size of the matrix. In this particular case, all faults falling into this category were discovered immediately by the function's own self-check parts, why none of these faults were included in this experiment.

6.3.6 VALIDITY THREATS

Before examining the results of this evaluation, some factors that could have affected this study are worth pointing out. First of all, the choice of what method to test, can be discussed. In the end, a well established method which have been used for many years, was deemed necessary for our purposes. By using several copies of the same method, instead of using many different methods, some certainty could be reached with respect to the method being identical for each test run—*except* for the seeded faults. A more general validity could be reached by repeating the procedure with additional methods, however.

The types of faults used in this evaluation does not represent the most common fault types but that was not the intent either. This evaluation explicitly aimed at testing the effectiveness of different testing techniques when it came to uncovering faults with low failure frequencies. Other studies, as mentioned in Section 6.2 and partly discussed

in Chapter 5, have already showed that some faults are easier to uncover with certain testing techniques.

Since the method used in this evaluation did not have any *real* users, an artificial specification was used. This is an obvious weakness since any such specifications have an element of subjectivity to it. On the other hand, it is essential for a fault seeding procedure, that the method used, is as trustworthy as possible and the real-life example of general interest. This, however, tend to result in non-exclusive specifications making partitioning problematic.

The main threats can thus be summarized as follows:

- Usage of a single method.
- Representation of faults.
- Artificial partitioning.

In the end, there will always be a problem when evaluating and comparing different testing techniques. Even though they, in this case, build upon the same assumption (black box), the steps these techniques take to uncover faults are radically different.

6.4 RESULTS

In Table 6.2 (page 98) a summary of the results is presented. Note, however, that a detection limit for EP (partition testing) was set to a minimum of 5% on average, in order to allow for a fault to be counted as 'detected'. In Figures 6.2 and 6.3 (next page) the results are presented with respect to the 4PS and the in-homogeneous 3PS respectively.

Examining the results from this evaluation, we notice a very low detection rate for partition testing (as can be seen from the first value on the x-axis in Figures 6.2–6.4 on pp. 96–97). This can partly be derived from the artificial specifications and partition schemes used, but is mainly connected to the type of faults seeded. Compiling the results from random testing, we also notice a typical 'detection build up' with the increasing number of test cases, see Figure 6.4 (please note that the first test case in the figure corresponds to the simple partition testing situation on average).

Although the fault finding ratio increases, together with increasing numbers of random tests performed, it is something that should be generally expected (sufficiently many random tests will be equivalent to exhaustive testing). However, the investigated FSSCs at hand clearly demonstrates the benefit of adding random testing to partition testing for the hard to detect fault types at hand. The full results as found in Table 6.2,

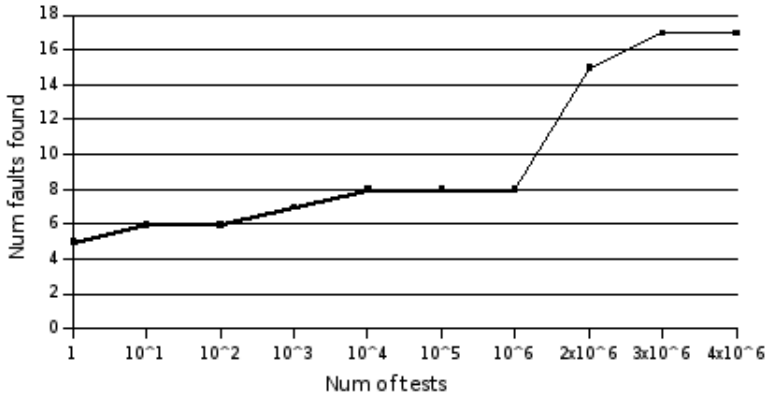

Figure 6.2: Number of detected faults (types A+B), using equivalent distribution of random test data on the 4PS (note the switch from logarithmic to linear scale).

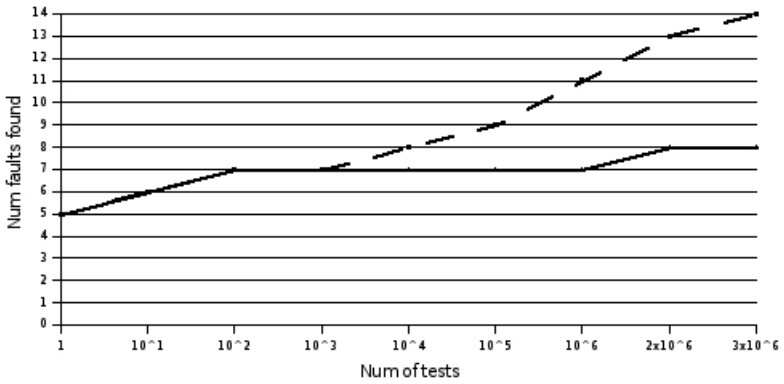

Figure 6.3: Compilation of detected faults using proportional (lower line) and equivalent (upper line) distribution of random test cases for the 3PS, i.e. inhomogeneous partitions (note the switch from logarithmic to linear scale).

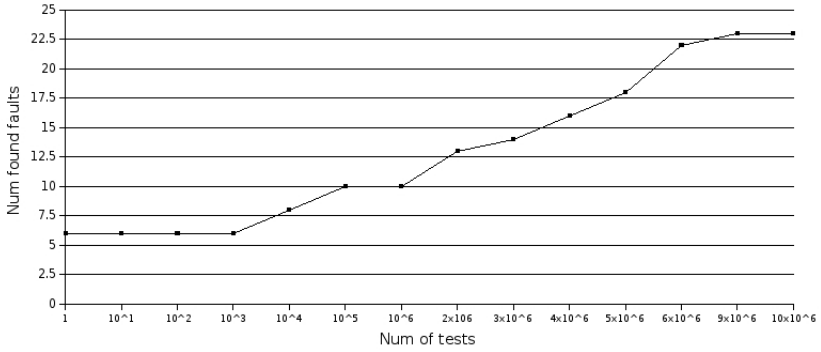

Figure 6.4: Number of detected faults (types A+B), using equivalent distribution of random test cases on the 10PS (note the switch from logarithmic to linear scale).

clearly indicates that no ordinary weak partition testing would have detected the seeded faults at hand, mainly since the faults not found by partition testing were discovered very few times by random testing (between 12 and 967 times).

While the 4PS (Figure 6.2) and 10PS (Figure 6.4) are rather similar, the inhomogeneous 3PS, with partitions of unequal size, makes the distribution of random testing important. The equivalent distribution of random tests assumes all partitions to be equally important, but the proportional distribution directs more effort to the largest partition. In this investigation it results in a less favorable outcome, which is demonstrated in Figure 6.3 by the lower-end proportional distribution line.

The different number of tests in each partition furthermore results in differences in the statistical validity of the results for each partition, while the equivalent distribution ensure identical quality estimations for each sub-domain (please see Chapter 4 for a wider discussion).

6.5 DISCUSSION

As we have seen, random testing initially find faults with a large size with respect to the variable's volume [211]. At the same time, it is not strange to find that running more random tests on a given partition also uncovers more faults.

Nevertheless, if a partition would have been classified in an incorrect way, random testing has a high, or at least higher, probability in uncovering any potential faults. In this evaluation, Table 6.2 (next page) indicates that random testing found faults that

Table 6.2: Fault detection rate for partition and random testing. Crosses marks 'detected' (partition testing) and numbers the number of times the seeded fault in each corresponding FSSC actually were detected by random testing.

	Fault type A		**Fault type B**	
Fault no.	**EP**	**3PS**	**EP**	**3PS**
1				111
2				43
3	X	1,131,342		44
4		141	X	3,445,677
5		64	X	2,322,112
6	X	864,554		762
7	X	2,298,533		
8	X	8,556,512		
9				12
10			X	4,444,421
11		37		
12				
13		967		43
14		23		
15				
16				
17				373
18		34		112
19				143
20		67		

would have been difficult to uncover by partition testing, even if the partition classification would have been done in a different way, e.g. Fault no. 13, Fault type A.

It is also well known that while partition testing displays difficulties in finding faults deviating from the equivalent assumption, e.g. faults depending on exact input, or simply changing the partitions themselves, random testing makes no use of any additional information about the software. Thus, employing proportional random testing within a given partitioning can be proven to be at least as good as only using (homogeneous) random testing [48].

On the other hand, the achievement of any degree of homogeneity within the chosen partitions will automatically improve the effectiveness drastically [55]. Thus, assuming truly equivalent partitions, and furthermore assuming equal importance among the partitions, the resulting effectiveness should be higher if this (equivalence) can be achieved. This is also indicated in our findings, although the underlying data is not enough to draw definitive conclusions from (see Figure 6.3 on page 96). An additional side effect is that an equal number of random test input ensures identical statistical quality boundaries for each partition.

Looking at the effectiveness, the results clearly indicate a definitive increase using random testing on top of partition testing. For the types of fault at hand, partition testing performs rather badly. In an automatized scheme, only relying on partition testing would consequently miss out on a large portion of low failure frequency faults. Hence, simply adding random testing in itself to partition testing increases the effectiveness for this type of faults. A properly conducted partition scheme would, on the other hand, increase the effectiveness of the random (like) testing further as noted above. Thus performing random testing, not only as a complement, but actually in combination with partition testing is of clear value to any automatization schemes. This is especially so since it is relatively straightforward to apply random testing automatically (given an oracle) within the sub-domains of the input, as defined by the partitions. Different weighting of the partitions would further increase the effectiveness of such an approach.

The indicated effectiveness/efficiency ratio in our findings is, however, not that advantageous. In order to find a reasonably large portion of the faults at hand, massive testing has to be performed, see Figures 6.2–6.4. Since the computational resources necessary for the random test cases executed, basically scales proportional to the number of test cases, the current approach becomes increasingly inefficient. It should be pointed out though, that the limits pursued in the current scenario is in the lower end of failure frequencies. Furthermore, the initial high efficiency of partition testing is of little use, if the associated effectiveness is too low, since no additional faults are found after the first test cases.

Implementing the partition and random combination approach on more powerful

computational resources (e.g. parallel execution) would reduce the efficiency problem even further. The possibility to implement an automated test generator/executor should also be taken into account; while the efficiency is lower than some other testing schemes [40], the implementation is likely to be correspondingly simpler. Taken together this clearly indicates the potential in using a combination of partition and random testing in future automatization despite the slightly low efficiency.

Since *efficiency* is one crucial factor to take into consideration in many software development projects, it might be worth noting that if a test runs for one minute then, looking at the *effectiveness* of e.g. random testing, letting it execute for ten minutes instead might be an easy choice to make. While, if a test takes eight hours to execute, it makes it somewhat harder to execute as a regression test during nighttime when developers are at home. Hence, the following question must be answered:

When is it economically motivated to continue to test software?

By executing tests in parallel using e.g. cluster technology, a software engineer might decrease the time it takes to test software considerably, thus postponing the answer to that question.

6.6 CONCLUSION

Creating multiple identical copies (clones) of a well established numerical method and seeding 40 different individual faults into 40 different clones, ensuring the single fault assumption, we investigated the practical benefits in combining partitioning and random testing in a black box testing scenario. The seeded faults consisted of 20 parameter faults, changing default values, and 20 faults concerning control statements, each fault injected in its separate copy of the software. All but a few of the seeded faults were low failure frequency faults, intentionally chosen to properly challenge both methodologies.

The input space was partitioned according to three straightforward artificial specifications. The performed random tests were furthermore distributed both proportional to the size of the partitions, as well as equivalently among the different sub-domains, assuming equal importance of all partitions. Additionally partition testing was performed (corresponding to a simple one test per partition) in all cases.

The current evaluation presented, demonstrates the added value of combining random testing, when an oracle is available, with partition testing. The optimal extent and usage of random testing, depends on the tested software and available CPU-power, though. In a future automatization scheme there is most likely a benefit in combining partition and random testing.

Chapter 7

A Literature Study of Software Testing and the Automated Aspects Thereof

To be submitted

R. Torkar

7.1 INTRODUCTION

This chapter gives an overview (by the use of a literature study) of two relating research fields with the purpose of finding key areas of interest. The fields covered are software testing and automated software testing.

A literature study of a field such as software testing is no simple task. During decades several interesting subjects, methods, techniques and tools have been presented and in many cases simply disregarded as uninteresting after a short while. Thus, in the case of describing software testing, including Automated Software Testing (AuST), one needs to focus mainly on principles that have stood the test of time.

On the other hand, when dealing with newly introduced techniques, tools, etc. one needs to instead make an educated guess regarding which particular strategy will hold

a long-lived value for the community. Hence, in the case of this chapter we try to look into the future by reviewing newly presented techniques, tools, etc. in well-known and highly regarded sources of publication [109].

This contribution is presented as follows. First we introduce the methodology used in this study (Section 7.2). Second, in Section 7.3 we present several definitions and the pattern used for categorizing the references found by applying our methodology. Third, in Section 7.4 the references found are explained, described and categorized. Finally, we discuss our findings (Section 7.5) and conclude this contribution (Section 7.6).

7.2 METHODOLOGY

As Kitchenham et al. [155] have explained it is important to justify the method of search while discussing the risks associated with the search method. This holds equally true when performing a literature study. To begin with the search was performed using three widely used search engines: IEEE Xplore [304], ACM Digital Library [2] and CiteSeer [52]. After finding a body of papers which focused on software testing or AuST we saw that some proceedings and journals were of more interest than others. We picked four conferences that showed up the most and performed a manual search of the proceedings from these conferences ten years back in time:

- International Symposium on Software Reliability Engineering (ISSRE).
- International Symposium on Software Testing and Analysis (ISSTA).
- International Conference on Automated Software Engineering (ASE).
- International Conference on Software Engineering (ICSE).

For each paper that was of interest (i.e. covering software testing in general or AuST in particular), a short summary was written. We limited ourselves to ten years since techniques that have been published before this date, and have made an impact, are likely to be established and thus published in text books.

Certainly there is a possibility that we overlooked papers not presented in any of the above conferences, but it is worth emphasizing that we, to begin with, used the search engines to search in the *whole* of the aforementioned databases and thus not restricted ourselves to certain proceedings or journals.

Several other journals and proceedings contribute to this chapter, but since the above four conferences seemed to cover many contributions an additional focus was put on them. In Figure 7.1 (next page) a view of the process used in this contribution can be found.

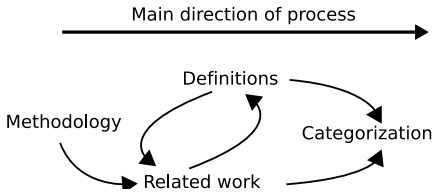

Figure 7.1: An overview of how the methodology led to a categorization.

Figure 7.2: A general overview of software testing.

7.3 DEFINITIONS USED

To be able to classify such a wide area as software testing and AuST one needs to classify certain parts of that area. A few sub-areas needed to be established, and in this case one could see that the contributions found assembled around five main areas:

- Test creation.
- Test execution.
- Result collection.
- Result evaluation.
- Test quality analysis.

Naturally, the above areas could also be depicted as the testing process in general (Figure 7.2).

Since these areas have been established over many years they were found to be solid and logical for categorizing the located contributions.

7.4 SOFTWARE TESTING AND AUTOMATED SOFTWARE TESTING

This section, and the following subsections, provides the reader with related work concerning software testing and AuST. When there is a discrepancy between our classifications (Section 7.3) and the references found it will be discussed in Section 7.5.

With respect to general information regarding software testing (and in some aspects AuST), the NIST and IEEE standards [49, 283] along with *Validation, Verification, and Testing of Computer Software* as published in ACM Computing Surveys [3], are in our opinion good introductions to these research subjects.

Since software testing and AuST use general techniques which are sometimes hard to strictly contain in one area, i.e. dynamic and static analysis, we merely refer the reader to the following papers before we cover the key areas of software testing.

Tsai's et al. [284], Sy's et al. [272] and Morell's contributions regarding fault-based testing [200], can all be used as an introduction on how to use static analysis as a supporting technology.

While Gupta's et al. [118], Artho's et al. [12] and Michael's et al. [193] contributions can be regarded as providing the same introduction concerning dynamic analysis.

For a more thorough description of static and dynamic analysis please see Subsection 3.1 in [78].

7.4.1 TEST CREATION

TEST DATA CREATION AND SELECTION

The same reference that encompasses information regarding static and dynamic analysis also contains an overview concerning test data generation and selection [78].

Test data creation is an old research topic that has many contributors. We start by covering some traditional test data creation techniques, i.e. partition or equivalence class partitioning, boundary value analysis, cause effect graphing and random data generation.

In addition to these techniques, we also cover the path- and goal-oriented approaches with their respective relatives constraint-based data generation and, in the case of goal-oriented data generation, the chaining and assertion-based approaches.

Data generation for partition testing, or equivalence class partitioning, was defined by Myers [204] already in 1978 as, *a technique that partitions the input domain of a program into a finite number of classes [sets], it then identifies a minimal set of well selected test cases to represent these classes. There are two types of input equivalence classes, valid and invalid.*

In other words, test data creation is performed with an introspective selection approach[1], since the test data that will be used is identified before it is actually generated.

Regarding boundary value analysis, NIST defines it as [49], *a selection technique in which test data are chosen to lie along 'boundaries' of the input domain [or output range] classes, data structures, procedure parameters*. This makes it identical to Myers' partition testing technique, even though with a slightly different aim, i.e. boundaries instead of partitions.

Boundary value analysis, above, was quite easy to describe as a data generation technique, on the other hand, cause effect graphing is not as easy to describe when looking at related work. Cause effect graphing, is an old technique which can be defined as either test case generation, according to Myers [204], or test case selection, as defined by NIST [49]. We choose to place cause effect graphing under the test data field instead. The reason for why we do this is that cause effect graphing's aim is to select the correct *inputs* to cover an entire effect set, and as such it deals with selection of test data. Nevertheless, we do not claim Myers or NIST to be wrong, since cause effect graphing can be placed into different definitions depending on the surrounding techniques being used.

Random data generation is another interesting technique, in which test data is generated in a random manner. Random testing [71, 122] and its derivate anti-random testing [184, 306] can be used as statistical test methods, and are unique in that they provide answers about possible remaining faults (see Chapter 4). Test case creation for random testing is also possible to automate to some extent [53, 143]. A somewhat newer contribution by Claessen and Hughes [53], where they use random data generation for testing Haskell[2] programs, is a significant addition to this research field and as such worth examining closer.

Thus, random testing can be seen as a test data generation technique with in some cases, i.e. anti-random testing, a retrospective approach[3], while otherwise with an introspective approach. In addition, test cases can be generated and selected, with an intro- or retrospective approach, i.e. create many test cases which we choose from, or create the appropriate test cases directly.

After covering the traditional test data creation techniques we now turn our attention to a few techniques that have been seen somewhat more recently in literature. First of all, path-oriented test data generation [168] is using dynamic, and in some cases static analysis, during its execution step. Secondly, Beydeda et al. [26] also defines it as, *a data generation technique which makes use of control flow information to de-*

[1] The selection process is implicitly performed during generation.

[2] `http://www.haskell.org`

[3] The selection process is performed after generation.

termine paths to cover and then generate test data for these paths. Thus, this makes path-oriented test data generation an introspective data generation technique.

Constraint-based test data generation [26, 65, 240] is based on the path-oriented techniques and described as, *a data generation technique which makes use of algebraic constraints to describe test cases designed to find particular types of faults*.

Finally, the goal-oriented test data generation technique with its two derivations, the chaining approach and the assertion-based approach, can generally be described as data generation techniques which aim to find program input and a sequence on which the selected statement is executed. According to Ferguson et al. [92] the chaining approach is described as, *a technique for automated software test data generation for distributed software*. The goal is to find program input and a rendezvous sequence on which the selected statement is executed. The chaining approach uses program dependency analysis to identify statements that affect the execution of the selected statement. These statements are then used to guide the test data generation process. Diaz et al. further refines this approach using meta-heuristic search techniques, i.e. genetic algorithms and simulated annealing [67].

The assertion-oriented approach can be described, according to Korel et al. [158, 161] as, *a technique which identifies test cases on which an assertion is violated*. If such a test is found then this test uncovers a fault in the program. Worth noting here is that Korel et al. do not make the distinction between test data and test cases. Andrews and Benson covered this technique already in 1981 in their paper *An Automated Program Testing Methodology and its Implementation* [9].

Hence, in the case of the assertion-based approach, the problem of finding input on which an assertion is violated may be reduced to the problem of finding program input on which a selected statement is executed [161]. Both the goal-oriented sub-categories can be seen as data generation techniques with an introspective approach, i.e. the appropriate test data is generated immediately and a selection process afterwards is not necessarily needed.

In Figure 7.3 (next page) an overview is shown depicting how the different techniques are related to each other. Another picture of this research field (test data selection and generation) can be found in [282], even though Tracey et al. exclude the path-oriented approach.

TEST CASE GENERATION AND SELECTION

The previous subsection covered test data generation approaches and the corresponding supporting techniques for generating test data.

As can already be seen, we make a distinction between test data and test cases. The main reason for making this distinction is the risk of having them treated as one.

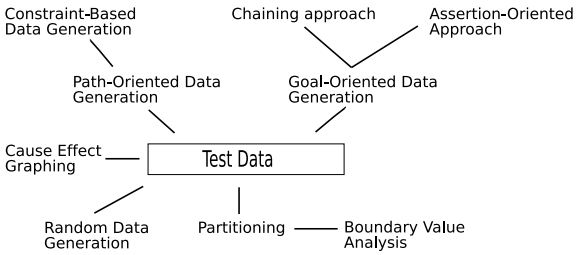

Figure 7.3: Different test data creation and selection techniques and their relationship.

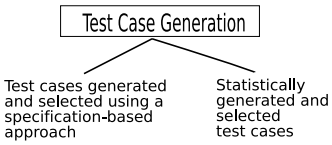

Figure 7.4: Test case creation and selection techniques.

Presently, there are difficulties in distinguishing between these two topics in literature, due to the unfortunate fact that researchers usually do not separate them. Nevertheless, we recognize that in some cases it is hard to distinguish between the two.

Related work covering test case generation can be divided into, when disregarding manual creation, specification-based and statistical ('intelligent' as defined by Pargas et al. in [224]) test case generation (see Figure 7.4). Since the word intelligent is, by us, considered to be inappropriate in this context, we intend to use the word statistical instead. After all, several statistical tools are used in this context.

Describing the specification-based approach to test case generation is straightforward, especially so if we first turn our attention to the IEEE definition of the word specification [283]:

> ... a document that specifies, in a complete, precise, verifiable manner, the characteristics of a system or component.

Hence, the above implicitly states that, and this is also supported by the contributions found, a specification-based test case generation and selection technique can use a formal [13, 214] or natural language [181] to automatically or semi-automatically generate and/or select test cases. Specification-based test case generation and selection has seen a lot of work the last decades [4, 58, 173, 188, 214, 220, 225, 262, 263, 269].

The above research provide different views on specification-based test case generation, e.g. Avritzer and Weyuker's work on Markov chains [13] and Lutsky's work on information extraction from documents [181] somewhat deviates from the more traditional specification-based approaches, but are nevertheless highly interesting. Chilenski and Newcomb's work [50] on using formal specification tools for coverage testing is, albeit not directly relevant in the case of test creation, still interesting as an overview on how to use specification-based technology for other purposes than 'traditional' test case generation.

Statistical test case generation and selection techniques, on the other hand, are a mixture of several topics. Simply put, we describe statistical test case generation and selection as a technique that relies on statistically based computations to identify test cases. These techniques can consist of e.g. mutation analysis [18] and genetic algorithms [176]. With respect to mutation analysis and genetic algorithms much research has been presented lately. Some of the more interesting contributions, in our opinion, are [67, 90, 113, 145, 195, 291].

It is worth mentioning that several other test creation and selection techniques do exist, but they are, as far as we can see, different combinations of statistical or specification-based test case generation and selection.

In this context we find it worthwhile to stop for a while and look at how literature treats the classification of different automation levels. Test case selection is a topic where, usually, the authors make a distinction between different levels of automation. Additionally, the distinction we made regarding the intro- and retrospective approaches corresponds well with literature as we will see next.

Test case selection, is a subject that covers e.g. graphical user interfaces for viewing large amount of feedback and select test cases that should be stored for later execution, or techniques for selecting the right test cases to run, in a more automatic manner, e.g. by prioritizing test cases [254].

Test case selection can be divided into three distinct categories, with respect to automation. To begin with there is the manual test case selection, whereas a software engineer or researcher manually select which test cases should be executed.

Automated test case selection is, as mentioned previously covered in Rothermel's et al. work on prioritizing test cases [254]. This is also an example on using a retrospective approach, i.e. where there are a large number of test cases and a selection must occur for some reason.

On the other hand, Baudry's et al. [18] and Jamoussi's work [143] on trying to improve test cases during the actual creation of said test cases, i.e. generate the correct test cases immediately, is by us considered to be an example of an introspective approach, which is highly resource-intensive during the creation, as opposed to Rothermel's concept [254] where computational resources are needed after the creation of test

cases.

Concerning semi-automatic test case selection a nice example can be found in Choi's et al. work on the Mothra tool set [51], while Feldt's discussion (Chapter 2, [90]), concerning the possibilities and problems of semi-automatic test case selection, is recommended reading.

As we have shown, related work in this area corresponds well with the concepts of different levels of automation along with the intro- and retrospective approaches.

