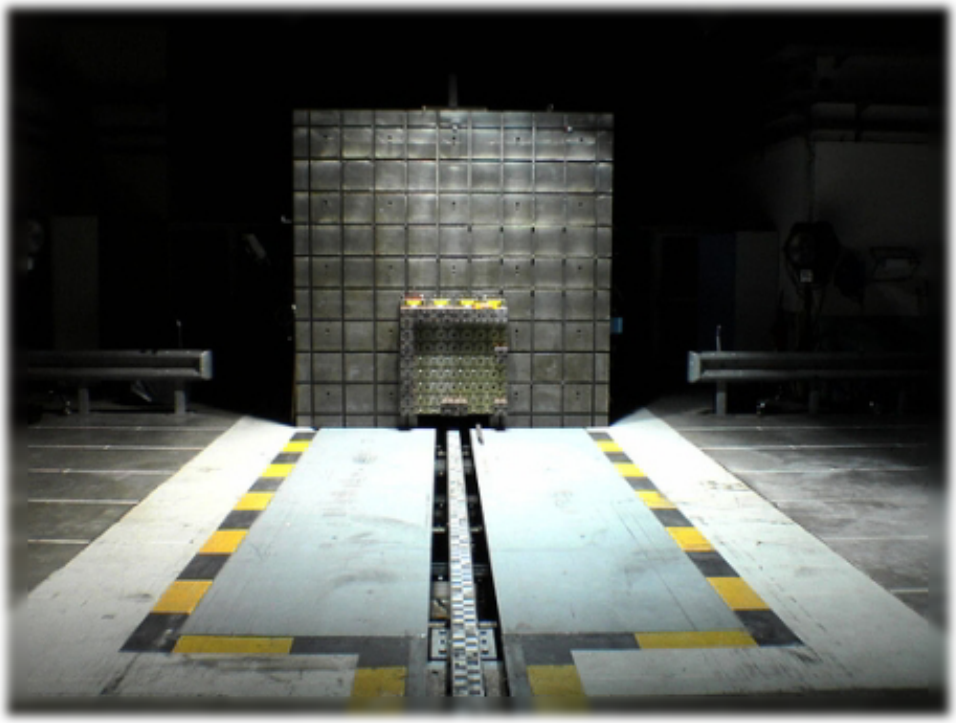
Path Analysis applied to Test Suites

Calculating the similarity of test cases

January 29, 2008



Willem van Gool

Path Analysis applied to Test Suites

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Willem van Gool
born in The Hague, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Info Support BV
Kruisboog 42
Veenendaal, the Netherlands
www.infosupport.nl

Path Analysis applied to Test Suites

Author: Willem van Gool
Student id: 1104233
Email: wvgool@xs4all.nl

Abstract

Because testing software is so important, many research has focused on how to create effective test suites. In addition, several metrics exist to estimate how thoroughly a system is being tested, which are commonly known as test adequacy criteria. However, less has been written about how to keep the size of a test suite manageable as the system evolves and more and more test cases are being added. Moreover, very little is usually known about how efficient test suites are actually implemented. In this thesis, we propose to measure the efficiency of a test suite by calculating the similarity between test cases. We base this similarity on the paths that are executed by each test case. Two case studies show that the calculated data can be used to obtain a general overview of how efficient certain parts of a test suite are implemented and can also be helpful to locate parts where redundant test cases may reside.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. Ing. L. Moonen, Faculty EEMCS, TU Delft
Company supervisor:	Ir. R. Brookman, Info Support
Committee Member:	Dr. Johan Pouwelse, Faculty PDS, TU Delft

Acknowledgements

I would like to take this opportunity to express my gratitude to all people that have helped and supported me during this project. First of all, I would like to thank Leon Moonen, who has been my overall supervisor and was always there to answer questions and provide detailed feedback on this report. Through his optimistic responses, I knew I was going into the right direction.

