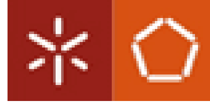


University of Minho
Informatics Department

Master Course in Informatics Engineering

Raikon

Invariant Detection Meets Random Test Case Generation

Tiago Alves Veloso
Informatics Department - University of Minho

Supervised by:

Professor José Nuno Oliveira, University of Minho, Braga, Portugal
Miguel Alexandre Ferreira, Software Improvement Group, Amsterdam, Netherlands

Braga, 26th October 2011

Acknowledgements

I would like to thank my supervisor, Professor José Nuno Oliveira from University of Minho, Braga, Portugal, for advice and guidance he has given to me while conducting the research which lead to this dissertation, and also for the time spent with me.

To my supervisor Miguel Alexandre Ferreira from Software Improvement Group, Amsterdam, The Netherlands, for always providing me with new views and advice on the problems at hand and guidance towards the best solution.

I would like to thank Carlos Pacheco and Jeff Perkins, of the Massachusetts Institute of Technology, USA and Michael Ernst of the University of Washington, USA, the developers of both DAIKON and RANDOOP, for their help and support in using, extending and integrating both tools.

To Peter Mehlitz, NASA Ames Research Center, USA and the rest of the JAVA PATHFINDER team, for providing me with help and support when using the Java Pathfinder tool, and for providing quick fixes to the tool when necessary.

To my friend and colleague, Márcio Coelho, for teaching me the Perl necessary to produce helpful scripts that were used throughout this thesis.

To my friends and family, for all the support they have given me, especially when helping to get my mind out of things allowing me to always have a fresh mind when working on my thesis.

Abstract

Full fledged verification of software ensures correction to a level that no other technique can reach. However it requires precise and unambiguous specifications of requirements, functionality and technical aspects of the software to be verified. Furthermore, it requires that these specifications together with the produced models and code be checked for conformity. This represents beyond doubt an investment that most developers and companies are neither able nor willing to make.

Although testing can not reach the same level of assurance as full fledged verification, it is the most widely accepted and used technique to validate expectations about software products. Testing is the most natural way of checking that a piece of software is doing what the developers expect it to do. Improvements to test case generation have the potential to produce a great impact in the state of the art of software engineering, by putting Software Testing closer to Formal Software Verification.

This is an exploratory project, aimed at surveying the current state of the art in the field of test case generation and related techniques for the Java language, eventually suggesting possible advancements in the field.

Resumo

A verdadeira verificação de software garante correcção de software a um nível que nenhuma outra técnica consegue igualar. No entanto, exige especificações precisas e inequívocas de requisitos de funcionalidade e aspectos técnicos do software a ser verificado. Além disso, é necessário que as especificações, juntamente com os modelos produzidos e código sejam verificados quanto à sua conformidade. Isto representa indubitavelmente um investimento que a maioria dos profissionais e empresas não são capazes, nem estão dispostos a fazer.

Embora os testes não alcancem o mesmo nível de garantia como a verificação completa, é a técnica mais amplamente aceite e usada para validar as especificações sobre produtos de software. O teste é a forma mais natural de verificar que um pedaço de software cumpre o que os programadores esperam que faça. Melhorias na geração de boletins de teste têm o potencial de produzir um grande impacto no estado da arte da engenharia de software, colocando o teste de software mais perto da Verificação Formal de Software.

Este projecto é de carácter exploratório, visando o levantamento do estado actual da área de geração de casos de teste para a linguagem Java e técnicas relacionadas, sugerindo avanços possíveis nesta área de validação de software.

Contents

Contents iii

List of Figures v

List of Tables vii

List of Listings ix

List of Acronyms xi

1 Introduction 3
- 1.1 Problem Statement . . . 3
- 1.2 Research Plan . . . 4
- 1.3 Background . . . 5
  - 1.3.1 Testing Philosophies . . . 5
  - 1.3.2 Levels of Testing . . . 6
  - 1.3.3 Goals of Testing . . . 6
  - 1.3.4 Adequacy Criteria . . . 7
- 1.4 Document Outline . . . 8

2 Literature Review 9
- 2.1 Review Process . . . 9
- 2.2 Test Case Generation Techniques . . . 10
  - 2.2.1 Random Testing . . . 10
  - 2.2.2 Bounded Exhaustive Testing . . . 12
  - 2.2.3 Mutation Driven Testing . . . 12
  - 2.2.4 Model Based Testing . . . 12
  - 2.2.5 Summary . . . 14
- 2.3 Auxiliary Techniques . . . 15
  - 2.3.1 Constraint Inference . . . 15
  - 2.3.2 Constraint Solving . . . 16
  - 2.3.3 Invariant Detection . . . 16
  - 2.3.4 Input Classification . . . 17
- 2.4 Challenges . . . 17
  - 2.4.1 Input Generation . . . 18
  - 2.4.2 Test Oracles . . . 18
  - 2.4.3 General Challenges . . . 18

3 Tool Review 21
- 3.1 Review Process . . . 21

3.2 Reviewed Tools 21
3.2.1 Test Case generation tools 21
3.2.2 Interesting Tools for Software Testing 37
3.3 A Case Study - Checkstyle 44
3.3.1 Subject of Study: Checkstyle 44
3.3.2 Tools Tested 46
3.3.3 Tools Not Tested 49
3.4 Summary 50

4 RAIKON - An Experiment in TCG Tool Interoperability **51**
4.1 Implementation 52
4.2 Evaluation Set Up 53
4.3 Results 54
4.3.1 Experiment A: Single Class 54
4.3.2 Experiment B: Package 56
4.3.3 Experiment C: Whole System 56
4.4 Discussion 56

5 Conclusions **59**

Bibliography **65**

Index of Terms **73**

A Detailed Results **77**

B Additional Tooling **99**
B.1 BIB2CSV 99
B.2 Fan-In/Fan-Out Calculator 99

List of Figures

1.1 Typical test case flow. 5

2.1 Concept lattice 11
2.2 Typical workflow of exhaustive testing. 13

3.1 How RANDOOP works [Pacheco and Ernst, 2007]. 21
3.2 Software Testing vs. Model Checking [NASA Ames Research Center, 2011] . . . 32
3.3 Example visualisation of a BinaryTree of size 5. 39
3.4 Applying REASSERT 41
3.5 AUTOMATIC JUNIT CREATION TOOL main window. 43

4.1 Architecture for RAIKON 51
4.2 Flow of the experiments. 54
4.3 Results for `FastStack` 55
4.4 Results for `org.apache.commons.collections.collection` 57
4.5 Results for Commons Collections 58

List of Tables

1.1 Cost to fix a defect in various project stages [Kaner et al., 2001] 4

3.1 Evaluation criteria for the case study. 22
3.2 Summary for RANDOOP. 24
3.3 Summary for QUICKCHECK. 26
3.4 Summary for JCHECK. 27
3.5 Summary for FEED4JUNIT. 28
3.6 Summary for T2. 30
3.7 Summary for JMLUNITNG. 31
3.8 Summary for JAVA PATHFINDER. 33
3.9 Summary for UNITCHECK. 35
3.10 Summary for UDITA. 36
3.11 Summary ECLAT. 36
3.12 Summary for PEX. 37
3.13 Summary for KORAT. 40
3.14 Summary for REASSERT. 42
3.15 Summary for DAIKON. 43
3.16 Summary for AUTOMATIC JUNIT CREATION TOOL. 44
3.17 Summary for JWalk. 45
3.18 Information collected from Checkstyle. 46
3.19 Coverage results achieved by the manual test cases. 46
3.20 Fan-In and Fan-Out by package of Checkstyle. 47
3.21 RANDOOP generated tests for the target package compared with manually written tests. 47
3.22 Comparison of JMLUnitNG results. 49

5.1 test case generation (TCG) Tools Summary. 61
5.2 Interesting Tools Summary. 62
5.3 RAIKON Summary. 63

A.1 Set up time for `FastStack` 77
A.2 Set up time for `org.apache.commons.collections.collection` 77
A.3 Set up time for Commons Collections 77
A.4 Detailed results for `FastStack` (0-55 seconds) 78
A.5 Detailed results for `FastStack` (60-120 seconds) 79
A.6 Detailed results for `org.apache.commons.collections.collection` (0-55 seconds) 80
A.7 Detailed results for `org.apache.commons.collections.collection` (60-120 seconds) 81
A.8 Detailed results for `org.apache.commons.collections.collection` (125-185 seconds) 82
A.9 Detailed results for `org.apache.commons.collections.collection` (190-240 seconds) 83
A.10 Detailed results for Commons Collections (0-55 seconds) 84

A.11 Detailed results for Commons Collections (60-120 seconds) 85
A.12 Detailed results for Commons Collections (125-185 seconds) 86
A.13 Detailed results for Commons Collections (190-250 seconds) 87
A.14 Detailed results for Commons Collections (255-315 seconds) 88
A.15 Detailed results for Commons Collections (320-380 seconds) 89
A.16 Detailed results for Commons Collections (385-445 seconds) 90
A.17 Detailed results for Commons Collections (450-510 seconds) 91
A.18 Detailed results for Commons Collections (515-575 seconds) 92
A.19 Detailed results for Commons Collections (580-640 seconds) 93
A.20 Detailed results for Commons Collections (645-705 seconds) 94
A.21 Detailed results for Commons Collections (710-770 seconds) 95
A.22 Detailed results for Commons Collections (775-835 seconds) 96
A.23 Detailed results for Commons Collections (840-900 seconds) 97

List of Listings

2.1 Example to illustrate symbolic execution. 15

3.1 Implementation of a random Stack generator 24
3.2 Application of the Stack generator in a test case 25
3.3 Application of the Stack generator in a parameterised test case 26
3.4 Example of a Feed4JUnit equivalence partition test. 28
3.5 Example of T2 specifications. 29
3.6 Example of a test case invoking T2. 29
3.7 Example JPF configuration. 33
3.8 The classic dining philosophers problem. 34
3.9 Example binary tree implementation. 38
3.10 Example binary tree imperative predicate. 38
3.11 Example binary tree finitisation. 39
3.12 Two Invariants inferred by DAIKON. 48
3.13 Example of a class annotated with invariants detected by DAIKON. 48
3.14 Example of incomplete invariants detected by DAIKON. 49

List of Acronyms

BBT . . . . Black-box testing

FSV . . . . Formal software verification

GBT . . . . Grey-box testing

PUT . . . . Parameterised unit testing

TCG . . . . Test case generation

WBT . . . . White-box testing

Chapter 1

Introduction

Quoting the co-founder of Microsoft, Bill Gates, in a recent interview [Eaton, 2011], *the importance of software is higher today than ever*. Software is developed by people with no completely reliable method to do so. This means that errors in software, which due to the importance of software, are more and more dangerous.

To verify software there are formal software verification (FSV) techniques, which allow practitioners to achieve the highest degree of confidence in its correction [Holzmann, 2010]. However, these techniques come with overloaded costs. For example, the estimated costs of completely verifying an operating system was reported to be around 20 people years [Klein et al., 2009]. This gives a rough estimate of 700 US$ per line of code. In the field of verification techniques we can find theorem proving and model checking.

Theorem proving is a formal verification technique that supports program mathematical correction arguments [Chang et al., 1973]. In order to verify a program using theorem provers, practitioners need to establish a sound link between the implementation in a programming language and its mathematical semantics. Only then *off-the-shelf* theorem provers be used to discharge the mathematical logic derived from the properties to be verified [Alexi, 1988; Vermolen, 2007]. Model checking is another technique to formally verify software. It is based on an exhaustive search for counter examples for a given property, within a bounded domain [Clarke and Emerson, 1981; Clarke, 2008]. Compared to theorem proving, it is easier to automate, but does not achieve the same degree of completeness, due to its bounded nature.

Neither of these techniques, reliable as they are, is widely adopted in mainstream software industry [Holzmann, 2010]. This is due to the cost/benefit trade-off in applying them [Klein et al., 2009]. The alternative that is most accepted is testing. Testing is even lighter than model checking [Myers, 1979], at the cost of being even less complete. What makes software testing more affordable is the fact that it can be done by regular developers using the same programming languages in which products are implemented. Testing relates inputs with outputs in accordance with an expected results [Myers, 1979].

1.1 Problem Statement

A good testing infrastructure for a product or service, it is estimated at least 50% of the total costs of the project [Myers, 1979]. This is due to the importance of detecting problems in the early stages of a project, when they are cheaper to fix [Sommerville, 2010]. An estimation of the costs required to fix problems detected in earlier product development stages can be found in Table 1.1 [Kaner et al., 2001]. Note however that, there is also a cost associated with the maintenance of the test suites throughout the entire project, to ensure the correct behaviour of the product throughout the various stages [Harrold et al., 1993]. An inadequate testing infrastructure can lead to even bigger losses [Tassey, 2002].

	Stage Detected				
Stage Introduced	Requirements	Architecture	Construction	System Test	Post-release
Requirements	1x	3x	5x	10x	10-100x
Architecture	-	1x	10x	15x	25-100x
Construction	-	-	1x	10x	10-25x

Table 1.1: Cost to fix a defect in various project stages [Kaner et al., 2001]

In order to reduce costs, or to better take advantage of them, researchers and practitioners focus their efforts in ways to automate test case generation (TCG) [Edvardsson, 1999]. This technique attempts to cover a considerable amount of inputs, while also testing more production code. TCG attempts to increase the effectiveness of testing, since it is less exposed to trivial mistakes. Also, due to the generation being automatic, there is considerably less effort with maintenance of a test suite. On the downside, automatically generated tests tend to be simpler than manually written ones, and therefore they are only able to detect simpler flaws in the software.

1.2 Research Plan

The ultimate goal of this research is to assess the prospect of advance in TCG and, if possible, provide an implementation of such advance. First we need to explore what techniques are available. Only then we will know whether or not potential for making a contribution to the field. For the purposes of this research, we will focus on TCG for the Java programming language. However, we will not limit the study of techniques and tools to the ones for Java, as there might be valuable lessons to learn from other languages or frameworks.

We propose to answer the following research questions:

RQ1 *What methods are there for test case generation?*

RQ2 *How do the existing methods compare to each other?*

RQ3 *Is there an opportunity to produce an advancement in existing methods for fully automated test case generation?*

In order to answer these research questions we have decided to break each of them into tasks in order to better structure the research. **RQ1** is broken down into the following tasks:

T1.1 Research the state of the art in TCG;

T1.2 Research techniques that might be used in the context of TCG;

For **RQ2** the break down in tasks is:

T2.1 Select of the most promising tools according to the survey done to answer **RQ1**;

T2.2 Conduct of a case study, where the tools selected in **T2.1** are applied to real world software;

Finally **RQ3** is broken down into the following tasks:

T3.1 Formulate requirements for our contribution;

T3.2 Based on the results of the case study conducted for **RQ2**, decide whether to improve one of the existing tools or implement a new tool chain;

T3.3 Implement a prototype of what was decided in **T3.2**;

1.3 Background

A test case consists of passing an input value to a method under test and checking if the output of that method conforms with a set of expected results in order to confirm that the method is behaving correctly [Myers, 1979]. Figure 1.1 illustrates a typical flow of a test case. The main advantage of manually written test cases is the control it provides to developers, allowing them to define input and expected output values. The main disadvantages are the high costs of the process and the obvious subject to human error.

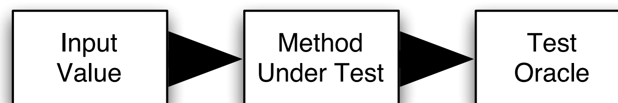


Figure 1.1: Typical test case flow.

The following sections provide a deeper understanding about testing.

1.3.1 Testing Philosophies

By tradition, testing philosophies are divided into two major factions, white-box testing (WBT) and black-box testing (BBT) [Myers, 1979]. Their combination leads to get another philosophy known as grey-box testing (GBT) [Kaner et al., 1999].

WBT is a testing methodology that checks the internal workings of a piece of software. To follow this approach the source code must be available to the practitioner, since test data are derived from the logic of the software. This approach is useful in detecting errors and problems within the code itself, but fails to detect unimplemented or incorrect specifications, because WBT completely disregards specifications. Ultimately, the goal of WBT is to check every possible execution of the software [Myers, 1979].

On the other hand, BBT validates that a software conforms to a given specification. Unlike WBT, tests following BBT philosophy do not require knowledge of the source code. This approach is used to detect incorrectly implemented specifications. However, it is not as accurate as WBT in pin-pointing problems with the internal logic. In the long run BBT attempts to test all possible input/output combinations [Myers, 1979].

Blending some aspects of both philosophies has lead to a third one, known as GBT. GBT does not fit into BBT because it leverages some knowledge of the internal logic. However, it can not be considered WBT either since it does not aim at exploring the problems in the internal logic of the software. The main idea is to leverage knowledge of the specification and the internal logic to produce better informed tests cases [Kaner et al., 1999].

Summing up, these philosophies offer guidelines to create reliable test suites. Neither BBT nor WBT, present a better offer than the other. On the other hand, GBT offers a good compromise between specification validation and structural verification.

1.3.2 Levels of Testing

In the process of testing software products, there are various *granularity* levels on which practitioners can test software. The finest of them is unit testing, which leads to integration testing, and finally to the coarsest level, system testing [Kaner et al., 1999]. Other levels exist, however, these three are considered the most relevant. This section will describe each of them in detail.

The finest level, unit testing, consists in testing software units, the smallest components of a software system [Kaner et al., 1999]. In essence a unit test is a call to the unit being tested with a predefined input value. By carefully selecting the inputs passed to the unit under test, the tester can predict the expected outputs, and then express that expectation as an assert statement. Such statements in a test are called oracles. In order to write high quality unit tests, both a very rigorous discipline and considerable effort are required, since each unit requires at least one test, but preferably one test per control path [Zhu et al., 1997].

There is, a technique that attempts to reduce efforts in unit testing, known as parameterised unit testing (PUT) [de Halleux and Tillmann, 2008; Tillmann et al., 2010]. The major contribution introduced by this technique is the possibility of providing parameterised input values, meaning that input values are not defined in the test itself, but at run-time and according to a predicate expression. PUT is therefore a generalisation of unit testing that requires more abstract oracles. Both approaches follow a WBT philosophy.

A coarser level of testing is integration testing. At this level, instead of testing single units, entire modules or sets of units are tested [Myers, 1979; Kaner et al., 1999]. The main goal is to check that the interactions of the modules under test are behaving as expected. However, there are some limitations to integration testing, for instance it makes it harder to pin-point the source of the detected problems [Kaner et al., 1999]. Integration testing usually follows a GBT philosophy.

The coarsest level of testing is called system testing [Kaner et al., 1999]. At this level the whole product is placed under test. This is done to validate that the software meets the requirements defined by the stakeholders, which makes system testing purely BBT. System testing suffers from the same limitations as integration testing, making it even harder to pin-point the source of flaws [Kaner et al., 1999].

Summing up, three of the most relevant levels of testing were described: unit testing, integration testing and system testing. No level stands out from the others, meaning they all complement each other. While testing a software system, one should consider more than one level of testing to ensure a higher degree of confidence [Kanstrén, 2008].

1.3.3 Goals of Testing

What do testers expect to learn from testing a piece of software? Testers may want to learn more about the performance of the system under test or if all requirements are met. Approaches known as non-functional testing, regression testing and acceptance testing provide for different goals in testing.

Non-functional testing addresses non-functional requirements, such as performance and scalability [Myers, 1979]. To evaluate performance of the software, the system is run several times in different machines and the performance on each machine is measured. To measure scalability,

the system is put through an increasing amount of workloads in order to assess the workload which it can handle.

Regression testing is a type of testing whose aim is to find errors that might have been introduced after applying changes to a certain functionality [Kaner et al., 1999]. This is done by recording in a regression test suite the behaviour of software before the changes are made. Then, after making the changes the regression test is used to check that only the intended new behaviour was affected by the change. Thus, no undesirable side effects were introduced. Regression tests may reveal newly introduced faults, or they may fail if the changes were meant to intentionally change functionality. Another use of regression testing, other than finding errors, is to assert that the performance of the system was not affected. Regression testing can, therefore, be accomplished at the various levels of testing.

On the other hand there is acceptance testing, also known as functional testing, which is a type of testing to check that a given implementation meets its requirements [Kaner et al., 1999]. This can be done by a tester with access to the requirements of the system, or by the final users of the system. This is usually a final stage of development, where any flaws may be uncovered before delivery. Usually, acceptance testing is done at system testing, but it is also possible to apply at the other levels.

These two forms of testing are not necessarily opposites, as they complement each other. An acceptance test can be later used as a regression test. While regression tests are based on previous versions of the production code, acceptance tests are based on a baseline specification that represents the intended functionality [Myers, 1979].

1.3.4 Adequacy Criteria

As previously mentioned, the main goal of software testing is to provide stakeholders with confidence in the software under test. In order to be easily understood by everyone, confidence should be expressed as a number. As such, the quality of a given test suite must be measured according to proper adequacy criteria [Zhu et al., 1997]. This section will describe some adequacy criterion: structural coverage; fault detection; input bound coverage and assertion density.

The first adequacy criterion covered in this section is also the most common, structural coverage. This criterion measures how much of the source code is executed by the test suite. Structural coverage can measure the coverage over structural elements like: line coverage, branch coverage and loop coverage [Zhu et al., 1997].

It is believed that when an element is executed by a test suite there are less chances of problems occurring in that element [Zhu et al., 1997]. This is, however, not enough because in addiction to executing the statement, it is also necessary to check that the result is the expected.

Another criterion of relevance is fault detection. This criterion measures how reliable a given test suite is in detecting defects in the code [Zhu et al., 1997]. This becomes especially useful while creating a test suite for regression testing. These tests aim to prevent the introduction of faults while maintaining the existing features and in developing new ones. A common form of evaluating the robustness of a test suite in relation to fault detection is *mutation analysis* [Zhu et al., 1997]. This analysis consists in injecting artificial changes, *mutations*, in the source code and then checking how many of the *mutations* the test suite is able to detect. The higher the number of *mutations* detected the more reliable a given suite is.

Another criterion is input bound coverage. This adequacy criterion measures how much of the input domain is being covered [Zhu et al., 1997]. The higher the input bound coverage, the more

likely the chances of detecting errors in the behaviour of the system under test are. However, covering the whole extent of the input domain is a virtually impossible task in most cases.

The last adequacy criterion covered in this section is known as assertion density. This criterion measures the number of assertions per lines of code [Foster et al., 2003], thus the higher the ratio the more assertions per each line of code. This metric leads to a higher confidence in the test suite, in the sense that more assertions can detect more discrete problems in the software.

In summary, this section provides some insight to several adequacy criteria in the field of software testing. These criteria are very important, because in order to measure the confidence in a software through its tests on should have confidence in the test suite itself.

1.4 Document Outline

The remainder of this dissertation is organised as follows. A review over TCG literature is presented in Chapter 2: Literature Review which also includes an overview of the current major challenges in TCG which allow us to answer **RQ1**. Chapter 3: Tool Review describes a review of chosen tools that can be used in TCG, we try to apply all the selected tools to a real world tool, Checkstyle.. Chapter 4: RAIKON - An Experiment in TCG Tool Interoperability describes the implementation of our tool, Raiko, which is a combination of a tool for emphdirected random testing, RANDOOP, with a constraint inference system, DAIKON. The document concludes in Chapter 5: Conclusions drawing conclusions and future work.

Chapter 2

Literature Review

This chapter reports on the literature review process conducted while researching into the state of the art in TCG. The most relevant TCG techniques found in the review are described, including random testing, bounded exhaustive testing, mutation-driven testing and finally model-based testing. Other techniques that can be used in the context of TCG are also discussed. The chapter closes with a discussion of the identified challenges in test case generation.

2.1 Review Process

This section will describe the literature review process conducted while studying the state of the art in software testing. The papers were found through searches in known search engines and through analysis of bibliography listings. It was decided to experiment with formal concept analysis [Wolff, 1993] in the context of literature reviewing. formal concept analysis provided a wider perspective over the field of TCG, while revealing interesting relations among the many contributions to the field.

Formal concept analysis is a technique that allows the automatic derivation of an ontology from a matrix of objects and attributes [Wolff, 1993]. For the purpose of this review, the objects are the papers and the attributes are the topics covered by the papers. A list of the attributes chosen follows.