7.4.2 TEST EXECUTION AND RESULT COLLECTION

Related work, with respect to test execution and result collection, can today be found in literature by mostly looking at software testing environments. The definition of software testing environments can include a wide range of environments and frameworks and consists of both commercial offerings and research contributions. The environments can be very large, covering several techniques [70], but also very specific, focusing on one technique [305].

Kropp's et al. work on automated robustness testing [162] and Schimkat's Tempto framework [258] are both interesting software testing environments. Additionally, Vogel's CONVEX environment [288], which is used for testing significant software systems, is very interesting since it covers something rather rare, i.e. testing large systems.

Saff and Ernst's work on continuous testing [255], with the accompanying evaluation [256], should also be mentioned here as an enabling technology where automated testing is playing a key role. Finally, Eickelmann and Richardson's evaluation of software test environments provides us with a reference architecture [80].

Several of the above mentioned frameworks and tools collect, and in addition store, results according to some automation criteria.

7.4.3 RESULT EVALUATION AND TEST QUALITY ANALYSIS

At the moment, a fully automated result evaluation technique would in most cases involve artificial intelligence and since this research field is not yet fully applicable on software testing, researchers presently, by large, concentrate on oracles; and it is also here one can find most contributions.

The oracle approach, or the oracle problem [293], is according to Chen et al. [47] one of two fundamental limitations in software testing. By using the description provided by Chen et al. as a basis, we can describe the oracle approach as, a technique used for automatically creating an oracle. The oracle can then be used for validating the correctness of test output. In other cases there is no need for an oracle and only a comparator is used, which returns true or false depending on the test result.

The oracle approach can, when looking at related work, be divided into three categories. First, as is usually the case, manual creation, which is labor intensive and prone to errors. Secondly, oracle creation using specifications [89, 91, 247] and finally, automated oracle creation which basically aims to remove the need for an oracle all together [47].

As we mentioned, a completely automated oracle creation process is at the moment probably not feasible. This, we believe, has lead to researchers focusing on specification-based approaches, a technique wherein a formal or natural language is used for expressing a specification and then generate oracles.

It is worth keeping in mind that an old version of a software, that has been verified as behaving in the correct way, can under certain circumstances act as an oracle when developing a new version. According to Binder this is known as the *gold standard oracle* (pp. 925 in [27]).

In the area of test quality analysis we find statistical tools for examining test quality with respect to test coverage [312, 313], software test data adequacy criteria [294, 295] and general measurement theory [250]. As can be seen, test quality analysis deals mainly with measurement of test adequacy. One way to categorize test quality analysis is to divide it into context insensitive and context sensitive approaches according to Zhu et al. [314].

The context insensitive approach generates test data at random using a probability distribution (see Chapter 4 for an example). Here we find probability-based [43], confidence-based [133] and reliability-based [123] measurements. The context sensitive approach on the other hand, is based on test data being selected from a specific domain and constitutes techniques such as e.g program-limited (testing on the domain of the software), specification-limited (testing on the domain of the specification), structure coverage (control-flow and data-flow criteria) and fault-based measurements [200].

7.5 DISCUSSION

By first establishing a methodology to use (described in Section 7.2) and then stringently applying it, we believe that we have minimized the risks regarding not finding significant contributions in this research field. Of course, minimizing means that there might still be contributions worth covering in this chapter. On the other hand, we believe that all major *fields* that might be of interest when describing software testing and AuST, have been found. This, in addition to dividing this research field into logical topics and describing everything as a process, should further strengthen the thesis that all vital areas have been covered. The various papers covering, what we refer to as *supporting techniques*, are to some extent covered in this chapter but not seen as main

contributions.

A few key areas are worth discussing with respect to this contribution. First, the distinction we make between test data and test cases. Presently, in literature, researchers sometimes do not make this distinction. We find this to be unfortunate, since in our opinion there are differences regarding problems, solutions and methodology when creating either test data or test cases. Unfortunately the problem regarding definitions and classifications of terms is more widespread and covers many areas; words, classifications and different terms are used interchangeable thus leading to confusion.

Secondly, since no researcher follows one and the same methodology or model for describing their research, it leads to even more confusion. This, in the end, makes it more difficult for researchers to get a good overview of this research field altogether.

Finally, the question of viability regarding some of the tools and techniques that we covered might be questionable. How do we know that the more recent techniques etc. will have an impact on this research field in the future? In our case we answer that question by simply selecting contributions that are published in proceedings and journals that are considered by researchers to be of high quality [109].

7.6 CONCLUSION

In this contribution we covered a large research field that has been active for several decades. By carefully following a described methodology, and then divide this field into naturally formed classifications, we believe that all major topics have been found.

In total, this chapter covered 69 references in the area of software testing. Of these, 45 were focused on test creation, 8 looked at test execution and result collection and 16 focused on result evaluation and test quality analysis.

Chapter 8

A Model for Classifying Automated Aspects Concerning Software Testing

Submitted to Software Testing,
Verification and Reliability

R. Torkar, R. Feldt &
S. Mankefors-Christiernin

8.1 INTRODUCTION

The total cost of inadequate software testing infrastructures, in the USA alone, is estimated to be as high as $59.5 billions per year [205]. Even source code which has been thoroughly scrutinized, averages three faults per hundred statements [22]. Boehm [30] has previously pointed out that about 60 percent of total product defects can be uncovered using inspections; the remaining faults show up in different types of application testing. In addition to the above it has been shown that application testing takes at least 50 percent of the total product labor costs [22].

One obvious conclusion is that software engineers in general need to write better software, but the people who write software are just - people. Another conclusion that can be made is that software has grown in size the last decades while computer resources (such as CPU hours) have become cheaper when at the same time human resources remain quite expensive. Thus, certain tasks which can be suitable for automation, e.g. software testing, should be automated to a high(er) extent. Tools are a vital contribution to the software engineer's landscape, and in the future these tools are likely to become even more focused on automation.

The last couple of decades more and more tools and frameworks have evolved. Most notably, different unit testing frameworks [138, 305] have gained widespread acceptance, helping test engineers to some extent create test cases by facilitating rudimentary support in creating skeletons which the developer then fill out. But the true value of these frameworks is not the support for developing test cases, but rather the role they play as regression testing frameworks [21, 255].

In our opinion software testing and especially the automated aspects thereof is and will continue to be an interesting research field in the future, with the potential to increase project success by automating a task that is often discarded when deadlines creep closer (see Chapter 2 for an overview regarding some of the problems facing software industry).

As such, (automated) software testing needs to be clearly defined, enabling researchers to understand each other in a satisfactory and concise manner. To further prove the point, we can see how software vendors describe their tools to be fully automatic with no additional need for developers to interact with the tool. A closer examination usually reveals that the tool might have some automatic aspects but by large is manual. The same problem area exists in academic research as well, e.g. searching databases for research contributions and in the end discovering that the contributions found, has in some cases nothing to do with automated software testing.

Compare this to Marinov and Khurshid's TestEra framework [188] where they make a more humble claim:

> TestEra automatically generates all non-isomorphic test cases [...]

The problem today is not that there is a lack of definitions per se, but rather that these definitions are interpreted differently. The definitions need to be collected and assembled into one comprehensive and uniform model, so as to provide researchers and developers with a clear taxonomy of this topic, thus hopefully increase researchers' comprehension of this topic in addition to avoid miscommunication. Marinov and Khurshid's claim above, is an example of how researchers should present their work—precisely and delimited if possible.

The model and its accompanying definitions, which we present here, are intended to be used mainly in three ways:

- To identify different characteristics, especially concerning automation, in present software testing tools, techniques and processes (technique $\mathbf{t}$ is classified according to definition $\mathbf{X}$ in our model).
- As guiding laws (technique $\mathbf{t}$ must support $1 \ldots n$, hence definitions $\mathbf{X}$ and $\mathbf{Y}$ in the model should be part of the requirements specification, thus fulfilling the requirements with respect to automation).
- To classify to what extent a tool, technique or process is automated.

In this chapter the model will be used for describing automated software testing, while later (Section 8.5) elaborated on as a framework for the software testing area as a whole.

The next section (Section 8.2) introduces the reader to automated software testing and its subtopics. In this introduction we also try to, as precisely as possible, define each entity in this area. The following section then present the model we propose (Section 8.3), while we later discuss the applicability of the model (Section 8.4). In the subsections following Section 8.4, a number of examples are covered when we apply them on our model. Finally, this contribution ends with a discussion (Section 8.5) and conclusion (Section 8.6).

8.2 SOFTWARE TESTING DEFINITIONS

By software testing we imply the process of identifying the correctness, completeness, security and quality (i.e. functionality, reliability, usability, efficiency, maintainability and/or portability [140]) of developed software. *Automated* software testing is no different, in the sense that it is applied to the exact same process as software testing, but with the addition that the process should be automated to some extent.

To be able to describe automated software testing in particular, and software testing in general, we need to systematically define and cover entire research areas, i.e. by simply collecting and systematizing what is available. Since we are covering a large research area, the model will be generally applicable for most types of software testing. But as will be evident, the strength will be in the way the model can be used to describe specific disciplines of software testing—in our case, automated software testing. To begin with, some basic definitions need to be established when covering automated software testing, i.e. the definitions of automatic, manual and semi-automatic.

According to The Oxford English Dictionary [261], the original definition of automatic is:

ORIGIN Greek *automatos* 'acting by itself'

Taking the Greek definition into account, and further conclude that one *part* of a software has the responsibility to fulfill a major *task* and a software is defined by the integration of N parts, we thus define automatic, manual and semi-automatic as:

Definition 5 (AUTOMATIC) *A software or a part of software is automatic if, and only if, the task executed by the software/part is performed autonomously with respect to human intervention.*

Remark 1 (HUMAN INTERVENTION) *Human intervention is not allowed, but interaction with other entities, e.g. a system, is.*

The above nevertheless means that a human, in most circumstances when it comes to software testing, must actually execute, i.e. start, the process. On the other hand, a human must not interfere with the execution as such.

Definition 6 (MANUAL) *A software or a part of software is manual if, and only if, the task executed by that software/part is performed by a human which takes all decisions.*

Definition 7 (SEMI-AUTOMATIC) *A software or a part of software that is neither automatic nor manual.*

A future extension of the above definitions is of course to define different, more fine-grained, levels of automation. Obviously there exist a wide spectrum ranging from manual to automatic [222, 223, 260], and we are well aware of the intricate problems of defining different grades of automation due to the vast amplitude of technologies (as will be discussed in Section 8.5). Despite that, the above three definitions are sufficient for our current purposes.

Next the reader will be provided with, what we consider to be, a reasonable classification of (automated) software testing into different entities.

8.2.1 ENTITY DEFINITIONS

Automated software testing can be divided into several entities by looking at how traditional software testing is defined by different people and organizations, notwithstanding the somewhat broader definitions by Dijkstra that "program testing can be used to

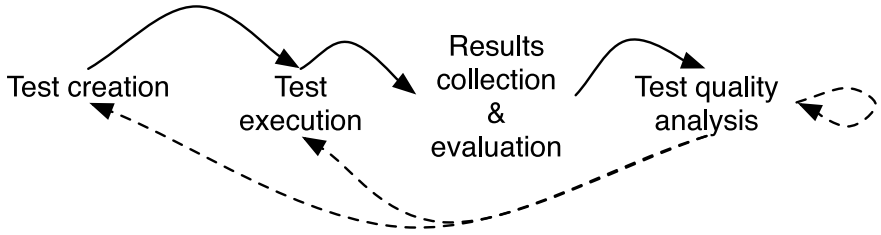

Figure 8.1: A general overview of software testing. Several iterative movements can be discerned—here depicted with dashed arrows.

show the presence of bugs, but never their absence", and Myers claiming it to be "the process of executing a program with the intent of finding errors."

A more practical, hands-on, view [204] of software testing is the need to exercise the software under test by using a set of test cases. By then performing a series of steps (possibly executing the software itself) the result is collected, evaluated and test quality analysis is performed (Figure 8.1). It might be worth noticing that, in some cases, different steps can be performed several times, while test quality analysis is executed last, after several iterations have been performed.

Some researchers' view on software testing resemble our view [93], while others prefer a somewhat simpler view [150, 233] in that they define it on a meta-level. We believe our view to be a good middle road i.e. neither too detailed nor too abstract.

Using Figure 8.1 as a basis, we now define the fundamental entities of our model, starting with the software under test.

Definition 8 (TESTEE) *A testee, denoted* **T***, is a part of, or an aspect of, a software under test.*

Thus, one aspect of **T** is the entire software, which is the case in e.g. system testing. Other aspects might be e.g. security, performance or domain-specific behavior.

Evidently we diverge from IEEE's definition of 'software item' [283]. The main reason for this is that it is not always the software item per se that is tested, but rather certain aspects of the testee. Thus, to not confuse the reader with IEEE's definition, we introduce the word testee.

We next define several fundamental entities using a 'bottom-up' perspective.

Table 8.1: An overview of how the automation levels are inherited.

	Man	**Semi**	**Auto**
Man	man	semi	semi
Semi	semi	semi	semi
Auto	semi	semi	auto

Definition 9 (TEST DATA) *Test data, denoted* Γ*, is the input(s), needed for testing the testee.*

Obviously, Γ can equal $\varnothing$. In addition, it seems, Γ can equal one input as can be the case in e.g. unit testing. When it comes to automation aspects, Γ can be generated or selected manually as is done in e.g. unit testing, or semi-automatically or automatically as is the case in e.g. random testing.

The question of generation and selection is interesting since if a technology implements both, then in our model, to be able to be classified as automatic, both generation and selection needs to be classified as automatic (logical conjunction). In all other cases, the rule 'semi-automatic is king' establishes the automation classification (see Table 8.1 for an overview). This rule is also applied on all entities in this case.

Definition 10 (TEST FIXTURE) *A test fixture, denoted* Φ*, is the environment wherein* Γ *acts.*

Thus, Φ sets up and tears down the surroundings in which $\mathbf{T}$ with Γ is executed, e.g. the steps taken before and after $\mathbf{T}$ is tested. In some cases, e.g. certain types of system testing, Φ is simply equal to $\mathbf{T}$ and thus in effect makes Φ non-existent, while in other cases Φ consists of $\mathbf{T}$ itself and a variant thereof ($\mathbf{T}'$), i.e. as is the case in mutation testing [132].

With respect to automation aspects, Φ can be classified as being selected or generated:

- manually—e.g. a human implements `SetUp()` and `TearDown()`.
- semi-automatically—e.g. a human selects a specific Φ from $\{\Phi_1, \Phi_2, \ldots, \Phi_n\}$ depending on $\mathbf{T}$ or Γ.
- automatically—e.g. using genetic algorithms the software generates and/or selects Φ.

The same applies to Φ as to Γ concerning inheritance of classifications of automation, i.e. 'semi-automatic is king'.

Definition 11 (TEST EVALUATOR) *A test evaluator, denoted Δ, is a superset of the expected output $\hat{o}$ ($\Delta \supset \hat{o}$). Δ compares the output o with the expected output $\hat{o}$.*

* *The output o can have one of three states:*
  * $o = \hat{o}$
  * $o \neq \hat{o}$
  * $o \approx \hat{o}$

Remark 2 (THE NATURE OF OUTPUT) *The last state, $o \approx \hat{o}$, occurs when an output can be approximately correct, i.e. the output needs to adhere to certain properties and not a particular value per see.*

Δ can be classified as being selected or generated:

- manually—e.g. a human is Δ.
- semi-automatically—e.g. from $\{\Delta_1, \Delta_2, \ldots, \Delta_n\}$ a human selects the Δ to be used.
- automatically—e.g. Δ is an old version of the software which can provide the known correct answer (this is also known as the *gold standard oracle* [27]).

Definition 12 (TEST CASE) *A test case, denoted Θ, is a tuple $\{\Gamma, \Phi, \Delta\}$.*

Thus, if all elements of the tuple are classified as being generated or selected automatically then Θ is classified as automatic (logical conjunction). To put it simple, for an entity (in the above case a tuple) to be classified as automatic all its sub-topics need to be automatic, i.e. boolean operator `AND` (see Definitions 5–7 and the paragraph covering the aspect of inheritance on the previous page).

Worth taking into account, e.g. in the case of the test case entity, is how automation levels are inherited upwards in the model. If we want to classify a technology that has automatic test data generation but manual test fixture generation (disregarding test evaluator for now), the definition of test case is to be classified as semi-automatic. As another example, if the test data entity is classified as semi-automatic, the test fixture as manual and the test evaluator as automatic, the test case will, again, be classified as

semi-automatic. Table 8.1 (page 118) provides an overview on how different classifications are inherited. So in short, semi-automatic properties are emphasized.

The above application of automation levels in our model leads to some interesting results when, in Section 8.4 and its subsections, our model is applied on a number of different tools and technologies.

Definition 13 (TEST RESULT COLLECTOR) *A test result collector, denoted* Π*, collects, and in some cases stores, the output* o *from the execution of a test case* Θ *or a set thereof.*

Definition 14 (TEST SUITE) *A test suite, denoted* Ξ*, is a superset of* Θ *(*$\Xi \supset \Theta$*).*

Definition 15 (SPECIFICATION) *A specification, denoted* β*, provides the requirements* $\mathbf{T}$ *needs to adhere to.*

Definition 16 (TEST SPECIFICATION) *A test specification, denoted* $\bar{\beta}$*, is a subset of a specification (*$\bar{\beta} \subset \beta$*) and stipulates e.g. a set of test cases* $\{\Theta_1, \Theta_2, \ldots, \Theta_n\}$.

Definition 17 (TEST ANALYZER) *A test analyzer, denoted* Σ*, provides the necessary functionality to determine if data provided by a test result collector* Π *is sufficient for determining if a set of test specifications* $\{\bar{\beta}_1, \bar{\beta}_2, \ldots, \bar{\beta}_n\}$ *are fulfilled. In addition,* Σ *might be able to answer if* β *has been fulfilled and to what extent this is the case.*

Σ can thus be used for e.g. prioritizing test cases, providing input for creating new test cases or changing the order of test case executions. Additionally, Σ can be classified as being conducted:

- manually—e.g. a human is the test analyzer and decides if $\mathbf{T}$ fulfills β.
- semi-automatically—e.g. a software provides a human with several alternatives concerning test prioritization, which the human can select between.
- automatically—e.g. a software is the test analyzer and decides if $\mathbf{T}$ fulfills β by e.g. concluding that all $\bar{\beta} \in \beta$ have been found.

Note here that Σ has a slightly different focus on the automation aspects since we here look at how Σ is exercised, and not how Σ is generated or selected as is the case with Γ, Φ and Δ.

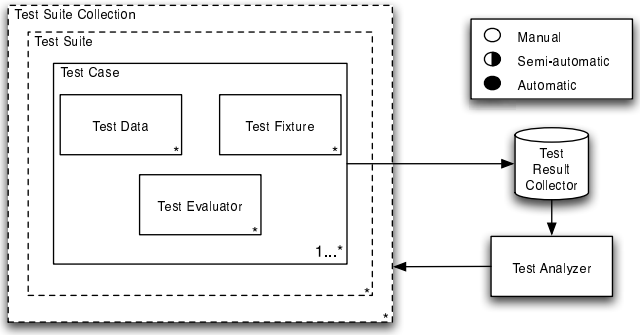

Figure 8.2: Proposed model for characterizing automatic aspects concerning software testing.

8.3 PROPOSED MODEL

All definitions are appropriate to classify according to the automation levels and as such can have an accompanying circle which would indicate the level of automation. The circle can be either empty, half filled or completely filled to indicate the degree of automation (as per Definitions 5–7). By assembling all definitions that apply, one ends up with Figure 8.2 which in this case is the proposed model.

By comparing our model (Figure 8.2) with the original process description as introduced in Figure 8.1 (page 117), one can see that they look somewhat differently. First of all, several entities has been created from the process model (and thus also defined as nouns). Secondly, due to the iterations, that are usually part of software testing, we chose to define our model with an iterative perspective as well. The iteration is illustrated by an arrow going from the test analyzer entity to the 'highest common denominator', i.e. the test suite collection entity (Figure 8.2). Thirdly, since we use an entity perspective, we have chosen to include the test evaluator to the test case entity (as per Definition 12). Finally, by allowing the aspect of quantity in the model (illustrated by e.g. $1 \ldots *$), we make a clear distinction between running a simple test case, test suite or a collection of test suites.

As is shown later, the focus on entities simplifies the automation classification tremendously, since, if all entities are defined as being automatic (for a specific technology) then the technology is automatic per definition.

It is now appropriate to look at some other models and definitions, and compare

how they relate to our classification model. First of all, models in literature can be divided into two categories when it comes to software testing, either methodologies in software testing (pp. 4–5, [146]) or different levels of software testing (pp. 187–188, [146]). In some cases we see other models such as Sommerville's description on the software testing process (pp. 539–540, [267]); unfortunately that model, as other similar descriptions of software testing, is not detailed enough to be used for classifying and comparing techniques or simply lack the distinctions concerning automation levels which we emphasize. These models were never intended, in the first place, to be used for that purpose—our model is.

Secondly, considering definitions, it is worth emphasizing that we lean towards Parasuraman's et al. [222, 223] and Sheridan's [260] work on the levels of automation of decision and action selection. But, in our model we provide rudimentary definitions of the different levels of automation. We do this mainly so as to facilitate the usage of the model in a straightforward way since we find Parasuraman's et al. and Sheridan's definitions to be somewhat misleading to use in the context of automated software testing (this will be covered more in Section 8.5).

We will demonstrate that our model is more fine-grained and thus provides the user with more freedom, especially regarding establishing the level of automation.

8.4 APPLICATION OF MODEL

Before adopting a model as the one we propose, the questions of validity and usability, needs to be investigated.

Validity was mainly ensured by searching research databases for literature concerning (automated) software testing. As Kitchenham et al. [155] has explained it is important to justify the method of search while discussing the risks associated with the search method. To begin with we performed our search using three widely used search engines: IEEE Xplore [304], ACM Digital Library [2] and CiteSeer [52]. After finding a body of publications which focused on software testing and automated software testing we immediately saw that some publications were of more interest than others. We picked four proceedings that showed up the most and performed a hand search of the proceedings from these conferences at least ten years back in time:

- International Symposium on Software Reliability Engineering (ISSRE).
- International Symposium on Software Testing and Analysis (ISSTA).
- International Conference on Automated Software Engineering (ASE).
- International Conference on Software Engineering (ICSE).

For each contribution that was of interest, i.e. covering software testing in general or automated software testing in particular, a short summary was written and the main parts were assembled in Chapter 7.

It is worth pointing out that each search engine was used for searching the *entire* database and thus not restricted to certain proceedings or journals to start with. In our opinion, in this way, a significant portion of the interesting contributions were gathered. In addition, this procedure should minimize experimenter bias and the risk of faulty selection (internal validity) [36, 41, 207]. It is worth pointing out that we did not need to assemble all references in this area; we simply needed a sample with which to evaluate the model with.

In the end, the model is applicable for all references found by our research as described in Chapter 7. This of course means that it is not unlikely that some references are lacking but, suffice to say, a portion of related work taken from Chapter 7 is presented, described and categorized in this chapter, which in the end would lead to having a theory that best explains the results (construct validity) [41].

The references found were divided into three populations: test data/fixture generation and selection, result collection and, finally, result evaluation and test analyzer.

The test data/fixture generation and selection population consists of [3, 4, 9, 13, 18, 26, 49, 50, 51, 53, 58, 65, 67, 71, 78, 90, 92, 113, 122, 143, 145, 158, 161, 168, 173, 176, 181, 184, 188, 195, 204, 214, 220, 224, 225, 240, 254, 262, 263, 269, 282, 283, 291, 306] in addition to Chapter 4, while the result collection population is described by [70, 80, 162, 255, 256, 258, 288, 305].

Finally, result evaluation and test analyzer is covered in the following contributions [27, 43, 47, 89, 91, 123, 133, 200, 247, 250, 293, 294, 295, 312, 313, 314].

Next, one reference was randomly selected from each population as defined above. By randomly (ensuring an unbiased selection [155]) selecting one reference from each population (applying blocking as discussed on pp. 102–104 in [36]), one can thus present different cases for a deeper analysis while at the same time control external validity [41]. The randomly selected references and the total sum of references per category can be found in Table 8.2.

By using the definitions introduced in this chapter, one can now establish where a certain technique can be placed. Before covering two introductory examples, three randomly selected examples from a defined population, and a comparison of two techniques, we first describe several 'old school' techniques. Note here, that we are not covering a specific implementation of these techniques, but rather look at them from a more general point of view (more comparisons of different techniques etc. will take place in Chapter 9).

To start with, partition testing [204, 267, 296], which boundary value analysis can be considered to be a subset of, is traditionally part of the test case entity as per our def-

Table 8.2: The number of references and the selected reference for each population.

Category	Num. of refs	Selected ref
Test data & test fixture	45	van Aertryck et al. [4]
Result collection	8	Schimkat et al. [258]
Test evaluator & Test analyzer	16	Fenkam et al. [91]

inition. As such, these technologies focuses on selecting the right partitions or values to test (defined as the test data entity).

Examining structural testing [209] closely, one reaches the conclusion that it basically belongs to the test analyzer entity, due to the fact that it provides answers on how well a software is exercised under test.

Finally, mutation testing [132, 216], in which two or more program mutations are executed using the same test cases to evaluate the ability to detect differences in the mutations [283], is somewhat more trickier to define. In this case one needs to look at the testee ($\mathbf{T}$) and its mutations $\{\mathbf{T'}_1, \mathbf{T'}_2, \ldots, \mathbf{T'}_n\}$ as part of the test case (Θ) and, more specifically, the test fixture entity Φ.