Secondly, I should thank Raimond Brookman and Marco Pil for all the fruitful discussions we had about technical issues. For some reason, we always needed twice the time we had scheduled for our meetings. Furthermore, I wish to thank Pascal Greuter and Marieke Keurntjes for their supportive attitude, even thought I ran over my own deadlines several times. I also must not forget to thank Arjan Seesing for his personal feedback on issues related to his framework.

A special thanks goes out to my parents, who have always been a solid support and have done so much to enable me to do the things I have done. I must say I dearly regret the fact that my grandma didn't make it to see this day. She would have loved it.

Willem van Gool
Delft, The Netherlands
January 29, 2008

Preface

When I chose to study Software Engineering in the summer of 2001, I really had no idea what software was about. At that moment, my main motivation was to pick a technical study which involved lots of math, and by becoming a software engineer, there was always that chance I would become a game-developer.

During my years at the TU Delft, as my understanding of computers and programming grew, I slowly became intrigued by the complexity and beauty of software. Why is it that something we have invented ourselves and have been studying for decades is still so hard to create without any errors?

Although I would have been glad to perform many other assignments, it was this thought that crossed my mind when I discussed the different subjects for my graduation project at Info Support. Another priority for me was to try and perform research in an area where few had gone before, to deliver some work with an innovative touch. Friends and family have always agreed that your graduation project is one of those rare opportunities to show what you are made off. I can only hope I have succeeded in doing that.

Contents

Chapter 1

Introduction

When I started my final year in September 2006, I chose to perform my graduation project at Info Support BV, a software development company in the Netherlands. They have developed a product called Endeavour, which is basically meant to provide more insight into the total development process of every running project using a web interface. This insight is obtained by showing, for example, the outcomes of nightly builds and the results of their test suites along with coverage measurements.

Since it is their desire to keep improving Endeavour, we started out a research project to look for tools or other innovative ideas that had not been integrated into their product, but for which it would seem to be effective to do so. The outcomes of this research motivated me to zoom in on the aspects of testing and especially on measuring the quality of a test suite.

The next section explains what we believe is meant by the quality of a test suite and why trying to measure it is so important. Hereafter, section 1.2 outlines our concrete problem statement, directly followed by our thesis objective. A complete outline of this paper is provided in section 1.3.

1.1 Test effectiveness and efficiency

In software development, there are basically two ways to improve the quality of a software product. The first is to try and prevent the introduction of bugs into the source code by creating clear, non-ambiguous specifications for a certain product before it is implemented. The second is to test the implemented system very thoroughly, which is the same as verifying that the software system functions according to its specifications.

In 1991, James L. Dalley concluded that the testing phase is generally responsible for more than 50% of total project costs [17]. Since software systems have only become larger and more complex since that day, while most testing is still performed by manually writing test cases, we may assume that this figure has gone up rather than gone done. Most of these costs are spent on salaries for programmers and testers that design and implement test cases by hand. Obviously, much money could be saved if the testing process was more automated, that is, if, for example, a large part of the test cases could be generated.

In a previous literature survey [41], we discussed, amongst other, some existing techniques that are applied to generate test cases and investigated their strengths and weaknesses. We concluded that many tools that are currently available have a good potential to be used as a way to test certain systems more thoroughly, but not to replace or reduce the work of implementing test cases by hand without the danger of compromising the overall quality of a test suite. We believe the quality of a certain test suite depends on two properties: its *effectiveness* and its *efficiency*.

Many research has focused on testing and how to do it properly [9, 49]. In most cases, proper testing means that the designed test suite has a high probability of revealing bugs in a software system. Since the goal of testing is to reveal as much faults as possible, a test suite that does so or at least has a high probability to do so can be said to be very *effective*. It follows instinctively that effective test suites result in software products with less faults and of higher quality.