Adequacy Criterion If the paper discusses relevant adequacy criteria for test suites;

Constraint Inference If the paper discusses constraint generation techniques;

Constraint Solving If the paper discusses constraint solving techniques;

Data Generation If the paper discusses or addresses the test data generation problem;

Dynamic Analysis If the paper discusses applications of concrete executions;

Path Selection If the paper discusses path selection decidability problems;

Programming Language If the paper relates to a specific programming language;

Software Failure If the paper discusses or describes software failures or faults;

Software Testing If the paper discusses or addresses software testing in some way;

Symbolic Execution If the paper discusses applications of symbolic execution;

TCG Technique If the paper describes a TCG technique;

TCG Problem If the paper describes the TCG problem;

TCG Tool If the paper describes a tool or system that either performs TCG or can be reused as part of a TCG tool-chain.

Having constructed the context matrix, the ConExp tool [Tilley, 2004; Yevtushenko, 2011] was used to generate the ontology illustrated in Figure 2.1. This ontology is represented by a concept lattice. The bottom node of the lattice collects all papers that do not fit into any of the attributes. By contrast, the top node contains all papers that fit into all of the attributes. However, in the lattice of Figure 2.1, there are no papers neither in the bottom nor top nodes of the lattice. This means that all the papers reviewed were both classifiable and diverse. By examining the remaining nodes, some conclusions can be drawn about the literature review thus carried out. Some of these conclusions follow:

- Most researched techniques for TCG refer to an implementation. This means that all techniques researched have associated tools.
- Not all papers are related to testing. This is due to the need of collecting some information, not directly related with software testing. For example, applications of given techniques in contexts other than software testing.

This experience with formal concept analysis was a pleasant experience. We have found that we had overlapping attributes which lead us to reduce them in number. It has allowed us to acquire a better perspective on what we were searching for and even find some related concepts. Bellow we fine a detailed account of what we have learned from the literature review process.

2.2 Test Case Generation Techniques

This section covers the most common techniques for TCG. These are random testing, bounded exhaustive testing, mutation-driven testing and finally model-based testing.

2.2.1 Random Testing

The most basic TCG technique is random testing. This technique ranks among the most simple techniques to implement, and also as one able to generate a high number of test cases per unit of time [Edvardsson, 2002]. This is due to the simple nature of this technique, since there is no sort of *guidance* in the generation process, i.e. it generates inputs regardless of the domain of the software, as any random stream of data can be used to produce an input. There are essentially three types of application of random testing. It can be used to generate random sequences of method invocations and random input values to execute them, as happens in Randoop [Pacheco and Ernst, 2007]. Another possible application of random testing is QuickCheck [Claessen and Hughes, 2000; Hughes, 2009].

As a downside to random testing, the chance of achieving an acceptable level of structural coverage is low [Deason et al., 1991; Offutt and Hayes, 1996]. This is due to the nature of the random test generator, which might not be able to produce the required input to cover relevant paths of a method. Recent research, however, has revealed some interesting properties about random testing [Ciupa et al., 2007]. For instance, the number of detected flaws seems to be

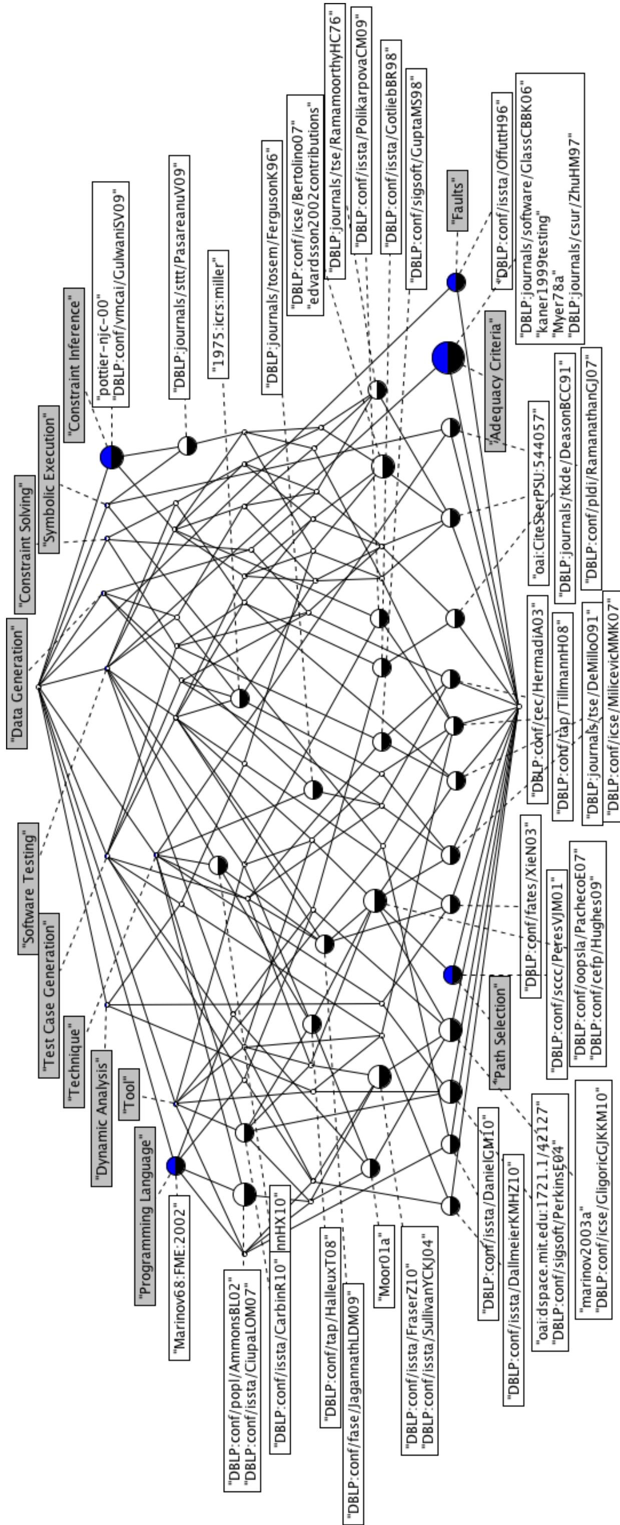


Figure 2.1: Concept lattice

inversely proportional to the elapsed testing time, meaning that there is no added gain in increasing the generation time of a random testing tool indefinitely. The same study has also reported that random testing is not limited to detecting artificial flaws seeded into the source code. It showed that this technique is also capable of detecting complex bugs in software.

2.2.2 Bounded Exhaustive Testing

Another TCG technique that is being brought forward is bounded exhaustive testing [Sullivan et al., 2004]. This technique generates test suites to meet a considerably high input bound coverage. The technique ensures that every input combination within a bounded scope is tested to check that the method under test is producing the expected outputs for each and every single one of them, making it a very effective technique for testing. There are two ways of using bounded exhaustive testing: 1) provide an oracle to be checked for a given input bound; 2) provide a description of valid inputs and generate appropriate inputs from the provided description, and pass them to a PUT.

There are two interesting tools in the field of bounded exhaustive testing, namely KORAT [Milicevic et al., 2007] and JAVA PATHFINDER [Visser and Mehlitz, 2005; Pasareanu and Rungta, 2010]. KORAT allows users to describe valid inputs of complex structures and a scope, upon which it generates proper inputs. JAVA PATHFINDER allows users to provide a test oracle. It will check that the software is meeting them. Nonetheless, there are other tools that could be considered for bounded exhaustive testing, even though they are not directed at this. Such is the case of QUICKCHECK [Claessen and Hughes, 2000; Hughes, 2009] with the aid of custom-made data generators for such a purpose [Runciman et al., 2008]. Figure 2.2 illustrates the typical workflow of a bounded exhaustive testing tool.

There are, however, serious drawbacks to bounded exhaustive testing. One of these is the large amount of computing resources required to generate sufficient inputs to cover a significant portion of the input domain, which, in many cases, is virtually unlimited. Nonetheless, recent research advances are attempting to reduce such costs by some orders of magnitude [Jagannath et al., 2009].

2.2.3 Mutation Driven Testing

Mutation-driven testing [Fraser and Zeller, 2010] consists in improving test suites by generating tests to detect a higher number of *mutations*. This technique uses *mutation analysis* techniques (recall in Section 1.3) to learn of possible *mutations* and then it attempts to generate appropriate oracles to properly detect those *mutations*. A mutation-driven testing technique which was implemented in a tool named $\mu Test$ [Fraser and Zeller, 2010] generates test cases while leveraging genetic algorithms to do so. This technique depends on the source code and an existing test suite.

2.2.4 Model Based Testing

Another research approach to TCG is model-based testing. This technique consists in extracting test cases from an existing model of the system under test [Bruno et al., 1995]. There are various approaches to model-based testing, from which, the following are singled out [Utting and Legeard, 2006]:

- Generation of test input data from a domain model;

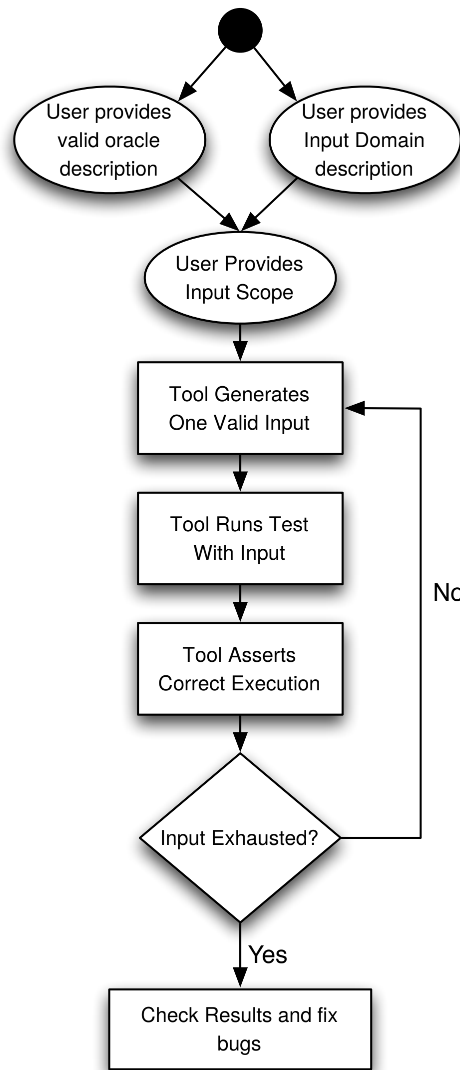


Figure 2.2: Typical workflow of exhaustive testing.

- Generation of test cases from an environment model;
- Generation of test cases with oracles from a behaviour model; and
- Generation of test scripts from abstract tests.

The first approach, input generation from a model consists in extracting information about the input domain from the model and generate appropriate test data that can be used while testing the software. The second approach, generation of test cases from an environment model where the system under test will be used, for instance a statistical study of the expected load of the system under test. From this model it is possible to construct sequences of calls to the system under test to validate the behaviour under the designated environment. The third approach to model-based testing, and perhaps the most interesting, allows the generation of test oracles, once a behavioural model is available. This type of model holds enough information to generate complete test cases with proper test oracles. Unlike the other approaches, this approach covers all the test construction phases since it contains the relationship between the inputs and the outputs. The last approach aims at the generation of executable test cases based on abstract representation, for example a UML sequence diagram.

The most serious limitation in applying model-based testing is the constant lack of appropriate models to generate tests from. But even in the presence of models, there are still limitations to model-based testing [Utting and Legeard, 2006]. For instance, no approach is able to detect all the differences between the model and the implementation, or the need to prove that the model is correct. Its costs in comparison to other TCG techniques are still relatively high, since it requires abstract models. Also there is the problem of keeping the software model synchronized with the software code. These limitations keep most mainstream software industries away from model-based testing [Sarma et al., 2010].

2.2.5 Summary

This concludes the state of art in TCG. The techniques surveyed were: random testing, bounded exhaustive testing, mutation-driven testing and model-based testing. Random testing is the simplest and cheapest technique studied. However, it is the one with the least chances of finding software bugs. This does not mean that random testing is worthless since there are reports of its usefulness [Ciupa et al., 2007]. Bounded exhaustive testing has the goal of covering a significant portion of the input domain, increasing the chances of finding a bug on the system under test. Mutation-driven testing is a technique that focuses on the generation of test cases specifically aimed at *mutation* detection. The last TCG studied, model-based testing, leverages a model of the software and generates complete test cases or appropriate input test data to check that the implementation meets the behaviour described by the model of the software.

In summary, no single technique presents a major advantage over the others and they rather complement each other. However, the technique with most potential for TCG is model-based testing, because it is capable of automatically generate the most complete input test cases with test data and test oracles. Also because the generated test cases are capable of validating the implementation against the model, in spite of the constant lack of suitable models for model-based testing preventing this technique from being more widely adopted.

2.3 Auxiliary Techniques

The literature review revealed software testing is commonly related to constraint satisfaction problems. This happens because constraint satisfaction problems are related to the generation of inputs and the inference of oracles, which are two important aspects of TCG.

This section explains constraint inference, constraint solving and related techniques, as well as their applications in the context of TCG.

2.3.1 Constraint Inference

Inference is the process of deriving strict logical consequences from assumed premises. A possible definition of constraint inference is the relation between constraints and their consequences. For example let D and C be two constraint sets. We say that $D \models C$ if and only if, for a given valuation[1] V, if V satisfies D then V also satisfies C. In (2.1) we attempt to express the definition of constraint inference as a mathematical equation.

$$D \models C \iff \forall V : (V(D) \vdash \top \implies V(C) \vdash \top) \tag{2.1}$$

In other words, constraint inference attempts to model a property system into a simpler one. This is done so as to better understand the constraint system and to facilitate its solving. In computer science, constraint inference is used in many fields, like symbolic execution [Ramamoorthy et al., 1976], dynamic analysis [Gulwani et al., 2009] and more recently a combination of both, named concolic execution [Pasareanu and Visser, 2009].

Symbolic Execution

Symbolic execution is a technique that executes programs using symbolic values, this is, instead of using actual values to execute the program, symbolic execution uses predicate expressions that represent the inputs. These expressions act like variable substitutions. The goal of symbolically executing a program is to construct an expression based on its input variables [Ramamoorthy et al., 1976]. This expression can then be solved to compute inputs for the program. Consider the statements in Listing 2.1. A symbolic execution on the statements would produce the expression `(((a * a)+ (b * b))== (c * c)`.

```
x = a * a;
y = b * b;
z = c * c;

result = ((x + y) == z);
```

Listing 2.1: Example to illustrate symbolic execution.

Using symbolic execution, there is no need to perform violation checks on each branch, since all of them can be resolved in a single execution [Ramamoorthy et al., 1976]. However, symbolic execution is entailed by some limitations, for instance function calls can not be resolved, unless we either perform an inline substitution of the function being called for its corresponding instructions or learn the symbolic expression of those functions before finishing the current function. Other

[1] The assignment of values

limitations of symbolic execution, are object, pointer and array referencing [Ramamoorthy et al., 1976]. Symbolic execution has no way of knowing their value since they are only instantiated during run-time.

Dynamic Analysis

Dynamic analysis, on the other hand, makes use of actual values while executing a program [Gulwani et al., 2009]. By using actual values to perform the executions, the problem of object, pointer and array referencing is much less accentuated since the actual value of those elements is always known throughout the execution of the program. Also, function calls do not present as much of a problem to dynamic analysis as compared to symbolic execution, since it can simply execute that call in order to know what the returned value is. There is a downside, however, and this is that dynamic analysis requires a considerably large number of executions in order to infer a reliable constraint system. This is due to the violation checks for each branch, which requires different executions, with different input values.

Concolic Execution

Recent research, has lead to a combination of both strategies, symbolic execution and dynamic analysis, leveraging the stronger points of each technique to balance each other weaknesses. This means that the ability to solve branch expressions from symbolic execution is used to reduce the number of executions needed for dynamic analysis. While dynamic analysis can provide values for referenced elements like objects, pointers and arrays, thus eliminating the referencing limitation of symbolic execution. This technique is known as concolic execution [Pasareanu and Visser, 2009]. The name is a contraction of concrete and symbolic execution.

2.3.2 Constraint Solving

Constraint solving is the process of attempting to find a solution to a constraint system. There are some constraint solving techniques that are used inside a TCG. These techniques are used by a TCG to search for solutions to the constraint system representing either the system under test or appropriate inputs to the system under test. Solutions to the system normally guide the TCG towards a desired adequacy criteria. The most relevant technique is known as satisfiability solving [Dixon et al., 2011]. An approach to solving a satisfiability system is conflict-driven [Dixon et al., 2011; Prasad et al., 2005]. This approach searches for a contradiction of the formula to satisfy and if no contradicting example is found, then the formula must be true. Another approach to solving a satisfiability system is based on stochastic local search methods [Prasad et al., 2005], but they do not perform well on structured instances obtained from real verification problems.

2.3.3 Invariant Detection

The previously discussed constraint inference techniques allow for the construction of constraint systems. These techniques have both direct and indirect applications to TCG. Direct applications include the construction of a constraint system that represents the system under test. This constraint system then can be solved to provide guidance towards given adequacy criteria. Indirect applications of constraint inference include invariant detection, a technique that attempts to infer

function invariants. There it more than one way to accomplish this, be it through specification mining [Ammons et al., 2002; Ernst et al., 2007] or static observation of source code [Ramanathan et al., 2007].

Specification mining attempts to dynamically infer program specifications from execution traces of the program [Ammons et al., 2002]. DAIKON [Ernst et al., 2007] is a tool that implements specification mining. It takes as input executions provided by a test suite and it observes properties that were kept along the execution. This leads to a very interesting approach to testing, since test cases can be created from specifications and specifications can be inferred from the execution traces provided by test cases. This creates a feedback loop [Xie and Notkin, 2003], where tests can be generated from specifications and specifications can be refined from tests. However, not all test cases contribute in the same way to this process [Dallmeier et al., 2010]. The more complete the test cases the more reliable the inferred properties will be.

There are also some applications of invariant detection using static analysis [Ramanathan et al., 2007]. By leveraging data flow analysis to gather predicates for each program point, it is possible to derive valid properties that should be valid before and after the execution of a method.

Nonetheless, automatically inferred specifications are not completely trustworthy. Studies reported in [Polikarpova et al., 2009] were made to compare programmer-written and automatically inferred invariants. These studies conclude that automatically inferred contracts do not always cover as much of the program behaviour as the ones manually defined by programmers. In some cases, automatically inferred contracts encounter unspecified or unexpected program behaviour that the developer missed. However, these studies suggest that the automatically inferred contracts could be used either to complete programmer-written contracts or to serve as a baseline for them.

2.3.4 Input Classification

Other relevant techniques that can aid in the process in testing, especially in test data generation, are known as input classification techniques [Carbin and Rinard, 2010; Pacheco and Ernst, 2005]. These techniques consist in classifying inputs based on their potential to reveal faults in the target software.

A possible approach to input classification is to use a model of the operation of the system under test to generate random inputs and then executing the target software with the generated input, checking its behaviour according to the previously extracted model of operation of the software. If the execution behaves incorrectly then it is possible that either an illegal input was found or a possible fault in the software was detected. This approach has been implemented in ECLAT [Pacheco and Ernst, 2005], a tool that generates fault revealing test cases, given an example execution of the system under test.

2.4 Challenges

Software testing and TCG remain a challenging field in software sciences. There are four major trends in software testing: Universal Test Theory, Test-Based Modelling, Fully Automated Testing and Efficacy-maximised Test Engineering [Bertolino, 2007]. To complete this chapter we discuss some of the most relevant challenges remaining in such trends.

2.4.1 Input Generation

The problem in input generation is the difficulty of generating appropriate inputs, i.e. that meet the entry condition for a given method. Some believe that a minimalist set of good program inputs covering a significant amount of branches can reveal the same faults as a huge test data set would [Carbin and Rinard, 2010; Pacheco and Ernst, 2005; Milicevic et al., 2007]. Also the automatic generation of test data is seen as an important step to achieve fully automated testing.

The areas expected to produce considerable advances in input generation are model-based approaches [Bruno et al., 1995], guided applications of random data generators [Pacheco and Ernst, 2007], and a wide variety of search-based techniques [Maragathavalli, 2011]. From model-based approaches it is possible to to use models as a baseline to derive complete test cases [Bertolino, 2007]. This approach could be used in conjunction with concolic execution to guide the execution process and thus traversing the most relevant paths to validate the model. Model-based approaches remain the most promising in their allowing developers to validate their implementation against application models defined by the stakeholders, assuming the model is correct.

Random generation techniques have been put aside in favour of other, more systematic, techniques that were deemed more comprehensive and capable of achieving much better results, like model-based testing and bounded exhaustive testing. Recently, however, random generation have been compared to such systematic techniques and shown to achieve almost identical results with much less effort [Ciupa et al., 2007]. For this reason, researchers currently believe that it is possible to somehow enhance random generation techniques by making them smarter [Pacheco et al., 2007], by leveraging previous executions to better predict the input bound.

2.4.2 Test Oracles

Another challenge in software testing and TCG in particular is the availability of test oracles [Staats et al., 2011; Bertolino, 2007]. Recall that the part testing that decides whether test cases succeed or fail is the oracle. There are several ways to produce test oracles. The most reliable ones involve manual interaction from the developer and even these are mostly incomplete, resulting in test oracles that do not cover all the behaviour they should. The current challenges is to improve current means of producing reliable test oracles, and of course to take test automation a step further by fully automating oracle generation. This is a challenge because there is no way of knowing what to expect from a method with incomplete information. It is only possible to capture the current result and assert that it is always the same. Even then it is necessary to convert observed behaviour into expressions to be asserted. Model-based testing and executable specification languages like JML are regarded as the most promising techniques capable of solving this challenge [Baresi and Young, 2001; Bertolino, 2007].

2.4.3 General Challenges

The so-called general challenges in software testing go beyond the technical aspects of the testing process. General challenges include enrichment to the whole theory behind software testing, like the assessment of the true effectiveness of testing and also how can the results of a test case could influence another test case, or even how the order of which test cases are executed influences the testing process. Also it is very important to know the real costs of testing and the real costs of not testing software applications.

Test effectiveness is related to adequacy criteria. It is considered of vital importance to determine which adequacy criteria or which combination of criteria is best in order to detect likely faults while minimising the costs and effort of testing at the same time. Currently there is no single criterion that stands out from the others, the general consensus being that a combination of criteria is considered best [Bertolino, 2007].

Test composition relates to the order by which test cases are executed. When a given test suite gets too big to run in a single execution, usually the approach is to apply the old divide and conquer - *divide et impera* - strategy [Bertolino, 2007]. However, it has been suggested that by properly ordering test cases it is possible to minimise the effort to run the test suite. Basically the results from previously executed test cases can be used to reduce the effort in executing the whole test suite.

Another interesting challenge in software testing is the inference of *Test Patterns*. Such patterns would be analogous to the well known *Design Patterns* [Gamma et al., 1995] for software. The purpose of these patterns would be to help in detecting the best strategy to thoroughly test a given component or system, thus allowing testing to go faster and simpler. It would also provide for reusable solutions to common problems.

Current state of the art cost evaluation of software testing is only based on estimations. What is also estimated is the cost of poorly testing or not testing applications [Tassey, 2002]. Knowing the real costs and risks of whether testing and not testing software is vital for the future of the software testing discipline as it will allow the assessment between cost and risk for each component and prioritise testing effort.

Finally, a challenge that will always prevail is the proper education of software testers [Bertolino, 2007]. Even with a wide array of tools and techniques available the human eyeball should always have the final say in testing, and good educators in the software testing will make them more aware of common problems and how to check for them.

Chapter 3

Tool Review

The motivation behind the tool review reported in this chapter is to find out what tools are available that either preform TCG, or aid in a testing environment. This chapter describes the review process, the tools reviewed and their range of application. Finally this chapter presents some considerations about the tools.

3.1 Review Process

To learn more about TCG methods some tools were reviewed and analysed under a given set of criteria in order to find out their usefulness in a software development environment. The evaluation criteria used throughout the review are presented in Table 3.1 as well as a summary of what each criterion means and what values it might take throughout the review.