Next, as an example, Claessen and Hughes' QuickCheck tool [53] is to be categorized (a technology which in many ways 'breaks' with current views on software testing in its implementation, thus providing the reader with a non-traditional and, perhaps, unknown case).

After that we cover a more widely known case—the XUnit framework [305]—a technology probably known by researchers and industry which, in addition, is a de facto standard with respect to test execution amongst other things.

8.4.1 CLASSIFYING QUICKCHECK

According to Claessen and Hughes, the QuickCheck tool can be used for random testing of Haskell[1] programs [53]. QuickCheck has a data generation engine which, by using a test data generation language, generates random data. These values can be of different types, and custom values can be specified manually (even though most types are defined already). By looking at the definition of test data (Definition 9) it is obvious that QuickCheck qualify for this criterion, in addition to being an automatic test data generation technique with a semi-automatic selection approach (thus classifying it in the end as semi-automatic). Test fixtures are created manually, as far as we understand.

[1] `http://www.haskell.org`

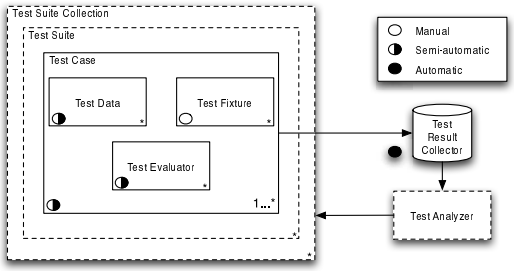

Figure 8.3: A classification of QuickCheck according to our model.

In addition to this, according to Claessen and Hughes, QuickCheck uses specification-based oracles [53]:

> We have taken two relatively old ideas, namely specifications as oracles and random testing[...]

Thus, since QuickCheck uses property-based specifications when generating oracles it would fall into Definition 11 (test evaluator) with a semi-automatic approach, as is shown in Figure 8.3 (oracles are more or less always executed automatically, but here we look at generation and selection).

Next we continue by going through the rest of the model. Result collection (with no storage) is performed in an automatic way in QuickCheck. Finally, there is no indication [53] that a test analyzer (Definition 17) is part of QuickCheck.

8.4.2 CLASSIFYING XUNIT

Different unit testing frameworks [305] have, the last decade or so, gained a widespread acceptance. Many developers usually mention unit testing in the same sentence as automated software testing. Is this correct? What is so automatic about unit testing frameworks?

First of all, by looking at any description of unit testing frameworks, one can see that test data generation and selection, in the case of unit testing frameworks, usually is performed manually.

Test fixture creation can, in some circumstances be considered semi-automatic (the framework generates stubs which the developer needs to fill in). In the same manner

can test case selection be seen as semi-automatic (when there is a selection process and the developer selects the test cases to run).

Result collection and execution is automatic. This is considered to be one of the strengths with unit testing frameworks, and as such indeed further refined by Saff and Ernst in [255, 256].

Comparison, in the case of unit testing frameworks, can be performed in an automatic way as long as there exist oracles. Oracles, on the other hand, need to be created manually and are usually (in the case of unit testing frameworks) seen as part of the test case creation step, i.e. the developer is the oracle by providing the correct answer when writing an assert. Finally, unit testing frameworks traditionally provides a test engineer with information from the test execution (fail and pass flags, execution time, coverage)—thus the test analyzer entity should be classified as being semi-automatic since unit testing frameworks can not, usually, decide if all $\bar{\beta} \in \beta$ has been found. Rather, unit testing frameworks, present the engineer with information if $\bar{\beta}$ has passed or failed.

In Figure 8.4 we now see how different levels of automation are scattered throughout our model. Thus, XUnit is according to this classification not automated as such, mainly since the techniques for generating and selecting test data, evaluators and fixtures are usually performed manually. But, and this is important to emphasize, XUnit has *some* entities that are automatic.

Especially worth noticing in Figure 8.4 is that the test evaluator is classified as being manual according to our usage of automation levels. This is rightly so since the creation of an evaluator, traditionally, is performed manually. Of course, the much simpler comparison process is automatic, but that on the other hand is more or less a rule in most software testing processes. To classify the result evaluator automatic in this case would be grasping for too much.

8.4.3 RANDOM EXAMPLE I—TEST DATA AND TEST FIXTURE GENERATION AND SELECTION

CASTING, a formally based software test generation method, as presented by van Aertryck et al. in [4], is a method for generating and selecting test data and fixtures. The initial version of CASTING describes test cases to be generated by using the B Formal Development Method notation [1]. This way a developer can build testing strategies, translate these to test case specifications and, in the end, execute them on the software.

When categorizing CASTING we end up with Figure 8.5. The selection technique is automatic while the generation technique is considered semi-automatic since

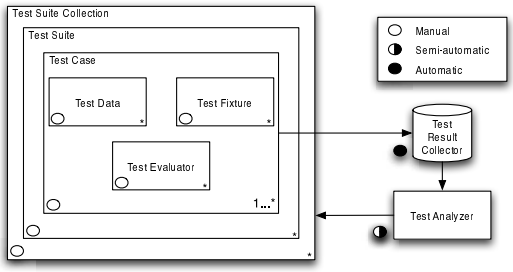

Figure 8.4: A classification of XUnit according to our model.

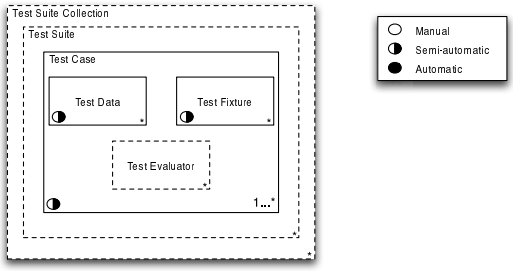

Figure 8.5: A classification of CASTING according to our model.

it evolves from a B specification [1] (which is usually created manually). Thus, since we are applying logical conjunction on this level, the end result is a semi-automatic classification of test data and test fixture.

The one question mark one can find in this particular case is if CASTING implicitly, by using a constraint solver, can be seen as containing a test analyzer. In our opinion this is not the case, since a constraint solver is considered to be a compulsory part when extracting and generating test cases from a B specification.

8.4.4 RANDOM EXAMPLE II—RESULT COLLECTION

In Section 8.4 several test execution and result collection references were presented and from these, Schimkat's et al. Tempto framework [258] was randomly selected. Tempto

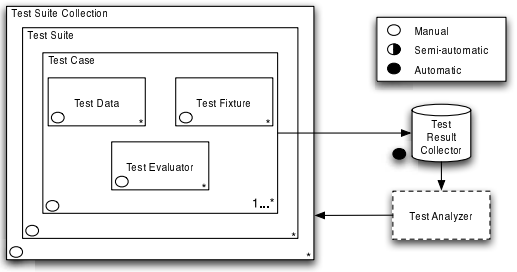

Figure 8.6: A classification of Tempto according to our model.

is, according to Schimkat et al., an object-oriented test framework and test management infrastructure based on fine-grained XML-documents. The Tempto framework focuses on test execution, result collection and to some extent result evaluation.

Tempto supports automated test execution (from prepared test suites), collection and storage of results and, finally, manually written comparators. In Figure 8.6 one can see the Tempto framework classified according to our model.

The authors mention that external management components, e.g. an expert system which could analyze test reports, can be added to the framework. Unfortunately, we did not find any proof that this had been done at the time of the writing.

8.4.5 RANDOM EXAMPLE III—RESULT EVALUATION AND TEST ANALYZER

The final, randomly selected example, is Fenkam's et al. work on constructing CORBA-supported oracles for testing [91]. Fenkam et al. show a technique for constructing oracles by using a specification. The specification is used to "automatically verify the results of operations", according to the authors.

Since the specifications for creating oracles are semi-automatically created and we are not dealing with 'traditional' comparators, there are no difficulties categorizing that particular part (Figure 8.7). Somewhat more difficult to understand is the authors' claim that they support automatic test creation and test case execution since they have no actual example [91]. Ultimately, it is clear that the specifications for creating oracles *could* be re-used (from the earlier test creation phase).

Finally, the test analyzer can be classified as manual since, in the end, it adheres to the test specification which is manually created.

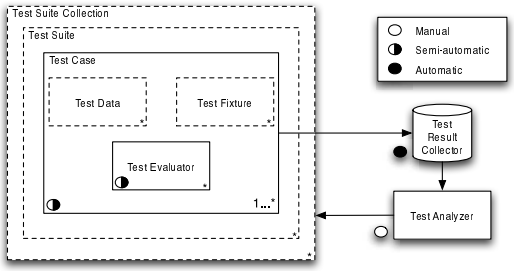

Figure 8.7: A classification of Fenkam's et al. work according to our model.

8.4.6 COMPARING AUTOMATION ASPECTS

The previous examples provide some insights with respect to the usage of the model. While the question on *how* one should use the model is important, the question on why is equally so.

In this subsection we will classify two similar approaches which, nonetheless, differs in how automation aspects are implemented. By doing so, we intend to show one of the strengths with this model, whereas it provides support in deciding what is more suitable to use from a strict automation perspective.

As an example we have chosen to compare Tempto (Random Example II, Subsection 8.4.4) with Eclipse [74]. Since Eclipse's default setup does not have any substantial software testing features, we include the following plug-ins: GERT [60], djUnit [69], EclipsePro Test [83] and continuous testing [57]. Both Tempto and Eclipse have support for, or are focused on, the Java programming language. In addition, Tempto and Eclipse (with the included plug-ins) attempt to support the developer with automated features (in this case, in particular, software testing), while at the same time being seen as frameworks which can be extended with miscellaneous features.

Figure 8.8 (next page) illustrates the differences, concerning automation aspects, between Eclipse and Tempto. In this case a developer might look at the figure and ask the question: "Does Tempto or Eclipse support automation in those areas where we have a weakness and how does that fit into the rest of our software testing process?"

Obviously, by looking at Figure 8.8 one can see two issues being revealed. First, none of the frameworks support automation to any higher extent (this by itself is noteworthy). Second, if one must select one of the frameworks then Eclipse (due to having more entities classified as at least semi-automatic) might be tempting to select. But on

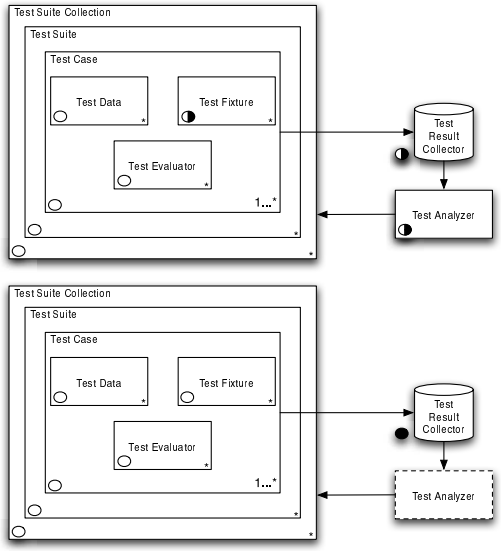

Figure 8.8: A comparison between Eclipse (upper figure) and Tempto (lower figure).

the other hand, in the case of Tempto, a software development team might have good test evaluators and test fixtures in place beforehand, hence making Tempto the obvious choice. In the end, the question of language support (in the case of Eclipse the Java programming language is very much supported) might be the point tipping the scale, which ultimately leads us to the next section (more comparisons and analyses will be presented in Chapter 9).

8.5 DISCUSSION

Applicability. The previous section presented several contributions and how they mapped against the model. However, the question of applicability, even though several examples have been presented and many more have been covered in the feasibility studies for this chapter (i.e. Chapter 7), is an interesting one since there will always be a risk that some tools and technologies might not be applicable to the model. During the review process of this chapter some contributions were brought forward which made

us reassess the model, usually by altering some details (e.g. by adding the distinction between test data and test fixture). What is the point of having a model which can not be used for describing the most common cases of reality?

One issue, not dealt with in our examples, is how one illustrates the behavior were a tool changes automation aspects over multiple iterations. In this case the model provides the user with two choices. Add new circles for each iteration to each entity (and as a subscript add a number indicating the order of iteration) or, if the iterations are few, show each iteration in its own view of the model where the new classification is shown for that particular iteration. After all, the different entities, e.g. $\Gamma, \Phi, \Delta, \Theta$, can simply be placed in a table for even easier comparison of different iterations if one would like.

Distinctions. One critique this model might face, is the distinctions made on the different definitions concerning automation—it might be worth mentioning that researchers and industry partners *using* automated software testing have been remarkably relieved by the clear distinctions, something many believe is lacking in marketing and research descriptions today. While, on the other hand, engineers and researchers *developing* these tools have been more dismissive in some cases.

The reason for the above critique is clear to us. A researcher working on developing these types of tools and a company selling them are, ever so interested in using the magical word *automatic* in either the title of their research paper or in the adds for their products. Unfortunately, neither of them are aware of the frustration end-users feel, when not being able to find the right tool for the job.

As an example, take contract-based software testing. If a developer needs to manually develop e.g. contracts for their oracles—then should the word automatic be used at all, as it is in some cases? After all, a comparator or oracle is usually performing the actual comparison automatically when being executed. Again, a more careful use of certain words might be useful to better describe ones research or product.

In our opinion, in the future, the levels between automatic and manual will probably be numerous. As an example we have Parasuraman's et al. [222, 223] and Sheridan's [260] work on defining different levels of automation. Unfortunately the current levels they propose are not applicable directly to automated software testing, hence indicating that more work is needed in this area.

Extensions. The question of future extensions is an interesting one since it is likely that the model might be used for more purposes than originally expected. For example, if an engineer searches for a technology with a certain level of automation, it is also likely that (s)he needs it for a particular domain, programming language, paradigm, or focusing on different levels of testing (e.g. unit, sub-system or system). For this purpose we propose a note next to the model describing certain aspects which would simplify a search when added to a classification (c.f. Figure 8.9).

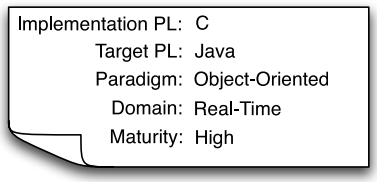

Figure 8.9: A note describing various features not part of the actual model. In this case an imaginary technology is described which is implemented in the C programming language, targeting software built in Java (using the object-oriented paradigm), contained in the real-time domain and with a high maturity.

8.6 CONCLUSION

In this chapter we defined several key parts of software testing especially focusing on the automated aspects thereof. Using these definitions, we created a model which can be used to e.g. compare, classify or elaborate on automated software testing. The model had its origin in the traditional software testing process but was then further refined using several supporting definitions which acted as a foundation to the model.

The validity of the model was covered by exemplification (two), by applying the model on three samples from a random population, and by comparing two similar techniques with each other. In addition several other traditional technologies were covered and the question of applicability, distinctions and future extensions was discussed and elaborated on.

In short, the proposed model can be used by the research community as a way to classify tools and technologies while at the same time significantly reduce the time spent to understand a technology as well as clearly grasp different automation aspects. In addition, industry could use the model to more clearly present how and in what way a specific technology adheres to a particular automation aspect.

The software engineering community needs to stop being fragmented and start contributing to a wider research agenda, while at the same time agree on some standard way of communicating. This model might be the first step towards such unification.

Chapter 9

Software Testing Frameworks—Current Status and Future Requirements

Submitted to The Journal of Systems and Software

R. Torkar, R. Feldt &
S. Mankefors-Christiernin

9.1 INTRODUCTION

Software testing plays a vital part in software engineering as a way to ensure that quality aspects have been reached. A study, performed by the National Institute of Standards and Technology in 2002 [205], showed that the costs for having inadequate software testing infrastructures (e.g. software testing environments and frameworks), in the USA alone, was estimated to be as high as $59.5 billions per year. Thus, the development of software testing frameworks, or software testing environments, play a crucial role in industry to reduce costs and improve quality [162, 246, 288].

Software testing frameworks that help test engineers with automatic and semi-automatic software testing are today usually concentrated on doing one thing, and most of the time do that quite well. Most notably, different unit testing frameworks [21, 138, 305] have gained widespread acceptance, helping test engineers to, to some extent create, but more importantly 'administrate' test cases, thus acting as regression testing frameworks. In our view, automation is mainly about efficiency (the state or quality of being efficient [292]), not effectiveness (producing a desired or intended result [292]). For many software testing techniques, effectiveness is the natural prolongation of efficiency, i.e. by being able to test more within a given time frame, because of increased efficiency, it will inadvertently lead to more faults being found. In other situations, increased efficiency will simply reduce the cost and time of a certain testing approach and make new resources available to other parts in a software development project. In principle such resources could very well be used for human-based testing such as code inspection to complement the automated approach.

Some frameworks that combine different testing approaches do exist, as will be shown later in this chapter. These types of frameworks usually combine a low level approach, such as unit testing, with a high(er) level approach, e.g. acceptance testing. We like to call this the *vertical perspective*.

On the other hand, frameworks that combine different testing techniques within the same principal level of testing, i.e. having a *horizontal perspective*, are harder to find, especially when considering how semi-automatic and automatic is defined by us in the previous chapter (page 116). Since, as will be shown later in this chapter, there is a lack of frameworks having a horizontal perspective the focus will primarily be on frameworks that try to combine one or, preferable, several aspects of a typical software testing process (always having automation aspects in mind). Additionally, this chapter elaborates on how software testing frameworks need to evolve in order to help developers that need to combine several different test activities under one and the same framework. The reasons, for having one framework combining different activities and techniques, are mainly:

- There is today not sufficient knowledge concerning the effectiveness and efficiency in combining testing techniques (see [148] and Chapters 2 and 5–7), i.e. introducing several techniques will most likely decrease efficiency (more time will be used per found fault), but on the other hand increase effectiveness (number of found faults might increase).

- The software testing community need to evolve and, to a higher extent, collect 'real life' data to better understand the current situation concerning different test techniques' efficiency [148, 306].

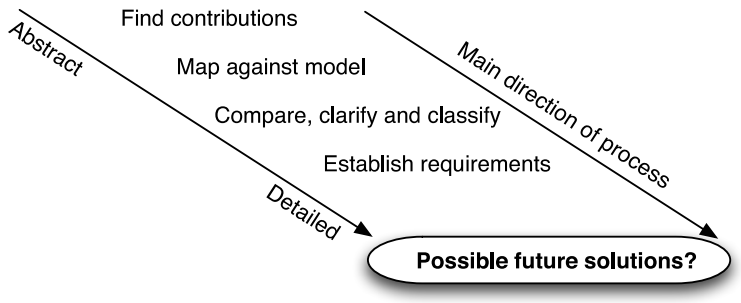

Figure 9.1: A general overview of the methodology used in this chapter.

- By allowing all test activities, as performed in a typical software testing process, to be performed within one framework certain synergy effects can be reached.

Simply put, by collecting data from different software projects a higher knowledge, regarding which software testing techniques are used the most and their respective efficiency, will lead to software testers being able to more easily select the most appropriate strategy for each unique software testing problem. This information can be obtained using a qualitative research approach. Nevertheless, a quantitative approach, as we propose, will give additional information of a statistical nature and data collection can be spread geographically and over time more easily.

By selecting a population of existing software testing frameworks (Subsection 9.2.2), and map the population against a software testing model (as introduced in Chapter 8), it enables us to compare, clarify and classify these frameworks. This then, gives an indication of what is missing and hence desiderata can be established stipulating what requirements a future software testing framework should fulfill (Section 9.3). Finally, this chapter ends with a short conclusion (Section 9.4).

Figure 9.1 gives an overview of the methodology used in this chapter.

9.2 DEFINITIONS AND POPULATION

In Subsection 9.2.2, several interesting software testing frameworks and environments are examined (the observant reader will notice that several of the samples have been introduced already in Chapter 8). Focus is put on automatic and semi-automatic aspects

and contributions that contain several parts of a typical software testing process, i.e. not contributions that one day *might* contain more parts. By mapping these contributions against a software testing model, as presented in Chapter 8 and further explained in Subsection 9.2.1, a clearer view is given, thus making it significantly easier to compare different frameworks. Special care is taken to make sure that all aspects of a typical software testing process (Figure 8.2, page 121) are covered.

9.2.1 CLASSIFICATION AND TERMS

To be able to classify and clarify different frameworks, one first needs to establish a few ways to measure them. First of all, the definitions of automatic, manual and semi-automatic should be clear, since the classification in this chapter focuses very much on this aspect.

To begin with, we claim that, one *part* of a software has the responsibility to fulfill a major *task* and a software is defined by the integration of N parts. This leads to the following definitions (see Chapter 8 for a discussion regarding these definitions):

> **Automatic**—A software or a part of software is automatic if, and only if, the task executed by the software/part is performed autonomously with respect to human intervention.

> **Manual**—A software or a part of software is manual if, and only if, the task executed by that software/part is performed by a human which takes all decisions.

Thus, leading to semi-automatic being defined as:

> **Semi-Automatic**—A software or a part of software that is neither automatic nor manual.

Next, by using a software testing model (as presented in Chapter 8) and applying the aforementioned definitions on the model, a more distinct way of depicting a software testing framework, technique or environment is possible (Figure 8.2, page 121). The model, as shown in Figure 8.2 derives very much from a typical software testing process. The model was created by looking at the current state of practice with respect to software testing, and as such should be easy to apply on different tools, framework and environments (several tools, frameworks and environments are classified and covered in Chapters 7–8).

As an example, the core of the model in Figure 8.2 (page 121) covers the test case entity. A test case is divided into test data, test fixture and test evaluator (the rational

for this can be found in Chapter 8). Every category or sub-category can then be labeled as having different types of automation aspects and, furthermore, each sub-category affects the automation label that the parent (i.e. test case) ultimately is labeled as.

Additionally, two other aspects of the model might be appropriate to clarify. First of all, the concept of comparators and oracles. A comparator only compares the execution with a known 'correct' solution. An oracle, on the other hand, can compare results (as a comparator), but also give the 'correct' solution for each and every varying execution. Oracles are simply put more general in that they can do more, while comparators can be seen as simple assertions in the traditional sense; in the model both are part of the test evaluator entity. For more background information regarding the differences between these two concepts, and the different variation that they might show, we refer you to [27].

Second, the concept of sensitive and insensitive test quality analysis as covered in Chapter 7 and more thoroughly described in [314]. The question of whether the tests are good enough is hard to answer. Usually, today, this question can be answered indirectly by quantifiable data, e.g. the number of paths that are exercised in the software during the test or the total amount of test data that have been used as input, as is the case with e.g. random testing. These two examples can be seen as acting in a context, with respect to test quality analysis, that is sensitive or insensitive [314]. In the model both sensitive and insensitive test quality analysis is part of the test analyzer entity.

A more thorough description of the model, together with the rationale regarding choices made when developing the model, can be studied separately in Chapter 8.

9.2.2 RELATED WORK

The related work that is covered in this subsection was collected by taking four factors into consideration. Primarily we looked at related work that either:

1. Included, if possible, several steps in the software testing process as described in Figure 8.2 (page 121).
2. Had an interesting aspect not usually seen in software testing frameworks.
3. Covered both the commercial and non-commercial actors.
4. Was considered to be established.

Of course, there is always a risk associated with trying to select the related work most suitable for a specific purpose. But the listing above fulfilled our purposes, since all steps in the software testing process were covered. Worth mentioning in this context

Table 9.1: Collection of software testing frameworks (in alphabetical order) supporting several parts of the software testing model. (full circle = automatic; half full circle = semi-automatic; empty circle = manual). An empty space indicates not applicable or information not found regarding this particular feature.

Reference	**Test data**	**Test fixture**	**Test evaluator**
.TEST [70]	○	◑	◑
Baudry et al. [17]	◑	◑	
CASTING [4]	◑	◑	
Claessen et al. [53]	◑	○	◑
Daley et al. [58]	●	◑	◑
Davidsson et al. [60]			◑
Eclipse[1][74]	○	◑	○
Edwards [79]	◑	◑	
Feldt [90]	◑	◑	◑
Fenkam et al. [91]			◑
Tempto [258]	○	○	○
XUnit [305]	○	○	○

[1] Eclipse with GERT [60], djUnit [69], EclipsePro Test [83] and continuous testing [57] plugins included.

is that related work concerning *comparisons* of different tools, techniques, etc. is not included here since the only references found were by commercial interests and thus considered as being somewhat biased.

For an overview on automatic testing we recommend [15, 78, 206, 231]. In addition, some other publications are of interest as well, since they all cover certain aspects of software testing and software quality, e.g. [72, 100, 248, 265, 310].

Most of the related work, with respect to existing software testing frameworks, can be divided into a few areas according to our software testing model. Tables 9.1 (above) and 9.2 (page 140) provide an overview of relevant related work mapped against the software testing model. The first row in each of the two tables is an enumeration of the main categories as can be found in Figure 8.2 (page 121). The references in Tables 9.1 and 9.2 will be covered in this subsection and a short rational as to why each reference has been chosen will also be given, e.g. having a possible place in future software testing environments.

But, to begin with, it might be appropriate to give an example of how a tool is classified using this model. For this we choose XUnit [305], a unit testing framework

known by many developers and researchers.

By looking at any description of unit testing frameworks, one can see that test data generation and selection, in the case of unit testing frameworks, usually is performed manually. Test fixture creation can, under some circumstances be considered semi-automatic (e.g. the framework generates stubs which the developer needs to fill in) but is traditionally seen as a manual effort.