Unfortunately, it is very hard to actually measure the effectiveness of a test suite, if possible at all. We cannot just count the number of faults it detects, since test suites do not have to actually reveal faults to be effective. Suppose, for example, we have created a test suite for a new product we are developing. During several runs of our test cases, we reveal a good amount of faults, fix them, and rerun the tests to be sure they are fixed correctly. After some time, all our test cases pass and the product seems to be working properly. At this time, our test suite is not actually revealing faults any longer, but clearly we have created a pretty effective test suite, for it helped us to reveal faults earlier: it's just that we fixed them.

One way to estimate the effectiveness of a test suite is by measuring *code coverage*. Generally, code coverage tells you how much code of a certain piece of software has been executed a certain way. We could argue that if a test suite A executes more statements of a certain piece of code or runs it more often with many different inputs than another test suite B, then A is probably more effective than B. This is because for a fault to surface, the code containing it must at least be executed once, and perhaps in a very specific way[1]. The advantage of measuring the effectiveness of a test suite is that such a measurement should provide a clear picture on the expected quality of the tested system.

A less frequently concerned property is the efficiency of a test suite. We define a test suite to be efficient if its effectiveness is obtained with as little test-effort as possible. Suppose two groups of people are to design a test suite for the same system. After several weeks, test suites A and B have been triggering almost the same set of faults, resulting in about the same set of bug-fixes, but test suite A contains a considerable smaller amount of test cases than B. Since the set of faults they have exposed are almost equal, we may argue that both test suites are probably equally effective, but test suite A is a lot more efficient.

The advantage of an efficient test suite is that it takes less time to run and causes only a minimal load on compilation and testing machines. Also, a very inefficient test suite may indicate that certain pieces of software were deliberately 'over-tested' because their correctness is essential (e.g. they belong to a highly critical system). Another possibility is that the

[1] That is, using a very specific set of inputs.

test case designers just did not think about the value of yet another test case too much. In any case, more insight is gained in how much test-effort was spent on certain pieces of the code which may be used to assess and steer the process of a certain development project.

1.2 Problem statement and general objective

Info Support tries to extract as much information in the form of concrete numbers and figures from a development project as possible. Currently, Endeavour provides an overview of the results of each compilation and test suite-executions, as well as an overview of code coverage measurements. Although the outcomes and coverage measurements give insight into the effectiveness of the test suites developed by their programmers, they have little knowledge about how efficient these suites were implemented.

Previous code surveys have also indicated that it is not uncommon for programmers to 'copy-and-paste' some of their own test cases and to only alter the input values of these test cases slightly, without paying attention to the added value of these tests. In doing so, they are creating less efficient test suites with a minimal increase in effectiveness. For these reasons, Info Support was very interested in a way to measure the efficiency of test suites. We then proposed to measure the paths that are executed by each test case in a test suite and compare these paths, since we assume that test cases that execute highly similar paths may be quite similar themselves. We will explain this concept and the techniques we use to measure and compare these paths in chapters 2 to 4.

ExsectJ In this thesis we also report on ExsectJ, an extension of the InsectJ framework developed by Alex Orso and Arjan Seesing [36]. ExsectJ is a prototype tool currently capable of extracting run-time paths from arbitrary Java programs and provides the ability to generally process or compare these paths using customized implementations. The tool has been designed and implemented in the same line as InsectJ: to be flexible and easy to extend. We use this tool to show a proof-of-concept for our theory on measuring the efficiency of a test suite. We provide detailed information about our tool in the appendices.

1.3 Outline

Our general solution to measuring the efficiency of test suites is explained in the next chapter. This chapter also summarizes some of the subjects that have been discussed in our previous literature survey and are applicable to this thesis project. Chapters 3 and 4 provide detailed technical descriptions that are part of this solution. In chapter 5 we evaluate our approach by performing two case studies. In chapter 6 we compare our approach with existing work. We conclude this paper in chapter 7, where we summarize our contributions and add some suggestions for future work.