The tools reviewed were found throughout the search for literature in the field, through searches in known search engines and discussion with peers. The tools are mostly focused around the Java programming language, with few a exceptions.

3.2 Reviewed Tools

3.2.1 Test Case generation tools

This section describes the most promising tools for TCG.

RANDOOP

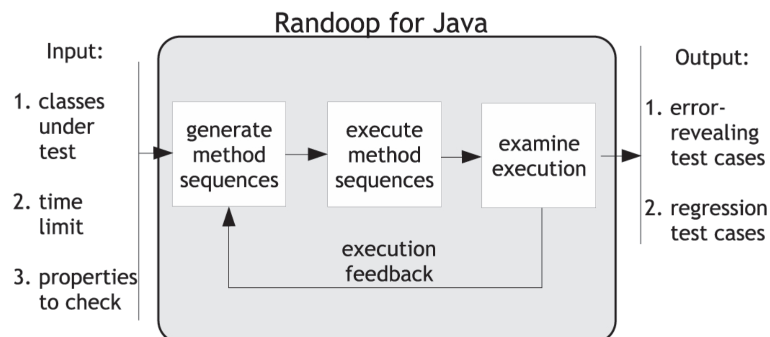


Figure 3.1: How RANDOOP works [Pacheco and Ernst, 2007].

Language	Indicates the language for which the tool is intended.
Price	Indicates the price or licensing limitations of the tool.
Level of Tests	Indicates the level of tests in which the tool operates, it may take the values Unit, Integration or System testing.
Scope	Indicates the type of the tool, framework, test case generator, etc.
Adequacy Criterion	Indicates the adequacy criteria that the tools attempts to meet.
Incremental	Indicates if the tool leverages any existing test suite.
Previously Generated Tests	Indicates the integration of the tool with previous generated tests.
Integration With Manual Tests	Indicates the level of integration between the tool and manually written tests.
Integration With Development Tools	Indicates the integration of the tool with other development tools.
Code or Binary Level	Indicates the level where the tool operates.
Configuration Effort	Indicates the level of effort required to use the tool: low, medium, high.
Quality of Produced Tests	Describes the quality of the generated tests.

Table 3.1: Evaluation criteria for the case study.

RANDOOP was developed by Carlos Pacheco and Michael Ernst [Pacheco and Ernst, 2007; Pacheco, 2009]. This tool is capable of generating regression testing and contract violation tests, with test inputs and oracles, for Java and .NET[1] software. The underlying technique in RANDOOP is *feedback-directed* random TCG [Pacheco et al., 2007]. To put it simply, RANDOOP generates sequences of method and constructor invocations. These sequences are built incrementally by randomly selecting which method to apply next to an already existing sequence. Once the sequence is built it is executed in order to determine if it is redundant, illegal, contract-violating, or useful for generating more inputs. When a method invocation requires input parameters, RANDOOP randomly selects suitable inputs from a pool of previous invocations or from a pre-defined set of values. Oracles are generated by observing the object state of the software under test and generating proper assertions that check that object state. Figure 3.1 [Pacheco and Ernst, 2007] attempts to illustrate this behaviour. RANDOOP has been validated using a considerable number of targets [Pacheco, 2009]. It showed potential as a reliable tool in generating and finding regression errors and contract violations. Do note that the figure presented may be misleading. Even though there are plans to allow user-defined contracts in the future, currently RANDOOP only checks for a handful of hard-coded contracts, as listed below:

- Equals to null: *o.equals(null) should return false*;
- Reflexivity of equality: *o.equals(o) should return true*;
- Symmetry of equality: *o1.equals(o2) implies o2.equals(o1)*;
- Equals-hashcode: *If o1.equals(o2)==true, then o1.hashCode() == o2.hashCode()*
- No null pointer exceptions: *No NullPointerException is thrown if no null inputs are used in a test.*

QUICKCHECK for Java

QUICKCHECK was originally a combinator library for the Haskell programming language [Claessen and Hughes, 2000]. It was later implemented for several languages like Java [Open Source Contributors, 2011b,a], Erlang [Hughes and Arts., 2011], Perl [Moertel, 2011], Ruby [Yeh, 2011] and JavaScript [Thompson, 2011] among others. These tools are all about test data generation. By generating high amounts of data and checking it against a given property, it is expected to cover a wide range of the input domain, thus increasing the changes of finding more faults [Hughes, 2009]. While there is no validation for the Java implementation of QUICKCHECK, there is a report on the effectiveness of QUICKCHECK for Haskell [Claessen and Hughes, 2000].

QUICKCHECK for Java [Open Source Contributors, 2011a] is one implementation of QUICKCHECK for the Java language that allows for the creation of PUT cases. Recall that PUT is to run the same test case with different input values. This implementation includes several statistical distributions allowing users to better guide data generation.

If applied like a third party framework, QUICKCHECK requires test cases to explicitly call its features. In some occasions it might be necessary to implement a custom data generator, which QUICKCHECK allows and provides an extensive library for. Listing 3.1 illustrates an example implementation of a generator. In this implementation a Stack instance is created with a random size between 1 and 100, and then filled with random integers.

[1] This requires a different version of the tool.

Language	Java and .NET
Price	Free under the MIT licence (see MIT [2011])
Level of Tests	Unit level
Scope	Test case generator
Adequacy Criterion	Fault detection
Incremental	Not influenced by existing test cases
Previously Generated Tests	Overwrites or complements previously generated test cases
Integration With Manual Tests	Possible to co-exist
Integration With Development Tools	Comes as an Eclipse IDE plugin and a command line tool. There is however, no integration with Apache Ant, Apache Maven and Netbeans
Code or Binary Level	Works at the binary level
Configuration Effort	Little effort in using the Eclipse plugin. The standalone command line tool requires a proper setup of the class path variable
Quality of Produced Tests	Produces complete test cases, with inputs and test oracles. Some of the produced tests seem to contain problems that prevent the test classes from compiling and therefore running.

Table 3.2: Summary for RANDOOP.

```java
public class StackGenerator implements Generator<FastStack<Integer>> {
  public FastStack<Integer> next() {
    Random rand = new Random();
    int size = 1 + rand.nextInt(100);
    FastStack<Integer> fs = FastStack.newInstance();

    for (int i = 0; i < size; i++) {
      fs.push(rand.nextInt());
    }
    return fs;
  }
}
```

Listing 3.1: Implementation of a random Stack generator

Implemented generators can then be applied within a test case as illustrated in Listing 3.2. In this test case, the generated Stack instance is tested through some common operations and checked for conformity. Then the Stack instance is regenerated and the process starts all over again.

```
@Test
public void testNormal() {
  FastStack<Integer> stack;
  for (int i = 0; i < NUMBER_OF_RUNS; i++) {
    stack = sg.next();

    int currentTop = stack.peek();
    int currentSize = stack.size();
    final int num = 100;

    for (int j = 0; j < num; j++) {
        stack.push(j);
      assertEquals(j, stack.peek());
    }

    assertEquals(currentSize + num, stack.size());
    assertEquals(1, stack.peek(currentSize + 1).intValue());

    for (int j = 0; j < num; j++) {
        stack.pop();
    }

    assertEquals(currentTop, stack.peek().intValue());
    assertEquals(currentSize, stack.size());
  }
}
```

Listing 3.2: Application of the Stack generator in a test case

JCheck

JCheck [Open Source Contributors, 2011b] is another Java implementation of the previously mentioned QuickCheck combinator library [Claessen and Hughes, 2000]. This implementation however presents a different approach to how the test cases are written. Instead of adding a loop to iterate over the randomly generated values, JCheck features its own test runner, which is compatible with the JUnit framework. This allows test cases to receive input parameters and run the test case with the received parameters. This implementation of QuickCheck lacks many of the data generation features present in QuickCheck for Java.

Much like QuickCheck for Java, to apply JCheck the test cases must include its functionalities. However, there is little difference while implementing a data generator, the difference being that the random value generator and the scope size are already passed on to the generator as parameters. A PUT written with JCheck is illustrated in Listing 3.3. In this case, the generator is specified through an annotation provided by JCheck. Notice the lack of a loop statement to refresh the instance under test, this being accomplished by the runner that comes packaged with JCheck.

Language	Java
Price	Free under the Apache 2.0 licence (see ASF [2011])
Level of Tests	Unit level
Scope	Parameterised test case framework
Adequacy Criterion	Input domain coverage
Incremental	Not influenced by existing test cases
Previously Generated Tests	Impossible to replay previous instantiated tests
Integration With Manual Tests	Possible to co-exist
Integration With Development Tools	Only available as a third party library
Code or Binary Level	Works at the source level
Configuration Effort	To use the library it is only required to add it to the build path and use its features
Quality of Produced Tests	Does not produce the code for the test case, it only instantiates a test case with different input values

Table 3.3: Summary for QUICKCHECK.

```
@Test
@Generator(klass = FastStack.class, generator = StackGenerator.class)
public void testNormal(FastStack<Integer> stack) {
  int currentTop = stack.peek();
  int currentSize = stack.size();

  final int num = 100;
  for (int i = 0; i < num; i++) {
      stack.push(i);
  }

  assertEquals(currentSize + num, stack.size());
  assertEquals(num - 1, stack.peek().intValue());

  for (int i = 0; i < num; i++) {
    stack.pop();
  }

  assertEquals(currentTop, stack.peek().intValue());
  assertEquals(currentSize, stack.size());
}
```

Listing 3.3: Application of the Stack generator in a parameterised test case

Language	Java
Price	Free under the CPL licence (see IBM [2011])
Level of Tests	Unit level
Scope	Parameterised test case framework
Adequacy Criterion	Input domain coverage
Incremental	Not influenced by existing test cases
Previously Generated Tests	Impossible to instantiate previous test cases
Integration With Manual Tests	Possible to co-exist
Integration With Development Tools	Only available as a third party library
Code or Binary Level	Works at the source level
Configuration Effort	To use the library it is only required to add it to the build path and use its features
Quality of Produced Tests	Does not produce the code for the test case, it only instantiates a test case with different input values

Table 3.4: Summary for JCheck.

Feed4JUnit

Feed4JUnit is another PUT generation framework. It is developed and maintained by Volker Bergmann [Volker Bergmann, 2011]. This tool allows users to specify an external `*.csv` file to feed input data and expect values to a given test case. This is especially useful since anyone from software analysts to developers can build a complete `*.csv` file with all the important input values to be tested. It also allows developers to produce equivalence class partition test cases, this meaning that it is possible to define strict input sets for each test case. Feed4JUnit is capable of exploring those input sets by following a combinatorial strategy. Consider the example in Listing 3.4, in which two sets for each input parameter are defined, Feed4JUnit would combine the sets and test them thoroughly.

There is little difference between tests written with this framework and the ones written with the previously described tools, QuickCheck for Java and JCheck. The differences in implementing custom generators and writing test cases are mostly aesthetic.

T2 Framework

T2 is a TCG framework developed by Wishnu Prasetya and collaborators [Prasetya et al., 2008]. This tool is a trace-based random test generation tool, similar to Randoop. However, there is a difference: T2 allows users to provide in-code specifications through the `assert` keyword of the Java programming language. Furthermore, this tool integrates with JUnit. Downside of this tool, is the fact that it can not test non-deterministic behaviour. For instance, classes that make use of randomly generated values or multi-threaded behaviour can not be tested with T2. A study of the effectiveness of T2 [Prasetya, 2011], reported that T2 can reduce the cost of testing software. Its speed and achieved coverage are its stronger points.

```
@RunWith(Feeder.class)
public class AddTest {
  @Test
  @Unique
  public void testAdd(
      @Values("-2147483648,-6,-1,0,1,42,2147483647") int param1,
      @Values("-2147483648,-6,-1,0,1,42,2147483647") int param2) {
    try {
      int result = MyUtil.add(param1, param2);
    } catch (Exception e) {
      // accept application exceptions, fail on runtime exceptions
      // like NullPointerException
      if (e instanceof RuntimeException)
        throw e;
    }
  }
}
```

Listing 3.4: Example of a Feed4JUnit equivalence partition test.

Language	Java
Price	Free under GPLv2 (see GNU [2011]) but depends in other components that are not entirely GPL (see Volker Bergmann 2011 for more details)
Level of Tests	Unit level
Scope	Parameterised test case framework
Adequacy Criterion	Input domain coverage
Incremental	Not influenced by existing test cases
Previously Generated Tests	Impossible to instantiate previous test cases, however it is possible to check which inputs were passed on to the test case
Integration With Manual Tests	Possible to co-exist
Integration With Development Tools	Only available as a third party library
Code or Binary Level	Works at the source level
Configuration Effort	To use the library it is only required to add it to the build path and use its features
Quality of Produced Tests	Does not produce the code for the test case, it only instantiates a test case with different input values

Table 3.5: Summary for FEED4JUNIT.

To apply T2, it is recommended that the source code should contain some specifications, otherwise the test process will only check for run-time exceptions. The specifications should be introduced through the `assert` keyword, either in the desired method or in a separate method with the same name appended with a `_spec` suffix, as illustrated in Listing 3.5. To invoke the T2 runner there are two possible ways. One way is to invoke the runner through the command line. However, it is also possible, and recommended, to invoke T2 within a JUnit test case. Running T2 from a test case allows other tools to measure the structural coverage that T2 managed to achieve. Listing 3.6 illustrates how T2 can be invoked from JUnit.

```
public class FastStack<E> implements Iterable<E> {
  private final List<E> mEntries = Lists.newArrayList();
  ...
  public void push(E aElement) {
    mEntries.add(aElement);
  }

  public void push_spec(E newElem) {
    assert newElem != null : ''PRE'';

    int size = this.size();
    this.push(newElem);

    assert (!this.isEmpty()) : ''POST'';
    assert (this.contains(newElem)) : ''POST'';
    assert (this.size() == size + 1) : ''POST'';
  }
  ...
}
```

Listing 3.5: Example of T2 specifications.

```
@Test
public void testClass() {
  Sequenic.T2.Main.Junit(FastStack.class.getName()
    + ''--nmax=2000 --lenexec=5 --violmax=8 --elemty=java.lang.Integer'');
}
```

Listing 3.6: Example of a test case invoking T2.

JMLUNITNG

JMLUNITNG [Zimmerman and Nagmoti, 2011; Zimmerman, 2011], is a TCG tool for JML-annotated source code. The tool is developed and maintained by Daniel Zimmerman. It aims to support the most recent features of the Java language, like generics and enumerated types. It leverages JML specifications to generate test oracles, it also attempts to use reflection mechanisms to generate test data, and also provides users the possibility to introduce their own input values. Currently JMLUNITNG relies on the TestNG framework for running the generated test cases and a compiler of the JML language (either `jmlc` or `jml4c`). Additionally, JMLUNITNG also depends on the next major version of the JDK, OpenJDK 7, which is, at the time of writing, under development.

Language	Java
Price	Free under GPLv3 licence (see GNU [2011])
Level of Tests	Unit level
Scope	Test case generator
Adequacy Criterion	Specification validation / input domain coverage
Incremental	Not influenced by existing test cases
Previously Generated Tests	Previously generated tests can be re-run, with the aim of fixing bugs
Integration With Manual Tests	Possible to co-exist
Integration With Development Tools	Only available as a third party library
Code or Binary Level	Works at the source level
Configuration Effort	To use the library it is only required to add it to the build path and use its features. Running it from the command line the class path needs to be properly configured. The specifications need to be added by other means
Quality of Produced Tests	Does not produce the code for the test cases, it only instantiates a test case with different input values

Table 3.6: Summary for T2.

The first step in applying this tool is to properly annotate the source code with JML specifications, whereafter the classes need to be compiled with a special compiler: `jmlc` if the source code does not require newer Java features or with `jml4c` if it requires newer features, like generic types. Then JMLUnitNG needs to run on the classes under test with `java -jar jmlunitng.jar path-list` command. This command will generate the required java classes which need to be compiled and executed with OpenJDK 7. Before compiling the generated classes, they can be edited to provide input values.

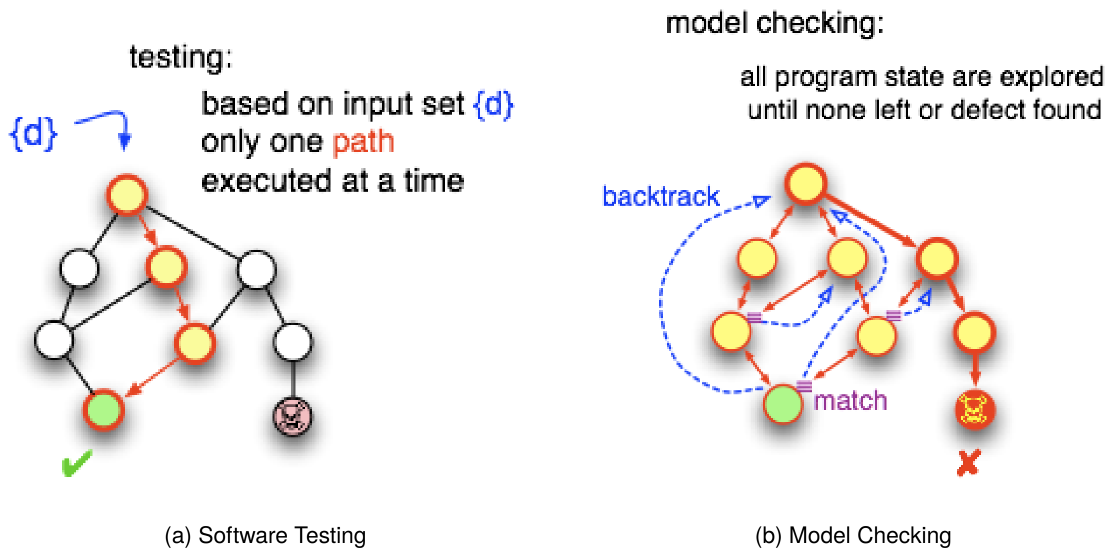
Language	JML annotated Java
Price	Free, not open source
Level of Tests	Unit level
Scope	Test case generator
Adequacy Criterion	Specification validation / input domain coverage
Incremental	Not influenced by existing test cases
Previously Generated Tests	Only generates test oracle, test cases can be added by extending the input set
Integration With Manual Tests	Possible to co-exist
Integration With Development Tools	Only available as a third party tool, there is no integration with any other development tool
Code or Binary Level	Works at the source level
Configuration Effort	This tool requires OpenJDK 7 installed and set up as the main JVM. The JML compilers, `jmlc` and `jml4c`, need to be set up, and lastly the class path variable needs to be set up to include JMLuniNG
Quality of Produced Tests	Does not produce the code for the test cases, it only instantiates a test case with different input values

Table 3.7: Summary for JMLUnitNG.

Java Pathfinder

Java Pathfinder is a tool developed by researchers at the NASA Ames Research Centre [Visser and Mehlitz, 2005; NASA Ames Research Center, 2011]. This tool is more a model checker for Java bytecode than a real TCG. This however does not mean that it can not be used to test software. In fact the type of model checking that this tool performs is commonly mistaken with systematic testing. The truth is that Java Pathfinder does a little more than traditional TCG tools. Tests generated with traditional tools only cover one execution path at a time, and some times some faults may go on undetected, while on the other hand model checking systematically checks all paths. This increases the chances of finding more faults (see Figure 3.2).

Java Pathfinder is a virtual machine on top of the regular JVM. This new virtual machine adds some model checking traits like backtracking and state matching among others to the original JVM.



(a) Software Testing

(b) Model Checking

Figure 3.2: Software Testing vs. Model Checking [NASA Ames Research Center, 2011]

JAVA PATHFINDER was successfully applied to check random algorithms [Zhang and van Breugel, 2010] and network protocols [Martínez and Jiménez, 2008] with success. There are, however, some limitations, for example it is only meant for non-deterministic software as quoted from the project page [NASA Ames Research Center, 2011]:

> *A word of caution: if you have a strictly sequential program with only a few well defined input values, you are probably better off writing a few tests - using JPF won't tell you much.*

PATHFINDER is also unable to check Java Native Interface methods, this is, JAVA PATHFINDER can not test most `I/O` operations that the language supports and in which most software currently build on. Out-of-the-box, JAVA PATHFINDER checks the following properties, leaving the possiblity to implement new properties:

Deadlocks gov.nasa.jpf.jvm.NotDeadlockedProperty - for every non-end state, test if there is any runnable thread left;

Assertion Violation gov.nasa.jpf.jvm.NoAssertionViolatedProperty - test if any assertion expression has been violated;

Uncaught Exceptions gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty - test if any exception was not handled inside the software.

To apply JAVA PATHFINDER to a software under test, it first needs to to be configured to define the intended software properties. Listing 3.7 illustrates a sample configuration, the `target` property defining the class where the software entry point is located, `DiningPhil` in the example. The `search.class` property defines the search strategy to be applied in the process, a breath-first approach in the example. Listing 3.8 illustrates the class that will be targeted. This class consists of the

traditional dinning philosophers problem, in which there are five philosophers and five forks. A philosopher is either thinking or eating, when a philosopher wants to eat it needs to take both the fork to its left and to its right. To start the JAVA PATHFINDER process the user just needs to invoke `jpf <path-to-jpf-configuration-file>` from the command line.

```
target = DiningPhil

search.class = .search.heuristic.BFSHeuristic
```

Listing 3.7: Example JPF configuration.

Language	Java
Price	Free under the NOSA licence (see NASA [2011])
Level of Tests	System level
Scope	Java bytecode model checker
Adequacy Criterion	Property validation / structural coverage
Incremental	Not influenced by existing test cases
Previously Generated Tests	Does note generate test cases
Integration With Manual Tests	Possible to co-exist
Integration With Development Tools	Plugins available for Eclipse and Netbeans, there is currently no support for Ant or Maven
Code or Binary Level	Works at the bytecode level
Configuration Effort	Configuration depends on each system system under test
Quality of Produced Tests	Does not produce test cases

Table 3.8: Summary for JAVA PATHFINDER.

UNITCHECK

UNITCHECK is a plugin a tool that was developed by Michal Kebrt [Kebrt and Sery, 2009]. This tool means to apply model checking techniques, through JAVA PATHFINDER to unit test cases, aiming to bring high quality model checking to the general software development process. UNITCHECK recognises JUnit tests and exhaustively explores their reachable state space including all admissible thread interleaving. This tool was used to check the behaviour of two test classes that lean on threaded behaviour to reveal faults [Kebrt and Sery, 2009].

UDITA

UDITA [Gligoric et al., 2010] is an extension to the standard Java language, that allows its users to describe their test cases. Relying in a modified version of JAVA PATHFINDER tool. This tool allows

```java
import gov.nasa.jpf.jvm.Verify;

public class DiningPhil {

  static class Fork {}

  static class Philosopher extends Thread {
    Fork left;
    Fork right;

    public Philosopher(Fork left, Fork right) {
      this.left = left;
      this.right = right;
      start();
    }

    public void run() {
      // think!
      synchronized (left) {
        synchronized (right) {
          // eat!
        }}}
  }

  static final int N = 6;

  public static void main(String[] args) {
    Verify.beginAtomic();
    Fork[] forks = new Fork[N];
    for (int i = 0; i < N; i++)
      forks[i] = new Fork();
    for (int i = 0; i < N; i++)
      new Philosopher(forks[i], forks[(i + 1) % N]);
    Verify.endAtomic();
  }
}
```

Listing 3.8: The classic dining philosophers problem.

Language	Java
Price	Free, no licensing information is available
Level of Tests	Unit level
Scope	Unit level model checker
Adequacy Criterion	Property validation / structural coverage
Incremental	Not influenced by previous executions
Previously Generated Tests	Does note generate test cases
Integration With Manual Tests	Required
Integration With Development Tools	Available as an Eclipse plugin and as an Ant Task, currently there is no support for Netbeans IDE or Maven
Code or Binary Level	Works at the binary level
Configuration Effort	Little effort in configuring the Eclipse plugin, running the Ant task however requires some understanding of Ant and how to set up the class path variable
Quality of Produced Tests	Does not produce test cases

Table 3.9: Summary for UNITCHECK.

users to describe either test oracles or the input domain for the test cases. The tool will generate input values or appropriate test cases to verify the correction of the oracles. These inputs could be used at either system testing level or at the unit testing level. It has been evaluated on the JAVA PATHFINDER and Eclipse IDE to reveal some flaws in both projects [Gligoric et al., 2010].