Comparison (test evaluation), in the case of unit testing frameworks, can be performed automatically as long as oracles or comparators are available. In the case of XUnit they are usually created manually, and are commonly (in the case of unit testing frameworks) seen as part of the test case creation step, i.e. the developer is the oracle by providing the correct answer.

The collection of test results is usually an automatic step in the sense that after a test engineer has started the execution of tests, XUnit collects the results during the execution. This is considered to be one of the strengths with unit testing frameworks, and as such indeed further refined by other researchers in [255, 256].

Finally, unit testing frameworks per see, do not traditionally include any test quality analysis, but nevertheless performs an 'analysis' by comparing the execution results with the expected (pre-defined) results.

The last row in Tables 9.1 and 9.2 provides a graphical notation of how XUnit is mapped against the software testing model used in this chapter. The rest of the related work will now be covered in the same manner.

Baudry et al. give an interesting insight on how to test .NET components with artificial intelligence [17]. One interesting thought is to take this further by making use of a back-end (i.e. a database) as a help to generate test cases. Hence, a future software testing framework might be able to 'learn' from previous experience.

van Aertryck, Benveniste and Le Métayer claim that their tool, CASTING [4], can generate tests in a formal (specification-based) and semi-automatic way. Generating tests from a formal language is probably something that future frameworks will need to support too, and since this is a good example on how one can automate this to a higher extent we include it in the population (Tables 9.1 and 9.2).

Daley et al. cover automatic testing of classes using the ROAST tool [58]. They use test case templates to define test cases which can then be used for generating boundary values. It is an interesting concept which we believe can be incorporated into other frameworks in the future as a complementary tool to reach higher effectiveness in test creation by adding manually written test templates. In addition, by using a storage some test templates could be generated a priori and provide a rudimentary foundation that a test engineer later could extend.

With a similar concept, but instead on the unit level, Claessen and Hughes give an introduction to the tool QuickCheck [53] which aids the developer in formulating and

Table 9.2: Collection of software testing frameworks (in alphabetical order) supporting several parts of the software testing model (full circle = automatic; half full circle = semi-automatic; empty circle = manual). An empty space indicates not applicable or information not found regarding this particular feature.

Reference	**TRC**[1]	**TA**[2]	**Supported language(s)**
.TEST [70]	◑	●	Misc. .NET languages
Baudry et al. [17]			C#
CASTING [4]			C
Claessen et al. [53]	●		Haskell[3]
Daley et al. [58]	○	●	Java
Davidsson et al. [60]	◑	●	Java
Eclipse[4][74]	◑	◑	Java
Edwards [79]	◑	◑	C++
Feldt [90]	●		Ruby[5]
Fenkam et al. [91]	●	○	Java
Tempto [258]	●		Java
XUnit [305]	●	◑	Misc. languages

[1] Test Result Collector.

[2] Test Analyzer.

[3] `http://www.haskell.org` (even though other languages have implemented QuickCheck too).

[4] Eclipse with GERT [60], djUnit [69], EclipsePro Test [83] and continuous testing [57] plugins included.

[5] `http://www.ruby-lang.org`

testing properties of programs. As a basis they use random testing for generating input, since they claim that research has shown that other techniques are at least not better in these particular circumstances. We believe that random testing is a good complement to other testing techniques no matter what type of software is tested and as such should always be considered as a complementary technique in future frameworks.

A similar framework, called Tempto [258], has been created with Java in mind. Tempto has one noticeable difference though; the whole test management infrastructure is based on XML declarations. Having all declarations and output written in XML could provide future frameworks some interoperability gains.

On a lower level, concerning testing techniques, we see Edwards' paper [79] on automating black box testing on component-based software with pre- and post-conditions as worthwhile pursuing. One interesting possibility is to instead use the Common Language Infrastructure (CLI) [141] as a testee, thus providing the possibility to be, as is the case with CORBA [91], more language independent.

Regarding frameworks for automated and/or semi-automatic testing some commercial tools are obviously also of interest. Especially .TEST from Parasoft [70] and IBM's Rational Robot [135] are worth mentioning here. Parasoft's framework (or maybe tool is a better word) can help the test engineer with automatic unit testing and automatic static and dynamic analysis on several .NET languages.

IBM's framework, on the other hand, helps the developer with functional and regression testing. However, in the case of these particular frameworks they usually create stubs for the developer to fill in. This is, in our opinion, something that should be automated to a higher extent in the future. In this chapter we will only cover .TEST and not Rational Robot since they are similar in all too many ways. Worth mentioning in these circumstances is the difficulty to obtain relevant technical information regarding commercial offerings such as [6, 70, 135]. This is of course something that needs to be taken into account when one compares different commercial and non-commercial frameworks.

In addition to the above commercial tools, we have GERT [60], a non-commercial tool for Eclipse [74]. GERT combines static source code metrics with dynamic test coverage and tries to predict the reliability of the software based on a linear combination of these values. In our opinion, this could be included in future frameworks as well and further refined using other reliability measurements (as discussed in Chapter 4), and as such is appropriate to include in the population. If Eclipse is combined with a few other tools (including GERT), such as djUnit [69], EclipsePro Test [83] and the continuous testing plugin [57], the making of a very competent software testing framework is starting to show. Unfortunately each plugin used by Eclipse in this case does not interact with other tools, hence missing an opportunity to reach interesting synergy effects. In addition, barring a few exceptions, the Eclipse project is focused

solemnly on the Java programming language.

Next, Feldt's contribution on the semi-automatic aspects of software testing [90] (i.e. the interaction between the test engineer and the software testing framework) gives an insight on what the future might look like, and as such is place in the tables as relevant related work.

Finally, we have two other contributions that are not included in the population, but are nevertheless interesting concepts regarding automatic software testing. Kung et al. present an object-oriented software testing environment in [164], and even though it is mainly for testing C++ programs, we believe that the fundamental principles of data flow testing and object state behavior can be incorporated into future frameworks focusing on testing software implemented using object-oriented technology. According to the authors the usefulness of the object-oriented test model is primarily to help:

> [...] testers and maintainer understand the structures of, and relations between the components of an OO program.

Collofello's et al. paper covering a testing methodology framework [56], which guides the developer in choosing the appropriate testing methodology is, even though a slight deviation from traditional software testing frameworks, still of interest since test technique selection etc. could be performed in a similar manner. Since this contribution does not focus on the typical software testing process, but instead on some 'meta-aspect' thereof, we will not include it in the population.

As mentioned previously there exist several examples of frameworks with different vertical approaches [134, 270, 285]. But as can be seen from the collection of links in [276], there is a lack of tools that combine different horizontal techniques. Hence, this chapter's focus on frameworks that fulfill certain aspects of a typical software testing process.

As an example, Figure 9.2 (next page) provides a graphical view of how the first reference in Tables 9.1 and 9.2, i.e. .TEST [70], map against our model; in addition one can see how the automation labels on each sub-category affects the parent, i.e. test case.

9.3 FRAMEWORK DESIDERATA

What, by looking at the references just covered, does the future hold then? For one, we can see a shift towards frameworks that try to combine several different techniques and tools (using a vertical approach). This is also indicated from the collection of references that can be found in Chapter 7 and [276].

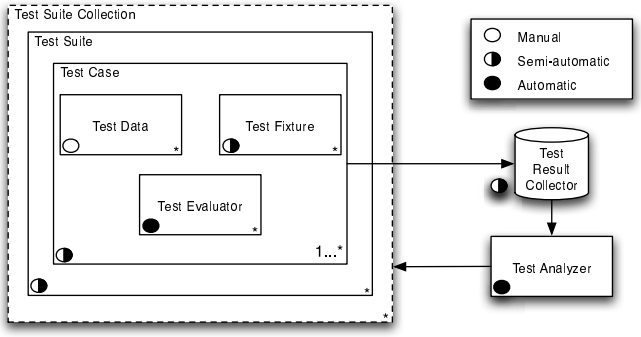

Figure 9.2: A complete graphical view of how .TEST [70] is classified using the software testing model.

Secondly, by examining each column in the tables (Tables 9.1 and 9.2), one can see that the only categories where there is a lack of 'pure' automation is (see Table 9.1) in the test fixture and test evaluator entities.

The reasons for this is simple. Regarding the test evaluator entity the problem is closely connected to the oracle problem. The oracle problem [293], is according to Chen et al. [47] one of two fundamental limitations in software testing. With respect to test quality analysis performed in an insensitive context we believe this is due to the immaturity of software testing techniques using mutation analysis and genetic algorithms (presently no commercial frameworks exist as far as we know). But to have a truly context insensitive approach in test quality analysis, with full automation, those two techniques are most likely to succeed in our opinion.

With respect to automatic test fixture it is not clear to us why there is no (as far as we know) framework supporting this concept. The technical problems should not be too difficult to solve (and indeed is partly solved in Chapter 10).

Other than that, the tables illustrate that all other parts can be fully automated, even though each contribution uses different techniques to reach that goal. Nevertheless, no tool, framework or environment covers all the steps in a typical software testing process.

As can be seen, partly from the collection of references (Tables 9.1 and 9.2 in addition to Chapter 7), software testing frameworks try to support more and more testing activities as seen in a typical software testing process, e.g. test creation, results collection and test analysis. In addition, a move towards combining different vertical test-

ing techniques can be visible (see Chapter 7). Of course, the different combinations, whether dealing with techniques or activities, should lead to more faults being found in software. For example, we have previously shown in Chapters 5 and 6 that combining different testing techniques within the same approach (e.g. combining partition, random and boundary testing) gives a higher effectiveness.

Thus, a framework where a test engineer might combine several unit testing techniques, e.g. partition testing [296], boundary value analysis [204] and random [122], as well as anti-random [306], testing, in addition to one or more high level testing approaches should, theoretically, lead to higher effectiveness in finding faults. What one test technique misses—another might catch.

By combining software testing approaches a higher effectiveness can be reached with respect to finding faults. Depending on whether these approaches are carried out in parallel to each other or not and the exact degree of automation of each technique, any combination also holds the potential of increased efficiency due to more found faults within a constant time frame. Saff and Ernst have in addition shown that higher efficiency can be reached by constant test execution [256].

Hence the end goal is significant since having one framework, which can combine both vertical and horizontal approaches in addition to having the continuous testing approach and the self-learning environment integrated, should lead to faults being found during less time. Simply put, even if we introduce a multiple of techniques that might take more time to execute in our framework, the total cost savings, i.e. since more faults will be found earlier in the development process, will make up for the effort (see Boehm's conclusions [30] and e.g. Damm's contribution on trying to solve the matter of fault-slip-through in software engineering processes [59]).

In addition, as we mentioned briefly, future frameworks could be used as a test bed for testing different test techniques' ability to find faults. As far as we know, there exist no such framework today.

With respect to our definitions of automatic, semi-automatic and manual it seems clear what the future might look like. First of all, allowing a test engineer to manually work on a framework's different parts should always be allowed since it is impossible to account for all different use cases. Thus, the guiding principle future software testing frameworks should adhere to is:

> Automatic where possible, semi-automatic otherwise

By having for example automatic, or when this is not possible, semi-automatic test case generation a test engineer will be able to save time. In addition it will require less in terms of knowledge and education. From the researcher's perspective, automatic creation of e.g. test data and test fixtures will be much more reliable compared

Table 9.3: Additional requirements not part of the software testing model.

Language agnostic
Combinations
Integration and extension
Continuous parallel execution
Human readable output
Back-end

to manual creation (with respect to repeatability). Thus, it will allow for a much easier comparison of different test case generation techniques in the future. Obviously, many testing techniques that are being used today are more or less inappropriate to automate to a high(er) extent.

To put it simple, look at Figure 8.2 (page 121) and imagine that each circle can be empty, partly filled and filled, at the same time. That kind of flexibility is something test engineers will probably look for in future frameworks, thus the baseline requirements future frameworks should strive towards is Figure 8.2 with varying levels of automation.

Future frameworks will most likely always provide the possibility to, first of all, continuously test in the background while the software engineer is developing the software and, secondly, perform parallel test execution. Parallel execution of test cases has the possibility to lead to higher efficiency. There are problems that need to be solved in this particular area, especially considering dependencies in test case hierarchies and prioritization of test case execution. But if these problems can be solved for the majority of test suites, it could lead to the breadth of testing techniques being used to be widened, i.e. techniques that generate massive amounts of tests.

But, as we already can discern, several other requirements need to be fulfilled. By itemizing these requirements into different properties we obtain Table 9.3. As can be seen from this figure there exists several requirements that, outside the software testing model being used in this chapter, accounts for a few features that future frameworks will probably contain.

Finally, a few more clarifications need to be made. Future frameworks should be extendable thus providing a test engineer the possibility to add functionality in the future. In addition, several 'common features' should always be provided in the future, as they are today. For example, features such as static source code analysis and dynamic execution tracing are used by many software testing techniques.

A quick look at Figure 8.2 and Table 9.3, which could be said to represent a combined view on the requirements, shows that the goal for future frameworks is ambitious.

The aim is to have all top-level definitions, i.e. test creation, test execution, result collection, result evaluation and test quality analysis to be automated, according to our definitions that were covered in Subsection 9.2.1.

In the following subsections several proposals will be made covering likely paths to take in the future with respect to software testing frameworks. Each subsection covers one topic that is of, in our opinion, particular interest (Table 9.3). Each proposal focuses, where appropriate, on solutions for the Common Intermediate Language [141]. Nevertheless, the discussions should also be applicable to other intermediate representations of code.

9.3.1 LANGUAGE AGNOSTIC

One initial focus that future frameworks could have is generating and executing test cases on Common Intermediate Language [141] code. First of all, since CIL can be extracted from any executable, that adheres to the standard [141], both white- and black box approaches can be used.

Secondly, CIL is suitable for low level testing techniques such as boundary value analysis, partition, random and anti-random testing, more or less in the same sense as source code. Obviously, on the other hand, other vertical testing approaches, such as acceptance testing or system testing, must be implemented on a more abstract and higher level.

Finally, the concept of an intermediate language (such as CIL) gives the software tester the possibility to test software written in different programming languages using the same tool, techniques, concepts and methods.

9.3.2 COMBINATIONS

The software testing model covers test generation but, as mentioned previously, combinations of testing techniques (vertical and horizontal) would be highly interesting to have in future frameworks. We believe that a test engineer will gain in ease of learning, ease of use and maintainability by having one framework. In addition to this (s)he will be able to reach a higher effectiveness by combining different (sometimes overlapping) techniques. A researcher, on the other hand, will be able to compare and evaluate different techniques more easily.

Random testing is a technique that is of interest to implement in a framework, when combining several different testing techniques, because of its rather straightforward manner. Random testing has been covered to a high extent the last decades especially by Hamlet (for a nice introduction please read [122]). Lately, progress have been made considering the semi-automatic creation of comparators using AI-related

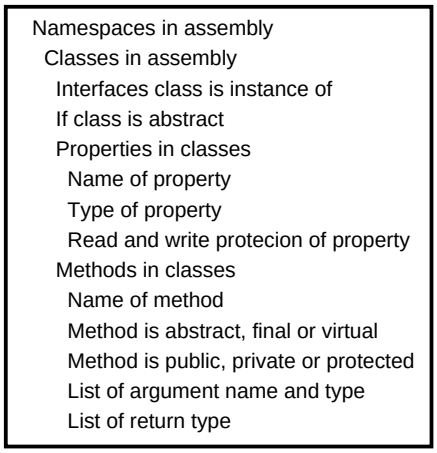

Figure 9.3: Example of metadata that can be collected using the Reflection namespace.

algorithms [90] or by using a test data generation language [53] in addition to the quality estimations considering random testing as a whole [187] (in addition please see Chapter 4).

One way to implement random testing in a framework would be to extensively make use of the System.Reflection namespace [242]. The Reflection namespace within the .NET Framework (as well as in Java) gives a developer access to metadata from assemblies (see Figure 9.3). In this way a semi black box approach can be used.

Another way to get this type of information from an assembly would be to directly analyze the CIL code. That way a test engineer, or a framework, will be able to get more information regarding the innards of the assembly e.g. the algorithms being used. Even though it would mean paying a much higher price in terms of complexity, than simply using the Reflection namespace, it would also make traditional white box approaches possible.

In addition to this, runtime instrumentation of assemblies [239] is also a possibility that might be worth pursuing in the future. This is however, as the name tells us, a pure runtime library and not for static testing.

Finally, reverse engineering assemblies using e.g. Reflector for .NET [243] or other tools for Java et al., would make it possible to use a wide range of white box approaches more easily, but on the other hand it could introduce some legal aspects with respect to intellectual property rights.

Implementing random testing in a framework is fairly straightforward and has been

done on several occasions. For example, using the technologies just described, a framework could get the signature of the method that will be tested (Listing 9.1).

Listing 9.1: Example of a signature obtained from e.g. an intermediate representation.

```
1 int SquareRoot (int number)
```

The framework could then generate an array of random integer values (e.g. $\{9,50\}$) which could be used when calling the method (as depicted in Listing 9.2). Worth mentioning here is that in this example scalar values are used but for a framework to be truly useful complex values need to be handled.

Listing 9.2: Example showing random testing of a unit.

```
1 //Generate new object o
2 o.SquareRoot(9);
3 //Returns 3
4
5 //Generate new object o
6 o.SquareRoot(50);
7 //Exception is cast
```

A simple analysis would then make the test engineer aware of the fact that an exception was thrown. In this case, the reason might be that the method is not able to find the square root for a particular integer.

Rudimentary support for generating stubs for unit test cases, by analyzing the CIL code extracted from an assembly, is already possible today. A similar approach, but where the actual source code is analyzed, has been taken before as far as we understand [70, 135]. In the future, combining stub generation with random testing in addition to boundary value analysis, is a likely path to gain higher effectiveness.

9.3.3 INTEGRATION AND EXTENSION

Since Integrated Development Environments (IDEs) are common today, we believe that there will be a shift towards integrating more and more features into established IDEs [74, 289], instead of creating 'traditional' test environments [70, 135, 6]. Obviously, why should a test engineer need to learn yet another testing environment when (s)he already knows an IDE?

Unfortunately, after examining the plugin interfaces for various IDEs, one quickly comes to the conclusion that each and every IDE is unique when it comes to adding functionality to it. As such, a future framework needs to be adapted for every IDE that will use the functionality provided by the framework. Alas there is no way today,

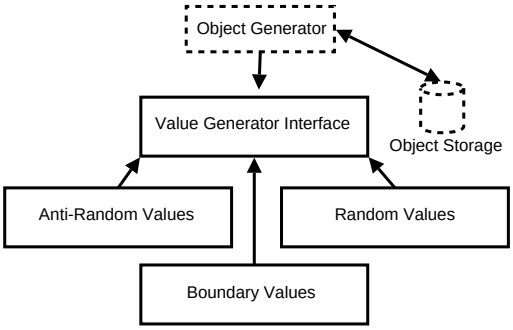

Figure 9.4: Schematic overview of a value generator.

standard or de facto standard, to extend different IDEs through one common plugin interface.

On the other hand, one way to extend a framework, would be by using the concept of, again, plugins. For example, one thing that could be used and reused many times over in a framework are values. A value generator could be extended, thus different values could be generated and tailored for a particular test. Primitive values, e.g. integers, floats and characters, are always of interest. In addition, by extending a value generator and having different types of values stored in the database, for generating anti-random [306] series of numbers or different types of mock-objects [197], might be of interest (Figure 9.4).

9.3.4 CONTINUOUS PARALLEL EXECUTION

A back-end for constant execution of test cases, as is described in [255] and [256], is naturally of interest for several reasons. First of all, having test suites executed constantly will let test engineers be notified immediately if any regression is found. Secondly, combining test execution and regression checking using clustered workstations during night time [31] would make the test engineers able to scale up the number of test cases to a very high degree. Although this technique may not increase efficiency per time unit (of execution) as such, it still increase efficiency in any software development project as a whole. One might add that this approach will put an additional emphasize on the statistical tools being used as well as the visualization and data mining parts of a framework. This approach is basically the opposite to [254] where Rothermel et al. are trying to keep the number of test cases to a minimum.

Of course, using clusters for software testing is not simple since the issue of test

case dependency creates a 'lose-lose' situation for the test engineers in that software slicing need to be performed to the extent that the tests in the end are testing more or less nothing. Random testing of software units (units in the sense of unit testing) might on the other hand benefit from this approach since dependency issues can be kept at a minimum.

9.3.5 HUMAN READABLE OUTPUT

Schimkat et al. [258] has showed us how to use XML to configure and execute tests in an object-oriented test framework. This is probably something that will be more common in the future.

If one in addition stores all output in XML then platform independence with respect to data storage will be readily available. Which operating system or test framework that is used will be more or less irrelevant as long as data is stored in a uniform human readable manner. The usage of XML schemes in the future would enforce a standardized look, but to create templates for something as complex and varying as software testing might prove to be all to difficult.

9.3.6 BACK-END

By using a database back-end that records test statistics and draw conclusion of said statistics, a higher knowledge can be gained with respect to effectiveness and efficiency of different testing techniques' ability to discover faults under different scenarios. In other words, some testing techniques might be better under certain circumstances e.g. type of software tested, programming languages or type of architectures being used. Having data gathering and analysis performed within one framework, while using different testing techniques and approaches, should ultimately give a higher quality with respect to data analysis. A self-learning environment, which analyzes old test results and draws conclusions, could naturally evolve in the end [17].

A future framework, would probably consist of a database used for collecting data of quantitative nature, i.e. test results stored for later evaluation, collection of statistics and regression monitoring. Information that might be appropriate to store in this case would be:

- Number of test cases.
- Type of test case (even storing the actual test case might be appropriate).
- Type of test technique being used.

- Found faults per test case.
- Test suite execution time.
- The number of lines of code (when applicable).
- Type of software, e.g. programming language, server or desktop application.

In addition to the above, being able to send the above information to either: a) a test manager at a company or, b) a researcher for further analysis, is obviously something of interest. Having quantitative data, regarding software testing (especially when combining data from several projects) readily available to a higher extent will increase the knowledge concerning various part of the software testing process, whether that may be test case creation or test quality analysis.

9.3.7 AUTOMATED TEST FIXTURE AND TEST EVALUATOR CREATION

Finally, as was concluded already in the beginning of this section automated creation of test fixtures and test evaluators is something not easily found in test frameworks today. In some cases, semi-automated approaches exist (see Tables 9.1 and 9.2), but usually they, in the case of test fixtures, allows for some stubs to be created (while the test engineer needs to fill out the rest), or in the case of test evaluators, are based on generating them via specifications.

In a future framework the above issues could be solved by retrieving method signatures etc. dynamically during run-time and thus, in addition, provide the test engineer with common input values and the results from executing said values. That way, it would at least provide the test engineer with an automated framework for regression testing, which later could be extended with other automated approaches.

9.4 CONCLUSION

This chapter consisted mainly of three parts. First of all, a software testing model was applied on a collection of frameworks for easier comparison. Next, an elaboration on the requirements that future frameworks probably need to adhere to was performed. Finally, a number of proposals regarding how the requirements might be implemented in future software testing frameworks were given.

In our opinion, we find it reasonable that frameworks should, in one form or another:

- Automate more parts in a typical software process, while still allowing test engineers to extend the functionality in different ways (e.g. automated creation of test fixture, test data and test evaluator by means of run-time extraction).
- Collect quantitative data that is stored for later analysis.
- Combine more testing techniques (vertical and horizontal).
- Apply techniques on intermediate code representations.
- Provide support for selecting the most appropriate technique when testing a particular software.
- Save results in a standardized format, e.g. XML.
- Integrate test activities in IDEs currently used in industry (as opposed to offer a completely new testing environment explicitly tailored for software testing).

If the above desiderata is to be implemented in one framework then another step, towards automated software testing, would be taken.

Chapter 10

Deducing Generally Applicable Patterns from Object-Oriented Software

Submitted to the Automated Software Engineering Journal

R. Torkar

10.1 INTRODUCTION

In 2002 the National Institute of Standards and Technology released a report [205] which revealed that the total cost of inadequate software testing infrastructures, in the USA alone, was estimated to be as high as $59.5 billions per year (80 percent of software development dollars were spent on correcting defects). The total market for automated software quality products (products that to some extent automatically or semi-automatically supports software testing) is today estimated to be as high as $1 billion with an annual growth of 9 to 10 percent [130]. These facts and predictions, together with the observation that application testing takes between 30 and 90 percent of total product labor costs [22], leads to an obvious conclusion—automated software

quality (ASQ) products will likely be an important part of the software engineering landscape for the foreseeable future and the focus on automation will increase so as to reduce expensive manual labor.

Several ASQ products are today automated to a higher or lesser extent, but few are completely automatic (see Chapters 7–9 for examples). The reasons for not reaching a level of full automation are several (one reason is the complexity of software), but nevertheless it is not an understatement to claim that much progress have been made the last decades with respect to automation in software testing. To be able to automate software quality assurance to a high(er) extent, one needs to look at other techniques as being more suitable for this purpose than others. In this respect random testing [122] is a likely candidate for automation due to its nature where a minimum of human intervention is needed. A future framework (as presented in Chapter 9) could encapsulate random testing as a viable option for testing object-oriented software.

Unfortunately, random testing of object-oriented software has not been researched widely (for an overview please see [27]). One of the reasons for this is surely the fact that random testing traditionally compares well to other techniques when it comes to dealing with scalar (pre-defined or primitive) types, while usually is seen to have weaknesses dealing with compound (programmer-defined or higher-level) types (semantic rules for these types can be found in e.g. [11]). Another reason might be that random testing, again traditionally, has been tested on small and delimited examples. However, these reasons are not as valid today, see [53, 219] and [268] respectively, as they were a decade or so ago. Since some progress has been made one can draw the conclusion that researchers need to perform more experiments and case studies using random testing on complex object-oriented software (using compound and programmer-defined types extensively as in this chapter).