Chapter 2

Measuring test suite quality

We have mentioned in the introduction that the primary focus of this thesis was to find a good method to measure the efficiency of arbitrary test suites. In this chapter, we will provide an overview of our solution to this problem. However, since the efficiency of test cases is highly correlated with its effectiveness, and because the effectiveness is generally the most important property of test suites, we will start with a more in-depth discussion on the general concept of testing.

In the first section, we will shortly discuss the implications of static and dynamic code analysis. We think it is important to discuss them here, because testing is actually a form of dynamic code analysis, and we feel it will be convenient to know the possibilities and limitations of these types of code analysis while reading the rest of this paper. The second section considers some techniques to make an assessment of the effectiveness of a test suite. In the last section we will go into the details of our own contribution: measuring the efficiency of test suites. To prevent misunderstandings, we will use Binder's terminology on errors, faults and failures during the remainder of this paper [9, p. 48]:

> *. . . A* failure *is the manifested inability of a system or component to perform a required function within specified limits. It is evidenced by incorrect output, abnormal termination, or unmet time and space constraints. A software* fault *is missing or incorrect code. When the executable code (translated from faulty code) is executed, it may result in a failure. An* error *is a human action that produces a software fault. . . .*

2.1 Potential and limitations of testing

Currently, the best way to locate faults in code is by simply performing manual code inspections. Unfortunately, even intensive code inspections cannot ensure that all faults are found and fixed. Errors made in complex code, such as code that is executed in a multi-threaded environment, are often overlooked and sometimes such errors do not surface as faults or failures for a long time because they only do in certain specific situations that are hard to simulate in a test suite.

In these situations, or perhaps generally, code analysis tools may help to find non-trivial faults. As the term suggests, such tools perform an automated analysis, often searching for and reporting on faults of a specific type. However, the problem of locating *all* faults in any non-trivial piece of software can be reduced to solving the *halting problem*, which has been proven to be undecidable [37]. Thus, it is not possible to develop a tool that will simply point out all our mistakes in the code, but this does not mean it is impossible to develop a tool that will automatically find *some* of the faults.

2.1.1 Soundness and Completeness

Consider any arbitrary program, which can be modeled as its own state space P. Within this set, there may exist a certain subset E which is the set of all *erroneous* states. An erroneous state is any state that is considered undesired and which can be reached only because of a fault in the program. Note that an erroneous may cause a system failure to occur, but does not necessarily have to.

Locating faults in a program is usually acquired by analyzing the states of a program and checking whether any erroneous states are encountered. If such a state has been reached, there is usually enough information available to locate the fault that caused the erroneous state to occur (although in a multi-threaded environment this may still be difficult, it should in theory be possible to locate a fault from an erroneous state when we have *all* information that somehow influenced the program to reach that state).

Tools that are capable of finding all erroneous states that may be caused by a certain type of fault are categorized as *sound* tools or algorithms. Note that such tools may also produce *false positives*: states that are reported as erroneous while they are actually valid. There also exist tools that avoid the production of false positives and only report those states that are ensured to be erroneous. These tools are called *complete*. Nonetheless, complete tools may produce *false negatives*, which means they cannot ensure that every erroneous state they are searching for is detected.

Ideally, a tool would be both sound and complete: detecting all erroneous states, no more, no less. This would mean such a tool would be capable of determining the set E in the program set P. However, the previous paragraph essentially tells us it is that is has been proven that it is not possible to systematically determine set E in any non-trivial program. Thus, a tool can not be both sound and complete at the same time, and there is always a trade-off to be made between the two when designing or choosing an algorithm with fault-finding capabilities.

The whole idea is visualized in figure 2.1. Next to the set P and E, the figure shows sets S and C, both representing a possible set of states that are reported on by a sound and a complete tool respectively. Set M may be a set of states that is reported on by a tool that is neither sound nor complete. When implementing a fault-finding algorithm, the obvious question we would ask ourselves is whether we should make it sound, complete, or somewhere down the middle. The answer to this question depends on the goals and purpose of the algorithm. The next few paragraphs will discuss some examples of sound and complete analysis tooling.