ECLAT

ECLAT is very similar to RANDOOP and has been developed by Carlos Pacheco and Michael Ernst [Pacheco and Ernst, 2005]. The major difference is that ECLAT integrates with DAIKON in order to produce an operational model of the system under test. This model is then used to generate test cases, by filtering out redundant test cases and improving the ability to check for faults. The developer can then check the failing test cases and decide if the test case is really revealing a fault of the software or not.

The tutorial available at [Pacheco and Ernst, 2005] provides an easy to understand example of usage. The basic usage is to tell ECLAT which classes should it generate test cases for and an example usage of those classes, for example existing test cases over those classes. A typical command line would be `java eclat.textui.Main generate-inputs --create-regression-suite --test ubs/BoundedStack.java ubs.BoundedStackJunitTest` which will generate a test suite. The resulting test suite can be found in a folder named `eclat-src`.

PEX

PEX is a tool developed by Nikolai Tillmann and Peli de Halleux and other staff at Microsoft Research labs [Tillmann and de Halleux, 2008]. PEX leverages concolic execution to execute pro-

Language	Java
Price	Free under the NCSA licence (see University of Illinois [2011])
Level of Tests	System / unit level
Scope	Input generator / Oracle generator
Adequacy Criterion	Input domain coverage
Incremental	Not influenced by previous executions
Previously Generated Tests	Not applicable
Integration With Manual Tests	Possible to coexist
Integration With Development Tools	Does not integrate with any software development tool
Code or Binary Level	Works at the binary level
Configuration Effort	Impossible to say, the instructions provided did not lead to a successful execution of the tool as described
Quality of Produced Tests	Does not produce test cases

Table 3.10: Summary for UDITA.

Language	Java
Price	Free, in spite of, the source code not being currently available
Level of Tests	Unit level
Scope	Test case generator
Adequacy Criterion	Fault detection
Incremental	Existing test cases improve the result
Previously Generated Tests	Can be ran to check remaining faults
Integration With Manual Tests	Possible to co-exist
Integration With Development Tools	Only available as a third party library
Code or Binary Level	Works at the source level
Configuration Effort	To use the library it is only required to add it to the build path and use its features
Quality of Produced Tests	Produces complete test cases, with inputs and test oracles.

Table 3.11: Summary ECLAT.

grams and capture their behaviour, generating a PUT and passing test inputs on to fully exercise the function under test and detect errors.

This tool is fully integrated with the Visual Studio IDE from Microsoft. After installing PEX the only thing to do is to press a single button and PEX will automatically start generating test cases.

Language	.NET
Price	Free for Visual Studio
Level of Tests	Unit level
Scope	Test case generator
Adequacy Criterion	Structural coverage / Specification validation (if combined with Code Contracts)
Incremental	Not influenced by previously generated tests
Previously Generated Tests	May remain for future notice or can be overwritten
Integration With Manual Tests	Possible to co-exist
Integration With Development Tools	Integrates with Microsoft Visual Studio, only
Code or Binary Level	Works at the binary level
Configuration Effort	Only installation is required
Quality of Produced Tests	Produces parameterised tests cases with test oracles and appropriate data generators

Table 3.12: Summary for PEX.

3.2.2 Interesting Tools for Software Testing

This section describes interesting tools in the software testing context, which are not TCG tools.

KORAT

KORAT is a tool that automatically generates structurally complex test inputs for the Java programming language. It is being developed by researchers at MIT and other universities [Milicevic et al., 2007]. Users must provide a predicate that describes a proper instantiation of the desired structure and a finitisation criterion to limit the scope. However, KORAT does not integrate with other testing frameworks like JUnit *out-of-the-box*.

To use KORAT, users must first provide at least two important methods to the target structure. These methods consist of (*a*) an imperative predicate that specifies the desired structural constraints; and (*b*) a finitisation criterion that bounds the desired test input size. An imperative predicate is a method that checks that all the desired properties about the structure are kept. It does not take any parameters and should always return a Boolean value, `true` in case the structure is valid or `false` otherwise. A finitisation is a method that tells the tool when to stop generating input values. In a way it bounds the input domain to prevent from memory exhaustion.

Consider the implementation of a binary tree, illustrated in Listing 3.9. This structure holds up to two nodes and each node can hold up to two other nodes and so on. A possible predicate over this structure would be Listing 3.10, that prevents the occurrence of cycles in the structure. The finitisation method (Listing 3.11), as previously mentioned, simply defines the size of the structure to generate. After enriching the desired structure with both these methods, KORAT can be applied. Currently, it is only available as a command line tool. To invoke KORAT on the presented examples, the following command could be used: `java korat.Korat --visualize --class korat.examples .binarytree.BinaryTree --args 5,5,5`. This command would produce some visualisations of the intended structure.

```
public class BinaryTree {
    public static class Node {
        Node left;
        Node right;
    }
    private Node root;
    private int size;
}
```

Listing 3.9: Example binary tree implementation.

```
public boolean repOK() {
    if (root == null)
        return size == 0;
    // checks that tree has no cycle
    Set visited = new HashSet();
    visited.add(root);
    LinkedList workList = new LinkedList();
    workList.add(root);
    while (!workList.isEmpty()) {
        Node current = (Node) workList.removeFirst();
        if (current.left != null) {
            if (!visited.add(current.left))
                return false;
            workList.add(current.left);
        }
        if (current.right != null) {
            if (!visited.add(current.right))
                return false;
            workList.add(current.right);
        }
    }
    // checks that size is consistent
    return (visited.size() == size);
}
```

Listing 3.10: Example binary tree imperative predicate.

So by applying KORAT to the previously illustrated binary tree it is possible to quickly generate various instances of the tree, one of them being illustrated in Figure 3.3.

```java
public static IFinitization finBinaryTree(int nodesNum, int minSize, int maxSize) {
    IFinitization f = FinitizationFactory.create(BinaryTree.class);
    IObjSet nodes = f.createObjSet(Node.class, nodesNum, true);
    f.set("root", nodes);
    f.set("Node.left", nodes);
    f.set("Node.right", nodes);
    IIntSet sizes = f.createIntSet(minSize, maxSize);
    f.set("size", sizes);
    return f;
}
```

Listing 3.11: Example binary tree finitisation.

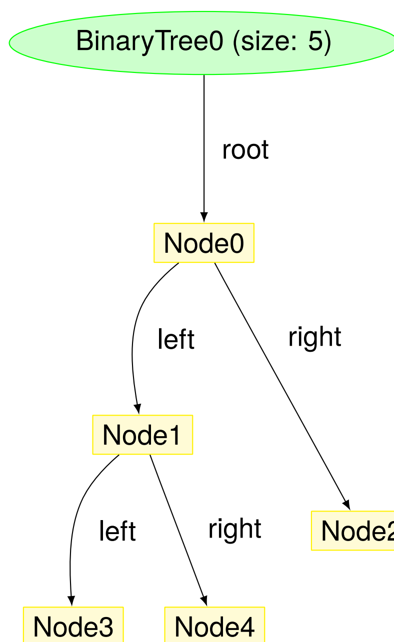


Figure 3.3: Example visualisation of a BinaryTree of size 5.

Language	Java
Price	Free under the GPLv2 licence (see GNU [2011])
Level of Tests	Not applicable
Scope	Input generator
Adequacy Criterion	Input domain coverage
Incremental	Not influenced by previously generated inputs
Previously Generated Tests	Does note generate test cases
Integration With Manual Tests	No integration
Integration With Development Tools	Does not integrate with IDEs nor other development tool
Code or Binary Level	Works at the bytecode level
Configuration Effort	KORAT only needs to be added to the class path; the imperative predicates and finitisation methods need to be written
Quality of Produced Tests	Does not produce test cases

Table 3.13: Summary for KORAT.

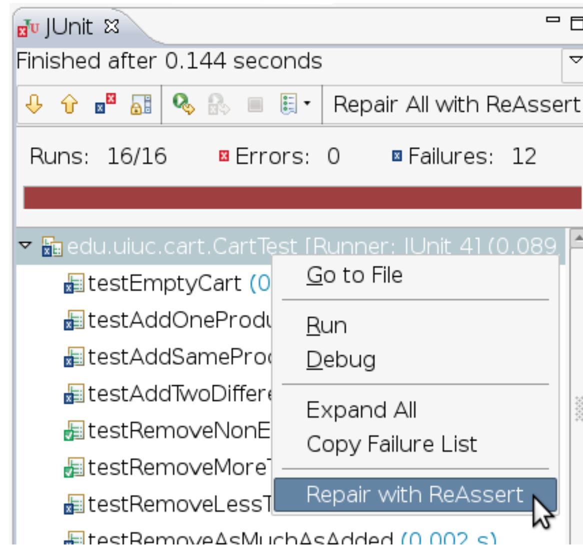
REASSERT

REASSERT is helpful as a maintenance tool. It was developed by Brett Daniel and Darko Marinov, [Daniel et al., 2011]. This tool helps its users in correcting broken tests. It must be used with caution as the tool will not know if developers have introduced a new bug into the function or just altered its behaviour to include new functionality. This tool executes the function whose tests got broken with the last revision and examines its new behaviour, then it suggests some corrections to the failing test that make it pass again, be it by changing an assertion or an input value. Figure 3.4 illustrates the typical usage of REASSERT

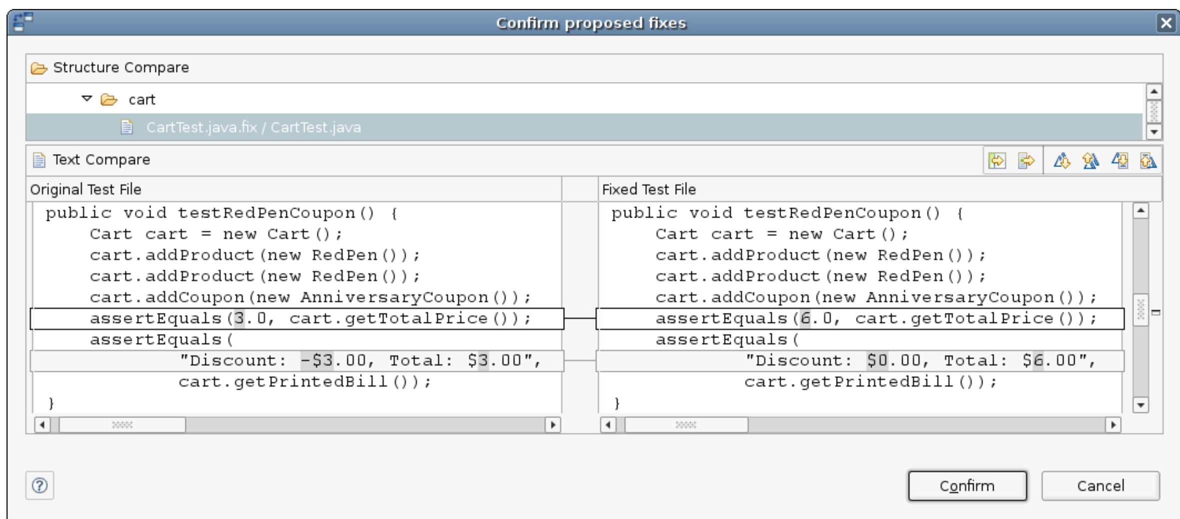
DAIKON

DAIKON is an invariant detection tool developed at MIT by Michael Ernst and collaborators [Ernst et al., 2007]. This tool is based in dynamic analysis of applications, this is, it detects likely program invariants based on values observed from actual executions of the software. Not directly related to TCG, DAIKON can be an intermediate step of it, since invariants may be used to aid the testing process, either manually or automatically. Manually, in the sense that it may help developers to better understand how a given function or method should behave. TCG tools could leverage the invariants to better guide their generation process.

To apply DAIKON, users should provide some executions over the intended software, which can be regular main methods exercising certain parts of the software, or regular JUnit tests. At the moment DAIKON is only available as a command line tool. To invoke it on an existing test suite the command `java daikon.Chicory --daikon TARGET_CLASS` should be enough. Then the tool will execute the target classes and capture the execution traces that are then used to infer the likely invariants. DAIKON is also able to annotate source code files with the detected invariants with the



(a) Select tests to repair.



(b) Confirm suggested corrections.

Figure 3.4: Applying ReAssert

Language	Java
Price	Free under the NCSA licence (see University of Illinois [2011])
Level of Tests	Unit level
Scope	Unit test correction tool
Adequacy Criterion	Not applicable
Incremental	Not applicable
Previously Generated Tests	Not applicable
Integration With Manual Tests	Possible, since it repairs test cases
Integration With Development Tools	Plugins available for Eclipse, only
Code or Binary Level	Works at the bytecode level
Configuration Effort	Only the plugin installation is required
Quality of Produced Tests	Does not produce test cases

Table 3.14: Summary for REASSERT.

following command `java daikon.tools.jtb.Annotate INVARIANT_FILE TARGET_SOURCE_FILE`.

AUTOMATIC JUNIT CREATION TOOL

AUTOMATIC JUNIT CREATION TOOL [Whitney, 2011] is a tool that allows the easy creation of JUnit test cases. This tool allows users to interact more closely with the methods under test by allowing the user to specify which inputs to pass to the method under test and what behaviour to expect. To put it simply, AUTOMATIC JUNIT CREATION TOOL is a JUnit test stub generator with the ability to interact directly with the JUnit test being generated. Once all the methods are properly set up the tool allows the user to export a complete JUnit test file. Perhaps the most interesting feature provided by the tools is the ability to automatically identify branches in the control flow of a method and allow the user to specify independent test cases for them.

This tool is only available as a standalone GUI application. In order to use the tool a user should first load a java class file using the menu-bar (File →New Test) and choose the file for which to stub out new test cases. The buttons at the bottom of the window (see Figure 3.5) can be used to cycle through the available methods and gradually build each test case, passing on inputs and defining appropriate test assertions. When all the methods have proper inputs and assertions the user may export the test cases to a Java file to be used by external tools like JUnit.

JWALK

JWALK is a tool that allows lazy systematic unit testing of Java classes. It was developed by Anthony Simons [Simons, 2007] and is currently licensed by the University of Sheffield. This tool preforms bounded exhaustive testing of any compiled java class and it tests it for conformance to a lazy specification inferred by the tool at run-time from the code.

Language	Various
Price	Free under the MIT licence (see MIT [2011])
Level of Tests	Not applicable
Scope	Invariant Detector
Adequacy Criterion	Not applicable
Incremental	The more executions available, the more trustworthy the invariants are
Previously Generated Tests	Not applicable
Integration With Manual Tests	Possible to leverage unit tests as executions
Integration With Development Tools	Does not integrate with any IDE or software developing tool
Code or Binary Level	Works at the binary level
Configuration Effort	Requires proper set up of the class path variable, whereafter it is fairly simple to use
Quality of Produced Tests	Does not produce test cases

Table 3.15: Summary for Daikon.

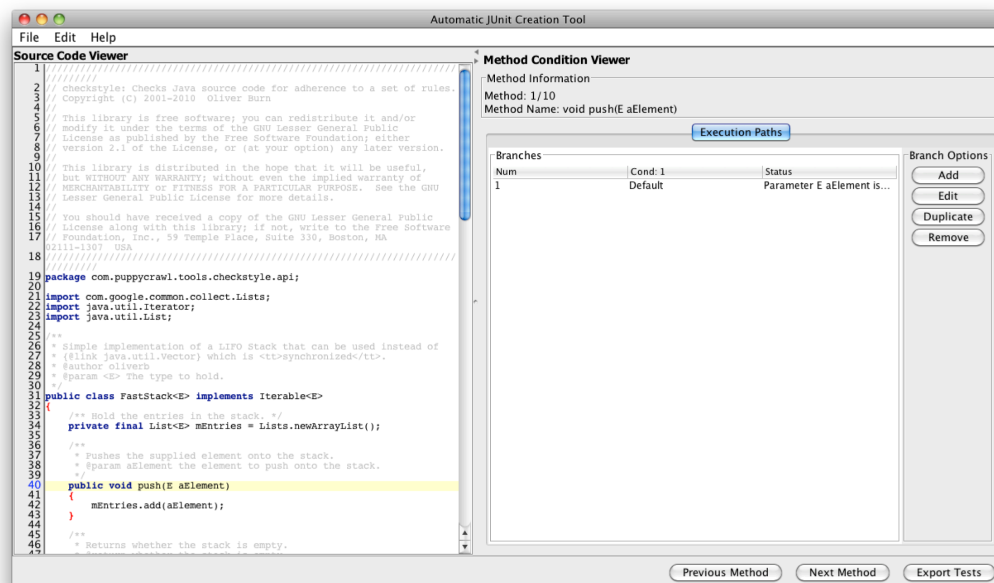


Figure 3.5: Automatic JUnit Creation Tool main window.

Language	Java
Price	Free under the GPLv3 licence (see GNU [2011])
Level of Tests	Unit level
Scope	Test case stub generator
Adequacy Criterion	Branch coverage
Incremental	Not influenced by existing test cases
Previously Generated Tests	Can not be opened by the tool again
Integration With Manual Tests	This tool facilitates writing of test cases
Integration With Development Tools	Only available as a standalone application
Code or Binary Level	Works at the source level
Configuration Effort	No configuration effort
Quality of Produced Tests	Produces complete test cases, with inputs and assertions

Table 3.16: Summary for AUTOMATIC JUNIT CREATION TOOL.

JWALK is available as a Java GUI application where the user is allowed to set the parameters for the test process. One of three strategies can be chosen: `protocol`, `algebra` and `states`. And one of three modalities: `inspect`, `explore` and `validate` and lastly the depth of the test process. The `protocol` strategy creates test sequences corresponding to every possible interleaved ordering of the public constructors and methods in the class under test. The `algebra` strategy creates sequences that drive the test object into different concrete states. The last strategy, `states`, creates sequences to cover most machine states of the class.

The `inspect` modality preforms a static analysis of the class to reveal the public API of the the test class or the state space of the class. The `explore` modality preforms a similar analysis, but dynamically instead. The `validate` modality creates and executes test sequences and validates the outcome of the sequence at hand. The test depth defines the bound the algorithms should use for bounded exhaustive testing. It turns out that this tool does not export any usable JUnit files that can be used by other tools.

3.3 A Case Study - Checkstyle

This section starts by presenting, Checkstyle, a piece of software chosen as target for the application of the testing tools described thus far. In this way we intend to “test such testing tools” checking how they behave and drawing conclusions on their usefulness.

3.3.1 Subject of Study: Checkstyle

To preform our study, Checkstyle was chosen as the target of the tools under review. Checkstyle was developed by Oliver Burn [Burn, 2011] and other freelance contributors. The purpose of this tool is to help developers in writing Java code that adheres to a given coding standard. It can

Language	Java
Price	Free for academic uses, not open source
Level of Tests	Unit level
Scope	Test case regeneration tool
Adequacy Criterion	Bounded exhaustive testing
Incremental	Previously generated test oracles can be used in generation of new tests
Previously Generated Tests	Stored as a binary file
Integration With Manual Tests	Possible to co-exist
Integration With Development Tools	Only available as a GUI application
Code or Binary Level	Works at the binary level
Configuration Effort	JWalk requires a licence under which the jar file executes
Quality of Produced Tests	Not producing the code for the test cases, it only produces a binary file that the tool uses to regenerate tests

Table 3.17: Summary for JWalk.

be configured to check virtually every coding style, but it defaults to the standard Java coding conventions [Sun Microsystems, Inc., 1999] that comes packaged with the tool. Other reasons for the success of Checkstyle, are easy the integration with other Java development tools like the Eclipse IDE and the Apache Maven project manager. It is important to note that Checkstyle performs several `I/O` operations, in reading: *a*) Checkstyle configuration files; *b*) Desired coding styles; and *c*) source files to check. There is also the printing of a report that can be analysed by the users of the tool, and information about the process that is sent to the default system logger.

This application was chosen for our case study because it is a common Java application, and for its history in software engineering research [Cornelissen and Moonen, 2007; Zaidman et al., 2007, 2008; Lubsen et al., 2009]. Before digging in to the case study itself, some information about the application was collected, especially information about its source code. First, an overview of Checkstyle has shown that the tool is composed of 22 packages and over 300 classes (Table 3.18). The first line of Table 3.18 presents the total metrics for the project while the other lines provide an average value for each level. There is also an existing test suite for Checkstyle, which allows some comparisons to be made.

Due to time limitations, it was decided to conduct this study in a smaller, but also relevant, part of the tool. In order to locate which part of the application should be used while evaluating TCG tools, a Fan-In/Fan-Out table of the packages of Checkstyle was derived (Table 3.20). The most relevant package was found to be `com.puppycrawl.tools.checkstyle.api`, due to the number of packages that depend on it. The existing test cases for the package cover about 3.5 % of the whole application as detailed in Table 3.19.

	Lines of Javadoc	Non Comment Source Statements	Methods	Classes	Packages
Project	1,674	20,781	2,240	327	22
Package	76	945	102	15	
Class	5	64	7		
Method	1	9			

Table 3.18: Information collected from Checkstyle.

Element	Coverage (%)	Covered Instructions	Missed Instructions	Total Instructions
Target Package	73.2	3,896	1,430	5,326
src/checkstyle	81.3	36,791	8,480	45,271
target/generated-sources/antlr	83.5	20,348	4,019	24,367
Totals	31.4	57,139	12,499	69,638
Average	43.4	31.4%	68.6%	100.00%

Table 3.19: Coverage results achieved by the manual test cases.

3.3.2 Tools Tested

The TCG tools that were successfully applied to the subject of our study, `com.puppycrawl.tools.checkstyle.api`, were: RANDOOP, QUICKCHECK for Java, JCHECK, FEED4JUNIT, T2, DAIKON, AUTOMATIC JUNIT CREATION TOOL, REASSERT and JWALK. In this section we will discuss the results obtained from exercising such tools.

Starting with RANDOOP, it was decided to run in two time spans, 2.5 and 5 minutes. Table 3.21 shows the coverage results obtained with both the test suites. Clearly there is little gain in increasing the running time from 2.5 to 5 minutes. Even with 5 minutes RANDOOP was unable to achieve a coverage result nearly as remarkable as the manually written test cases.

Let us now proceed by discussing the three PUT tools that were covered by this study, QUICKCHECK for Java, JCHECK and FEED4JUNIT. These tools were applied successfully to the target package and all of them achieved the same line coverage as the manually written tests. They have, however, allowed the coverage over the input domain to increase, by instantiating the test cases for several inputs.

The T2 Framework was also applied with success to a class of the target package, `com.puppycrawl.tools.checkstyle.api`. Because this tool requires some specifications in order to reveal its full potential, it was decided to produce specifications for one class (`FastStack`). This tool was able to cover 12.8% of the total instructions in comparison with the 15.9% achieved by the manually written test cases. This is not a bad result considering it was done automatically with just some simple specifications.

The application of DAIKON used only a fraction of the test suite of the case study. It was possible to infer some invariants from test cases, an example of which can be found in Listing 3.12. An

Package	Fan-In	Fan-Out
com/puppycrawl/tools/checkstyle	96	166
com/puppycrawl/tools/checkstyle/api	993	351
com/puppycrawl/tools/checkstyle/checks	180	211
com/puppycrawl/tools/checkstyle/checks/annotation	20	49
com/puppycrawl/tools/checkstyle/checks/blocks	17	31
com/puppycrawl/tools/checkstyle/checks/coding	82	232
com/puppycrawl/tools/checkstyle/checks/design	11	38
com/puppycrawl/tools/checkstyle/checks/duplicates	31	56
com/puppycrawl/tools/checkstyle/checks/header	5	10
com/puppycrawl/tools/checkstyle/checks/imports	29	58
com/puppycrawl/tools/checkstyle/checks/indentation	118	145
com/puppycrawl/tools/checkstyle/checks/javadoc	22	77
com/puppycrawl/tools/checkstyle/checks/metrics	26	55
com/puppycrawl/tools/checkstyle/checks/modifier	2	7
com/puppycrawl/tools/checkstyle/checks/naming	27	48
com/puppycrawl/tools/checkstyle/checks/regexp	24	34
com/puppycrawl/tools/checkstyle/checks/sizes	14	41
com/puppycrawl/tools/checkstyle/checks/whitespace	26	54
com/puppycrawl/tools/checkstyle/doclets	10	10
com/puppycrawl/tools/checkstyle/filters	24	66
com/puppycrawl/tools/checkstyle/grammars	10	8
com/puppycrawl/tools/checkstyle/gui	153	173
Total	**1,920**	**1,920**

Table 3.20: Fan-In and Fan-Out by package of Checkstyle.