In order to apply random testing on software, one can use dynamic (runtime) analysis to be able to get a deeper understanding regarding the software item under test (testee). Extracting patterns from a testee, by injecting code (aspects) which records the usage (dynamically) and then later apply these patterns on the testee itself, is one possible approach for random testing object-oriented software (this way one would random test the software using a likely use case profile). In addition, it is not unlikely that these patterns could be generally applicable on other software once extracted using e.g. statistical pattern recognition (due to the share volume of data needed to be processed). The word pattern, in this context, constitutes several entities such as: execution path(s) (sequences of method calls), method input type(s) and value(s), and method return type(s) and value(s), with which the testee is executed.

As an example, regarding the general usage of patterns in object-oriented software, one can see a scenario where one examines the methods that have been executed, how many times they have been executed, and the specific order they are executed in.

As an example, assume that `ObjectOfClassA` is always executed in the following order: `.ctor()`, `Foo()` and `Bar()` or that `ObjectOfClassA`'s method `Foo()` is consistently called from `ObjectOfClassB` or `ObjectOfClassC`, which in the end could imply that all objects of type `ClassA` in any software item should at least be tested using these orders of execution.

The research contribution of this chapter is to, a) show how object message pattern analysis (from automatically instrumented applications) can be used for automatically creating test cases. Test cases in this context equals test data, evaluators and fixtures (for a discussion regarding the different entities in software testing please see Chapter 8), b) point to how random testing can be performed on these test cases and, finally, c) examine the possible existence of object message patterns in object-oriented software in addition to the question of general applicability of said patterns. In our opinion, these questions need to be examined before another step can be taken towards a framework for automated random testing of object-oriented software as previously partly described in Chapter 9.

Throughout this chapter an empirically descriptive model is used, i.e. explaining a phenomenon sufficiently on 'real life' software.

Next, related work is presented. Following that, the setup of the experiment is presented (Section 10.2) while the results are covered in Section 10.3. Finally, this chapter ends with a discussion and a conclusion (Sections 10.4–10.5).

10.1.1 RELATED WORK

Related work for this chapter can be divided mainly into four categories: random testing, type invariants, dynamic analysis and patterns in object-oriented software.

Random testing. Random testing [122, 124], which acts a basis for our approach, is a fairly old research area which has seen several new contributions lately which directly or indirectly affect this chapter. One point to make in this context is that random testing has advantages which other testing techniques lack. To begin with, due to the nature of random testing, automatic aspects are fairly straightforward to implement. Furthermore, statistical test methods, such as random testing, are unique in that they do provide a type of answers regarding possible remaining faults ([122] and Chapter 4), the downside with this information is its probabilistic nature and interchangeability between failure probability and test reliability. Finally, even though much debated, random testing compared with e.g. partition testing, has in some circumstances been shown to have very small differences in effectiveness [48, 71, 210].

Looking at Claessen and Hughes' work on a lightweight property-based tool called QuickCheck [53] one can see a novel technique in using random testing. In this example they use random testing together with property-based oracles, thus introducing new

ways of automating several parts which earlier used to be extensively manual (Chapter 8). Hence, further proving the point that random testing is suitable for automation, albeit only on Haskell[1] programs in this particular case (other languages have implemented QuickCheck too lately).

But is random testing the answer to all our problems? Certainly not, random testing has at least one substantial disadvantage, i.e. low code coverage (see [215] for an excellent overview). However, by using the approach in which this experiment is conducted, low code coverage is mostly inconsequential since directed coverage is applied (in most cases), i.e. the most likely paths are always executed since they are exercised by the use case when applied on a testee, which in the end means that the test strategy used in this experiment focuses on exercising values within these paths. We do not claim, in any way, that this framework reaches complete coverage—it is better to focus on what random testing does best.

Likely invariants. Likely invariants has gained considerable attention lately following the work of Ernst et al. [84, 219, 230] and Lam et al. [126]. The concept of likely invariants is built upon the assumption that one can find *different* input values, for testing a software item, by looking at the *actual* values the software uses during a test run. In our opinion, using likely invariants is an appropriate way of gaining semi-randomness in software testing when a traditional random testing approach is not feasible (likely invariants in these contributions are based on likely deviations from used, or recorded, values). By having likely invariants for the input domain, i.e. method input values, the test engineer will always have the possibility to avoid testing an extremely large input volume when using random testing, which might be the fact in many circumstances due to the combinatorial nature of software.

Our approach does not focus on likely invariants per see but rather class interactions which in its turn provides the order of the methods being executed as well as method signatures and actual input and return values and types. While likely invariants are mostly focused on unit testing, this chapter, on the other hand, focuses on object interactions and sequences of method calls. The contributions regarding likely invariants can in the future be added to our framework if needed, but will ultimately lead to a deviation from the path of true random testing, although provide an excellent way towards automated software testing (please see [126] for a discussion on different usage models for likely invariants).

Dynamic analysis. The concept of dynamic analysis, i.e. analysis of the software based on its execution, can in our approach best be described as a see-all-hear-all strategy (for more recent contributions on how to use dynamic analysis for program comprehension please see [125, 309]). This way it resembles the omniscient debug-

[1] `http://www.haskell.org`

ger [174], which takes snapshots of every change in the state of the running application thus allowing the developer to move backwards and forwards when debugging the application. The difference in our approach is that every important piece of information during an execution, is stored for later off-line analysis (while in [174] it is used during run-time). Thus, in our case, we are avoiding costly run-time computations while focusing on data-mining concepts [169] and, in the long run, statistical pattern recognition [142] as a way to extract valuable information from the rich information source.

Object-oriented patterns. An overview of object-oriented software testing, and especially patterns in this area, can be found in the following contributions [27, 62, 97, 167, 227]. Important to note, in this circumstance, is that the word pattern (as used by almost all of the references) has in many ways a different meaning compared to how it is used in this chapter. In [27] and [167] the word pattern is used for the purpose of testing systems (not adhering to the detailed lower level as proposed by us), while in [62] the word pattern is used to describe *how* system testing should be performed from an organizational perspective. In [97] the definition of pattern follows the predominant definition of patterns as introduced by Gamma et al. in [105]. Finally, in [227] the authors examine the subject of patterns in object-oriented software (further researched in [228]), but on the other hand they do not to make a connection with regards to testing object-oriented software but rather focus on analysis in a more broad sense. Nevertheless, the abstraction mechanism as applied in this chapter is very much the same as in [227] albeit the purpose is completely different.

In able to clarify the concept of patterns in this chapter, we use the name *Object Message Patterns* for our purposes. *Object* stands for the fact that the focus is set on object-oriented software. *Message* is short for the message-driven perspective as employed by object-oriented software and finally, *patterns* stands for the execution traces as found when analyzing software.

As far as we know, the only contributions that bear resemblance to this chapter (looking at it from a test framework perspective) are Godefroid's et al. paper on DART [111] and Lei and Andrews' contribution on minimization of randomized unit test cases [172].

DART, or Dynamic Automated Random Testing, implements an automatic extraction of interfaces using static source code parsing (whereas we focus on intermediate representations) which then is used to create a test fixture. The fixture is in its turn used for random testing the method. Unfortunately, DART, as most other contributions in the area of random testing, focuses on scalar types (no evidence is found stating otherwise in the reference) and is not automated to the extent one might like, e.g. a test engineer needs to define the self-contained unit to test.

Lei and Andrews' contribution on the other hand has, as far as we understand, a slightly different focus in that they concentrate solely on minimizing the length of

the sequences of method calls using Zeller and Hildebrandt's test case minimization algorithm [311], which significantly reduces the length of sequences. In addition, to our knowledge, existing (manually created) oracles are a prerequisite to their approach (further clarified in [10]), and the case studies performed are applied on small units—compared to our larger full-scale applications as introduced in the next section.

10.2 EXPERIMENTAL SETUP

In this experiment the focus is set around testing intermediate representations of source code. Today, in industry, many would say that the center of attention is mostly around the Java Virtual Machine [177] and the Common Language Infrastructure (CLI) [77] with its Common Language Runtime (both inheriting from the original ideas brought forward by the UCSD P-System's developers in the late 70's [137]).

To dwell on the pros and cons of intermediate representations of source code is beyond the scope of this chapter but, suffice to say, having an intermediate representation allows the developer to perform dynamic manipulations easily (using e.g. reflection [63, 274] which can be found on the Java 2 Platform [42] and the .NET framework [131]). When reflection is not enough to solve the problem, one can easily instrument the intermediate representation either directly [85, 170] or indirectly [106, 115]. In this experiment we make use of the Common Language Runtime and instrument the intermediate representation, but nevertheless, the concept proposed in this chapter should be applicable to most if not all types of intermediate representations.

The experiment conducted in this chapter was performed on three different software items: Banshee [28], Beagle [104] and the Mono C# compiler [208] (Mcs). The selection of the software items was performed with the following in mind:

- The application should be written in a language which can be compiled to an intermediate representation (in this case the Common Intermediate Language, CIL [77]).
- The application should be sufficiently large and thus provide large amount of data for analysis.
- The applications should be of GUI or console type and targeted towards end-users or developers.
- The applications should be developed by separate development teams.

In the end, Banshee (media player), Beagle (search tool) and Mcs (C# compiler), were considered to fit the profile for the case study. For each application one common use case was selected to be executed after the application was instrumented:

Table 10.1: An overview of the experiment showing the number of LOC (lines including comments, white spaces, declarations and macro calls), the size of the assemblies (in KB), the number of classes that were instrumented and the number of instrumentation points, for each application.

App.	LOC	IL (KB)	# Classes	# Instrumentation Points
Banshee	53,038	609	414	2,770
Beagle	146,021	1,268	1,045	5,084
Mcs	56,196	816	585	2,640

Table 10.2: The time to instrument (TTI) and execute (TTE) without and with instrumentation for each application. Time command execution according to IEEE Standard 1003.2-1992 (POSIX).

App.	TTI	TTE w/o Instr.	TTE w Instr.
Banshee	28s	1m 3s	7m 4s
Beagle	1m 11s	6s	1m 14s
Mcs	2m 46s	0.8s	34s

- **Banshee**—Start application, select media file, play media file, stop playback, shut down application.
- **Beagle**—Start search daemon in console (background process), perform query in console, close the GUI which presents the search results, stop search daemon.
- **Mcs**—Compile a traditional 'Hello World' application.

Tables 10.1 and 10.2 provides an overview of the different software items that were tested, as well as the number of classes, methods and instrumentation points. Correspondingly, for reference, the time to instrument the assembly and execute the test case with and without instrumentation is presented (the last numbers should be considered approximate since some of the use cases did require manual intervention by the software tester).

After the selection of the candidate applications was accomplished the actual instrumentation took place. To begin with, each valid class in the testee (disregarding abstract, extern and interface annotated signatures) in every assembly (exe and dlls),

had instructions inserted in each method which would collect runtime input and return value(s) and type(s), as well as the caller (i.e. what invoked the method). All this data, together with a time stamp, was then stored during runtime in an object database while the testee was executed following a valid use case. That is to say, each time a method was executed, during the execution of a use case, an object containing all the values necessary to recreate that state, was stored in the object database. Having to serialize the data beforehand would be too resource intensive for obvious reasons not to mention posing some difficulties from a technical perspective, as is discussed in Section 10.4.

Next, the object database was used for an analysis of data looking for patterns and discovering likely critical regions. The selected paths could then be used for creating a test case (using the actual runtime values as test data and test evaluators). The execution of the use cases, as well as the instrumentation of the assemblies, was performed under Linux 2.6.15, Mac OS X 10.4.4 and Windows XP SP2 using Cecil 0.3 [85], AspectDNG 0.47 [106] and the open source (in-memory) object database engine db4o 5.2 [61].

To be able to better explain our approach (which can be applied directly on data from the experiment), an example is introduced next. The example is very simple in its nature (two classes: `Foo` and `Bar`), and as such also provides a good overview regarding the type of data that can be extracted from the experiment (Listing 10.1).

Listing 10.1: Example which will be used in this chapter.

```
1  using System;
2
3  public class Bar {
4          public static void Main(string[] args) {
5                  Foo myFoo = new Foo();
6                  string exec = myFoo.Run("Our␣string");
7          }
8  }
9
10 public class Foo {
11         public string Run (string myString) {
12                 return myString;
13         }
14 }
```

The disassembled intermediate representation [77]) of the method `Run()` (when compiled with the Microsoft C# compiler) can be seen in Listing 10.2.

Listing 10.2: The disassembled method `Run`.

```
1  .method public hidebysig instance string
2                  Run(string myString) cil managed
3  {
```

```
4           // Code size 6 (0x6)
5           .maxstack 1
6           .locals init (string V_0)
7           IL_0000: ldarg.1
8           IL_0001: stloc.0
9           IL_0002: br.s IL_0004
10          IL_0004: ldloc.0
11          IL_0005: ret
12  } // end of method Foo::Run
```

After performing the automatic process of instrumenting the assembly, the method `Run()`'s name is thus changed to `Run__Wrapped__0()`, i.e. everything in Listing 10.2 stays the same except for the string '`Run`' being exchanged to '`Run__Wrapped__0`'.

`Run__Wrapped__0()` is then referenced from the new 'implementation' of method `Run()` which now looks completely different (Listing 10.3, please note that some CIL instructions have been removed to decrease the number of lines and that a pseudo code example is presented in Listing 10.4). The instrumentation is performed in the same manner no matter what type of value (compound or scalar) the method takes as input or returns.

Listing 10.3: The new implementation of method `Run()`. On line 19 and 20 one can see the original implementation of `Run()` being referenced by its new name, and on lines 34–36 the original implementation's execution continues. As a side note, the implementation of `MethodJoinPoint` (referenced on lines 21–26), stores the data in the object database.

```
1   .method public hidebysig instance string
2                   Run(string myString) cil managed
3   {
4           // Code size 68 (0x44)
5           .maxstack 8
6           .locals init (object V_0,
7                         object[] V_1)
8           IL_0000: ldc.i4 0x1
9           IL_0005: newarr [mscorlib]System.Object
10          IL_000a: stloc V_1
11          IL_000e: ldarg myString
12          IL_0012: stloc V_0
13          IL_0016: ldloc V_1
14          IL_001a: ldc.i4 0x0
15          IL_001f: ldloc V_0
16          IL_0023: stelem.ref
17          IL_0024: ldarg.0
18          IL_0025: ldloc V_1
19          IL_0029: ldtoken method instance string Foo::
20                                          Run__Wrapped__0(string)
```

```
21          IL_002e: newobj instance void
22                   [aspectlib]MethodJoinPoint::
23                   .ctor(
24                   object,
25                   object[],
26                   valuetype [mscorlib]System.RuntimeMethodHandle)
27          IL_0033: dup
28          IL_0034: ldtoken method object Torkar.AspectsSample::
29                   BodyInterceptor(class [aspectdng]MethodJoinPoint)
30          IL_0039: call instance void
31                   [aspectlib]Joinpoints.JoinPoint::
32                   AddInterceptor(
33                   valuetype [mscorlib]System.RuntimeMethodHandle)
34          IL_003e: call instance object
35                   [aspectlib]Joinpoints.JoinPoint::
36                   Proceed()
37          IL_0043: ret
38  } // end of method Foo::Run
```

Listing 10.4: A pseudo code example displaying what a method looks like when instrumented.

```
1  Foo(input data and type)
2          execute Foo_Wrapped
3          store input data and type(s)
4          store return data and type(s)
5          proceed
```

Figure 10.1 (next page) depicts an excerpt from the object database regarding the execution of class `Foo`. The result of the query shows that everything is stored in proxy objects (first line). In addition one can see that the time stamps (lines 3 and 13) and the result from executing the method `Run()` are present (line 11). This together with the type of the input value on line 21 provides us with much information regarding the runtime behavior of the application as will be covered next in Section 10.3.

Finally, Figure 10.2 (c.f. 164) provides an overview of how the complete process was performed (from application selection to object message pattern analysis).

10.3 RESULTS

Next, follows several subsections whereas the statements, posed at the end of Section 10.1, are covered with respect to the results. First, the issue of generality regarding the discovered paths is covered, e.g. if a pattern in one software can be found in other software. Here we reference directly the object message pattern analysis (OMPA) per-

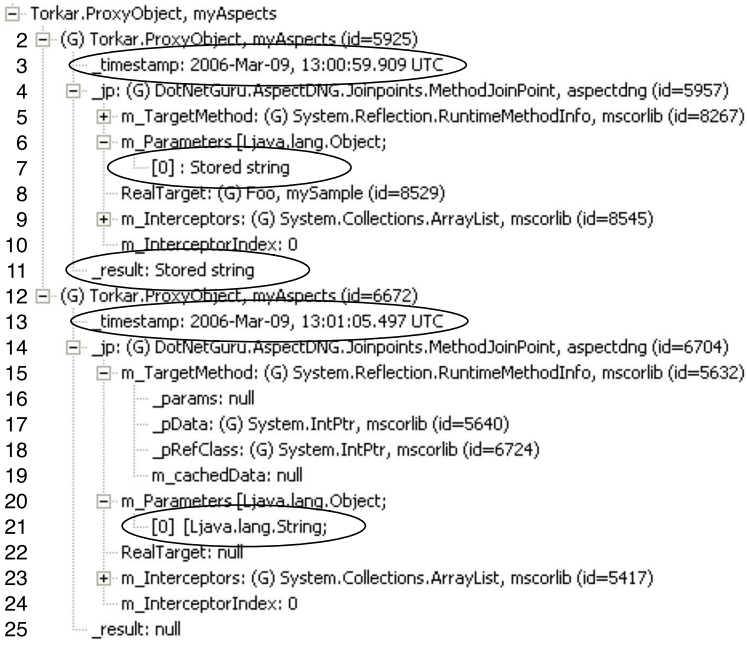

Figure 10.1: An excerpt from the object database showing relevant information regarding the class `Foo`.

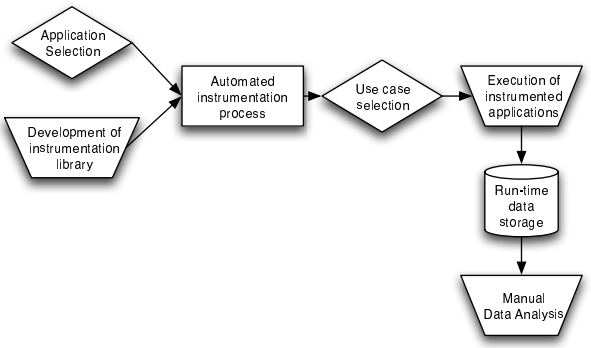

Figure 10.2: An overview of how the experiment was performed.

formed in the experiment, and distance us from the previously introduced example for obvious reasons, i.e. a comparison can not be done using an example only.

Secondly, we cover the issue of using OMPA for the creation of test cases (by showing an example from the experiment). We then point to how random testing and likely invariants can be applied to this solution. These steps are performed on a small and delimited example, from the experiment, to thus be able to better present and explain the approach.

In Section 10.4 the results in these categories are discussed and elaborated on.

10.3.1 OBJECT MESSAGE PATTERNS

The intention of performing object message pattern analysis (OMPA) on data in this experiment is to find and generalize patterns which then can be used when testing the software item. In addition to this the hypothesis is that patterns, if found, could be generally applicable to most, if not all, object-oriented software. Since the analysis is currently performed manually a limitation on the number of analyzed objects was needed. Thus, 300 objects (in a call sequence) from each application were analyzed from an arbitrary point in the object database (selected by a pseudo-random generator as found on pp. 283–284 in [236]).

Eight object message patterns were found during the analysis of the data stored in the object database (Tables 10.3 and 10.4 on page 166 and 167, respectively).

The patterns found are of two different categories. Four patterns belong to, what has by us been defined as object unit patterns. Object unit patterns constitutes of a sequence

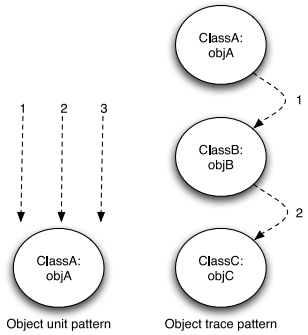

Figure 10.3: The different categories of patterns found during the OMPA. Object unit patterns cover sequences of method invocations on one object, while object trace patterns cover sequences of method invocations through more than one object.

of method invocations on one object, i.e. methods in an object has been executed in a certain order. Object trace patterns, on the other hand, are slightly different. They cover the path of execution through more than one object. In Figure 10.3 one can see the differences between object unit and object trace patterns.

Object Unit Patterns. Four object unit patterns were found during the OMPA (c.f. Table 10.3)—these patterns exercised only one object consistently over time and was found in all three applications (needless to say, the names of the objects and classes differed in all three applications, but the pattern can nevertheless generally be applied on all three applications).

The first pattern, the Vault pattern (c.f. Table 10.3), is a straightforward pattern which is executed by first invoking a constructor, then invoking a setter and finally, multiple times, a getter (before a destructor is invoked). This can be seen as a very rudimentary pattern for storing data in an object which then is fetched by one or many objects, and as such is suitable to always execute in a testing scenario, i.e. data is stored in a simple vault. During the analysis the intermediate representation was used for examining if a method was defined as a getter or setter (property) by searching for the keyword `.property`. There is of course a possibility that a method is *acting* as getter or setter while not being defined as such, but in this analysis these types of methods are disregarded and an emphasize is put on the proper definition of a getter or setter according to the CIL [77].

Next, the Storage pattern is an evolved Vault pattern and the combinations of setter and getter invocations can be many (c.f. Table 10.3). Hence, the Storage pattern can

Table 10.3: Different object unit patterns found in all three applications. The first column shows the name selected for the pattern and the second column the actual pattern. Abbreviations used: ctor and ∼ctor is short for constructor and destructor respectively, while setter and getter is a method which sets or gets data stored in the object.

Name	*Pattern*
Vault	ctor → setter → $1 \ldots n$ getter → ∼ctor
Storage	ctor → setter → $1 \ldots n$ getter → $1 \ldots n$ setter → ... → ∼ctor
Worker	ctor → setter → method invocation → ∼ctor
Cohesion	ctor → $1 \ldots n$ method invocation → ∼ctor

be constructed in different ways and a combinatorial approach might be suitable in a testing scenario (compared to the Vault pattern which is very straightforward), i.e. data is stored in a storage and the storage has (many) different ways of adding or offering content. The reason for distinguishing between a Vault and a Storage is that a Vault was common during OMPA and as such should always be used when testing object-oriented software, while Storage, on the other hand, is a more complex pattern (more steps performed) and as such needs additional analysis.

The Worker pattern at first glance looks like bad design. An object gets instantiated, and immediately filled with data. A method is next invoked which manipulates the data, returns the manipulated data and, finally, a destructor is invoked. The reason for this design might be to make sure the method's implementation can be used by different objects (extended) since it is declared `public`. If one would have opted for a method declared as `private` or even `protected`, which could be invoked when the getter is invoked, then there would be no simple way to reuse the implementation.

Finally, the Cohesion pattern is a pattern which executes one or more methods in one object. It does this without a priori setting any values and the order of executing the methods is not always important, i.e. each and every method was found to be (by analyzing objects in the object database) an atomic unit with no dependency on other methods in the class and as such the word cohesion (united whole) was found to be appropriate to use.

Object Trace Patterns. Looking at the object trace patterns, one can see four patterns that can be generally applicable (Table 10.4); these patterns exercise several objects and constitutes sequences of object:method invocations (as depicted in Figure 10.3 on page 165).

Table 10.4: Different object trace patterns found in all three applications. The first column shows the name selected for the pattern and the second column the actual pattern. Abbreviations used: ctor and ∼ctor is short for constructor and destructor respectively, while setter and getter is a method which sets or gets data stored in the object. An alphabetical character in front of the abbreviation, i.e. A:ctor, indicates that a type A object's constructor is invoked.

Name	*Pattern*
Cascading	A:ctor → B:ctor → C:ctor → ...
Storing	A:ctor → B:ctor; A:method → B:setter
Fetching	A:method → B:getter
Dispatch	A:method → B:method → C:method → ...

A fairly common pattern which seems to come up on a regular basis is the Cascading pattern. This pattern instantiates object after object (which can all be of different or same types). The approach seems to be quite common when object-oriented applications are starting up, but in addition shows up in several phases of an application's life time (from start up to shutdown).

Next, the Storing pattern and the Fetching pattern showed up many times as well. These patterns are directly connected to the object unit Storage and Vault patterns, and as such can be combined in many ways.

The final pattern which has been named is the Dispatch pattern. The Dispatch pattern simply invokes one method (not a constructor though) after another. In most cases the Dispatch patterns ends up with executing the Storing or Fetching pattern as a final step.

10.3.2 TEST CASE CREATION

Applying test case creation on the example (and on data in the experiment) is fairly straightforward when all entities needed can be found in the object database. From the previous example the following data is available:

- Method's name.
- Method's input type and values.
- Method's return type(s) and value(s).

In addition information regarding the caller can be extracted from the object database (not part of the previous example) by simply examining the caller value in the current method being executed (a value stored in the database) or, as can be seen from Figure 10.1 (page 163), by examining the timestamp for each stored entity in the database.