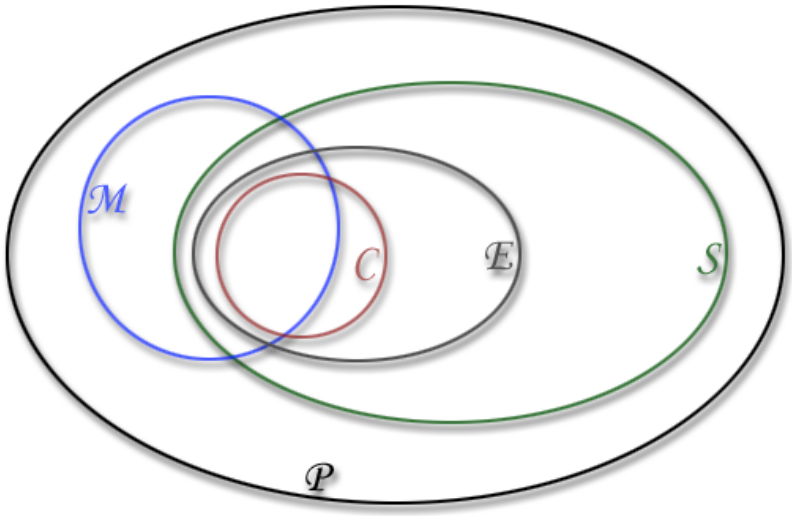


Figure 2.1: A program state space P with a set of erroneous states E. Fault-finding algorithms that return the sets S or C are respectively sound and complete.

2.1.2 Static Code Analysis

Static code analysis is a form of code analysis in which the observed code is not actually executed. The strategy behind applying static code analysis to locate specific faults in software is to try and *prove the absence* of such specific faults, or to return detailed information on the location of faults in the code if they are present and detected, which is called *formal verification* of programs.

Formal verification tools are often sound, which means that once they do not report any fault that it is capable to find, they have proven the absence of this type of fault and therefore we are *sure* that no failure will *ever, in any execution* be caused by that type of fault[1]. The downside is that on top of the implicit cost of being sound (the possibility of producing false positives), many sophisticated provers require huge amounts of memory and lots of processing time to complete. For example, many tools have to solve large amounts of complex theorems [19, 12, 7, 22, 5, 6] or analyze a large part of the state space (P) of a program [14, 43, 42], which is typically huge or infinite for any non-trivial program.

2.1.3 Dynamic Code Analysis

Dynamic code analysis is the form of code analysis in which the code that is being analyzed is observed while it is executing. Testing is an obvious example of dynamic code analysis, but also path profiling, memory profiling and debugging are forms of dynamic code analysis which are not the same as testing.

[1]Of course, this statement only holds if the applied prover functions correctly and the code has not been altered since the prover has been applied.

By executing the program under observation, we avoid most of the memory-problems that can occur in static code analysis, because we do not consider all states of a program, but only the ones that we encounter during that specific execution. However, observed behavior in one execution is never guaranteed to be the same in the next execution. Although we assume most programs will act the same each time they are executed with the same inputs, multi-threading and dependencies of a program to its environment (e.g. files, the operating system, other processes and hardware) may change the behavior of a program from one execution to another in a very unpredictable way. Moreover, the dynamic analysis performed during run-time often influences the behavior of a program directly, especially in its run-time speed or memory use. This is a very delegate problem, since we are never able to accurately measure a property cleanly when we change this property to measure it.

These apparent limitations render tools that are based on dynamic code analysis complete at best, which implies they may locate some of the faults they are designed to find, but may also remain silent while such faults do exist. This directly relates to testing, where test cases may reveal the faults they are designed to find, but may just as well miss these faults. Consequently, no test suite will ever actually guarantee that a product is fault-free, or as Dijkstra once put it[2]:

> *Testing shows the presence of errors, not their absence.*

2.2 Effectiveness of a test suite

The theory explained in the previous section really sheds some new light on the effectiveness of test suites that was introduced in the first chapter, and especially on the difficulty that arises when trying to measure it. What test cases actually do, is *checking* whether some states of the methods, classes or modules they test are correct, while using some specific inputs in a certain environment. If a test case passes, then all the states that are checked have been verified. If it fails, it has detected an erroneous state. Note that it is possible for a test case to allow a program or object to reach an erroneous state and not detect it.