Element	Test Suite	Coverage (%)	Covered Instructions	Missed Instructions	Total Instructions
Target Package	RANDOOP 2.5 min	38.60	2,057	3,269	5,326
	RANDOOP 5 min	39.30	2,092	3,234	
	Manual Tests	73.20	3,896	1,430	

Table 3.21: RANDOOP generated tests for the target package compared with manually written tests.

example of an annotated method with the extracted invariants is illustrated in Listing 3.13.

```
===========================================================================
com.puppycrawl.tools.checkstyle.api.FastStack.clear():::ENTER
  Variables: this this.mEntries this.mEntries.getClass()
this != null
this.mEntries != null
===========================================================================
com.puppycrawl.tools.checkstyle.api.FastStack.clear():::EXIT
  Variables: this this.mEntries this.mEntries.getClass() orig(this) orig(this.
      mEntries) orig(this.mEntries.getClass())
this.mEntries == \old(this.mEntries)
this.mEntries != null
this.mEntries.getClass() == \old(this.mEntries.getClass())
===========================================================================
```

Listing 3.12: Two Invariants inferred by DAIKON.

```
public class FastStack<E> implements Iterable<E> {
...
  /*@ requires this != null; */
  /*@ requires this.mEntries != null; */
  /*@ ensures this.mEntries != null; */
  /*@ ensures \typeof(this.mEntries) == \old(\typeof(this.mEntries)); */
  public void clear() {
    mEntries.clear();
  }
...
}
```

Listing 3.13: Example of a class annotated with invariants detected by DAIKON.

Albeit useful, the invariants detected by DAIKON are incomplete in general. Take for instance the annotated method in Listing 3.14, which only take one of three possible values: 0, 1 or 100. This is obviously incomplete, since there is nothing preventing a Stack instance from having any particular size.

AUTOMATIC JUNIT CREATION TOOL was also successfully applied to the case study. The work flow of this tool is not much of an improvement over manually writing the test cases. After the first few test methods the user will probably be fed up with constantly having to manually introduce test code in so many different text boxes without any handy auto complete feature like in modern IDEs. The tool also lacks some relevant features, like the possibility of creating tests that expect a given exception to be thrown, or to generalise on a generic data type.

To apply REASSERT, changes were introduced to methods of the target of our study, `com.puppycrawl.tools.checkstyle.api`. After introducing the changes, the tests started failing, as expected. However, using REASSERT it was possible to fix them, using the new functionality that was used, by altering the assertions of the test case.

Finally, JWALK was successfully applied to a class of the case study, the `FastStack` class. This had to undergo some changes in order for JWALK to properly test it and produce accurate test oracles. The problem was that some code used in the class was not quite deterministic and was causing some problems for JWALK.

```
public class FastStack<E> implements Iterable<E> {
...
  /*@ requires this != null; */
  /*@ requires this.mEntries != null; */
  /*@ ensures this.mEntries != null; */
  /*@ ensures \result == 0 || \result == 1 || \result == 100; */
  /*@ ensures \typeof(this.mEntries) == \old(\typeof(this.mEntries)); */
  public int size() {
    return mEntries.size();
  }
...
}
```

Listing 3.14: Example of incomplete invariants detected by DAIKON.

3.3.3 Tools Not Tested

The tools that were not applied with success to `com.puppycrawl.tools.checkstyle.api` were: JMLUNITNG, JAVA PATHFINDER, UNITCHECK, UDITA, ECLAT and PEX. In this section we discuss why this happened.

The first tool that was not able to run successfully on the subject of this study was JMLUNITNG, the problem arising from the classes not being able compile under the `jml4c` compiler, a compiler for JML specifications, which is a requirement for JMLUNITNG. The tool was applied to an example annotated class provided at `jml4c`'s website. This new class consists of a simple representation of a bank account and JML is used to specify legal operations over it. Executing the generated test class without providing any additional inputs the tool was able to generate 75 test cases of which 51 were skipped and 5 failed. It is worth mentioning that the tests that failed were due to the incorrect handling of null values in the constructor and main methods of the class. The test cases skipped are due to the inputs not meeting the specified pre-condition. Some more values were provided to the test cases through strategy files generated by the tool, which was then able to generate a total of 588 test cases of which 402 were skipped and the same 5 failed, as is illustrated in Table 3.22.

	Total	Passed	Skipped	Failed	Used Tests (%)	Unused Tests (%)
Without custom values	75	19	51	5	25.33	74.67
With custom values	588	181	402	5	30.78	69.22

Table 3.22: Comparison of JMLUnitNG results.

Continuing to JAVA PATHFINDER and related tools, UNITCHECK and UDITA, it was not possible to successfully apply them to `com.puppycrawl.tools.checkstyle.api`, nor Checktyle, due to the dependency of Checkstyle on Java Native interface methods. A discussion was opened with the team behind JAVA PATHFINDER, to learn a way to work around this limitation. It was concluded that making Checkstyle work with JAVA PATHFINDER would consume too much time and there would be little or no return from it. In addition, UNITCHECK is very unstable at the time and UDITA is not working properly.

ECLAT was unable to run due the lack of maintenance of the project. Dependencies were broken and the tool will not run regardless of the target system. PEX was not applied, because it is meant for systems implemented with the .NET framework.

KORAT was not successfully applied to Checkstyle. The results achieved with KORAT were either too loose or too strict, and proper ones were never generated. However, it was applied to some examples provided by their respective developers.

3.4 Summary

The tools covered in this case study are regarded as the ones that could aid in the testing phase of the software development cycle. Other tools exist which either were not available for public use or there is no usable implementation available. These include TestEra [Marinov and Khurshid, 2001], Rostra [Xie et al., 2004], Symstra [Xie et al., 2005] and JTestCraft [den Hollander, 2010].

Some commercial tools were also found, like JTest [Parasoft, 2011] and Agitar [Agitar Technologies, 2011], though very little is known of how these solutions work. However, according to Boshernitsan et al. [2006], Agitar makes use of the the DAIKON invariant detector to generate invariants and then uses them to improve the test generation process. At the time of writing there is no pricing information for these tools in their homepage. Evaluations are available for enterprises.

Other tools that were not considered for this case study are tools that rely on the existence of models for the tool under test, like TGV [Jard and Jéron, 2005] and TorX [van Osch, 2005]. The reason these tools were discarded was due to the common lack of a model from which to extract test cases, like Checkstyle, or in some cases the existence of an outdated model.

Chapter 4

Raikon - An Experiment in TCG Tool Interoperability

There are two ways to contribute to the advance of technology - either by inventing something new and useful, or by showing that the current state-of-the-art tools can be put together in a way such that the whole is better than the sum of its parts.

Tool integration, the second alternative above, is a challenging activity. We follow its path in building a new TCG tool - Raikon - by putting together two tools reviewed in the previous chapters - Randoop and Daikon. The inspiration came from Eclat, a tool that attempts to infer an operational model of the target software from existing test cases and then applies random testing to check the inferred model . Our tool is named Raikon, a construction from Randoop and Daikon. Raikon aims to be a replacement to the abandoned Eclat, while introducing some new functionality.

Like Eclat, Raikon attempts to infer a model of correct behaviour of a system from an existing test suite using Daikon. Then, it uses Randoop to generate assertions that check the model, achieving some degree of acceptance testing. The new assertions go beyond the ones that are generated by default by Randoop, which are meant for regression testing. The most interesting feature introduced over Eclat is the possibility to generate input values based on the inferred model. Figure 4.1 depicts the overall architecture of Raikon.

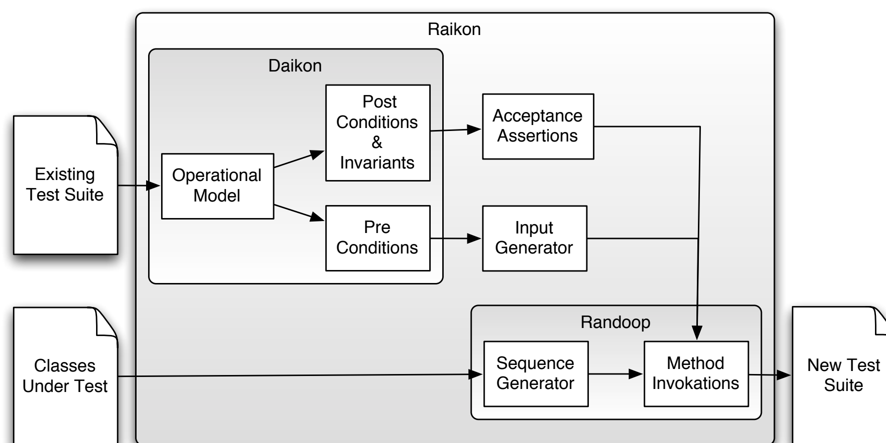


Figure 4.1: Architecture for Raikon

Raikon should improve upon the input domain coverage and help developers in creating complete test suites. Raikon should also be able to find more flaws in faulty software, since failing tests may be violating the inferred model and that may mean that there is an error in the software

or that the inferred model is incomplete. This is one of the reasons not to completely trust RAIKON, as the model may be incomplete. Ideally RAIKON would use a user provided model of operation. Inferring the model from an existing test suite is meant as a backup solution.

4.1 Implementation

To implement the core of RAIKON, first we have studied how RANDOOP implements checks and how the checks are added to sequences. We also had to study how DAIKON handles invariants. Starting with DAIKON, we have learned that the invariants are divided as unary, binary and ternary according to the number of variables involved in it. After analysing RANDOOP, we have learned that there are no classes implementing generic *n*-ary checks. It became clear that it would be necessary to implement a similar architecture for the checks in RAIKON. So it was decided to implement one class for each arity: unary, binary and ternary. To simplify the conversion between invariants from DAIKON to checks of RANDOOP, a field was added to the newly implemented classes to indicate what operation it intends to express. This allowed a reduction of the number of classes needed to convert the invariants into checks. Finally, to glue the whole process, a sequence visitor (which is how RANDOOP finds which checks should be added to a given statement) was implemented to convert invariants from DAIKON into checks of RANDOOP on a statement-by-statement approach.

In order to support pre-state invariants inferred by DAIKON, the sequence visitor was extended to create a deep copy of the object on which the statement is being invoked. The deep copy is done with a small library named JAVA DEEP-CLONING LIBRARY[1]. This library relies on standard reflection mechanisms. This decision introduced a limitation in the tool. When the object to be cloned is too big and complex, i.e. involving too many nested objects, the sequence generation process is delayed.

Since DAIKON refers to parameters by name, and since parameter names are not available by reflection, it was necessary to use a byte code engineering library to gather that information. DAIKON relies on the BCEL[2] library. However, using BCEL proved not easy to use, and we eventually opted for another such library, PARANAMER[3], which provides a much simpler way to get parameter names. This decision has limited the tool to only being able to get parameter names for classes whose debugging information is available, meaning that the classes need to be compiled with the debug flag on.

To generate input values from the inferred entry conditions, it would be necessary to use a constraint solving library to find all the values in range of the bounds defined by the conditions. Two major libraries were found in this respect, JACOP[4] and CHOCO[5]. Experiments were made with both solvers and JACOP was the chosen as the solver to include in RAIKON. In our experiments with both solvers, we have learned that CHOCO was not as reliable and not as scalable as JACOP and that both solvers however lack the support for floating point numbers.

Some pragmatic choices were also made in order to make for some missing functionality in DAIKON. For instance, there is no simple way of getting the name of a variable without tokens introduced by DAIKON. Therefore, it was decided to apply regular expression recognition to the variable names to eliminate each token. This introduces some delay while processing the correct

[1]Available at http://code.google.com/p/cloning/
[2]Available at http://commons.apache.org/bcel/
[3]Available at http://paranamer.codehaus.org/
[4]Available at http://www.jacop.eu/
[5]Available at http://www.emn.fr/z-info/choco-solver/

variable name. The internal API of DAIKON also has some limitations when attempting to learn what the type of a given variable is. Some methods for such a purpose are incomplete, others non existing, so it was decided to attempt to complete such methods as much as possible.

Currently, RAIKON is available at `http://code.google.com/p/raikon/`. At the moment it is only available as a command-line tool. All the usual RANDOOP options are still available, new options being added to control the extra functionality. The options are the following:

`use-daikon:boolean` Signals that this run will use DAIKON artefacts (Slower) [defaults to false].

`invoke-daikon:boolean` Signals that this run will invoke DAIKON to produce an invariants file (Slower) [defaults to false].

`test-case:string` The fully-qualified name of a test case.

`projectPrefix:string` The project prefix to filter out unwanted classes. Includes only classes with the prefix.

`exclusionPattern` Complements `projectPrefix`. Excludes specified classes.

`heapSize:string` The heap size for invoking DAIKON. Formatted as for the JVM [defaults to "512m"].

`invFileName:string` The name of the file that will store the invariants [defaults to "invariants"].

`disable-regression:boolean` Signals that this run will disable regression checks [defaults to false].

`daikon-config:string` Specifies the DAIKON configuration file, if any.

`lowBound:int` Lower bound for the input search domain [defaults to -100].

`uppBound:int` Upper bound for the input search domain [defaults to 100].

`maxSolutions:int` Maximum number of solutions [defaults to 100].

4.2 Evaluation Set Up

To evaluate RAIKON, three experiments were conducted. The first one addressed FASTSTACK class of the subject of our tool review, Checkstyle. The other two experiments were conducted on the Apache Commons Collection[6], one on an arbitrary package of the project and the other on the whole project. The reason for choosing Commons Collections is due to RANDOOP having been evaluated on it with success [Pacheco, 2009]. These three experiments also allowed us to evaluate RAIKON on three different levels, a single class, a package and a whole system.

We will measure the number of generated test cases and the input bound coverage, which is measured by the tool itself. We measure line and method coverage to compare both tools effectiveness in covering code elements. They are measured with the EclEmma[7] plugin for Eclipse. We also measure a ratio of assertions by cyclomatic complexity, to achieve a better idea of the amount

[6] Available at `http://commons.apache.org/collections/`
[7] Available at `http://www.eclemma.org/`

of assertions generated. Assertions are measured with an `egrep` script and cyclomatic complexity is measured with JavaNCSS[8]. Finally, we also compare the number of generated failing test cases, measured with JUnit.

To perform the experiments we ran both tools, RANDOOP and RAIKON, in increasing time spans, starting at 0 and incrementing 5 seconds after each run. We repeat this process until the number of generated test cases starts to stabilise, or the time reaches 900 seconds. In between runs we take note of the desired metrics. The sequences generated have a maximum size of 5 statements. The flow chart in Figure 4.2 illustrates the steps for the experiments, the results of which are detailed in the following sections.

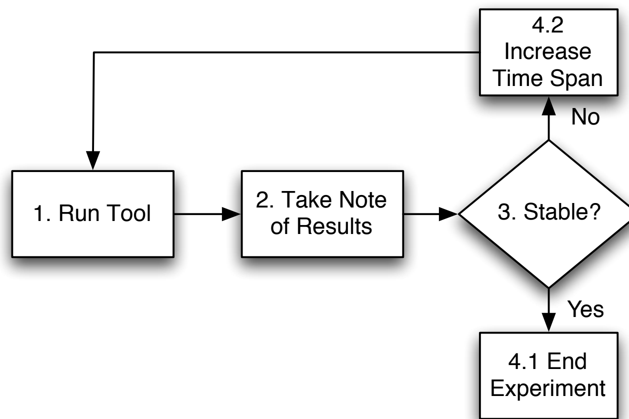


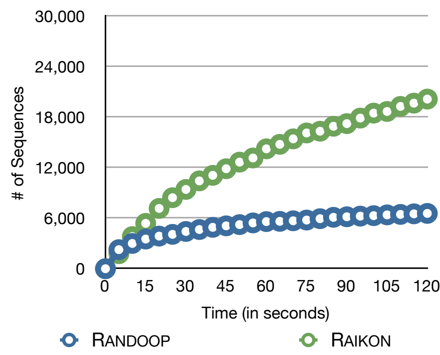
Figure 4.2: Flow of the experiments.

4.3 Results

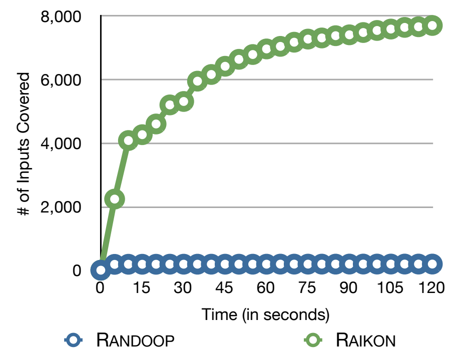
4.3.1 Experiment A: Single Class

This experiment started with the most simple case where RAIKON was used, the `FastStack` class from `com.puppycrawl.tools.checkstyle.api`. Both tools were ran up to 120 seconds. Figure 4.3 provides a comparison of results over the elapsed time. RAIKON was able to generate more sequences, i.e. more test cases in a 120 seconds time span, RANDOOP was able to generate approximately 6,500 while RAIKON was able to generate around 21,000. RAIKON was also able to significantly increase the input coverage in comparison to RANDOOP. RAIKON managed to cover around 7,700 distinct inputs while RANDOOP only managed to cover about 200. Both tools managed to cover all the lines and methods of the `FastStack` class. When comparing the ratio of number of assertions by cyclomatic complexity, RAIKON also came ahead of RANDOOP achieving a ratio of almost 15,000 against the ratio of 1,000 achieved by RANDOOP. When comparing the number of failed tests RANDOOP did not generate any failing test cases while RAIKON generated around 15,000 failing test cases. Running RAIKON adds an overhead of about one second to read the invariant file and to generate the new inputs.

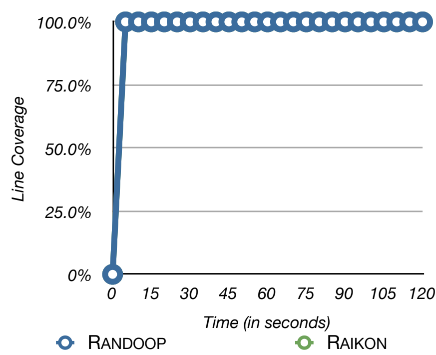
[8]Available at `http://www.kclee.de/clemens/java/javancss/`



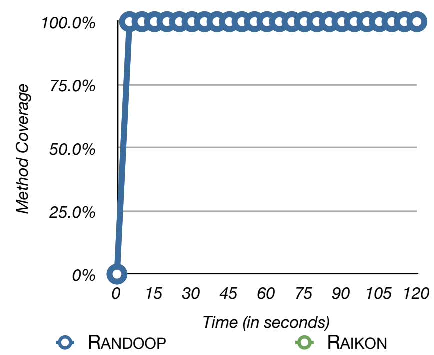
(a) Comparison of generated sequences.



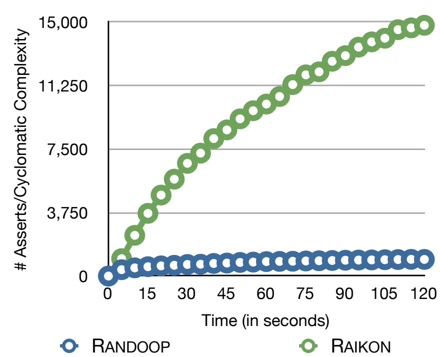
(b) Comparison of covered inputs.



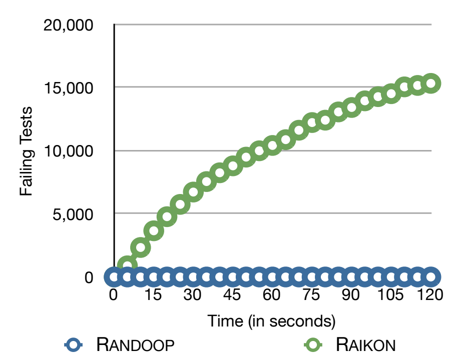
(c) Comparison of covered lines.



(d) Comparison of covered methods.



(e) Comparison of assertions by cyclomatic complexity (14).



(f) Comparison of failing tests.

Figure 4.3: Results for `FastStack`

4.3.2 Experiment B: Package

The package from Commons Collections chosen to perform our second experiment was, `org.apache.commons.collections.collection` which holds classes that extend the standard Java interface `java.util.Collection`. Results for the experiment can be found in Figure 4.4. This experiment took about 240 seconds to stabilise. The results for this package are also very encouraging. RAIKON generated around 22,750 test cases while RANDOOP generated 22,000. Still, the number of covered inputs was significantly increased with RAIKON managing to cover 8,672 distinct inputs and RANDOOP covering 2,781. Both tools achieved similar results in line and method coverage. The line coverage came close to 42% and method coverage was about 40%. When comparing the ratio of assertions by cyclomatic complexity, RAIKON was also able to surpass RANDOOP with a ratio of around 1,400 against the 400 achieved by RANDOOP. RAIKON generated close to 15,500 failing test cases while RANDOOP generated around 1,000 failing test cases. RAIKON took around two seconds in reading the invariants file and generating new inputs.

4.3.3 Experiment C: Whole System

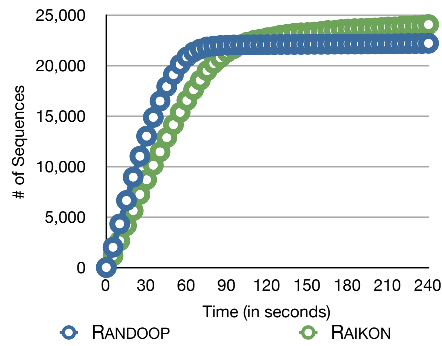
Our experiment with the whole Commons Collections library, was not as successful. This experiment was conducted to a limit of 900 seconds after which only RANDOOP was able to stabilise. RANDOOP was able to generate around 360,000 test cases achieving a coverage of 1,450,000 inputs while RAIKON was only able to generate about 123,000 test cases, that cover around 410,000 inputs. In terms of coverage both tools were close to each other. However, RANDOOP came out on top with a line coverage of 41.9% and method coverage of 60%, while RAIKON managed a line coverage of 39.7% and method coverage of 57.4%. Comparing the rate of assertions by cyclomatic complexity, RANDOOP and RAIKON were equal up until 600 seconds, where RANDOOP stopped generating assertions. RAIKON achieved a rate of 147.41 assertions by cyclomatic complexity and RANDOOP achieved a rate of 105.50. RAIKON also generated more failing tests than RANDOOP with a total of 21,040 failing tests and RANDOOP generating 3,940 failing tests. Figure 4.5 Illustrates the results obtained over time.

4.4 Discussion

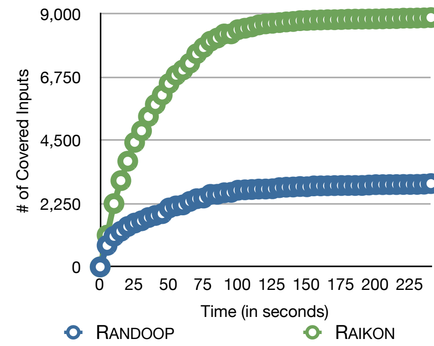
We can see that the metrics measured for both tools increase proportionally over time. It can also be seen that the generation of test cases starts growing at a lower pace after a given moment. This is due to the tools generating unique test cases, so the number of generated tests decreases over time. When the tools test all methods, they permute the sequences in order to generate new ones.

Overall, RAIKON preformed better than RANDOOP. In the smaller experiments RAIKON was able to surpass RANDOOP. It significantly exceeded the coverage over input domain, while generating more test cases. The largest experiment, with the whole library, was less successful. RANDOOP was able to generate more test cases and to cover more inputs. This is due to RAIKON's overhead when visiting sequences. Nonetheless, when comparing the number of failing tests, RAIKON surpassed RANDOOP. This could mean that RAIKON has more potential to find more flaws in the software under test.