The above information provide us with an opportunity to automatically create test cases and, in the end, provide us with a simple smoke test [189] or regression test [116] mechanism, depending on the aim of our testing efforts. In short, the software item is executed, test cases are created from the database, patterns are extracted from the database and, in the end, the testee is validated using the aforementioned test cases and patterns.

Looking at an example taken from Banshee [28], one method is invoked consistently when playing a media file (`PlayerEngineCore::OpenPlay(TrackInfo track)`). The `OpenPlay` method takes as argument an object (of type `TrackInfo`), which in its turn follows the Vault object unit pattern, i.e. a number of values are set once and then retrieved when needed.

If a test engineer would like to store data values and types for later regression testing, then a simple test case can be automatically created using the information in the object database (in Listing 10.5 some lines have been omitted to save space).

Listing 10.5: Example of test case creation (reusing stored values for regression testing).

```
1  [Test]
2  public void TestOpenPlay() {
3          PlayerEngineCore pec = new PlayerEngineCore ();
4
5          //Get *any* object of type TrackInfo
6          Query query = db.Query();
7          query.Constrain(typeof(TrackInfo));
8          ObjectSet result = query.Execute();
9
10         foreach (object item in result) {
11             TrackInfo ti = (TrackInfo)item;
12                 try {pec.OpenPlay(ti);}
13                 catch (Exception e) {Assert.IsNotNull(e);}
14         }
15 }
```

Which in the end would allow the developer to choose between two regression testing scenarios (c.f. Figure 10.4).

Traditional random testing of the unit `OpenPlay` is fairly straightforward since, in this case, the object `TrackInfo` is composed of five scalar and five compound types. Data generation of scalar values is not something which will be discussed here (much

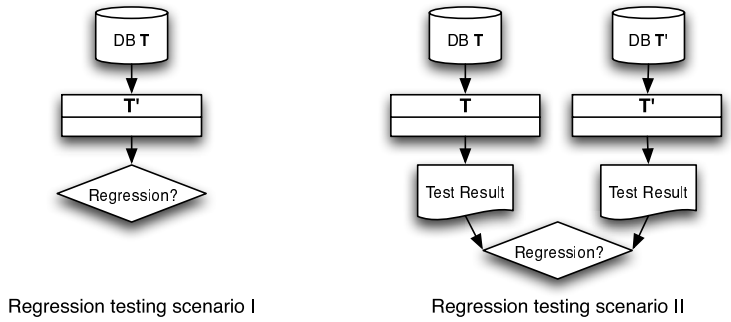

Figure 10.4: Two possible regression testing scenarios. T could be v1 of a software item, while T' might be v2. DB T and T' contain values from executing an instrumented T and T' respectively.

research contributions can be found in this research field), but the five compound types are worthwhile examining closer.

10.3.3 RANDOM TESTING AND LIKELY INVARIANTS

If one is dealing with scalar types, a traditional random testing approach can be used whereas one bombards the complete input volume with random (scalar) values. Other random testing strategies can be used as well: directed random testing [111] (by using random values which are 'close' to the input values) or anti-random testing [184] (always trying to find values which are as far a part from the input values as possible).

Turning our attention, yet again, to the example in the previous subsection, the conclusion was that five compound types were used by the type `TrackInfo`. Of the five compound types, three are found in the .NET framework (`System.TimeSpan`, `System.Uri` and `System.DateTime`), one is found in the application library and one in a third-party graphical library (`RemoteLookupStatus` and `Gtk.TreeIter` respectively).

There are now at least three test data generation strategies one can follow for generating values from these types:

- **Basic**—Retrieve values from the object database and use these as test input (Listing 10.5 and Figure 10.4).
- **Random**—Create a random data generator (manually) for each of these four types and use it to generate random, semi-random or anti-random values.

Table 10.5: Scalar or pre-defined higher-level types found in the compound types as per the example. LCD is short for Lowest Common Denominator and stands for the simplest way to use the constructor.

Compound	*Type(s) to use for generation*	*LCD*
TimeSpan	Int64, Int32	Int64
Uri	String, Uri, Boolean	String
DateTime	Int64, Int32, Calendar	Int64
RemoteLookupStatus	enum	enum
TreeIter	Int32	Int32

- **Likely Invariants**—Since the state of each and every method invocation is readily available in the object database the concept of likely invariants should be applicable [126].

The random test data generation approach have at least one disadvantage—someone or something needs to provide the correct answer. Having ready-made oracles available is uncommon in most circumstances, thus leading us to other modus operandi (as is discussed in the next section).

But nevertheless, the point to make here is that a breakdown of each compound type should occur. That is, since every compound type consists of one or more compound and/or scalar types, in most cases what should be left are scalar types (e.g. `int`, `float`, `enum`) or at the very least pre-defined higher-level compound types (e.g. the type `System.String` as found in the .NET framework).

Table 10.5 depicts a breakdown of each of the compound types left, as previously mentioned, with their respective scalar types that can be used to create random values of these types.

Important to notice in this case is that each type can now be generated automatically since they all break down to scalar pre-defined and primitive types (albeit not discrete). Furthermore, since some input values for these types are stored in the database a directed random testing approach is feasible [111].

To conclude this matter it might be appropriate to mention likely invariants. If one would try to add likely invariants, on top of the test cases created with the help of OMPA, an automated approach could be possible, i.e. enough information for generating oracles is available (according to Lam et al. [126]).

10.4 DISCUSSION

This section begins with a discussion covering different usage models for our framework and different issues concerning inspection and instrumentation of intermediate representations before discussing the concept of object message patterns.

If one would look at the general usage models, for the scenario as described in Section 10.2, a developer is faced with three possible directions to take when using the test cases created by the framework.

First, the test cases that are created can be stored and used to check for regressions in the testee, e.g. either if the testee is manipulated directly by the developer or as a way to check backwards compatibility in components and thus notify developers if a syntactic fragile base class problem, also known as the fragile binary interface problem, has occurred (for a good overview concerning the syntactic and semantic fragile base class problems please see [273]).

Secondly, collected and stored test data can be used as input for creating likely invariants (as per the proposal in [126]) and, consequently, lead to a semi-random testing approach. In addition, by using the object database, it is straightforward to analyze which objects are among the most common or uncommon and, hence, making it possible to direct testing efforts to the appropriate entities.

Finally, the test cases can, when appropriate, be used for traditional random testing, i.e. by allowing the complete input space to be flooded with data input taken randomly from the complete volume of a (scalar) type. If the last approach is selected (traditional random testing) then the developer will be facing some manual work if no oracles exist, e.g. using an interactive approach that remarks on conspicuous output which might guide the developer when examining large quantities of output data [90]. In short, since all method invocations are stored in the object database, information such as input and return values as well as internal states of a specific method, is readily available. This in the end allows a test engineer to choose which data generation technique to apply.

Regarding the collection of runtime data much can be said about how it should be performed. Usually, when dealing with dynamic features in the Java Virtual Machine or the Common Language Runtime, a developer can reach far by simply using the frameworks themselves, but sometimes they are unfortunately not enough. While, on the one hand, it is possible for a developer to discover the currently executing method quite easily during runtime, i.e. using `StackFrame`, `StackTrace` and `MethodBase` in the .NET framework [277], it is on the other hand not simple to find out the runtime input values for a method (except for e.g. editing the source code, recompiling and writing out the values explicitly). Additionally, some of these classes, which are part of the .NET framework, do not work as intended when compiled as software to be released (some code is inlined thus rendering the stack analysis to be close to useless),

and collecting the values from a stack analysis still leaves the developer with the issue of storing these values for later off-line analysis, e.g. using a simple approach as serialization is out of the question due to the fact that most types etc. are not set as `Serializable` in the .NET framework per default.

Hence, storing objects as-is in the object database has several advantages. First, there is no need to convert the objects to another format before storing them and, second, there is no need, during the test phase, to bring objects back to life from that representation—they are preserved as they once were stored in the database and can thus be retrieved easily. Second, the as-is storage of objects in an object database provides the test engineer with an excellent opportunity to extract operational profiles for e.g. random testing (see Chapter 4 and [187] for a clarification regarding the term operational profiles). Finally, finding how many objects of a particular type are created is as simple as performing a traditional database search. Nevertheless, to be able to perform more thorough analyses in the future, a stricter categorization of existing object-oriented types needs to be carried out, e.g. a method might be a property manipulator even though it is not formally defined as one.

Object Message Patterns. The concept of object message patterns, as introduced in this chapter, has potentially many interesting applications. Some of the patterns are very simplistic in its nature which is a good thing. Having a simple pattern implicitly means that in most circumstances applying that pattern during testing will be straightforward. As an example one can take the Vault pattern (Table 10.3 on page 166). The Vault pattern, could with very few steps be applied on every object that has getters and setters. It is effortless to find out if a method belonging to a particular object is formally defined as a getter or setter method in many intermediate representations (note: no source code is presumed to be available).

If one would add some statistics to how a pattern is used in most cases, it might be possible to generalize these patterns further. As an example, the OMPA might show that the Storage pattern is used, in six out of ten times, in the following sequence: set, get, get, set, get and set. Then, of course, applying that pattern on that type of object is self-evident, but might there be a point to always use that sequence when facing an object (from any application) that can be tested using that pattern? Other, more or less self-evident sequences, within the Storage pattern might be to invoke, first the constructor and then all getters in the object to see if they implement some sort of exception handling when no data is stored.

Finally, it is worthwhile to mention that querying the object database and discovering patterns is extremely time consuming (manual) work and as such should be automated to a higher extent in the future. We are convinced that tens, if not hundreds of patterns, can be found when instrumenting even small applications. Since the amount of data usually is very large there is an incitement to apply different techniques to au-

tomatically, or at least semi-automatically, discover patterns. Statistical pattern recognition [142] is one area that comes to mind which might help out in this task. By automating the extraction of patterns tedious manual work can be avoided and in the end provide the test engineer with a completely automated framework.

In the end, the analysis indicates that patterns can be found that are generally applicable to more than one application, in the same manner as design patterns are generally applicable to many problems in software development. This is in no way a surprise but, nevertheless, this chapter presents results that these patterns are to be found in very different types of software developed by completely different software development teams. As such, the general nature of these object message patterns, is likely to support testing different intermediate representations of object-oriented software. The question remains, how many object message patterns can be found that are generally applicable on most software and what do we imply with the word 'most'?

10.5 CONCLUSION

This chapter presents eight software object message patterns, for testing object-oriented software. As such it indicates, in our opinion, that the first steps have been taken on the road to deduce generally applicable object message patterns for the purpose of testing object-oriented software. In this case study, the patterns are accompanied with automatically generated test cases whose entities (test data, test fixture and test oracle) are retrieved from a database which stores runtime values that are collected when executing a use case on the testee.

The patterns were found by instrumenting three very different applications developed by three separate development teams. All in all $10,494$ instrumentation points were inserted which enabled data to be collected during runtime by the execution of use cases.

The results from the experiment reveals how object message pattern analysis, performed on data collected from executed applications, provides necessary information for automatically creating test cases using the approach presented in this chapter. On top of this, several data generation techniques can be applied, e.g. (directed) random, anti-random or the concept of likely invariants.

Chapter 11

Conclusions

11.1 SUMMARY OF RESEARCH RESULTS

The research results as presented in this section are spread over three areas: research covering the situation in small- and medium-sized projects (in 2002) and research concerning effectiveness and efficiency. To simplify for the reader, the results from this research are presented in the current section together with, when appropriate, a paraphrase of the research questions as presented in Chapter 1. In Section 11.2 the conclusions drawn from the results are presented while Section 11.3 provides an overview of issues remaining to be solved.

11.1.1 CONTEXTUAL SETTING

In Chapter 2 a survey was presented which was performed on a population constituted by software developers from industrial, open source and free software projects. In this chapter several questions were posed and the analysis of these questions brought forward a number of issues that were considered to be of importance to the area of software testing and thus found to be of interest regarding further research (as presented in this thesis). Research Question 2 (see page 15) was answered in a satisfactory manner by the survey in Chapter 2 (all research questions are to be found in Chapter 1, Section 1.3).

As is evident from reading Chapter 2, software development projects at that time, encountered many problems in different parts of the software development process (we suspect they still do). Chapter 2 placed a focus on software testing and reuse, and a picture showed itself where a number of issues needed to be examined further:

- Developers reused code but did not test the code being reused.
- Simple to use tools were lacking when it came to test creation and test analysis.
- Knowledge, on the importance of testing in general and unit testing in particular, appeared to be subpar.
- Certification of software and/or components was more or less non-existent.

Looking at the above list, it became clear that more research was needed concerning the issues of low effectiveness and efficiency in software testing. It was furthermore unambiguous that reuse in projects was taking place on a regular basis, but testing the reused code seldom happened. The matter of 'simple to use' tools was also of interest and something that was constantly in the mind of the research group at all times (the partial realization of such a tool is also presented in Chapter 10).

To this end, research needed to be performed wherein software testing of reusable components and libraries was examined. In addition, the issue of effectiveness, i.e. improving current testing techniques and combining testing techniques, and the issue of efficiency, i.e. trying to find more faults in a shorter time frame by automating more activities, needed to be investigated.

Hence, the research results from the survey (together with conclusions from related work for Chapter 2) led to the division of this thesis in two parts: effectiveness and efficiency in software testing (the definitions of effectiveness and efficiency can be found on page 88).

11.1.2 EFFECTIVENESS RESULTS

Chapter 3 presents research results from an exploratory study. The results from the study indicated that the findings in Chapter 2 were correct and RQ3 ("Is the current practice, within software development projects, sufficient for testing software items?") was answered. The current practice was not in any way encouraging and a meager picture showed itself once again. The software testing activities in the investigated project was simply not effective to the extent they were supposed to be.

In Chapter 4 the issue of effectiveness (and to some extent efficiency) was once again scrutinized and the concept of random testing was improved by introducing new quality estimations, hence answering RQ4 (see page 17). The results were encouraging and further research, in this area, has been conducted by other parts of the research group and should in the end lead to industry being able to use random testing much more than is currently the case today.

Research Question 5, or "How do different traditional software testing techniques compare with respect to effectiveness?", was answered in Chapter 5 by comparing different testing techniques. The findings were pleasing in that they demonstrated that different testing techniques have completely different results in varying environments. Hence a higher degree of effectiveness could be reached by combining different testing techniques in a fairly simple manner. Research Question 6 (see page 17) was consequently answered in Chapter 6 when several combinations of testing techniques were studied.

After looking at research results dealing with issues on effectiveness, the attention was placed on the matter of efficiency.

11.1.3 EFFICIENCY RESULTS

The case of low effectiveness is dual; a project manager would, in most cases, prefer that software testing activities were *efficient* in that they do not take up too much time in the project. The findings in Chapters 2–6 led us to believe that to be able to reach a higher efficiency in software testing more activities needed to be automated, hence a literature study was conducted trying to answer RQ7 (see page 17). In Chapter 7 a research field is presented (automated software testing) and the current status of several of the sub-disciplines in this field is clarified. The research results from Chapter 7, indicated a need, in the software testing community by large, to have clear definitions readily available to describe, compare and perform research in this field.

To this end, Chapters 8 and 9 (RQ8 and RQ9 on page 18) presented a model for classifying and comparing software testing tools and techniques. The results will hopefully be of use to industry and the research community when comparing and classifying techniques and tools for automated software testing. Additionally, the results pointed at a need to further clarify certain automatization aspects with respect to software testing (further discussed in Section 11.3).

The usage of the model in research, but more importantly in industry, should lead to projects being able to compare tools more easily and, this is even more important, look at automation benefits in the projects. In the end this would, implicitly, lead to a higher degree of efficiency by having the right tool for the job at hand.

Finally, the research results from Chapter 10 (Research Question 10 on page 18) constitutes of a framework for automated object pattern extraction and storage. This framework will hopefully be improved further and thus provide the testing community with a research and development platform for these research issues. It is nevertheless clear that the framework performs well on 'real life' software and, hence, can probably already be used in projects primarily as a way to understand complex software systems better but, in addition, as a framework for regression testing.

11.2 CONCLUSIONS

The main purpose of the research, as presented in this thesis, was to examine and improve software testing activities in small- and medium-sized projects. To this end, several case studies and experiments were conducted wherein efficiency and effectiveness issues, directly connected to 'real life' software, were investigated.

The basic assumption during the work on this thesis was that there is room for improving efficiency and effectiveness in software testing, and that the issue of low efficiency can be solved, in parts or whole, by automating more software testing activities to a higher extent. That there was a potential in being able to reach a high(er) efficiency and effectiveness was early on uncovered, but furthermore a need for classifying different entities in a software testing process was also considered mandatory for uncovering some of the more subtle issues with respect to automatization in software testing.

The overall situation, we dare say, has probably not improved in industry despite technology issues, programming languages and paradigms constantly improving or being introduced. UML [34], Java/.NET, the usage of object-oriented and component-based software, have probably all, while not providing a silver bullet [38], at least provided some help. Unfortunately, software systems have grown in size and complexity in the meantime and, by large, the software engineer is still playing catch-up with software quality issues (see e.g. [127] where the complete issue is dedicated to this subject).

Considering the main research question, as introduced in Chapter 1 (page 15):

> *Main Research Question:* How can software testing be performed efficiently and effectively especially in the context of small- and medium-sized enterprises?

The answer would be, not very surprisingly:

- Classify software testing, and especially the automated aspects, in a way to make tools, frameworks, etc. comparable.
- Use and develop frameworks which are adapted to the needs of small- and medium-sized enterprises (object-oriented software, Java/.NET, component-based development).
- Combine different software testing techniques.
- Automate more entities in the software testing process.

In our opinion, this thesis partly provides the answers by presenting a classification model, a framework for object message pattern analysis (which builds on current technology trends in industry and, in addition, is likely to be possible to automate to a high extent) and an examination on the pros and cons of combining test techniques. To this end, the road towards automated software testing is likely to be substantially improved.

11.3 FURTHER RESEARCH

The research in this thesis paves the way for several questions that need to be answered and, in addition, more research areas in need for further investigation. In this section, the two most interesting areas (with, in our opinion, the highest potential to increase efficiency in projects) are covered. It is worth mentioning, in this context, that there of course exist many other interesting research questions that need to be answered (especially when one considers the question marks left in Chapter 2, which by large probably is still current).

11.3.1 IMPROVING THE MODEL

There are several paths concerning future work one can take in the case of Chapters 8 and 9.

First of all, the model should be further evaluated on even more contributions (it is likely that there might be a need to expand the model).

Second, the issue concerning different levels of automation should be researched more as was discussed in Chapter 8 (Section 8.5 on page 130) and mentioned in this chapter (Subsection 11.1.3 on page 177). Parasuraman's et al. [222, 223] and Sheridan's [260] contributions on grades of automation, even though too general in the applicability of the definitions, will most likely play an important role forming the definitions for the specific purposes of automated software testing.

Finally, a tool for semi-automatically creating a classification of a particular technology should be implemented. For example, using a programming language to set attributes on different classes (templates) which in the end is used to generate the classification model for a particular technology.

11.3.2 IMPROVING THE FRAMEWORK

The framework as presented in Chapter 10 has many potentially interesting usages, some are presented in Chapter 10, but taking a more general view beyond the imme-

diate focus on the framework, a few more exist. For example, the following matters should be examined in the future:

- How common are the patterns, and how many patterns can we find in different types of applications?
- What pattern recognition strategies can be applied on the data collected from applications?
- Are there more 'traditional' testing techniques that can be applied on-top of the framework to further increase effectiveness?
- Use runtime data as collected, and stored in the object database, for calculating likely invariants off-line instead of doing it during runtime [126].
- How can the framework be adapted to other integrated development environments?

A case study encompassing even more applications should be performed to answer the first point, while, in order to answer the second bullet, a survey of statistical pattern recognition should be performed. The third bullet, should be clarified by further experimentation in, and development of, frameworks such as the one laid forward in this thesis. Finally, experiments and case studies should provide an answer to the last two bullets.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[2] The ACM Digital Library. `http://portal.acm.org/dl.cfm`, May 2006.

[3] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky. Validation, Verification, and Testing of Computer Software. *ACM Computing Surveys*, 14(2):159–192, 1982.

[4] L. van Aertryck, M. Benveniste and D. Le Métayer. CASTING: A Formally Based Software Test Generation Method. In *Proceedings of the 1st International Conference on Formal Engineering Methods*, page 101. IEEE Computer Society, 1997.

[5] S. Agerholm and P. G. Larsen. A Lightweight Approach to Formal Methods. In *FM-Trends 98: Proceedings of the International Workshop on Current Trends in Applied Formal Method*, pages 168–183, London, UK, 1999. Springer-Verlag.

[6] Agitar Software, Enterprise Developer Testing for Java. `http://www.agitar.com`, May 2006.

[7] A. Aksaharan. cUnit. `http://cunit.sf.net/`, May 2006.

[8] P. Ammann and J. Offutt. Using Formal Methods To Mechanize Category-Partition Testing. Technical Report ISSE-TR-93-105, Dept. of Information & Software Systems Engineering, George Mason University, USA, September 1993.

[9] D. M. Andrews and J. P. Benson. An Automated Program Testing Methodology and its Implementation. In *Proceedings of the 5th International Conference on Software Engineering*, pages 254–261. IEEE Computer Society, 1981.

[10] J. H. Andrews. A Case Study of Coverage-Checked Random Data Structure Testing. In *ASE 2004: Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pages 316–319, Linz, Austria, September 2004. IEEE Computer Society.

[11] N. W. A. Arends. *A Systems Engineering Specification Formalism*. PhD thesis, Eindhoven University of Technology, The Netherlands, 1996.

[12] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Rosu, K. Sen, W. Visser, and R. Washington. Combining Test Case Generation and Runtime Verification. *Theoretical Computer Science*, 2004. To be published.

[13] A. Avritzer and E. J. Weyuker. The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software. *IEEE Transactions on Software Engineering*, 21(9):705–716, 1995.

[14] E. R. Babbie. *Survey Research Methods*. Wadsworth Pub. Co., Inc., 1990.

[15] J. Bach. Test Automation Snake Oil. *Windows Tech Journal*, October 1996.

[16] V. R. Basili and R. W. Selby. Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering*, 13(12):1278–1296, 1987.

[17] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. Computational Intelligence for Testing .NET Components. In *Proceedings of Microsoft Summer Research Workshop*, Cambridge, UK, September 9–11 2002.

[18] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. Genes and Bacteria for Automatic Test Cases Optimization in the .NET Environment. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*, pages 195–206. IEEE Computer Society, 2002.

[19] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Pub. Co., Inc., USA, 2002.

[20] K. Beck and E. Gamma. Test Infected: Programmers Love Writing Tests. *Java Report*, pages 51–66, 1998.

[21] K. Beck and E. Gamma. Test Infected: Programmers Love Writing Tests. *Java Report*, pages 51–66, 1998.

[22] B. Beizer. *Software Testing Techniques*. van Nostrand Reinhold Computer, New York, NY, USA, 2nd edition, 1990.

[23] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, Inc., 1995.

[24] T. Bell. The Concept of Dynamic Analysis. In *Proceedings of the 7th European Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 216–234. Springer-Verlag, 1999.

[25] A. Benso and P. Prinetto. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Kluwer Academic Publishers, 2003.

[26] S. Beydeda and V. Gruhn. Test Data Generation Based on Binary Search for Class-Level Testing. In *Book of Abstracts: ACS/IEEE International Conference on Computer Systems and Applications*, page 129. IEEE Computer Society, July 2003.

[27] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Pub. Co., Inc., 1999.

[28] A. Bockover. Banshee. `http://banshee-project.org`, May 2006.

[29] B. W. Boehm. *Software Engineering Economics*. Addison-Wesley Pub. Co., Inc., 1983.

[30] B. W. Boehm. *Software Cost Estimation with COCOMO II*. Prentice-Hall, 2000.

[31] A. Boklund, C. Jiresjö, and S. Mankefors. The Story Behind Midnight, a Part Time High Performance Cluster. In *PDPTA'03: Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 173–178, Las Vegas, NV, USA, June 23–26 2003. CSREA Press.

[32] P. J. Boland, H. Singh, and B. Cukic. Comparing Partition and Random Testing via Majorization and Schur Functions. *IEEE Transactions on Software Engineering*, 29(1):88–94, 2003.

[33] F. Bolton. *Pure CORBA*. SAMS, 2001.

[34] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley Pub. Co., Inc., 1998.

[35] J. P. Bowen and M. G. Hinchey, editors. *Applications of Formal Methods*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1995.

[36] G. E. P. Box, W. G. Hunter, and J. Stuart. *Statistics for Experimenters*. John Wiley & Sons, Inc., New York, NY, USA, 1978.

[37] S. Boztas. A Necessary and Sufficient Condition for Partition Testing to be Better than Random Testing. In *IEEE International Symposium on Information Theory*, pages 401–. IEEE Computer Society Press, 1997.

[38] F. P. Brooks, Jr. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, 1987.

[39] A. W. Brown. The Current State of CBSE. *IEEE Software*, 15(5):37–46, 1998.

[40] C. J. Burgess. Using Artificial Intelligence to solve problems in Software Quality Management. In *SQM 2000: 8th International Conference on Software Quality Management, Software Quality Management VIII*, pages 77–89. British Computer Society, April 2000.

[41] D. T. Campbell and J. C. Stanley. *Experimental and Quasi-Experimental Designs for Research*. Houghton Mifflin Company, June 1966.

[42] M. Campione, K. Walrath, and A. Huml. *The Java™ Tutorial: A Short Course on the Basics*. Addison-Wesley Pub. Co., Inc., 3rd edition, 2000.