2.2.1 Effectiveness defined

Since a test suite is just a collection of test cases, a test suite is checking all the states that its contained test cases are checking. Hence, we can model a test suite as the set $T \subseteq P$ in figure 2.1 (not shown), where T contains all the states that are checked. This implies that the states in $T \cap E$ will be detected by the test suite.

It may be clear at this point that it is desirable to have a test suite that verifies as much states as possible. In other words, we would like T to be as large and as close to P as possible. Unfortunately, it is not practical and usually impossible to aim for $T = P$. Nonetheless, the true effectiveness of a test suite obviously depends on the relative sizes of T and P.

[2] Unfortunately, we were unable to find a published article in which Dijkstra mentions this widely cited statement.

Effectiveness. Let $T(\Theta) \subseteq P$ be a set of states that are checked by a certain test suite Θ within a program state space P, then the effectiveness τ of Θ can be calculated by

$$\tau(\Theta) : \frac{size(T(\Theta))}{size(P)}$$

2.2.2 Measuring effectiveness

It follows from the previous definition that we will have to measure both T and P to calculate the effectiveness of a certain test suite. However, because of the sizes of these sets it is highly impractical if not impossible to do this. The alternative is to try and get a good estimation of $\frac{size(T)}{size(P)}$ or at least of the size of T. Trying to estimate the effectiveness of a test suite has been a widely investigated topic in computer science, and there have been invented several methods to do this. We discuss two common approaches below.

2.2.3 Code coverage

In the introduction we have already shortly discussed the concept of code coverage. We will look at it more closely in this section. Code coverage results are expressed as a percentage and essentially tell you how much of the code has been executed a certain way. The exact meaning is defined by the type of coverage that is measured. Arnold Zanderink from the UT Twente mentions four very regular used types of coverage [48, p. 1-2]:

Statement coverage. *is used to measure the percentage of statements that has been executed during the testing. Full statement coverage is reached if all the code under test has been executed at least once...*

Branch coverage. *measures the percentage of branches that has been reached during the testing. Before reaching a new branch, like the body of an if-statement, a condition has to be evaluated. Full branch coverage is reached when at every condition statement both the true and the false branch is tested...*

Condition coverage. *is used to check if both the true and the false condition of every subexpression within an expression have been tested at least once...*

Multiple-condition coverage. *coverage is a more detailed kind of coverage than condition coverage, because it checks if all possible combinations of conditions of subexpressions within an expression have been tested...*

Note that the types are placed in order of 'complexity': generally, a 100% score on multiple-condition coverage is a lot harder to achieve than full statement coverage, since in the latter case, the code has to be executed more times, with different inputs or parameters, to achieve the same coverage score. In this respect, we may add the most complex coverage type to the end of the list:

Path coverage. *is defined as all the paths that have been executed within a method, divided by the total number of paths.*[3]

[3]The term *path* can also relate to a path of execution within a program, instead of just within a single method. A more in-depth explanation can be found in chapter 3.

However, it is not guaranteed that a certain coverage score obtained in a single execution on a certain coverage type is always higher than if the coverage were measured using a more complex type, and vice versa. More formally, let c_i be a coverage type of a certain complexity i, where c_i is more complex than c_k if $i > k$. Let $0 \leq s_e(c_i) \leq 1$ be the coverage score obtained in some execution e while measuring c_i, then it is safe to say that for *most* e the statement $s_e(c_i) \leq s_e(c_k) \qquad i > k$ holds, but *not for all* e.