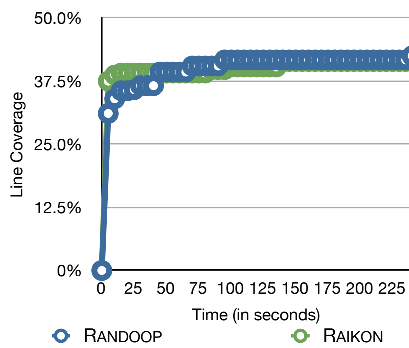
Over time, we believe that RAIKON can preform better than RANDOOP on all scenarios, when its limitations are lessened.



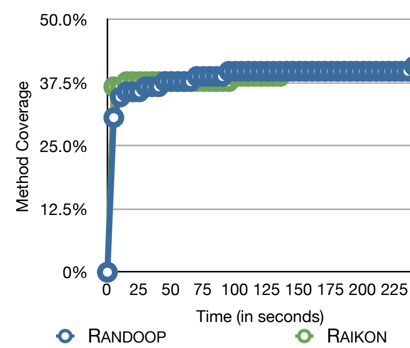
(a) Comparison of generated sequences.



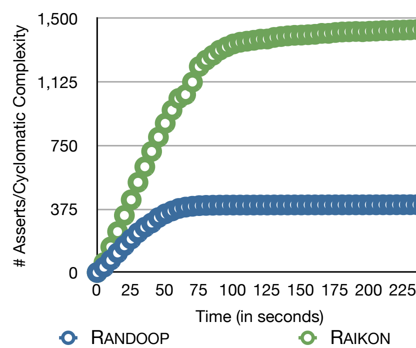
(b) Comparison of covered inputs.



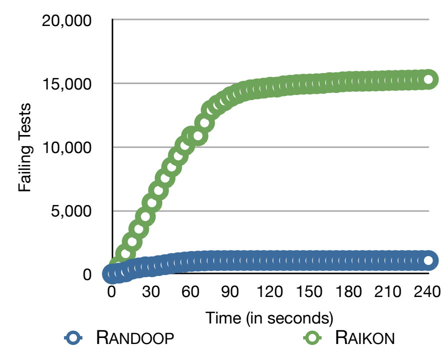
(c) Comparison of covered lines.



(d) Comparison of covered methods.

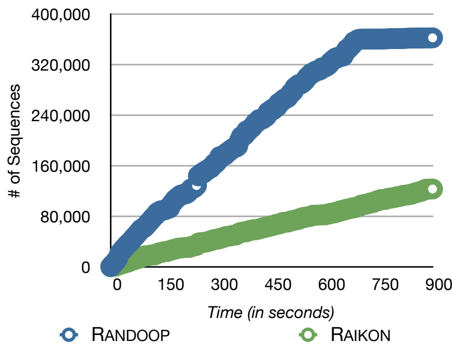


(e) Comparison of assertions by cyclomatic complexity (142).

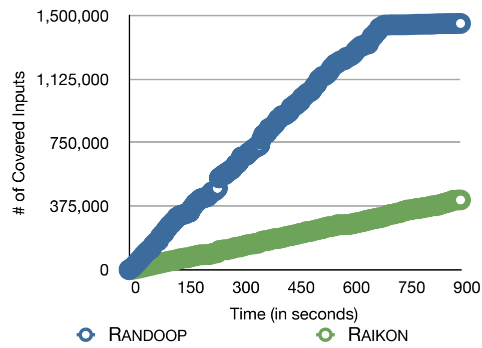


(f) Comparison of failing tests.

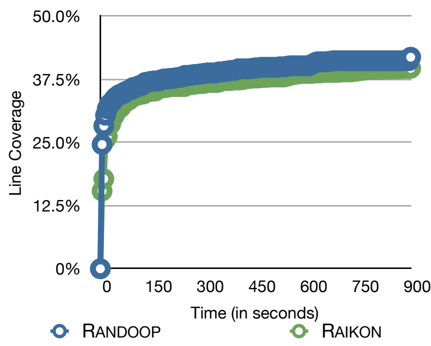
Figure 4.4: Results for `org.apache.commons.collections.collection`



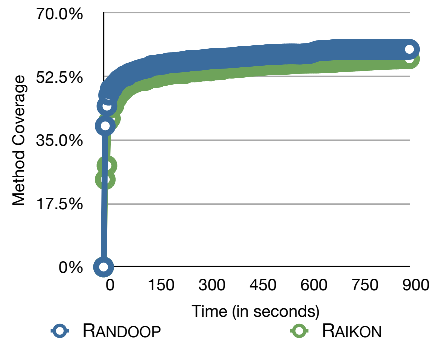
(a) Comparison of generated sequences.



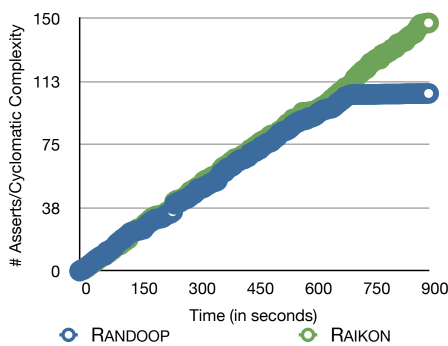
(b) Comparison of covered inputs.



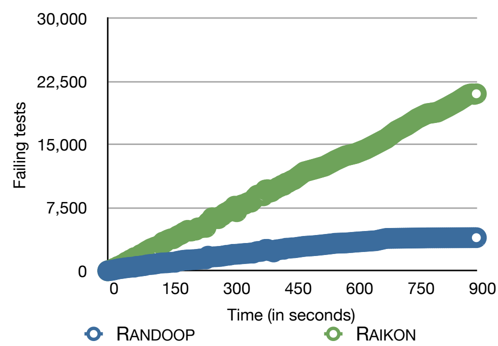
(c) Comparison of covered lines.



(d) Comparison of covered methods.



(e) Comparison of assertions by cyclomatic complexity (142).



(f) Comparison of failing tests.

Figure 4.5: Results for Commons Collections

Chapter 5

Conclusions

This is the final chapter of this dissertation which sums up the whole work, presents answers to the research questions and finally, it opens up avenues for future research.

Summing up, we have conducted two reviews: a literature review, where literature of the prominent topic, software testing, of our work was reviewed and a tool review, where we have evaluated existing tools. Then we gave our own contribution to the field by showing that, rather than inventing new tools for TCG one benefits from combining existing tools so as to get the best of them. Our choice was to merge RANDOOP with DAIKON, leading to RAIKON, whose implementation and evaluation was performed.

Our literature review allowed us to learn more about TCG and related techniques in a theoretical approach. We have also learned of the major challenges in testing. We have studied random testing, bounded exhaustive testing, mutation-driven testing and model-based testing. These techniques have their advantages and disadvantages, but they all aim at reducing the cost of testing in a development environment. We have also learned the current state of the art in constraint inference/solving techniques, since they are very closely related with software testing. The most relevant uses of constraint inference techniques were found to be: invariant detection, symbolic execution, dynamic analysis and concolic execution. While the constraint solving techniques studied amounted to TCG satisfiability solving. This study allows to answer our first research question, *What methods are there for Java unit test case generation?*. Our research has revealed that the most common methods of for the Java language are random testing, bounded exhaustive testing and model-based testing. We have also learned about mutation-driven testing, but this is a recent technique and no public implementation of this technique was available.

Finally, our literature review has shown some techniques that could either directly or indirectly useful to TCG, namely invariant detection and input classification. Invariant detection is a technique that relies on constraint inference to infer invariants from source code and input classification is a technique that classifies inputs based on their effects on the target software.

Our tool review, compared many of the tools uncovered in our literature review. Not all of the tools were successfully applied to our study, Checkstyle. Nonetheless, it was shown that the tools that were not applied, were either inappropriate for the tool under study or simply unstable at the time of the case study. The most interesting tools were RANDOOP, FEED4JUNIT and JMLUNITNG. RANDOOP is a very interesting tool that automatically generates a regression testing suite. This tool captures program executions and builds test cases from the observable states of the software. It is however, incapable of exploring a significant part of the software mostly due to its random input generation strategy, which lacks the capability to generate more complex inputs. FEED4JUNIT is very similar to both QUICKCHECK for Java and JCHECK. It does however, provide some additional features that make it more appealing that the other two. Like the possibility to feed test data through a `csv` file.

Even though JMLUNITNG was not successfully applied to the subject of our study, it was found to

be very interesting by being able to extract oracles from JML specifications. It also features some reflection mechanisms helping to improve test data generation. Nonetheless, it allows users to provide their own input data. The major set back of JMLUnitNG is that it requires JML specifications. Another problem of JMLUnitNG is that it generates too many unused test cases, leading to a considerable overhead in TCG.

In our tool review we have also found some very interesting tools. The ones that we would like to point out are Daikon and ReAssert. Daikon is able to take some of the already written test cases and infer code invariants. This is an interesting functionality, which we found could be used by other tools to improve TCG, as was shown with Raikon. ReAssert is another interesting tool to help developers in fixing broken test cases.

Let us raise some considerations about the tool review. Firstly, it was noted that some of the tools covered by our study were not suited for the case study, Checkstyle. However, it was decided to continue using Checkstyle as case study due to the previously stated reasons, i.e. for being a common Java application with a history in software engineering research [Cornelissen and Moonen, 2007; Zaidman et al., 2007, 2008; Lubsen et al., 2009]. Another relevant observation about the tools that were covered by this study is that they are far more usable when they come integrated in an IDE, or some form to apply the tool through a GUI. Command line tools are, most of the times, quite complicated to set up, especially in real life applications whose class paths are not always easy to configure.

A last consideration about the studied tools, is that they do not seem ready to be used by the software industry. Some tools, like QuickCheck for Java and Feed4JUnit, even though capable of significantly improving test cases by increasing their input space coverage, are still not automatic enough to efficiently reduce the test case creation cost. In addition, development and maintenance of these tools appear to be fairly slow. Other tools like Java Pathfinder and Udita are more of research tools than proper tools for use in industrial environments, as their set up is not always simple and in some cases there is very little or no introductory documentation available. However, due to their nature these tools are in constant development. Finally, this study covers some tools that are both stable and easy to use, as is the case of Randoop and Daikon, but are also usable in research contexts by allowing new approaches to solve the problems they aim at solving.

Tables 5.1 and 5.2 sum up all information about the tools covered by the review, enabling us to answer the second research question which triggered the work. *How do the existing methods compare to each other?* The answer is that there are several tools available for TCG. However, some tools are widely unstable or unable to efficiently reduce the costs of software testing.

Concerning the last research question, *Is there an opportunity to integrate such a method in a tool for fully automated test suite augmentation?*, we have implemented, Raikon, a tool that integrates Randoop and Daikon. The tool uses an operational model inferred by Daikon to improve the assertions used by Randoop and to generate a higher number of valid inputs. This effectively increases the coverage over the input domain and in some cases improves the line coverage in comparison with the test suites generated by Randoop. However, in larger systems, Raikon works significantly slower in comparison with Randoop, but that could be due to some limitations of Raikon, which can be improved. Nonetheless, the results obtained from Raikon were good enough to inspire future work on the tool. Which answers our last research question affirmatively.

There are, however, some future improvements to Raikon. First and foremost, the tool limitations should be amended. In an attempt reduce the delay introduced by the deep copy strategy there is a possibility of using serialisation mechanisms instead of reflection. To get the parameter names of a statement, a new strategy also needs to be implemented. For example using the same library as Daikon, BCEL. Or by extending Daikon to also provide the parameter position in

	RANDOOP	QUICKCHECK	JCHECK	FEED4JUNIT	T2	JMLUNITNG	UDITA	PEX
Language	Java & .NET	Java	Java	Java	Java	Java	Java	ṄET
Price	Free	Free	Free	Free	Free	Free	Free	Free
Level of Tests	Unit	Unit	Unit	Unit	Unit	Unit	Unit	Unit
Scope	TCG	PUT framework	PUT framework	PUT framework	PUT framework	TCG	Input / Oracle Generator	TCG
Adequacy Criterion					/ Spec. Valid.	Spec. Valid.		/ Spec. Valid.
Incremental	No	No	No	No	No	No	No	No
Previously Generated Tests	Overwrites	Not applicable	Not applicable	Not applicable	Regression tests	Not applicable	-	Overwrites
Integration With Manual Tests	Co-Exist	Co-Exist	Co-Exist	Co-Exist	Co-Exist	Co-Exist	Co-Exist	Co-Exist
Integration With Development Tools	Eclipse	None	None	None	None	None	None	Visual Studio
Code or Binary Level	Binary	Code	Code	Code	Code	Source	Binary	Binary
Configuration Effort	Low	Low	Low	Low	Low	Median	Impossible to say	Low

Table 5.1: TCG Tools Summary.

	JAVA PATHFINDER	UNITCHECK	KORAT	REASSERT	DAIKON	AUTO JUNIT
Language	Java	Java	Java	Java	Java	Java
Price	Free	Free	Free	Free	Free	Free
Level of Tests	System	Unit	Unit	Unit	Unit	Unit
Scope	Bytecode Model Checker	Unit Model Checker	Input Generator	Test Maintenance	Invariant Detection	Test Stub Generator
Adequacy Criterion	Prop. Valid. /	Prop. Valid. /		-	-	Branch Coverage
Incremental	No	No	No	-	Yes	Mo
Previously Generated Tests	-	-	-	-	-	No Integration
Integration With Manual Tests	Co-Exist	Co-Exist	Mo	Possible	Can Help	No
Integration With Development Tools	Eclipse / Netbeans	Eclipse / Ant	None	Eclipse	None	None
Code or Binary Level	Binary	Binary	Binary	Code	Binary	Code
Configuration Effort	Varies with the	Low	Low	Low	Low	Low

Table 5.2: Interesting Tools Summary.

the parameter array, thus dispensing the need for both PARANAMER and BCEL. To patch up the input generation limitation, of only being able to generate integer values, the solver, JACOP could be extended to support floating point numbers, or we could implement a similar library, whose focus would be the generation of input data based on the conditions inferred by DAIKON. Also, the improvement of the internal API of DAIKON should somewhat improve performance and simplify some internal logic of RAIKON.

Other than repairing the discussed limitations, RAIKON could greatly benefit of more integration with other development tools. Like the Eclipse IDE or the Maven project manager. Other improvements that could benefit RAIKON are improvements on both of the tools that make RAIKON. A suggestion to improve DAIKON would be use existing specifications when inferring invariants. Instead of relying solely in execution traces, DAIKON could also attempt to use existing specifications, such as JML specifications or existing abstract models of the software, to enrich the one it infers. RANDOOP could offer more control on the sequences it generates, allowing a differentiation of *set up* statements, i.e. statements that prepare the instance, from statements that test the instance. Also, in the future, RANDOOP could benefit from a combinatorial testing strategy, in which all possible input combinations are tested. Table 5.3 sums up a profile of RAIKON.

Language	Java
Price	Free under the MIT licence (see MIT [2011])
Level of Tests	Unit level
Scope	Test case generator
Adequacy Criterion	Fault detection / specification coverage
Incremental	Existing test cases are used to infer a model of correct operation
Previously Generated Tests	Overwrites or complements previously generated test cases
Integration With Manual Tests	Co-exist with manual tests
Integration With Development Tools	Only available as a command line tools
Code or Binary Level	Works at the binary level
Configuration Effort	The standalone command line tool requires a proper setup of the class path variable
Quality of Produced Tests	Produces complete test cases, with inputs and test oracles. Some of the produced tests seem to contain problems that prevent the test classes from compiling and therefore running. Nonetheless, they are easy to fix and execute

Table 5.3: RAIKON Summary.

In conclusion, our work has opened some paths for future research. There is already a wide variety of TCG tools whose source code is open and in some cases, there is room for improvement. For instance, integrating KORAT with FEED4JUNIT in order to allow the generation of more complex

data types to pass them on to a PUT. Daikon could be combined with Korat, in order to produce a valid predicate for the structure to be generated from the invariants inferred by Daikon. Another avenue for future research is the generation of more abstract test oracles. The invariants inferred by Daikon are influenced by execution traces, and when the number of execution traces is not enough, the invariants inferred will reflect the execution traces, thus the invariants will not reveal appropriate properties for acceptance testing. Allowing Daikon to gather information from other means than executions might improve the inferred invariants. These experiments could produce advancements in TCG.

Summing up, tool interoperability shows potential [Sun et al., 2009; Marín et al., 2011]. The effort put in combining tools is payed off by the results, which in most cases, are better than the sum of its parts. This dissertation is proof of that, as our work showed with Raikon.

Bibliography

29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007, 2007. IEEE Computer Society.

Agitar Technologies. Agitar, March 2011. URL http://www.agitar.com/.

Werner Alexi. Extraction and verification of programs by analysis of formal proofs. *Theor. Comput. Sci.*, 61:225–258, 1988.

Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.

ASF. Apache License, Version 2.0, February 2011. URL http://www.apache.org/licenses/LICENSE-2.0.html.

Luciano Baresi and Michal Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001. http://www.cs.uoregon.edu/~michal/pubs/oracles.html.

Bernhard Beckert and Reiner Hähnle, editors. *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science*, 2008. Springer. ISBN 978-3-540-79123-2.

Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In Lionel C. Briand and Alexander L. Wolf, editors, *FOSE*, pages 85–103, 2007.

Marat Boshernitsan, Roong-Ko Doong, and Alberto Savoia. From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. In Lori L. Pollock and Mauro Pezzè, editors, *ISSTA*, pages 169–180. ACM, 2006. ISBN 1-59593-263-1.

Giorgio Bruno, Mauro Varani, Valter Vico, and Chris Offerman. Benefits of using model-based testing tools. In Paolo Nesi, editor, *Objective Software Quality*, volume 926 of *Lecture Notes in Computer Science*, pages 224–235. Springer, 1995. ISBN 3-540-59449-3.

Oliver Burn. Checkstyle Homepage, February 2011. URL http://checkstyle.sourceforge.net/.

Michael Carbin and Martin C. Rinard. Automatically identifying critical input regions and code in applications. In Tonella and Orso [2010], pages 37–48. ISBN 978-1-60558-823-0.

Chin-Liang Chang, Richard C. T. Lee, and John K. Dixon. The specialization of programs by theorem proving. *SIAM J. Comput.*, 2(1):7–15, 1973.

Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Experimental assessment of random testing for object-oriented software. In David S. Rosenblum and Sebastian G. Elbaum, editors, *ISSTA*, pages 84–94. ACM, 2007. ISBN 978-1-59593-734-6.

Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

Edmund M. Clarke. The birth of model checking. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2008. ISBN 978-3-540-69849-4.

Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981. ISBN 3-540-11212-X.

B. Cornelissen and L. Moonen. Visualizing similarities in execution traces. *Program Comprehension through Dynamic Analysis*, 1:6–10, 2007.

Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In Tonella and Orso [2010], pages 85–96. ISBN 978-1-60558-823-0.

Brett Daniel, Danny Dig, Tihomir Gvero, Vilas Jagannath, Johnston Jiaa, Damion Mitchell, Jurand Nogiec, Shin Hwei Tan, and Darko Marinov. Reassert: a tool for repairing broken unit tests. In Taylor et al. [2011], pages 1010–1012. ISBN 978-1-4503-0445-0.

Jonathan de Halleux and Nikolai Tillmann. Parameterized unit testing with pex. In Beckert and Hähnle [2008], pages 171–181. ISBN 978-3-540-79123-2.

William H. Deason, David B. Brown, Kai-Hsiung Chang, and James H. Cross. A rule-based software test data generator. *IEEE Trans. Knowl. Data Eng.*, 3(1):108–117, 1991.

Menno den Hollander. Automatic Unit Test Generation. Master's thesis, Delft University of Technology, 2010.

Heidi E. Dixon, Matthew L. Ginsberg, and Andrew J. Parkes. Generalizing boolean satisfiability i: Background and survey of existing work. *CoRR*, abs/1107.0040, 2011.

Nick Eaton. Bill gates: 'the importance of software is higher today than ever', October 2011. URL http://blog.seattlepi.com/microsoft/2011/05/18/bill-gates-the-importance-of-software-is-higher-today-than-ever/.

J. Edvardsson. Contributions to program and specification-based test data generation. Master's thesis, Linköpings universitet, 2002.

Jon Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering in Linkoping*, pages 21–28, October 1999. URL http://citeseer.ist.psu.edu/544057.html;http://www.ida.liu.se/~joned/papers/class_atdg.pdf.

Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

Harry Foster, David Lacey, and Adam Krolnik. *Assertion-Based Design*. Kluwer Academic Publishers, Norwell, MA, USA, 2 edition, 2003. ISBN 1402074980.

Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In Tonella and Orso [2010], pages 147–158. ISBN 978-1-60558-823-0.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in udita. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *ICSE (1)*, pages 225–234. ACM, 2010. ISBN 978-1-60558-719-6.

GNU. GNU LIcences, February 2011. URL `http://www.gnu.org/licenses/`.

Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Constraint-based invariant inference over predicate abstraction. In Neil D. Jones and Markus Müller-Olm, editors, *VMCAI*, volume 5403 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 2009. ISBN 978-3-540-93899-6.

M.J. Harrold, R. Gupta, and M.L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):270–285, 1993.

Gerard J. Holzmann. Formal software verification: How close are we? In John Hatcliff and Elena Zucca, editors, *FMOODS/FORTE*, volume 6117 of *Lecture Notes in Computer Science*, page 1. Springer, 2010. ISBN 978-3-642-13463-0.

John Hughes. Software testing with quickcheck. In Zoltán Horváth, Rinus Plasmeijer, and Viktória Zsók, editors, *CEFP*, volume 6299 of *Lecture Notes in Computer Science*, pages 183–223. Springer, 2009. ISBN 978-3-642-17684-5.

John Hughes and Thomas Arts. QuickCheck for Erlang Homepage, April 2011. URL `http://www.quviq.com/`.

IBM. Common Public License, February 2011. URL `http://www.ibm.com/developerworks/library/os-cpl.html`.

Vilas Jagannath, Yun Young Lee, Brett Daniel, and Darko Marinov. Reducing the costs of bounded-exhaustive testing. In Marsha Chechik and Martin Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2009. ISBN 978-3-642-00592-3.

Claude Jard and Thierry Jéron. Tgv: theory, principles and algorithms. *STTT*, 7(4):297–315, 2005.

C. Kaner, J.L. Falk, and H.Q. Nguyen. *Testing computer software*. John Wiley & Sons, Inc. New York, NY, USA, 1999. ISBN 0471358460.

Cem Kaner, James Bach, and Bret Pettichord. *Lessons Learned in Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 2001. ISBN 0471081124.

Teemu Kanstrén. Towards a deeper understanding of test coverage. *Journal of Software Maintenance*, 20(1):59–76, 2008.

Michal Kebrt and Ondrej Sery. Unitcheck: Unit testing and model checking combined. In Zhiming Liu and Anders P. Ravn, editors, *ATVA*, volume 5799 of *Lecture Notes in Computer Science*, pages 97–103. Springer, 2009. ISBN 978-3-642-04760-2.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *SOSP*, pages 207–220. ACM, 2009. ISBN 978-1-60558-752-3.

Z. Lubsen, A. Zaidman, and M. Pinzger. Using association rules to study the co-evolution of production & test code. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 151–154. IEEE Computer Society, 2009.

P. Maragathavalli. Search-based software test data generation using evolutionary computation. *CoRR*, abs/1103.0125, 2011.

César A. Marín, Gabriel Alejandro Lopardo, and Nikolay Mehandjiev. A diversity analysis of the impact of an interoperability tool to a business ecosystem. In Sumitra Reddy and Samir Tata, editors, *WETICE*, pages 35–40. IEEE Computer Society, 2011.

Darko Marinov and Sarfraz Khurshid. Testera: A novel framework for automated testing of java programs. In *ASE*, pages 22–31. IEEE Computer Society, 2001. ISBN 0-7695-1426-X.

Jesús Martínez and Cristóbal Jiménez. Software model checking for internet protocols with java pathfinder. In Ulrich Ultes-Nitsche, Daniel Moldt, and Juan Carlos Augusto, editors, *MSVVEIS*, pages 91–100. INSTICC PRESS, 2008. ISBN 978-989-8111-43-2.