[43] M. Caramma, R. Lancini, and M. Marconi. A Probability-Based Model for Subjective Quality Evaluation of MPEG-2 Coded Sequences. In *Proceedings of the 3rd Workshop on Multimedia Signal Processing*, pages 521–525. IEEE Computer Society, 1999.

[44] A. Chamillard. An Empirical Comparison of Static Concurrency Analysis Techniques. Technical Report 96-84, University of Massachusetts, 1996.

[45] F. T. Chan, T. Y. Chen, I. K. Mak, and Y. T. Yu. Practical Considerations in Using Partition Testing. In *Third International Conference on Software Quality Management*, pages 425–433. ACM Press, 1995.

[46] R. N. Charette. Why Software Fails. *IEEE Spectrum*, 42(9):42–49, 2005.

[47] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Fault-Based Testing Without the Need of Oracles. *Information & Software Technology*, 45(1):1–9, 2003.

[48] T. Y. Chen and Y. T. Yu. On the Relationship Between Partition and Random Testing. *IEEE Transactions on Software Engineering*, 20(12):977–980, 1994.

[49] J. C. Cherniavsky. *Validation, Verfication and Testing of Computer Software*. National Institute of Standards and Technology (NIST), February 1981. NIST 500-75.

[50] J. J. Chilenski and P. H. Newcomb. Formal Specification Tools for Test Coverage Analysis. In *Proceedings of the 9th Knowledge-Based Software Engineering Conference*, pages 59–68. IEEE Computer Society, September 1994.

[51] B. J. Choi, R. A. DeMillo, E. W. Krauser, R. J. Martin, A. P. Mathur, A. J. Offutt, H. Pan, and E. H. Spafford. The Mothra Tool Set. In *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, volume 2, pages 275–284. IEEE Computer Society, 1989.

[52] CiteSeer. `http://citeseer.ist.psu.edu`, May 2006.

[53] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.*, 35(9):268–279, 2000.

[54] M. S. Cline and L. L. Werner. An Emprical Study of the Branch Coverage of Different Fault Classes. Technical Report UCSC-CRL-94-30, University of California, Santa Cruz, 1994.

[55] W. G. Cochran. *Sampling Techniques*. Wiley, 2nd edition, 1977.

[56] J. Collofello, T. Fisher, and M. Rees. A Testing Methodology Framework. In *COMPSAC'91, Proceedings of the 15th Annual International Computer Software and Applications Conference*, Tokyo, Japan, September 11–13 1991. IEEE Computer Society.

[57] Continuous Testing—PAG. `http://pag.csail.mit.edu/continuoustesting/`, May 2006.

[58] N. Daley, D. Hoffman, and P. Strooper. Unit Operations for Automated Class Testing. Technical Report 00-04, Software Verification Research Center, School of Information Technology, University of Queensland, January 2000.

[59] L.-O. Damm, L. Lundberg, and C. Wohlin. Faults-Slip-Through—A Concept for Measuring the Efficiency of the Test Process. *Software Process: Improvement and Practice*, 11(1):47–59, 2006.

[60] M. Davidsson, J. Zheng, N. Nagappan, L. Williams, and M. Vouk. GERT: An Empirical Reliability Estimation and Testing Feedback Tool. In *ISSRE 2004: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 269–280, Saint-Malo, Bretagne, France, November 2004.

[61] db4objects, Inc. db4objects. `http://www.db4objects.com`, May 2006.

[62] D. E. DeLano and L. Rising. Patterns for System Testing. In *Pattern Languages of Program Design 3*, pages 403–503. Addison-Wesley Pub. Co., Inc., Boston, MA, USA, 1997.

[63] F.-N. Demers and J. Malenfant. Reflection in Logic, Functional and Object-Oriented Programming: A Short Comparative Study. In *IJCAI '95: Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, August 1995.

[64] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Data Selection: Help for the Practicing Programmer. *IEEE Computer*, pages 34–41, 1978.

[65] R. A. DeMillo and A. J. Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[66] A. van Deursen. Program Comprehension Risks and Opportunities in Extreme Programming. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 176–185. IEEE Computer Society, 2001.

[67] E. Diaz, J. Tuya, and R. Blanco. Automated Software Testing Using a Metaheuristic Technique Based on Tabu. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 310–313. IEEE Computer Society, 2003.

[68] E. Dijkstra. Notes On Structured Programming. Technical Report 70-WSK-03, Dept. of Mathematics, Technological University of Eindhoven, The Netherlands, April 1970.

[69] djUnit. `http://works.dgic.co.jp/djunit/`, May 2006.

[70] Parasoft. `http://www.parasoft.com`, May 2006.

[71] J. W. Duran and S. C. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, 1984.

[72] E. Dustin. *Effective Software Testing: 50 Specific Ways to Improve Your Testing*. Addison-Wesley Pub. Co., Inc., USA, 2002.

[73] E. Dustin, J. Rashka, and J. Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley Pub. Co., Inc., Boston, MA, USA, 1999.

[74] Eclipse.org Main Page. `http://www.eclipse.org`, May 2006.

[75] The Class Library Specification. `http://msdn.microsoft.com/net/ecma/All.xml`, May 2006. Microsoft Corp.

[76] ECMA International—European Association for Standardizing Information and Communication Systems, ECMA International, Rue de Rhône 114, CH-1204, Geneva, Switzerland. *ECMA-334, C# Language Specification*, June 2005.

[77] ECMA International—European Association for Standardizing Information and Communication Systems, ECMA International, Rue de Rhône 114, CH-1204, Geneva, Switzerland. *ECMA-335, Common Language Infrastructure (CLI)*, June 2005.

[78] J. Edvardsson. A Survey on Automatic Test Data Generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, pages 21–28. Linköping University, October 1999.

[79] S. H. Edwards. A Framework for Practical, Automated Black-Box Testing of Component-Based Software. *Software Testing, Verfication and Reliability*, 11(2), 2001.

[80] N. S. Eickelmann and D. J. Richardson. An Evaluation of Software Test Environment Architectures. In *Proceedings of the 18th International Conference on Software Engineering*, pages 353–364. IEEE Computer Society, 1996.

[81] I. K. El-Far and J. A. Whittaker. Model-Based Software Testing. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 834, 908, 1097–1098. John Wiley & Sons, Inc., 2001.

[82] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing Test Cases for Regression Testing. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 102–112. ACM Press, 2000.

[83] EclipsePro Test. http://www.instantiations.com/eclipsepro/test.htm, May 2006.

[84] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[85] J.-B. Evain. Cecil. http://www.mono-project.com/Cecil, May 2006.

[86] Evans Data Corp. American Developer Survey Volume 2, 2002. http://www.evansdata.com, May 2006.

[87] M. E. Fagan. Advances in Software Inspections. *IEEE Transactions on Software Engineering*, 12(7):744–51, 1986.

[88] T. Faison. *Component-Based Development with Visual C#*. Hungry Minds, Inc., 2002.

[89] M. S. Feather and B. Smith. Automatic Generation of Test Oracles—From Pilot Studies to Application. *Automated Software Engineering*, 8(1):31–61, 2001.

[90] R. Feldt. *Biomimetic Software Engineering Techniques for Dependability*. PhD thesis, Chalmers University of Technology, Dept. of Computer Engineering, 2002. ISBN: 91-7291-241-3.

[91] P. Fenkam, H. Gall, and M. Jazayeri. Constructing CORBA-Supported Oracles for Testing: A Case Study in Automated Software Testing. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, pages 129–138. IEEE Computer Society, 2002.

[92] R. Ferguson and B. Korel. Generating Test Data for Distributed Software Using the Chaining Approach. *Information & Software Technology*, 38(5):343–353, 1996.

[93] M. Fewster and D. Graham. *Software Test Automation: Effective Use of Test Execution Tools*. ACM Press/Addison-Wesley Publishing Co., 1999.

[94] M. Fewster and D. Graham. *Software Test Automation: Effective Use of Test Execution Tools*. ACM Press/Addison-Wesley Pub. Co., Inc., New York, NY, USA, 1999.

[95] D. Flanagan. *Java Enterprise in a Nutshell*. O'Reilly, 2nd edition, 2002.

[96] Flashline. Software Component Certification Program. `http://www.flashline.com`, May 2006.

[97] B. Foote, H. Rohnert, and N. Harrison. *Pattern Languages of Program Design 4*. Addison-Wesley Pub. Co., Inc., Boston, MA, USA, 1999.

[98] J. Forrester and B. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 59–68, Seattle, WA, USA, August 2000. USENIX.

[99] W. B. Frakes and G. Succi. An Industrial Study of Reuse, Quality, and Productivity. *The Journal of Systems and Software*, 57(2):99–106, June 2001.

[100] P. G. Frankl. *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM Press, 2002.

[101] P. G. Frankl, R. G. Hamlet, B. Littlewood, and L. Strigini. Evaluating Testing Methods by Delivered Reliability. *Software Engineering*, 24(8):586–601, 1998.

[102] P. G. Frankl and S. N. Weiss. An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, 1993.

[103] P. G. Frankl and E. J. Weyuker. Testing Software to Detect and Reduce Risk. *The Journal of Systems and Software*, 53:275–286, 2000.

[104] N. Friedman, D. Drake, D. Camp, D. Cutting, F. Hedberg, J. Gasiorek, J. Shaw, J. Trowbridge, L. Lipka, R. Love, and V. Varadhan. Beagle. `http://beaglewiki.org`, May 2006.

[105] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub. Co., Inc., Boston, MA, USA, 1995.

[106] T. Gil. AspectDNG: a .NET Static Aspect Weaver. `http://www.dotnetguru.biz/aspectdng`, May 2006.

[107] T. Gilb. Evolutionary Delivery Versus the Waterfall Model. *ACM SIGSOFT Software Engineering Notes*, 10(3):49–61, 1985.

[108] T. Gilb, D. Graham, and S. Finzi. *Software Inspection*. Addison-Wesley Pub. Co., Inc., 1993.

[109] R. L. Glass and T. Y. Chen. An Assessment of Systems and Software Engineering Scholars and Institutions (1999–2003). *The Journal of Systems and Software*, 76:91–97, April 2005.

[110] The GNU project. `http://www.gnu.org`, May 2006.

[111] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI '05: ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 213–223, New York, NY, USA, June 2005. ACM Press.

[112] G. H. Golub and C. F. van Loan. *Matrix Computations*. John Hopkins University Press, 2nd edition, 1989.

[113] A. Gotlieb and B. Botella. Automated Metamorphic Testing. In *Proceedings of the 27th Annual International Conference on Computer Software and Applications*, pages 34–40. IEEE Computer Society, 2003.

[114] K. J. Gough. Stacking Them Up: A Comparison of Virtual Machines. In *Proceedings of the 6th Australasian Conference on Computer systems Architecture*, pages 55–61. IEEE Computer Society, 2001.

[115] J. D. Gradecki and N. Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley & Sons, Inc., March 2003.

[116] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An Empirical Study of Regression Test Selection Techniques. *ACM Transactions on Software Engineering Methodology*, 10(2):184–208, 2001.

[117] J. Guo and Luqi. A Survey of Software Reuse Repositories. In *Proceedings of the 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 92–100. IEEE, 2000.

[118] N. Gupta, A. P. Mathur, and M. L. Soffa. UNA Based Iterative Test Data Generation and its Evaluation. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, pages 224 –. IEEE Computer Society, 1999.

[119] W. J. Gutjahr. Partition Testing vs. Random Testing: The Influence of Uncertainty. *IEEE Transactions on Software Engineering*, 25(5):661–674, 1999.

[120] D. Hamlet. Connecting Test Coverage to Software Dependability. In *Proocedings of the 5th International Symposium on Software Reliability Engineering*, pages 158–165. IEEE Computer Society Press, 1994.

[121] D. Hamlet. Foundations of Software Testing: Dependability Theory. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1994.

[122] D. Hamlet. Random Testing. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[123] D. Hamlet, D. Mason, and D. Woitm. Theory of Software Reliability-Based on Components. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 361–370. IEEE Computer Society, 2001.

[124] D. Hamlet and R. Taylor. Partition Testing Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, 1990.

[125] A. Hamou-Lhadj. The Concept of Trace Summarization. In *PCODA '05: Program Comprehension through Dynamic Analysis*, pages 43–47, Antwerpen, Belgium, 2005. Universiteit Antwerpen.

[126] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, New York, NY, USA, 2002. ACM Press.

[127] S. Hassler, editor. *IEEE Spectrum*, volume 42. The Institute of Electrical and Electronics Engineers, Inc., September 2005.

[128] L. Hatton. N-Version Design Versus One Good Design. *IEEE Software*, pages 71–76, November 1997.

[129] D. L. Heine and M. S. Lam. A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 168–181. ACM Press, 2003.

[130] S. D. Hendrick, M. Webster, and K. E. Hendrick. *Worldwide Distributed Automated Software Quality Tools 2005–2009 Forecast and 2004 Vendor Shares*. IDC, Inc. Market Analysis, July 2005. Doc #33771.

[131] D. Hovel. ASP.NET Page Persistence and Extended Attributes. *Dr. Dobb's Journal of Software Tools*, 27(12):30, 32, 34–36, December 2002.

[132] W. E. Howden. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, 1982.

[133] W. E. Howden. Confidence-Based Reliability and Statistical Coverage Estimation. In *Proceedings of the 8th International Symposium on Software Reliability Engineering*, pages 283–291. IEEE Computer Society, November 1997.

[134] Eclipse Projects. `http://www.eclipse.org/hyades/`, May 2006.

[135] IBM Rational Software. `http://www-306.ibm.com/software/rational/`, May 2006.

[136] D. C. Ince and S. Hekmatpour. Empircal Evaluation of Random Testing. *Computer Journal*, 29:380, 1986.

[137] Institute for Information Systems, UCSD, University of California. *UCSD PASCAL System II.0 User's Manual*, March 1979.

[138] Institute of Electrical and Electronics Engineers. *Standard for Software Unit Testing, ANSI/IEEE 1008-1987*, 1993.

[139] International Institute of Infonomics, University of Maastricht, The Netherlands. Free/Libre and Open Source Software: Survey and Study. `http://www.infonomics.nl/FLOSS/`, May 2006.

[140] International Organization for Standardization, International Organization for Standardization (ISO), rue de Varembé, Case postale 56, CH-1211 Geneva 20, Switzerland. *International Standard ISO/IEC 9126*, 1991.

[141] International Organization for Standardization, International Organization for Standardization (ISO), rue de Varembé, Case postale 56, CH-1211 Geneva 20, Switzerland. *Common Language Infrastructure (CLI), Partition I-V, ISO/EIC 23271*, April 2003.

[142] A. K. Jain, R. P. W. Duin, and J. Mao. Statistical Pattern Recognition: A Review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):4–37, 2000.

[143] A. Jamoussi. An Automated Tool for Efficiently Generating a Massive Number of Random Test Cases. In *Proceedings of the 2nd High-Assurance Systems Engineering Workshop*, pages 104–107. IEEE Computer Society, 1997.

[144] B. Jeng and E. J. Weyuker. Some Observations on Partition Testing. In *Proceedings ACM SIGSOFT 3rd Symposium on Software Testing, Analysis and Verification*, Key West, Florida, USA, December 1989. ACM, ACM Press.

[145] B. F. Jones, H. Sthamer, and D. E. Eyres. Automatic Structural Testing Using Genetic Algorithm. *Software Engineering Journal*, 11(5):299–306, 1996.

[146] P. C. Jorgensen. *Software Testing—A Craftsman's Approach*. CRC Press, 2nd edition, 2002.

[147] JUnit. `http://www.junit.org`, May 2006.

[148] N. J. Juzgado, A.-M. Moreno, and S. Vegas. Reviewing 25 Years of Testing Technique Experiments. *Empirical Software Engineering*, 9(1-2):7–44, 2004.

[149] E. Kamsties and C. M. Lott. An Empirical Evaluation of Three Defect-Detection Techniques. In *Proceedings of the 5h European Software Engineering Conference*, pages 362–383. Springer-Verlag, 1995.

[150] C. Kaner, J. L. Falk, and H. Q. Nguyen. *Testing Computer Software*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1999.

[151] H. V. Kantamneni, S. R. Pillai, and Y. K. Malaiya. Structurally Guided Black Box Testing. Technical report, Dept. of Computer Science, Colorado State University, Ft. Collins, CO, USA, 1998.

[152] D. Karlström. Introducing eXtreme Programming—an Experience Report. In *First Swedish Conference on Software Engineering Research and Practise*, Ronneby, Sweden, October 2001.

[153] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice-Hall, Inc., 1978.

[154] A. I. Khincin. *Mathematical Foundations of Statistical Mechanics*. Dover publications, New York, 1949.

[155] B. Kitchenham, T. Dybå, and M. Jørgensen. Evidence-Based Software Engineering. In *Proceedings of the 26th International Conference on Software Engineering*, pages 273–281. IEEE Computer Society, May 2004.

[156] N. Kobayashi, T. Tsuchiya, and T. Kikuno. Non-Specification-Based Approaches to Logic Testing for Software. *Information and Software Technology*, 44:113–121, 2002.

[157] P. J. Koopman, Jr., A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic Automated Software Testing. In R. Peña, editor, *IFL '2: Proceedings of the 14th International Workshop on the Implementation of Functional Languages Selected Papers*, volume 2670 of *LNCS*, pages 84–100, Madrid, Spain, September 2002.

[158] B. Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[159] B. Korel. Automated Test Data Generation for Programs with Procedures. In *ISSTA '96: Proceedings of the 1996 International Symposium on Software Testing and Analysis*, pages 209–215. ACM Press, 1996.

[160] B. Korel. Black-Box Understanding of COTS Components. In *Seventh International Workshop on Program Comprehension*, pages 92–105. IEEE Computer Society Press, 1999.

[161] B. Korel and A. M. Al-Yami. Assertion-Oriented Automated Test Data Generation. In *Proceedings of the 18th International Conference on Software Engineering*, pages 71–80. IEEE Computer Society, 1996.

[162] N. P. Kropp and P. J. Koopman, Jr. and D. P. Siewiorek. Automated Robustness Testing of Off-the-Shelf Software Components. In *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing*, pages 230–239. IEEE Computer Society, 1998.

[163] S. Kuball, G. Hughes, and I. Gilchrist. Scenario-Based Unit-Testing for Reliability. In *Proceedings of the Annual Reliability and Maintainability Symposium, 2002*, 2002.

[164] D. Kung, J. Gao, and P. Hsia. Developing an Object-Oriented Software Testing Environment. *Communications of the ACM*, 38(10):75–87, 1995.

[165] O. Laitenberger. Studying the Effects of Code Inspection and Structural Testing on Software Quality. Technical report, International Software Engineering Research Network, 1998.

[166] O. Laitenberger and J.-M. DeBaud. An Encompassing Life Cycle Centric Survey of Software Inspection. *The Journal of Systems and Software*, 50(1):5–31, 2000.

[167] M. Lange. It's Testing Time! In *EuroPLoP 2001: Proceedings of the 6th European Conference on Pattern Languages of Programming and Computing*, Irsee, Germany, July 2001. UVK Universitatsverlag Konstanz GmbH.

[168] S. Lapierre, E. Merlo, G. Savard, G. Antoniol, R. Fiutem, and P. Tonella. Automatic Unit Test Data Generation Using Mixed-Integer Linear Programming and Execution Trees. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 189–198. IEEE Computer Society, 1999.

[169] M. Last, M. Friedman, and A. Kandel. The Data Mining Approach to Automated Software Testing. In *KDD '03: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 388–396, New York, NY, USA, 2003. ACM Press.

[170] H. B. Lee and B. G. Zorn. BIT: A Tool for Instrumenting Java Bytecodes. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[171] M. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[172] Y. Lei and J. H. Andrews. Minimization of Randomized Unit Test Cases. In *ISSRE 2005: Proceedings of the 16th International Symposium on Software Reliability Engineering*, pages 267–276, Chicago, IL, USA, November 2005. IEEE Computer Society.

[173] K. R. P. H. Leung, W. Wong, and J. K.-Y. NG. Generating Test Cases from Class Vectors. *The Journal of Systems and Software*, 66(1):35–46, 2003.

[174] B. Lewis and M. Ducasse. Using Events to Debug Java Programs Backwards in Time. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 96–97, New York, NY, USA, 2003. ACM Press.

[175] N. Li and K. Malaiya. On Input Profile Selection for Software Testing. Technical report, Colorado State University, 1994.

[176] J.-C. Lin and P.-L. Yeh. Automatic Test Data Generation for Path Testing Using GAs. *Information Sciences: An International Journal*, 131(1–4):47–64, 2001.

[177] T. Lindholm and F. Yellin. The Java™ Virtual Machine Specification, 1999. `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`.

[178] J. L. Lions. ARIANE 5: Flight 501 failure. Technical report, Ariane 5 Inquiry Board Report, European Space Agency, Paris, July 1996.

[179] V. B. Livshits and M. S. Lam. Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs. In *Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 317–326. ACM Press, 2003.

[180] G. Luo, R. L. Probert, and H. Ural. Approach to Constructing Software Unit Testing Tools. *Software Engineering Journal*, 10:245–252, November 1995.

[181] P. Lutsky. Information Extraction from Documents for Automating Software Testing. *Artificial Intelligence in Engineering*, 14(1):63–69, 2000.

[182] T. MacKinnon. XUnit—a Plea for assertEquals. `http://citeseer.nj.nec.com/525137.html`, May 2006.

[183] H. Madeira, D. Costa, and M. Vieira. On the Emulation of Software Faults by Software Fault Injection. In *Proceedings International Conference on Dependable Systems and Networks*, pages 417–426. IEEE Computer Society, 2000.

[184] Y. K. Malaiya. Antirandom Testing: Getting the Most Out of Black-Box Testing. In *ISSRE 1995: Proceedings of the 6th International Symposium on Software Reliability Engineering*, pages 86–95, Los Alamitos, CA, USA, October 1995. IEEE Computer Society.

[185] Y. K. Malaiya and J. Denton. Estimating Defect Density Using Test Coverage. Technical report, Colorado State University, 1998.

[186] A. Malec. Check: a unit test framework for C. `http://check.sf.net/`, May 2006.

[187] S. Mankefors-Christiernin and A. Boklund. Applying Operational Profiles to Existing Random Testing Results: Meeting the Square Root Efficiency Challenge. In Nuno Guimarães and Pedro Isaías, editors, *IADIS AC*, pages 233–244, Algarve, Portugal, February 2005. IADIS.

[188] D. Marinov and S. Khurshid. TestEra: A Novel Framework for Automated Testing of Java Programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, pages 22–31. IEEE Computer Society, 2001.

[189] S. McConnell. Daily Build and Smoke Test. *IEEE Software*, 13(4):143–144, 1996.

[190] G. McGraw, C. Michael, and M. Schartz. Generating Software Test Data by Evolution. Technical report, RST Corporation, 1998.

[191] C. Meudec. *Automatic Generation of Software Test Cases from Formal Specifications*. PhD thesis, The Queen's University of Belfast, Belfast, Northern Irland, UK, 1998.

[192] B. Meyer. On to Components. *IEEE Computer*, 32(1):139–140, 1999.

[193] C. Michael and G. McGraw. Automated Software Test Data Generation for Complex Programs. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, pages 136–146. IEEE Computer Society, 1998.

[194] C. Michael, G. McGraw, M. Schatz, and C. Walton. Genetic Algorithms for Dynamic Test Data Generation. Technical report, RST Corporation, 1997.

[195] C. Michael, G. McGraw, M. Schatz, and C. C. Walton. Genetic Algorithms for Dynamic Test Data Generation. In *Proceedings of the 12th IEEE International Conference on Automated Software Engineering*, pages 307–308. IEEE Computer Society, 1997.

[196] J. Miller, M. Roper, M. Wood, and A. Brooks. Towards a Benchmarking for the Evaluation of Software Testing Techniques. *Information and Software Technology*, 37:5–13, 1997.

[197] NMock: A Dynamix Mock Object Library for .NET. `http://nmock.truemesh.com/`, May 2006.

[198] T. Moher and G. M. Schenider. Methodology and Experimental Research in Software Engineering. Technical Report UM-79-2, University of Minnesota, February 1980.

[199] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, 19 April 1965.

[200] L. J. Morell. A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, 1990.

[201] M. Morisio, M. Ezran, and C. Tully. Success and Failure Factors in Software Reuse. *IEEE Transactions on Software Engineering*, 28(4):340–357, April 2002.

[202] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability, Measurement, Prediction, Application*. McGraw-Hill, New York, 1987.

[203] M. Musuvathi and D. Engler. Some Lessons from Using Static Analysis and Software Model Checking for Bug Finding. In *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[204] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[205] National Institute of Standards and Technology (NIST), Program Office Strategic Planning and Economic Analysis Group. *The Economic Impacts of Inadequate Infrastructure for Software Testing*, May 2002. NIST 02-3.

[206] Nester Home Page. `http://nester.sourceforge.net/`, May 2006.

[207] M. J. Norušis. *SPSS 13.0 Statistical Procedures Companion*. Prentice Hall PTR, 2005.

[208] Novell, Inc. Mono. `http://www.mono-project.com`, May 2006.

[209] S. C. Ntafos. A Comparison of Some Structural Testing Strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874, 1988.

[210] S. C. Ntafos. On Random and Partition Testing. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 42–48, New York, NY, USA, 1998. ACM Press.