Aleksandar Milicevic, Sasa Misailovic, Darko Marinov, and Sarfraz Khurshid. Korat: A tool for generating structurally complex test inputs. In *ICSE* DBL [2007], pages 771–774.

MIT. The MIT License, February 2011. URL `http://www.opensource.org/licenses/mit-license.php`.

Tom Moertel. Test::LectroTest, April 2011. URL `http://search.cpan.org/~tmoertel/Test-LectroTest-0.3600/lib/Test/LectroTest/Tutorial.pod`.

Glenford J. Myers. *The art of software testing*. Business data processing - A Wiley-Interscience publication. Wiley, New York, 1979. ISBN 0-471-04328-1.

NASA. NASA Open Source Agreement, March 2011. URL `http://ti.arc.nasa.gov/opensource/nosa/`.

NASA Ames Research Center. Java PathFinder Project WIki, March 2011. URL `http://babelfish.arc.nasa.gov/trac/jpf/wiki`.

A. Jefferson Offutt and Jane Huffman Hayes. A semantic model of program faults. In *ISSTA*, pages 195–200, 1996.

Open Source Contributors. QuickCheck for Java Homepage, April 2011a. URL `http://java.net/projects/quickcheck/pages/Home`.

Open Source Contributors. Jcheck homepage, April 2011b. URL http://www.jcheck.org/.

Carlos Pacheco. *Directed Random Testing*. Ph.D., MIT Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts, June 2009.

Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 504–527. Springer, 2005. ISBN 3-540-27992-X.

Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *OOPSLA Companion*, pages 815–816. ACM, 2007. ISBN 978-1-59593-865-7.

Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE* DBL [2007], pages 75–84.

Parasoft. Jtest homepage, March 2011. URL http://www.parasoft.com/jsp/products/jtest.jsp/.

Corina S. Pasareanu and Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *ASE*, pages 179–180. ACM, 2010. ISBN 978-1-4503-0116-9.

Corina S. Pasareanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *STTT*, 11(4):339–353, 2009.

Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. In Gregg Rothermel and Laura K. Dillon, editors, *ISSTA*, pages 93–104. ACM, 2009. ISBN 978-1-60558-338-9.

M.R. Prasad, A. Biere, and A. Gupta. A survey of recent advances in sat-based formal verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(2):156–173, 2005.

Wishnu Prasetya. T2: Experiment with reversi, October 2011. URL http://www.cs.uu.nl/wiki/bin/view/WP/TTSomeResult.

Wishnu Prasetya, Tanya Vos, and Arthur I. Baars. Trace-based reflexive testing of oo programs with t2. In *ICST*, pages 151–160. IEEE Computer Society, 2008.

C. V. Ramamoorthy, Siu-Bun F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Trans. Software Eng.*, 2(4):293–300, 1976.

Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static specification inference using predicate mining. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 123–134. ACM, 2007. ISBN 978-1-59593-633-2.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In Andy Gill, editor, *Haskell*, pages 37–48. ACM, 2008. ISBN 978-1-60558-064-7.

Monalisa Sarma, P. V. R. Murthy, Sylvia Jell, and Andreas Ulrich. Model-based testing in industry: a case study with two mbt tools. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, pages 87–90, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-970-1. doi: http://doi.acm.org/10.1145/1808266.1808279. URL http://doi.acm.org/10.1145/1808266.1808279.

Anthony J. H. Simons. Jwalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction. *Autom. Softw. Eng.*, 14(4):369–418, 2007.

Ian Sommerville. *Software Engineering*. Addison-Wesley, Harlow, England, 9. edition, 2010. ISBN 978-0-13-703515-1.

Matt Staats, Michael W. Whalen, and Mats Per Erik Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In Taylor et al. [2011], pages 391–400. ISBN 978-1-4503-0445-0.

Kevin J. Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. Software assurance by bounded exhaustive testing. In George S. Avrunin and Gregg Rothermel, editors, *ISSTA*, pages 133–142. ACM, 2004. ISBN 1-58113-820-2.

Y. Sun, Z. Demirezen, F. Jouault, R. Tairas, and J. Gray. A model engineering approach to tool interoperability. *Software Language Engineering*, pages 178–187, 2009.

Sun Microsystems, Inc. Code Conventions for the Java Programming Language, April 1999. URL http://www.oracle.com/technetwork/java/codeconv-138413.html.

G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 2002.

Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors. *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, 2011. ACM. ISBN 978-1-4503-0445-0.

Darrin Thompson. QuickCheck for JavaScript homepage, April 2011. URL https://bitbucket.org/darrint/qc.js/.

Thomas Tilley. Tool support for fca. In Peter W. Eklund, editor, *ICFCA*, volume 2961 of *Lecture Notes in Computer Science*, pages 104–111. Springer, 2004. ISBN 3-540-21043-1.

Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In Beckert and Hähnle [2008], pages 134–153. ISBN 978-3-540-79123-2.

Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. Parameterized unit testing: theory and practice. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *ICSE (2)*, pages 483–484. ACM, 2010. ISBN 978-1-60558-719-6.

Paolo Tonella and Alessandro Orso, editors. *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, 2010. ACM. ISBN 978-1-60558-823-0.

University of Illinois. University of Illinois/NCSA Open Source License, March 2011. URL http://www.opensource.org/licenses/UoI-NCSA.php.

Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. ISBN 0123725011.

Michiel van Osch. Automated model-based testing of x simulation models with torx. In Ralf Reussner, Johannes Mayer, Judith A. Stafford, Sven Overhage, Steffen Becker, and Patrick J. Schroeder, editors, *QoSA/SOQUA*, volume 3712 of *Lecture Notes in Computer Science*, pages 227–241. Springer, 2005. ISBN 3-540-29033-8.

Sander Daniël Vermolen. Automatically discharging vdm proof obligations using hol. Master's thesis, Radboud University Nijmege, 2007.

Willem Visser and Peter C. Mehlitz. Model checking programs with java pathfinder. In Patrice Godefroid, editor, *SPIN*, volume 3639 of *Lecture Notes in Computer Science*, page 27. Springer, 2005. ISBN 3-540-28195-9.

Volker Bergmann. Feed4JUnit Homepage, February 2011. URL http://databene.org/feed4junit.

William Whitney. Automatic junit creation tool homepage, April 2011. URL http://sourceforge.net/apps/trac/amaticjunittool/.

K.E. Wolff. A first course in formal concept analysis. *SoftStat*, 93:429–438, 1993.

Tao Xie and David Notkin. Mutually enhancing test generation and specification inference. In Alexandre Petrenko and Andreas Ulrich, editors, *FATES*, volume 2931 of *Lecture Notes in Computer Science*, pages 60–69. Springer, 2003. ISBN 3-540-20894-1.

Tao Xie, Darko Marinov, and David Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE*, pages 196–205. IEEE Computer Society, 2004. ISBN 0-7695-2131-2.

Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 365–381. Springer, 2005. ISBN 3-540-25333-5.

Howard Yeh. Rantly homepage, April 2011. URL https://github.com/hayeah/rantly#readme.

Serhiy A. Yevtushenko. Conexp homepage, February 2011. URL http://conexp.sourceforge.net/.

A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen. On how developers test open source software systems. *Arxiv preprint arXiv:0705.3616*, 2007.

A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen. Mining software repositories to study co-evolution of production & test code. In *2008 International Conference on Software Testing, Verification, and Validation*, pages 220–229. IEEE, 2008.

Xin Zhang and Franck van Breugel. Model checking randomized algorithms with java pathfinder. In *QEST*, pages 157–158. IEEE Computer Society, 2010. ISBN 978-0-7695-4188-4.

Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

Daniel Zimmerman. JMLUnitNG Homepage, March 2011. URL http://formalmethods.insttech.washington.edu/software/jmlunitng/.

Daniel Zimmerman and Rinkesh Nagmoti. JMLUnit: the next generation. *Formal Verification of Object-Oriented Software*, pages 183–197, 2011.

Index of Terms

acceptance testing
a test conducted to determine if the requirements of a specification or contract are met. 6, 7, 51, 64

adequacy criterion
measures the quality of the test suite following a given criterion. 7–9, 19, 73

assertion density
adequacy criterion that measures number of assertions per line of code. 7, 8

black-box testing
tests to validate a specification without any knowledge of the code. xi, 5, 6, 74

bounded exhaustive testing
testing technique that exhaustively tests the input domain. 9, 10, 12, 14, 18, 42, 44, 59

concolic execution
a technique that results from the combination of symbolic execution and dynamic analysis. 15, 16, 18, 35, 59

constraint inference
techniques to infer simpler constraint systems from more complex ones. 8, 9, 15, 16, 59

constraint satisfaction problems
mathematical problems defined as a set of objects whose state must satisfy a number of constraints. 15

constraint solving
techniques to solve constraint systems. 9, 15, 16, 52, 59

cyclomatic complexity
metric that measures the number of linearly independent control flow paths. 53–58, 78–97

dynamic analysis
techniques that attempt to infer constraint systems from actual executions. 9, 15, 16, 59, 73

fault detection
adequacy criteria that measure defect detection effectiveness. 7, 61, 74

formal concept analysis
a way of automatically deriving an ontology from a map of objects and their properties. 9, 10, 99

formal software verification
a process of formalising software validation. xi, 3, 76

functional tests
type of tests that verify that a is abiding a design specification. 7

genetic algorithms
group of algorithms that search for solutions by evolving existent candidates. 12

grey-box testing
a method that combines black-box testing and white-box testing, validates a certain specification using knowledge of internal algorithms and structures. xi, 5, 6

input bound coverage
measure of how much of the input is covered. 7, 12, 23, 37, 46, 51, 53, 60–62

input classification
technique to classify inputs by order of criticality. 17, 59

integration testing
tests for modules or sets of units. 6

invariant detection
research field that aims to detect invariants from source code. 16, 17, 59

model checking
technique to formally validate software. 3, 31

model-based testing
technique that generates test cases from models. 9, 10, 12, 14, 18, 59

mutation
artificial change introduced in the source code, with the aim of detecting it. 7, 12, 14

mutation analysis
technique to measure how effective a test suite is in fault detection. 7, 12

mutation-driven testing
Test case generation technique to detect mutants. 9, 10, 12, 14, 59

non-functional tests
tests for non-functional requirements, such as performance, stability, etc. 6

ontology
a formal representation of knowledge as a set of concepts within a domain. 9

parameterised unit testing
generalisation of unit testing allowing inputs as parameters. xi, 6, 12, 23, 25, 27, 37, 46, 61, 64

random testing
Test case generation technique that randomly generates test cases. 9, 10, 12, 14, 18, 27, 51, 59

regression testing
a type of testing that seeks to uncover new errors in existing functionality after making changes. 6, 7, 23, 51, 59

satisfiability solver
determine if the variables of a given boolean formula can be assigned. 16, 59

software
set of functions that compose an application. 3–6, 8, 9, 12, 14–19, 23, 31, 32, 35, 40, 45, 50, 51, 59, 60, 74, 75

software bug
flaw that causes incorrect behaviour of code. 10, 12

software testing
process intended to check the quality and correctness of software. 3, 5–10, 12, 14, 15, 17–19, 21, 27, 37, 40, 42, 44, 50, 59, 60

specification mining
techniques that attempt to generate program specifications from executions. 17

stakeholder
a person with an interest or concern in a product. 6, 7, 18

structural coverage
coverage criteria that measure how many code elements are executed. 7, 29, 46, 53, 54, 56, 60–62

symbolic execution
techniques that attempt to generate a constraint system through static execution. 9, 15, 16, 59, 73

system testing
type of tests that check if a given software is behaving as expected. 6, 7, 35

system under test
application or component being targeted for testing. 6, 12, 14, 16, 17, 35, 62

test
sequence of instructions that assert a given expected behaviour. 4–8, 10, 12, 14, 17–19, 23, 27, 35, 42, 46, 48, 51, 54, 59, 60, 76

test case generation
techniques that automatically generate test cases. vii, xi, 4, 8–10, 12, 14–18, 21, 23, 27, 29, 31, 37, 40, 45, 46, 51, 59–61, 63, 64, 74–76

test data generator
component of a test case generation that generates data. 9, 17, 23, 60

test oracle
component of a test case generation that asserts expected behaviour. 6, 12, 14, 15, 18, 23, 35, 60, 64

test suite
a set of test cases. 3, 4, 6–9, 12, 17, 19, 51, 52, 60, 73

theorem proving
a formal software verification technique that proves program correctness. 3

unit testing
testing of small application units. 6, 35, 74

white-box testing
tests to check internal workings and structures. xi, 5, 6, 74

Appendix A

Detailed Results

The tables that lead to Figure 4.3, Figure 4.4, Figure 4.5 can be found bellow.

Time Reading Invariants	0:00:00.738
Time Solving Constraints	0:00:00.260
Total Time	0:00:00.998

Table A.1: Set up time for `FastStack`

Time Reading Invariants	0:00:01.502
Time Solving Constraints	0:00:00.647
Total Time	0:00:02.149

Table A.2: Set up time for `org.apache.commons.collections.collection`

Time Reading Invariants	0:02:01.674
Time Solving Constraints	0:00:03.270
Total Time	0:02:04.944

Table A.3: Set up time for Commons Collections

Time	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (14)		Failing Tests	
	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON
0	0	0	0	0	0.0	0.0	0.0	0.0	0	0	0	0
5	2,266	1,783	184	2,240	100.0	100.0	100.0	100.0	376.14	1,028.21	0	886
10	3,001	3,817	187	4,085	100.0	100.0	100.0	100.0	483.64	2,405.93	0	2,337
15	3,524	5,391	187	4,269	100.0	100.0	100.0	100.0	552.79	3,701.79	0	3,660
20	3,875	7,163	187	4,606	100.0	100.0	100.0	100.0	601.64	4,784.21	0	4,777
25	4,082	8,438	189	5,207	100.0	100.0	100.0	100.0	635.79	5,722.50	0	5,751
30	4,425	9,419	189	5,315	100.0	100.0	100.0	100.0	684.79	6,657.07	0	6,728
35	4,655	10,437	189	5,955	100.0	100.0	100.0	100.0	698.36	7,260.07	0	7,558
40	4,917	11,095	189	6,165	100.0	100.0	100.0	100.0	744.00	8,119.21	0	8,255
45	5,095	11,880	189	6,424	100.0	100.0	100.0	100.0	767.29	8,633.43	0	8,804
50	5,241	12,664	189	6,643	100.0	100.0	100.0	100.0	804.21	9,285.93	0	9,495
55	5,443	13,156	189	6,794	100.0	100.0	100.0	100.0	823.71	9,765.93	0	9,998

Table A.4: Detailed results for `FastStack` (0-55 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (14)		Failing Tests	
Time	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON
60	5,599	14,244	189	6,970	100.0	100.0	100.0	100.0	854.21	10,158.50	0	10,420
65	5,627	14,779	191	7,051	100.0	100.0	100.0	100.0	864.93	10,604.79	0	10,879
70	5,695	15,429	191	7,182	100.0	100.0	100.0	100.0	874.00	11,308.79	0	11,625
75	5,781	16,131	191	7,280	100.0	100.0	100.0	100.0	890.93	11,885.00	0	12,234
80	5,944	16,336	191	7,316	100.0	100.0	100.0	100.0	904.00	12,063.00	0	12,422
85	6,107	16,925	191	7,387	100.0	100.0	100.0	100.0	919.29	12,686.86	0	13,076
90	6,171	17,242	193	7,415	100.0	100.0	100.0	100.0	934.64	13,018.21	0	13,427
95	6,248	17,897	193	7,487	100.0	100.0	100.0	100.0	945.71	13,521.64	0	13,954
100	6,305	18,488	193	7,552	100.0	100.0	100.0	100.0	952.79	13,832.21	0	14,287
105	6,397	18,684	193	7,596	100.0	100.0	100.0	100.0	965.29	14,047.79	0	14,521
110	6,452	19,286	193	7,649	100.0	100.0	100.0	100.0	974.36	14,553.43	0	15,060
115	6,521	19,673	193	7,669	100.0	100.0	100.0	100.0	986.36	14,659.29	0	15,174
120	6,571	20,155	193	7,709	100.0	100.0	100.0	100.0	982.43	14,811.07	0	15,338

Table A.5: Detailed results for `FastStack` (60-120 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (142)		Failing Tests	
Time	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON
0	0	0	0	0	0.0	0.0	0.0	0.0	0	0	0	0
5	2,008	1,154	766	1,131	31.1	37.5	30.6	36.7	31.63	58.15	64	617
10	4,315	2,636	1,089	2,253	34.0	38.7	34.7	36.7	73.32	150.80	164	1,597
15	6,637	4,142	1,269	3,071	35.6	39.1	35.7	37.8	115.27	241.37	382	2,537
20	8,945	5,628	1,438	3,762	35.6	39.1	35.7	37.8	157.85	337.18	474	3,537
25	11,017	7,232	1,560	4,433	35.8	39.1	35.7	37.8	197.50	429.25	562	4,514
30	13,009	8,688	1,642	4,844	36.6	39.1	36.7	37.8	234.63	532.39	567	5,604
35	14,864	10,125	1,757	5,354	36.6	39.1	36.7	37.8	267.07	624.35	656	6,577
40	16,481	11,446	1,831	5,784	36.6	39.1	36.7	37.8	292.37	714.97	732	7,547
45	17,918	12,857	1,896	6,118	39.3	39.1	37.8	37.8	324.20	799.34	823	8,424
50	19,103	14,143	2,098	6,530	39.3	39.1	37.8	37.8	346.18	881.18	887	9,300
55	20,119	15,364	2,171	6,823	39.3	39.1	37.8	37.8	363.35	958.64	940	10,122

Table A.6: Detailed results for `org.apache.commons.collections.collection` (0-55 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (142)		Failing Tests	
Time	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON
60	20,851	16,543	2,206	7,009	39.3	39.1	37.8	37.8	378.17	1,027.96	988	10,862
65	21,340	17,596	2,330	7,249	39.3	39.1	37.8	37.8	386.13	1,050.78	1,011	10,878
70	21,637	18,552	2,427	7,588	40.5	39.1	38.8	37.8	391.11	1,124.59	1,034	11,899
75	21,841	19,486	2,451	7,808	40.5	39.1	38.8	37.8	394.88	1,216.68	1,051	12,902
80	21,927	20,205	2,582	8,033	40.5	39.1	38.8	37.8	396.30	1,254.15	1,053	13,313
85	21,976	20,764	2,584	8,146	40.5	39.8	38.8	37.8	397.14	1,282.98	1,056	13,637
90	22,024	21,316	2,639	8,292	40.5	39.8	38.8	37.8	397.82	1,313.06	1,058	13,971
95	22,040	21,677	2,653	8,295	41.7	39.8	39.8	37.8	398.15	1,332.27	1,058	14,199
100	22,050	22,022	2,730	8,445	41.7	40.3	39.8	38.8	398.26	1,349.65	1,058	14,392
105	22,051	22,308	2,744	8,495	41.7	40.3	39.8	38.8	398.32	1,359.85	1,058	14,505
110	22,056	22,487	2,755	8,544	41.7	40.3	39.8	38.8	398.35	1,365.96	1,058	14,575
115	22,061	22,622	2,770	8,614	41.7	40.3	39.8	38.8	398.42	1,371.32	1,058	14,633
120	22,065	22,756	2,772	8,651	41.7	40.3	39.8	38.8	398.65	1,378.10	1,058	14,707

Table A.7: Detailed results for `org.apache.commons.collections.collection` (60-120 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (142)		Failing Tests	
Time	Randoop	Raikon	Randoop	Raikon	Randoop	Raikon	Randoop	Raikon	Randoop	Raikon	Randoop	Raikon
125	22,075	22,853	2,781	8,672	41.7%	40.3%	39.8%	38.8%	398.46	1,378.99	1,058	14,716
130	22,084	23,014	2,821	8,696	41.7%	40.3%	39.8%	38.8%	398.89	1,388.27	1,058	14,814
135	22,089	23,102	2,837	8,720	41.7%	40.3%	39.8%	38.8%	398.81	1,393.61	1,058	14,863
140	22,091	23,187	2,844	8,736	41.7%	41.4%	39.8%	39.8%	398.92	1,397.64	1,058	14,911
145	22,096	23,239	2,861	8,765	41.7%	41.4%	39.8%	39.8%	399.11	1,400.68	1,058	14,944
150	22,101	23,262	2,853	8,773	41.7%	41.4%	39.8%	39.8%	399.02	1,402.04	1,058	14,957
155	22,108	23,314	2,884	8,785	41.7%	41.4%	39.8%	39.8%	399.23	1,404.32	1,058	14,981
160	22,108	23,365	2,885	8,791	41.7%	41.4%	39.8%	39.8%	399.23	1,406.62	1,058	15,006
165	22,110	23,461	2,888	8,799	41.7%	41.4%	39.8%	39.8%	399.26	1,412.30	1,058	15,071
170	22,123	23,462	2,894	8,799	41.7%	41.4%	39.8%	39.8%	399.49	1,412.28	1,058	15,071
175	22,125	23,558	2,897	8,806	41.7%	41.4%	39.8%	39.8%	399.53	1,417.49	1,058	15,130
180	22,125	23,618	2,898	8,816	41.7%	41.4%	39.8%	39.8%	399.53	1,420.46	1,058	15,159
185	22,127	23,648	2,898	8,819	41.7%	41.4%	39.8%	39.8%	399.59	1,421.65	1,058	15,170

Table A.8: Detailed results for `org.apache.commons.collections.collection` (125-185 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (142)		Failing Tests	
Time	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON
190	22,129	23,659	2,899	8,820	41.7%	41.4%	39.8%	39.8%	399.66	1,423.48	1,058	15,174
195	22,134	23,691	2,914	8,823	41.7%	41.4%	39.8%	39.8%	399.56	1,422.06	1,058	15,187
200	22,139	23,722	2,928	8,825	41.7%	41.4%	39.8%	39.8%	399.74	1,424.58	1,058	15,198
205	22,139	23,758	2,928	8,830	41.7%	41.4%	39.8%	39.8%	399.74	1,425.99	1,058	15,212
210	22,144	23,819	2,930	8,838	41.7%	41.4%	39.8%	39.8%	399.94	1,428.90	1,058	15,236
215	22,150	23,828	2,931	8,839	41.7%	41.4%	39.8%	39.8%	399.82	1,428.49	1,058	15,240
220	22,164	23,886	2,934	8,854	41.7%	41.4%	39.8%	39.8%	400.12	1,430.91	1,058	15,259
225	22,171	23,903	2,935	8,854	41.7%	41.4%	39.8%	39.8%	400.21	1,431.56	1,058	15,265
230	22,174	23,938	2,937	8,859	41.7%	41.4%	39.8%	39.8%	400.25	1,433.06	1,058	15,278
235	22,184	23,971	2,945	8,864	41.7%	41.4%	39.8%	39.8%	400.41	1,434.80	1,058	15,296
240	22,199	24,056	2,966	8,875	42.5%	41.4%	40.8%	39.8%	400.63	1,437.62	1,058	15,322

Table A.9: Detailed results for `org.apache.commons.collections.collection` (190-240 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (6,848)		Failing Tests	
Time	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON
0	0	0	0	0	0.0	0.0	0.0	0.0	0	0	0	0
5	3,137	780	7,488	1,939	24.6	15.4	38.9	24.1	0.81	0.47	27	88
10	6,049	968	14,557	1,939	28.3	17.8	44.4	28.0	1.05	0.81	62	129
15	9,661	1,010	25,663	5,307	30.4	25.6	47.4	40.2	1.86	1.87	106	295
20	13,585	2,112	37,338	6,000	31.7	26.2	49.1	40.9	2.78	2.26	170	332
25	20,848	3,629	49,403	9,293	32.5	28.3	49.1	43.9	3.68	3.52	216	519
30	24,417	4,197	60,541	10,309	32.8	28.7	50.0	44.4	4.53	4.03	260	587
35	27,979	5,277	73,164	13,386	33.1	29.8	50.6	45.9	5.46	5.10	287	733
40	30,767	5,834	86,797	15,100	33.6	30.3	51.1	46.5	6.49	5.49	334	814
45	34,223	7,051	98,025	19,423	34.0	31.2	51.6	47.7	7.33	6.67	360	1,038
50	36,771	7,470	108,565	20,677	34.2	31.4	51.9	47.9	8.12	7.28	393	1,124
55	39,515	9,136	118,557	24,260	34.5	32.0	52.2	48.8	8.84	8.52	419	1,299

Table A.10: Detailed results for Commons Collections (0-55 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (6,848)		Failing Tests	
Time	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON
60	42,884	10,594	124,008	25,722	34.6	32.0	52.7	48.8	9.24	9.02	435	1,384
65	46,279	10,799	143,930	29,159	34.6	32.9	52.7	49.6	9.27	10.01	438	1,569
70	49,103	12,293	155,477	31,437	35.0	33.1	52.7	49.9	11.57	10.82	527	1,597
75	52,503	12,546	165,331	33,555	35.1	33.1	53.4	50.0	11.76	10.86	531	1,690
80	55,883	14,071	170,882	35,936	35.3	33.5	53.5	50.5	12.71	12.30	565	1,926
85	59,242	15,369	195,100	38,760	35.4	33.5	53.7	50.5	13.52	12.61	601	1,959
90	60,662	15,694	207,554	40,642	35.5	33.9	53.9	50.7	14.43	14.02	628	2,181
95	62,595	17,157	214,439	43,717	35.9	34.2	54.0	51.0	15.97	14.82	696	2,342
100	65,700	17,347	234,944	45,582	35.9	34.2	54.2	51.1	17.09	15.13	732	2,473
105	69,029	17,348	246,313	48,518	36.0	34.4	54.4	51.2	17.91	15.64	760	2,618
110	72,143	18,304	261,937	50,643	36.2	34.4	54.5	51.3	19.02	16.93	770	2,711
115	75,420	18,594	269,775	52,375	36.3	34.5	54.6	51.3	19.38	17.53	806	2,814
120	78,718	18,594	283,644	53,692	36.3	34.6	54.7	51.4	20.69	18.43	852	2,895

Table A.11: Detailed results for Commons Collections (60-120 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (6,848)		Failing Tests	
Time	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON
125	81,247	19,007	293,331	55,428	36.6	34.6	54.8	51.4	21.58	19.01	894	2,290
130	85,177	20,601	308,713	58,759	36.9	34.9	54.9	51.7	22.73	21.12	924	3,170
135	86,932	21,906	315,484	62,935	36.9	35.0	55.2	52.0	23.23	22.65	941	3,416
140	88,413	22,182	321,377	63,909	37.0	35.1	55.5	52.1	23.66	23.32	954	3,401
145	88,985	23,129	323,595	66,739	37.1	35.1	55.6	52.2	23.83	24.21	958	3,620
150	89,874	23,843	327,278	68,867	37.1	35.2	55.6	52.2	24.07	25.07	989	3,726
155	90,724	23,846	330,453	68,869	37.2	35.2	55.7	52.3	24.30	25.07	1,014	3,811
160	92,380	24,309	337,234	70,280	37.2	35.2	55.7	52.3	24.77	25.72	1,021	3,811
165	92,840	25,429	338,972	73,587	37.2	35.6	55.8	52.5	24.90	26.99	1,021	3,970
170	98,824	26,441	362,421	76,775	37.2	35.7	55.9	52.6	26.62	28.28	1,123	4,141
175	102,296	27,080	376,274	78,900	37.4	35.8	55.9	52.8	27.63	29.15	1,168	4,256
180	106,172	27,660	391,729	80,766	37.5	35.8	56.0	52.8	28.76	30.09	1,211	4,357
185	108,381	28,777	400,411	84,555	37.6	35.9	56.1	52.9	29.45	31.33	1,251	4,545

Table A.12: Detailed results for Commons Collections (125-185 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (6,848)		Failing Tests	
Time	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON
190	110,885	29,339	409,989	86,287	37.6	35.9	56.2	53.0	30.13	32.06	1,271	4,662
195	113,079	29,742	418,731	87,634	37.6	36.0	56.3	53.1	30.77	32.60	1,295	4,814
200	113,931	29,836	422,074	87,920	37.7	36.0	56.3	53.1	31.03	32.69	1,297	4,722
205	114,779	30,233	425,441	89,173	37.7	36.1	56.3	53.2	31.28	33.08	1,315	4,738
210	115,313	30,413	427,414	89,679	37.7	36.1	56.4	53.2	31.44	33.26	1,323	4,837
215	116,833	30,414	433,702	89,686	37.8	36.1	56.4	53.2	31.89	33.26	1,341	4,839
220	119,782	30,418	445,671	89,688	37.8	36.1	56.5	53.3	32.74	33.26	1,366	5,101
225	123,173	32,005	459,152	95,017	37.9	36.1	56.5	53.3	33.71	35.13	1,406	5,130
230	123,900	32,159	462,095	95,542	37.9	36.1	56.6	53.3	33.92	35.30	1,406	5,130
235	125,495	33,244	468,451	99,150	38.0	36.1	56.6	53.3	34.38	36.47	1,436	5,306
240	128,458	33,622	479,927	100,399	38.0	36.1	56.6	53.3	35.25	36.87	1,412	5,165
245	143,710	37,398	540,743	112,991	38.3	36.1	56.6	53.7	39.71	41.21	1,742	5,761
250	145,799	37,971	549,433	115,001	38.3	36.4	57.0	53.7	40.31	41.98	1,631	5,966

Table A.13: Detailed results for Commons Collections (190-250 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (6,848)		Failing Tests	
Time	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON
255	148,062	38,274	558,658	116,048	38.3	36.5	57.0	53.8	40.98	42.24	1,656	6,314
260	148,383	38,589	559,951	117,068	38.3	36.5	57.0	53.8	41.08	42.65	1,674	6,162
265	151,561	38,796	572,833	117,822	38.4	36.5	57.0	53.8	42.02	42.91	1,676	6,310
270	152,731	39,465	577,701	119,763	38.4	36.5	57.0	53.8	42.36	43.70	1,705	6,197
275	155,135	40,253	587,689	122,411	38.5	36.6	57.0	53.8	43.08	44.76	1,714	6,463
280	156,307	40,884	592,537	124,547	38.5	36.6	57.0	53.9	43.43	45.53	1,758	6,556
285	159,558	42,371	605,620	129,238	38.5	36.7	57.0	54.0	44.38	47.31	1,790	6,800
290	163,281	42,839	620,745	131,405	38.6	36.7	57.1	54.3	45.48	47.54	1,844	6,937
295	164,431	43,518	625,525	133,113	38.6	36.7	57.2	54.0	45.83	48.61	1,855	7,037
300	168,090	43,583	640,522	133,420	38.7	36.7	57.2	54.1	46.91	48.72	1,891	7,230
305	174,219	44,718	665,667	137,185	38.7	36.8	57.3	54.1	48.75	50.07	1,957	7,299
310	174,762	45,198	667,910	138,916	38.8	36.9	57.3	54.3	48.91	50.44	1,961	7,684
315	175,524	46,426	671,033	143,416	38.8	36.9	57.4	54.0	49.14	51.49	1,972	7,024

Table A.14: Detailed results for Commons Collections (255-315 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (6,848)		Failing Tests	
Time	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON
320	176,768	47,313	676,136	146,150	38.8	36.9	57.5	54.5	49.51	53.33	1,987	7,555
325	180,422	47,915	691,263	148,294	38.8	36.9	57.5	54.3	50.60	54.06	2,022	7,789
330	181,877	48,394	697,528	150,165	38.8	36.9	57.5	54.3	51.04	54.48	2,035	7,872
335	184,499	48,628	708,378	150,897	38.8	36.9	57.5	54.3	51.82	54.84	2,064	7,914
340	186,229	49,387	715,693	153,482	38.9	36.9	57.5	54.4	52.33	55.88	2,075	8,065
345	188,366	50,039	724,512	155,527	39.0	37.0	57.6	54.4	52.96	56.42	2,093	8,158
350	188,913	50,042	726,659	155,537	39.0	37.1	57.6	54.4	53.13	56.42	2,121	8,158
355	190,702	51,711	733,984	160,984	39.0	37.2	57.6	54.6	53.67	57.48	2,104	8,427
360	199,420	53,424	769,785	166,619	39.1	37.3	57.6	54.8	56.26	58.23	2,218	8,727
365	205,825	54,412	796,429	169,975	39.1	37.3	57.6	54.8	58.18	60.26	2,291	8,984
370	207,633	54,751	804,112	171,191	39.2	37.3	57.7	54.8	58.72	61.69	2,306	9,059
375	208,457	55,261	807,409	172,950	39.2	37.3	57.7	54.8	58.97	62.31	2,314	9,062
380	214,953	55,279	833,900	173,015	39.3	37.3	57.8	54.8	60.95	63.03	2,400	8,913

Table A.15: Detailed results for Commons Collections (320-380 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (6,848)		Failing Tests	
Time	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON
385	215,027	56,746	834,250	178,423	39.3	37.4	57.8	55.0	61.34	63.05	2,550	9,604
390	217,998	57,139	846,924	179,633	39.3	37.4	57.8	55.0	61.88	64.87	2,553	9,646
395	220,223	58,211	856,156	183,198	39.4	37.4	57.8	55.0	62.55	65.51	2,386	9,336
400	223,878	58,388	871,203	183,821	39.6	37.4	58.0	55.0	63.66	66.75	2,415	9,401
405	227,876	58,416	888,120	183,895	39.6	37.4	58.0	55.0	64.86	67.00	2,242	9,642
410	229,219	58,884	893,815	185,383	39.6	37.5	58.0	55.1	65.26	67.01	2,474	9,732
415	232,807	59,959	908,862	188,979	39.6	37.5	58.0	55.2	66.36	67.46	2,512	9,910
420	233,220	60,415	910,602	190,594	39.6	37.6	58.0	55.3	66.49	68.74	2,522	9,986
425	233,817	61,531	913,157	194,491	39.6	37.7	58.0	55.3	66.67	69.32	2,584	10,167
430	236,174	61,889	922,722	197,557	39.6	37.7	58.1	55.4	67.39	70.05	2,559	10,163
435	237,765	62,959	929,230	199,329	39.6	37.7	58.1	55.4	67.88	70.55	2,680	10,421
440	244,570	63,235	957,030	200,234	39.7	37.8	58.2	55.4	69.92	72.28	2,691	10,476
445	244,803	64,114	957,961	203,493	39.7	37.8	58.2	55.4	69.99	72.61	2,694	10,623

Table A.16: Detailed results for Commons Collections (385-445 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (6,848)		Failing Tests	
Time	Randoop	Raikon	Randoop	Raikon	Randoop	Raikon	Randoop	Raikon	Randoop	Raikon	Randoop	Raikon
450	248,642	64,826	973,418	205,995	39.8	37.8	58.4	55.4	71.13	73.74	2,737	10,753
455	248,643	65,345	973,422	207,726	39.8	37.8	58.4	55.4	71.13	74.59	2,737	10,842
460	252,403	66,492	989,478	211,809	39.8	37.8	58.4	55.5	72.26	75.19	2,771	11,025
465	252,892	67,078	991,568	213,874	39.9	37.9	58.4	55.5	72.41	76.44	2,805	11,208
470	258,700	67,540	1,016,022	215,543	39.9	37.9	58.4	55.6	74.16	77.13	2,839	11,391
475	259,984	68,176	1,021,425	217,751	39.9	37.9	58.5	55.6	74.54	77.65	2,873	11,574
480	262,807	69,793	1,032,845	223,684	39.9	37.9	58.5	55.7	75.39	78.12	2,910	11,761
485	265,175	70,309	1,042,652	225,374	39.9	37.9	58.5	55.7	76.11	78.48	2,926	11,827
490	266,216	70,492	1,047,283	225,935	39.9	38.0	58.5	55.7	76.42	80.21	2,942	11,893
495	267,770	70,558	1,053,870	226,234	39.9	38.0	58.6	55.7	76.90	80.89	2,958	11,959
500	271,553	71,446	1,070,121	229,422	39.9	38.0	58.6	55.8	78.05	81.01	2,974	12,025
505	275,296	72,674	1,086,331	233,774	39.9	38.0	58.6	55.8	79.17	81.15	2,994	12,091
510	277,314	73,090	1,094,944	235,236	39.9	38.0	58.6	55.8	79.80	81.82	3,019	12,161

Table A.17: Detailed results for Commons Collections (450-510 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (6,848)		Failing Tests	
Time	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON
515	283,379	73,743	1,120,118	237,440	39.9	38.0	58.6	55.8	81.63	83.20	3,044	12,231
520	285,701	74,436	1,130,015	241,161	40.0	38.0	58.6	55.7	82.33	83.81	3,069	12,301
525	288,053	74,784	1,140,169	241,345	40.0	38.0	58.7	55.7	83.03	84.54	3,094	12,371
530	288,647	76,519	1,142,513	247,325	40.0	38.0	58.7	55.8	83.22	85.94	3,121	12,443
535	290,262	76,861	1,149,230	248,481	40.0	38.0	58.7	55.9	83.71	87.35	3,149	12,572
540	292,560	78,176	1,158,853	252,923	40.1	38.1	58.7	55.9	84.38	88.01	3,177	12,701
545	295,767	78,193	1,172,399	252,981	40.1	38.2	58.7	55.9	85.36	88.62	3,205	12,830
550	300,035	78,760	1,190,364	254,908	40.2	38.3	58.7	56.0	86.63	90.34	3,233	12,959
555	302,458	79,305	1,200,758	256,763	40.2	38.4	58.7	56.0	87.37	90.35	3,273	13,100
560	305,521	79,872	1,213,669	258,702	40.2	38.5	58.7	56.0	88.30	90.90	3,282	13,211
565	306,325	81,533	1,217,109	264,919	40.2	38.5	58.7	56.1	88.55	91.44	3,291	13,322
570	308,540	81,550	1,226,636	264,987	40.2	38.6	58.7	56.1	89.22	92.09	3,300	13,433
575	309,894	81,555	1,232,400	265,003	40.2	38.6	58.7	56.2	89.63	94.29	3,309	13,544

Table A.18: Detailed results for Commons Collections (515-575 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (6,848)		Failing Tests	
Time	Randoop	Raikon	Randoop	Raikon	Randoop	Raikon	Randoop	Raikon	Randoop	Raikon	Randoop	Raikon
580	310,391	81,564	1,234,629	265,053	40.2	38.7	58.7	56.2	89.78	94.32	3,311	13,656
585	311,311	81,918	1,238,343	266,273	40.2	38.7	58.8	56.2	90.06	94.32	3,331	13,715
590	314,625	81,998	1,252,440	266,550	40.2	38.7	58.8	56.2	91.08	94.34	3,351	13,774
595	314,864	82,065	1,253,521	266,794	40.2	38.7	58.8	56.2	91.15	94.76	3,371	13,833
600	316,101	82,071	1,258,750	266,801	40.2	38.7	58.8	56.2	91.53	94.92	3,391	13,892
605	317,646	83,204	1,265,543	271,026	40.2	38.7	58.8	56.2	92.00	94.94	3,413	13,952
610	319,895	83,206	1,274,789	271,035	40.4	38.7	59.0	56.2	92.69	94.94	3,430	14,049
615	323,595	84,417	1,290,234	275,547	40.5	38.7	59.1	56.2	93.80	96.35	3,447	14,146
620	326,020	84,932	1,300,696	277,321	40.7	38.8	59.3	56.2	94.54	96.35	3,464	14,243
625	328,398	84,934	1,311,038	278,676	40.9	38.8	59.5	56.2	95.27	97.73	3,481	14,340
630	330,509	86,240	1,319,560	281,849	41.0	38.8	59.6	56.2	95.91	98.21	3,498	14,437
635	330,609	86,584	1,319,978	283,102	41.0	38.8	59.6	56.2	95.94	98.65	3,515	14,562
640	332,355	88,152	1,327,418	288,612	41.0	38.8	59.6	56.3	96.46	99.93	3,532	14,687

Table A.19: Detailed results for Commons Collections (580-640 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (6,848)		Failing Tests	
Time	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON
645	332,860	88,175	1,329,356	288,691	41.0	38.9	59.6	56.3	96.62	100.42	3,549	14,812
650	333,315	89,616	1,331,269	293,647	41.0	38.9	59.6	56.4	96.76	102.16	3,566	14,937
655	338,692	89,787	1,354,333	294,261	41.0	38.9	59.7	56.4	97.42	102.18	3,584	15,065
660	343,661	91,050	1,375,412	298,407	41.0	38.9	59.7	56.4	98.39	104.14	3,638	15,193
665	344,870	91,429	1,380,771	299,755	41.1	38.9	59.8	56.4	99.91	104.37	3,692	15,321
670	348,527	91,880	1,396,217	301,289	41.1	38.9	59.8	56.4	100.27	106.14	3,746	15,449
675	350,784	93,085	1,405,870	305,413	41.2	38.9	59.9	56.4	101.39	106.59	3,800	15,577
680	355,725	93,117	1,426,241	305,554	41.2	38.9	59.9	56.4	102.08	107.32	3,851	15,703
685	356,695	93,626	1,430,176	307,523	41.2	38.9	59.9	56.5	103.57	109.01	3,854	15,889
690	358,880	94,706	1,439,305	311,339	41.2	38.9	59.9	56.5	103.87	109.07	3,858	16,075
695	359,773	95,483	1,442,851	313,900	41.2	39.0	59.9	56.6	104.54	109.94	3,861	16,261
700	359,977	97,371	1,443,643	320,751	41.2	39.0	59.9	56.6	104.81	111.08	3,865	16,447
705	359,977	98,049	1,443,643	322,853	41.2	39.0	59.9	56.6	104.87	111.93	3,868	16,638

Table A.20: Detailed results for Commons Collections (645-705 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (6,848)		Failing Tests	
Time	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON
710	360,104	98,447	1,444,321	326,451	41.2	39.0	59.9	56.6	104.87	114.50	3,871	16,764
715	360,105	99,073	1,444,321	327,007	41.2	39.0	60.0	56.7	104.91	115.46	3,875	16,890
720	360,105	99,660	1,444,321	328,496	41.2	39.0	60.0	56.7	104.91	116.79	3,878	17,016
725	360,105	100,845	1,444,321	332,705	41.2	39.0	60.0	56.7	104.91	117.66	3,882	17,142
730	360,108	101,362	1,444,322	334,519	41.2	39.0	60.0	56.7	104.91	119.04	3,885	17,264
735	360,120	101,698	1,444,368	335,632	41.2	39.1	60.0	56.8	104.91	119.73	3,888	17,425
740	360,120	103,072	1,444,368	340,553	41.2	39.1	60.0	56.8	104.91	120.20	3,892	17,586
745	360,120	104,455	1,444,368	345,565	41.2	39.2	60.0	56.8	104.91	121.96	3,895	17,747
750	360,123	104,620	1,444,378	346,094	41.2	39.2	60.0	56.8	104.91	123.62	3,899	17,908
755	360,199	106,281	1,444,671	352,095	41.2	39.3	60.0	56.8	104.91	123.96	3,902	18,079
760	360,223	106,281	1,444,766	352,095	41.2	39.3	60.0	56.8	104.94	125.00	3,905	18,198
765	360,290	106,303	1,445,035	352,180	41.2	39.4	60.0	56.9	104.94	125.95	3,909	18,317
770	360,515	106,700	1,446,041	356,387	41.2	39.4	60.0	56.9	104.96	125.95	3,912	18,436

Table A.21: Detailed results for Commons Collections (710-770 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (6,848)		Failing Tests	
Time	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON
775	360,515	108,133	1,446,041	358,428	41.2	39.5	60.0	57.0	105.03	125.96	3,916	18,555
780	360,831	108,503	1,447,210	359,735	41.2	39.5	60.0	57.0	105.03	128.54	3,922	18,670
785	360,836	108,550	1,447,210	359,903	41.2	39.5	60.0	57.0	105.12	129.14	3,923	18,707
790	360,839	109,240	1,447,283	362,176	41.2	39.5	60.0	57.0	105.13	129.21	3,924	18,744
795	360,839	109,612	1,447,314	363,497	41.2	39.5	60.0	57.1	105.13	130.14	3,924	18,781
800	360,891	110,713	1,447,515	367,619	41.2	39.5	60.0	57.1	105.13	130.54	3,925	18,818
805	360,903	110,936	1,447,515	368,415	41.2	39.5	60.0	57.1	105.14	131.88	3,926	18,863
810	360,903	111,280	1,447,638	369,579	41.2	39.5	60.0	57.1	105.15	132.23	3,927	18,986
815	360,951	112,501	1,447,710	371,586	41.2	39.5	60.0	57.1	105.15	132.63	3,928	19,109
820	360,952	113,016	1,447,710	375,511	41.2	39.6	60.0	57.2	105.16	134.88	3,928	19,232
825	361,379	113,285	1,449,262	376,460	41.2	39.6	60.0	57.2	105.16	135.35	3,929	19,355
830	361,379	113,435	1,449,262	376,891	41.2	39.6	60.0	57.2	105.30	135.61	3,930	19,478
835	361,670	114,222	1,450,554	379,712	41.2	39.6	60.0	57.2	105.30	136.53	3,931	19,613

Table A.22: Detailed results for Commons Collections (775-835 seconds)

	Generated Sequences		Covered Inputs		Line Coverage (%)		Method Coverage (%)		Asserts/ Cyclomatic Complexity (6,848)		Failing Tests	
Time	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON	RANDOOP	RAIKON
840	361,670	114,269	1,450,554	379,915	41.2%	39.6%	60.0%	57.3%	105.38	136.61	3,932	19,748
845	361,691	115,614	1,450,625	384,830	41.2%	39.6%	60.0%	57.3%	105.38	138.43	3,932	19,883
850	361,765	116,329	1,450,912	387,399	41.2%	39.6%	60.0%	57.3%	105.39	139.38	3,933	20,018
855	361,777	117,493	1,450,912	391,724	41.2%	39.6%	60.0%	57.3%	105.40	140.12	3,934	20,149
860	361,777	117,935	1,451,327	393,503	41.2%	39.6%	60.0%	57.3%	105.41	140.86	3,935	20,306
865	361,796	118,729	1,451,331	397,548	41.2%	39.6%	60.0%	57.3%	105.41	141.39	3,936	20,463
870	361,866	119,616	1,451,348	399,334	41.2%	39.7%	60.0%	57.4%	105.41	143.38	3,936	20,620
875	361,895	121,166	1,451,407	404,758	41.2%	39.7%	60.0%	57.4%	105.44	145.27	3,937	20,777
880	361,942	122,394	1,451,657	409,183	41.2%	39.7%	60.0%	57.4%	105.44	146.61	3,938	20,930
885	361,942	122,536	1,451,657	409,771	41.2%	39.7%	60.0%	57.4%	105.46	146.81	3,938	21,007
890	361,942	122,796	1,451,657	410,652	41.2%	39.7%	60.0%	57.4%	105.46	147.10	3,941	21,033
895	362,033	122,913	1,452,247	411,058	41.2%	39.7%	60.0%	57.4%	105.48	147.25	3,941	20,964
900	362,076	122,940	1,452,275	411,131	41.9	39.7	60.0	57.4	105.50	147.41	3,940	21,040

Table A.23: Detailed results for Commons Collections (840-900 seconds)

Appendix B

Additional Tooling

During this master thesis project some tools were created to aid in several aspects of the process. This chapter describes the developed tools and what they are meant for.

B.1 bib2csv

The first tool developed was `bib2csv`. This tool was developed in collaboration with Márcio Coelho and is meant to take a BibTeX file properly annotated with keywords, and converts it to a `csv` file compatible with the ConExp tool [Yevtushenko, 2011]. This was done in order to facilitate the application of formal concept analysis in the context of literature revision. The tool is available as an open source Perl script at `https://github.com/TiagoVeloso/Bib2CSV`.

B.2 Fan-In/Fan-Out Calculator

This is another tool that was developed in the context of this master thesis was a simple Fan-In/Fan-Out Calculator generator. This tool was built on top of the already existing `javap` tool, which is a Java class disassembler. The `javap` tool is able to extract class dependencies. From this, all it was done was to apply a sequence of regular expressions to transform the output of `javap` into a useable format, `dot` or `csv`. It was written in Perl, the tool is available under an open source licence at `https://github.com/TiagoVeloso/CallGraph`.