[211] S. C. Ntafos. On Comparisons of Random, Partition, and Proportional Partition Testing. *IEEE Transactions of Software Engineering*, 27(10):949–960, 2001.

[212] NUnit. `http://www.nunit.org`, May 2006.

[213] K. Nursimulu and R. L. Probert. Cause-Effect Graphing Analysis and Validation of Requirements. In *Proceedings of the 1995 Conference of the Centre for Advanced Studies on Collaborative research*, page 46. IBM Press, 1995.

[214] A. J. Offut and S. Liu. Generating Test Data from SOFL Specifications. *The Journal of Systems and Software*, 49(1):49–62, 1999.

[215] A. J. Offutt and J. H. Hayes. A Semantic Model of Program Faults. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 195–200, New York, NY, USA, 1996. ACM Press.

[216] A. J. Offutt and S. D. Lee. An Empirical Evaluation of Weak Mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, 1994.

[217] D. O'Neill. Peer reviews. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 930–931, 937–938, 940, 1221–1227, 1506. John Wiley & Sons, Inc., 2001.

[218] Open Source Initiative. `http://www.opensource.org`, May 2006.

[219] C. Pacheco and M. D. Ernst. Eclat: Automatic Generation and Classification of Test Inputs. In *ECOOP 2005: Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 504–527, Glasgow, UK, July 2005.

[220] A. M. Paradkar, K. C. Tai, and M. A. Vouk. Automatic Test Generation for Predicates. In *Proceedings of the 7th International Symposium on Software Reliability Engineering*, page 66. IEEE Computer Society, 1996.

[221] A. M. Paradkar, K.-C. Tai, and M. A. Vouk. Specification-Based Testing Using Cause-Effect Graphs. *Annals of Software Engineering*, 4:133–157, 1997.

[222] R. Parasuraman. Designing Automation for Human Use: Empirical Studies and Quantitative Models. *Ergonomics*, 43(7):931–951, 2000.

[223] R. Parasuraman, T. B. Sheridan, and C. Wickens. A Model for Types and Levels of Human Interaction with Automation. *IEEE Transactions on Systems, Man and Cybernetics, Part A*, 30(3):286–297, 2000.

[224] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-Data Generation Using Genetic Algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[225] I. Parissis and J. Vassy. Strategies for Automated Specification-Based Testing of Synchronous Software. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, pages 364–367. IEEE Computer Society, 2001.

[226] D. L. Parnas and P. C. Clements. A Rational Design Process: How and Why to Fake it. In *IEEE Transactions in Software Engineering*, volume 12, pages 251–257. IEEE, 1986.

[227] W. De Pauw, D. Kimelman, and J. M. Vlissides. Modeling Object-Oriented Program Execution. In *ECOOP '94: Proceedings of the 8th European Conference on Object-Oriented Programming*, pages 163–182, London, UK, 1994. Springer-Verlag.

[228] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution Patterns in Object-Oriented Visualization. In *COOTS '98: Proceedings of the Fourth USENIX Conference on Object-Oriented Technologies and Systems*, pages 219–234, Santa Fe, New Mexico, April 1998. USENIX Association, 2560 Ninth Street, Suite 215 Berkeley, CA 94710 USA.

[229] Z. Pawlak. *Rough Sets: Theoretical Aspects of Reasoning about Data*. Kluwer Academic Publishers, 1992.

[230] J. H. Perkins and M. D. Ernst. Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants. In *FSE-12: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 23–32, New York, NY, USA, 2004. ACM Press.

[231] B. Pettichord. Seven Steps to Test Automation Success. In *STAR West*, San Jose, NV, USA, November 1999.

[232] P. Piwowarski, M. Ohba, and J. Caruso. Coverage Measurement Experience During Function Test. In *Proceedings of the 15th International Conference on Software Engineering*, pages 287–301. IEEE Computer Society Press, 1993.

[233] R. Poston. Test Generators. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 1757–1759. John Wiley & Sons, Inc., New York, NY, USA, 2005.

[234] M. Prasanna, S. N. Sivanandam, R. Venkatesan, and R. Sundarrajan. A survey on automatic test case generation. *Academic Open Internet Journal*, 15, 2005.

[235] T. W. Pratt and M. V. Zelkowitz. *Programming Languages: Design and Implementation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 3 edition, 1995.

[236] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.

[237] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in Fortran: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1999.

[238] L. Råde and B. Westergren. *Beta—Mathematics Handbook*. Studentlitteratur, Lund, 1993.

[239] Runtime Assembly Instrumentation Library. `http://rail.dei.uc.pt`, May 2006.

[240] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. On the Automated Generation of Program Test Data. *IEEE Transactions on Software Engineering*, 2(4):293–300, 1976.

[241] E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Inc., 2001.

[242] System.Reflection Namespace (.NET Framework). `http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemreflection.asp`, May 2006.

[243] Reflector for .NET. `http://www.aisto.com/roeder/dotnet/`, May 2006.

[244] S. C. Reid. An Empirical Analysis of Equivalence Partitioning, Boundary Value Analysis and Random Testing. In *4th International Software Metrics Symposium*, pages 64–73. IEEE Computer Society Press, 1997.

[245] Resoning Inc. How Open Source and Commercial Software Compare: A Quantitative Analysis of TCP/IP Implementations in Commercial Software and in the Linux Kernel. `http://www.reasoning.com/downloads/opensource.html`, May 2006.

[246] D. J. Richardson. TAOS: Testing with Oracles and Analysis Support. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 138–153. ACM Press, August 1994.

[247] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-Based Test Oracles for Reactive Systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118. ACM Press, 1992.

[248] D. J. Richardson and M. J. Harold. *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM Press, 2000.

[249] D. C. Rine and N. Nada. An Empirical Study of a Software Reuse Reference Model. *Information and Software Technology*, 42(1):47–65, January 2000.

[250] F. S. Roberts. *Measurement Theory*. Encyclopedia of Mathematics and its Applications. Addison-Wesley Pub. Co., Inc., 1979.

[251] M. Roper. Software Testing—Searching for the Missing Link. *Information and Software Technology*, 41:991–994, 1999.

[252] D. Rosenblum. Adequate Testing of Component-Based Software. Technical report, Dept. of Information and Computer Science, University of California, CA, 1997. http://citeseer.nj.nec.com/rosenblum97adequate.html.

[253] P. E. Ross. The Exterminators. *IEEE Spectrum*, 42(9):36–41, 2005.

[254] G. Rothermel, R. J. Untch, and C. Chu. Prioritizing Test Cases For Regression Testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[255] D. Saff and M. D. Ernst. Reducing Wasted Development Time via Continuous Testing. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 281–292. IEEE Computer Society, November 2003.

[256] D. Saff and M. D. Ernst. An Experimental Evaluation of Continuous Testing During Development. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 76–85, Boston, MA, USA, July 12–14, 2004.

[257] H. Schildt and ANSI C Standard Committee. *The Annotated ANSI C Standard: American National Standard for Programming Languages-C: ANSI/ISO 9899-1990*. Osborne Publishing, 1993.

[258] R.-D. Schimkat, M. Schmidt-Dannert, W. Küchlin, and R. Krautter. Tempto—An Object-Oriented Test Framework and Test Management Infrastructure Based On Fine-Grained XML-Documents. In *Net.ObjectDays 2000*, pages 274–287, Erfurt, Germany, October 2000. Net.ObjectDays-Forum.

[259] M. Shaw. Truth vs. Knowledge: The Difference Between What a Component Does and What We Know It Does. In *Proceedings of the 8th International Workshop on Software Specification and Design*, March 1996.

[260] T. B. Sheridan. *Humans and Automation: System Design and Research Issues*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[261] J. A. Simpson and E. S. Weiner. *The Oxford English Dictionary*. Oxford University Press, 2nd edition, 1989.

[262] A. Sinha, C. S. Smidts, and A. Moran. Enhanced Testing of Domain Specific Applications by Automatic Extraction of Axioms from Functional Specifications. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 181–190. IEEE Computer Society, 2003.

[263] R. O. Sinnott. Architecting Specifications for Test Case Generation. In *Proceedings of the 1st International Conference on Software Engineering and Formal Methods*, pages 24–32. IEEE Computer Society, 2003.

[264] H. P. Siy and L. G. Votta. Does the Modern Code Inspection Have Value? In *IEEE International Conference on Software Maintenance*, pages 281–, 2001.

[265] M. L. Soffa, M. Young, and W. Tracz. *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM Press, 1998.

[266] I. Sommerville. *Software Engineering*, chapter 3.5, pages 61–62. Addison-Wesley Pub. Co., Inc., 6th edition, 2001.

[267] I. Sommerville. *Software Engineering*. Addison-Wesley Pub. Co., Inc., 7th edition, 2004.

[268] J. Soto and L. Bassham. *Randomness Testing of the Advanced Encryption Standard Finalist Candidates*. National Institute of Standards and Technology (NIST), Computer Security Division, March 2000.

[269] J. Srinivasan and N. Leveson. Automated Testing from Specifications. In *Proceedings of the 21st Digital Avionics Systems Conference*, volume 1, pages 6A2–1–6A2–8, 2002.

[270] Software Testing Automation Framework (STAF). `http://staf.sf.net`, May 2006.

[271] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer Verlag, 1980.

[272] N. T. Sy and Y. Deville. Automatic Test Data Generation for Programs with Integer and Float Variables. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, pages 13–21. IEEE Computer Society, 2001.

[273] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[274] É. Tanter, N. M. N. Bouraqadi-Saâdani, and J. Noyé. Reflex—Towards an Open Reflective Extension of Java. In A. Yonezawa and S. Matsuoka, editors, *Reflection*, volume 2192 of *Lecture Notes in Computer Science*, pages 25–43, Kyoto, Japan, September 2001. Springer.

[275] A. Tashakkori and C. Teddlie. *Mixed Methodology—Combining Qualitative and Quantitative Approaches*. Sage Pub. Co., Inc., 1998.

[276] Software Testing Resources and Tools. `http://www.aptest.com/resources.html`, May 2006.

[277] T. L. Thai and H. Q. Lam. *.NET Framework Essentials (O'Reilly Programming Series)*. O'Reilly & Associates, Inc., 3rd edition, June 2003.

[278] R. Thayer, M. Lipow, and E. Nelson. *Software Reliability*. North-Holland, 1987.

[279] The Standish Group. The CHAOS Report. `http://www.standishgroup.com`, May 2006.

[280] R. Torkar. *Empirical Studies of Software Black Box Testing Techniques: Evaluation and Comparison*. Licentiate thesis, Blekinge Institute of Technology, School of Engineering, April 2004. ISBN: 91-7295-036-6.

[281] R. Torkar. A Literature Study of Software Testing and the Automated Aspects Thereof. Technical Report S-CORE 0501, `http://www.s-core.htu.se/reports/SCORESR0501.pdf`, Dept. of Computer Science, University West, Sweden, 2005.

[282] N. Tracey, J. Clark, and K. Mander. Automated Program Flaw Finding Using Simulated Annealing. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 73–81. ACM Press, 1998.

[283] L. Tripp. *IEEE Standards Collection—Software Engineering*. Institute of Electrical and Electronics Engineers, 1994 edition, 1994.

[284] B.-Y. Tsai, S. Stobart, N. Parrington, and I. Mitchell. An Automatic Test Case Generator Derived from State-Based Testing. In *Proceedings of the 5th Asia Pacific Software Engineering Conference*, pages 270–277. IEEE Computer Society, 1998.

[285] Unite.NET. `http://www.ascentiv.ch/products/products.htm`, May 2006.

[286] J. Voas and G. McGraw. *Software Fault Injection: Innoculating Programs Against Errors*. John Wiley & Sons, Inc., 1997.

[287] J. Voas and J. Payne. Dependability Certification of Software Components. *The Journal of Systems and Software*, pages 165–172, 2000.

[288] P. A. Vogel. An Integrated General Purpose Automated Test Environment. In *ISSTA '93: Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 61–69. ACM Press, 1993.

[289] Visual Studio Home. `http://msdn.microsoft.com/vstudio`, May 2006.

[290] C.-C. Wang. An Automatic Approach to Object-Oriented Software Testing and Metrics for C++ Inheritance Hierarchies. In *Proceedings of the International Conference on Information, Communications and Signal Processing*, 1997.

[291] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary Test Environment for Automatic Structural Testing. *Information & Software Technology*, 43(14):841–854, 2001.

[292] E. S. Weiner and J. Simpson, editors. *The Compact Oxford English Dictionary*. Oxford University Press, 2nd edition, November 1991.

[293] E. J. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4):465–470, 1982.

[294] E. J. Weyuker. Axiomatizing Software Test Data Adequacy. *IEEE Transactions on Software Engineering*, 12(12):1128–1138, 1986.

[295] E. J. Weyuker. The Evaluation of Program-Based Software Test Data Adequacy Criteria. *Communications of the ACM*, 31(6):668–675, 1988.

[296] E. J. Weyuker and B. Jeng. Analyzing Partition Testing Strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, 1991.

[297] D. A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`, May 2006.

[298] J. A. Whittaker. *How to Break Software*. Addison-Wesley Pub. Co., Inc., 2002.

[299] T. W. Williams, M. R. Mercer, J. P. Mucha, and R. Kapur. Code Coverage, What Does it Mean in Terms of Quality? In *Proceedings of the 2001 Annual Reliability and Maintainability Symposium*, pages 420–424, Philadelphia, PA, January 2001.

[300] C. Wohlin and P. Runeson. Defect Content Estimations from Review Data. In *20th International Conference on Software Engineering*, pages 400–409. IEEE Computer Society, 1998.

[301] K. W. Wong, C. C. Fung, and H. Eren. A Study of the Use of Self-Organising Map for Splitting Training and Validation Sets for Backpropagation Neural Network. In G. L. Curry, B. Bidanda, and S. Jagdale, editors, *1996 IEEE TENCON Digital Signal Processing Applications Proceedings (Cat. No. 96CH36007)*, volume 1, pages 157–62. Inst. Ind. Eng, Norcross, GA, USA, 1997.

[302] W. E. Wong. *On Mutation and Dataflow*. PhD thesis, Purdue University, Dept. of Computer Science, 1993.

[303] T. Xie. *Improving Effectiveness of Automated Software Testing in the Absence of Specifications*. Ph.D., University of Washington, Dept. of Computer Science and Engineering, Seattle, Washington, USA, August 2005.

[304] IEEE Xplore. `http://ieeexplore.ieee.org`, May 2006.

[305] The XUnit Home Page. `http://www.xprogramming.com/software.htm`, May 2006.

[306] H. Yin, Z. Lebne-Dengel, and Y. K. Malaiya. Automatic Test Generation Using Checkpoint Encoding and Antirandom Testing. In *Proceedings of the 8th International Symposium on Software Reliability Engineering*, pages 84–95. IEEE Computer Society, 1997.

[307] R. K. Yin. *Case Study Research: Design and Method*. Sage, 1994.

[308] H. Yuan and T. Xie. Substra: A Framework for Automatic Generation of Integration Tests. In *AST 2006: Proceedings of the 1st Workshop on Automation of Software Test*, May 2006.

[309] W. Yuying, L. Qingshan, C. Ping, and R. Chunde. Dynamic Fan-In and Fan-Out Metrics for Program Comprehension. In *PCODA '05: Program Comprehension through Dynamic Analysis*, pages 38–42, Antwerpen, Belgium, 2005. Universiteit Antwerpen.

[310] S. J. Zeil and W. Tracz. *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM Press, 1996.

[311] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[312] H. Zhu and P. A. V. Hall. Test Data Adequacy Measurement. *Software Engineering Journal*, 8(1):21–29, 1992.

[313] H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

[314] H. Zhu and L. Jin. *Software Quality Assurance and Quality Testing*. Academic Press Professional, Inc., 1997.

Appendix A: Survey Questions

All numbers in this appendix are, if not otherwise stated, in percentages

Q1. What is your educational level?

High School	College degree/B.Sc.	M.Sc. or higher
15	40	55

Q2. Are you male or female?

Male	Female
98	2

Q3. How old are you?

$\leq$20	21-29	$\geq$30
4	51	45

Q4. Do you consider yourself being an industry developer or open source developer?

Open Source	Business
43	57

Q5. How many developers do you usually have in one team?

Less than 10	Between 10 and 20	More than 20
77	14	9

Q6. Is there usually interaction between your team/project and other team(s)/project(s)?

Yes	Rarely	No
26	2	72

Q7. Do you find yourself having much freedom in your work as a developer?

A lot of freedom	Some freedom	Quite limited freedom
75[1]	22	3

Q8. Are you involved in setting requirements or specifications for software?

Yes	Rarely	No
94[2]	2	4

Q9. How do you know if you've fulfilled the requirements, specifications or goals for a particular software?
Only a few answers presented

1. Customer satisfaction.
2. Time will tell.
3. I just know.
4. Through testing of software and user feed-back.
5. Through customers quality assurance testing.

Q10. Do you have flexible time frames when working in projects?

Yes	Rarely	No
67[3]	22	11

[1] Of the developers experiencing 'A lot of freedom' 80% where from the open source world.

[2] Open source developers seem to be somewhat more involved in this case - although this falls within the $\pm 5\%$ error margin.

[3] Not surprisingly, the vast majority that answered yes in this question, were from the open source world (85%).

Q11. How often do you move deadlines in projects?

Often	Rarely	Never
37[4]	59	4

Q12. During a project that is proceeding according to plan, how much of your total time spent on the project do you approximately spend on:[5]

	Lot of time	Some time	No time
Analysis	33	61	6
Design	45	53	2
Implementation	70	30	0
V & V / testing	35	63	2

Q13. Which part do you find most troublesome?

Analysis	Design	Implementation	V & V / testing
25	16	35	24

Q14. Have the projects you've been taking part in stored and documented components/libraries in a systematic way for later reuse in other projects?

Yes, often	Yes, but seldom	No
47	39	14

Q15. How often do you search for re-usable code (i.e. libraries, components, classes) instead of doing it yourself?

Always	Rarely	Never
36	62	2

Q16. How many classes does your average project encompass?

Less than 100	between 500 - 10,000	More than 10,000	Uncertain
35	41	21	3

Q17. How many lines of code does your average project encompass?

Less than 500	Between 500 and 10,000	More than 10,000
4	43	53

[4] 34 developers answered yes in this question, almost all of them were from industry.
[5] This was unfortunately a question that became very hard to analyze.

Q18. What sort of development environment do you usually use?

Console editor and compiler	Fancy GUI editor + compiler	Visual Studio etc.
35	31	34

Q19. How often do you test your software?

Often	Rarely	Never
52	34	14

Q20. What type of structured approach do you use when testing software?

Black box testing	White box testing	OO testing	Other
72	10	12	6

Q21. The method long foo(int i) takes an int and converts it to a long, which is then returned.
Which approach would you like use to test the above method's ability to convert every possible int to long?

The absolute majority (>75%), that first and foremost tested their software, only tested boundary values. In some cases this was complemented by random values.

Q22. Do you test a single component or class in any way?

Yes	Rarely	No
67	22	11

Q23. Do you test an assembly of components in any way?

Yes	Rarely	No
67	16	17

Q24. Do you test a component in any way before you reuse it?

Yes	Rarely	No
43	41	16

Q25. Do you use some sort of certification when you have developed a component within your project/company?

Yes	Rarely	No
6	13	81

Q26. Do you use any specific test framework?

Yes	Rarely	No
35	18	47

Q27. If the answer was yes/rarely in the previous question, please stipulate which framework you use.

Most developers used a variant of Unit testing (usually derived from JUnit), i.e. NUnit, CppUnit, COMUnit, pyUnit, cUnit, JUnit

Q28. Does—in your opinion—the choice of framework (.NET, EJB, CORBA) affect the possibility for the software to be easily upgradable in the long term?

Yes	Rarely	No
53	16	31

Q29. Which framework do you use (e.g. Java, CORBA, .NET[6] et al.)?[7]

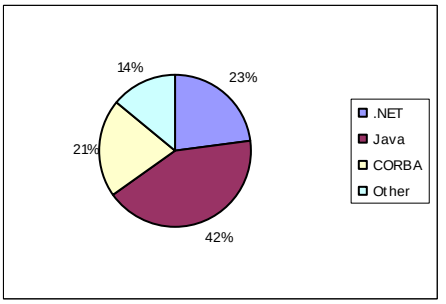

Figure 1: Usage of Frameworks

Q30. How often do you rather spend time writing glue code and reuse a class/component, than rewriting it?

Often	Rarely	Never
46	48	6

Q31. How often do you reuse a piece of code/class/component from another project?

Often	Rarely	Never
53	47	0

Q32. Does the size of (a) component(s) seriously affect decisions on whether you develop it yourself or buy it?

Often	Rarely	Never
43	31	26

[6]The rather large amount of .NET developers in the open source world was not expected initially. After contacting several of the respondents it became clear that they participated in several open source .NET implementations.

[7]Open source developers were spread over all four categories—fairly equally. Business developers on the other hand focused mostly on Java and .NET.

Q33. Does the complexity of (a) component(s) seriously affect decisions on whether you develop it yourself?

Often	Rarely	Never
59	24	17

Q34. Do open source or commercial projects usually enforce a certain technology, e.g. .NET, EJB, CORBA?

Yes[8]	Rarely	No
55	22	23

Q35. What do you think should be improved in today's component technologies (e.g. EJB, .NET, CORBA)? Do they miss a feature that a developer might need?
(Only a few answers presented)

1. The "[. . .] ability to concentrate on the business domain. Still have to write too much plumbing."

2. (1) Easy finding of existing components for reuse. (2) Certification of components. (3) Compatibility between different component technologies.

3. Those guys need to agree on ONE standard so you do not need to waste time learning new stuff all the time.

4. Today's component technologies lack maturity. half of them will be gone in 10 years.

5. They are way too bloated.

6. I stick with the smaller more specialized components written as libraries or DLL's. They do what they shall, are easier to modify and adapt to special needs.

7. Too big, too general, one-fits-it-all will more often fail than help.

8. Performance/Portability/Speed.

9. Make the technologies more portable between platforms and application server vendors.

[8] Of the 55% answering 'Yes' almost 80% came from industry.

Q36. In case of time shortage during a project, which part do you find is being reduced firstly?

Analysis	Design	Implementation	V & V / testing
16	20	4	60

Q37. When/if you have written test cases for your software, do you feel confident that you have written enough or the right tests?

Yes	Rarely	No
17	10	73

Q38. What do you feel is the main disadvantage with test frameworks being in use today? (Only a few answers presented)

1. Hard to get real numbers on how well my tests are written.
2. Most unit test case generators only do stubs. That is bad...
3. I shouldn't need to write even one test case for my software. This should be automated.
4. They are not mature enough yet. I don't want to learn a test tool that doesn't give me much help!

Appendix B: Distribution of Original and Seeded Faults in Different Functions

Table 1: Distribution of original and seeded faults in different functions.

Severity	SFF	DFF	*Func. with BVF*	*Func. with FLR*	♯ func. tested	EP	Random	BVA
T/R[1]	80	5	*6*	*79*	85	64 (4D)	7 (2D)	6
P/R[2]	20	5	*6*	*19*	25	13 (4D)	4 (1D)	6
T/U[3]	80	5	*6*	*79*	85	53 (4D)	3 (1D)	6
P/U[4]	20	5	*6*	*19*	25	9 (4D)	4 (1D)	6
Σ	200	20	*24*	*196*	220	139 (16D)	18 (5D)	24

[1] Transient/Recoverable.
[2] Permanent/Recoverable.
[3] Transient/Unrecoverable.
[4] Permanent/Unrecoverable.

LIST OF FIGURES

LIST OF TABLES

LIST OF DEFINITIONS

LIST OF LISTINGS

NOMENCLATURE

Author Index

Detective Spooner (the human):

You are a clever imitation of life... Can a robot write a symphony? Can a robot take a blank canvas and turn it into a masterpiece?

Sonny (the robot):

Can you?

From the movie "I, Robot"
(originally from the book with the same title by Isaac Asimov, 1950)

ABSTRACT

Software is today used in more and different ways than ever before. From refrigerators and cars to space shuttles and smart cards. As such, most software, usually need to adhere to a specification, i.e. to make sure that it does what is expected.

Normally, a software engineer goes through a certain process to establish that the software follows a given specification. This process, verification and validation (V & V), ensures that the software conforms to its specification and that the customers ultimately receive what they ordered. Software testing is one of the techniques to use during V & V. To be able to use resources in a better way, computers should be able to help out in the "art of software testing" to a higher extent, than is currently the case today. One of the issues here is not to remove human beings from the software testing process altogether—in many ways software development is still an art form and as such pose some problems for computers to participate in—but instead let software engineers focus on problems computers are evidently bad at solving.

This dissertation presents research aimed at examining, classifying and improving the concept of automated software testing and is built upon the assumption that software testing could be automated to a higher extent. Throughout this thesis an emphasis has been put on "real life" applications and the testing of these applications.

One of the contributions in this dissertation is the research aimed at uncovering different issues with respect to automated software testing. The research is performed through a series of case studies and experiments which ultimately also leads to another contribution—a model for expressing, clarifying and classifying software testing and the automated aspects thereof. An additional contribution in this thesis is the development of framework desiderata which in turns acts as a broad substratum for a framework for object message pattern analysis of intermediate code representations.

The results, as presented in this dissertation, shows how software testing can be improved, extended and better classified with respect to automation aspects. The main contribution lays in the investigation of, and the improvement in, issues related to automated software testing.

ISSN 1653-2090
ISBN 91-7295-089-7

2006:04