For instance, there are some particular cases in which it is possible to obtain a higher branch coverage than statement coverage. To clarify this example, consider the control-flow graph in figure 2.2. The graph in this figure has five distinct branches and two possible paths. Suppose the body of the `if`-statement, block C, contains n statements. When in a certain execution path A-B-D-E is taken, our branch coverage measurement would become $\frac{3}{5}$. Assuming both B and D contain one statement, our measured statement coverage would become $\frac{2}{2+n}$. Solving for n leads us to the conclusion that the measured statement coverage would be *lower* than the measured branch coverage if $n > 1$. Note that a more in-depth explanation concerning control-flow will given in chapter 3.

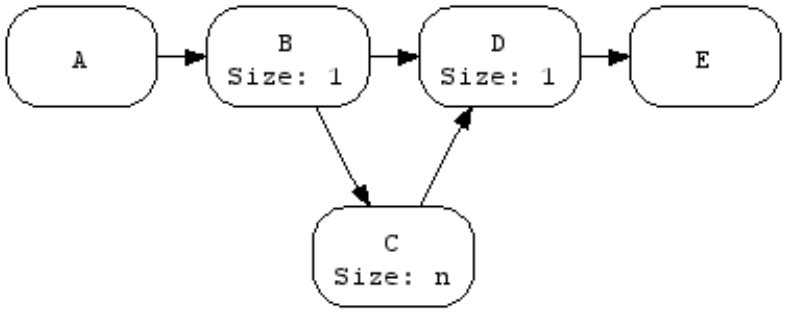


Figure 2.2: Simple control-flow graph of a method where the path A-B-D-E generates a higher branch than statement coverage score if `n > 1`

It is, nonetheless, generally assumed that a certain coverage score obtained on a more complex type implies more thorough testing than the same score obtained on a less complex type, simply because this is *usually* the case. We also assume, intuitively, that the more thorough a piece of code is tested, the better chance we have in finding faults, and thus the more effective a certain test suite is conceived to be.

To be more precise, when we say one test suite is executing a certain piece of code more thoroughly than another one, we actually mean that it checks more states of the program under test. Hence, a higher coverage score generally implies a larger T, and a larger T implies more effectiveness of the test suite: a higher probability of finding faults.

2.2.4 Mutation adequacy

Another way to estimate the effectiveness of a test suite is by introducing custom faults and to see if the current test suite detects them. This is called *fault injection* [45] or *mutation*

testing. The code in which an error was intentionally planted is called a *mutant*. If the error is detected, the mutant is said to be killed by the test suite; otherwise, it is said to be still alive. The *mutation score* is based on the number of killed mutants compared to the number of mutants that are not equivalent to the original program. Unfortunately, it is generally not possible to detect whether a mutant is equivalent to the original program [37], so mutants have to be assumed to be different.

Of course, we feel that the more mutants are killed, the higher the effectiveness of the test suite must be. The embedded strategy is actually as follows. Each time a new mutant is created, one which is not equivalent to the parent program, a new randomly chosen part $e \in P$ is added to E. The systematic checking whether the test suite kills the mutant is the same as checking whether e is in T. It follows statistically that the more mutants are killed, the larger T must be, and thus the more effective the test suite actually is.

2.3 Efficiency of a test suite

So far we have primarily focused on the effectiveness of a test suite. Partly, this is because the effectiveness of a test suite is the most important property: it completely embodies the whole reason for creating tests, namely detecting faults in software. Another reason for discussing the effectiveness of a test suite first is that it greatly increases the understanding of what testing is and what test suites are actually doing. Now that we have a clear definition of effectiveness, it is much easier to also define the efficiency of a test suite.

2.3.1 Efficiency defined

Whereas effectiveness denotes a certain degree up to which someone or something reaches its goal, efficiency denotes the amount of resources that are used to perform a certain job. With respect to testing, we may consider each line of test code or each test case as a single resource unit: since writing and running test cases is expensive, we would like to reach a certain effectiveness with a minimal set of test cases. All other things remaining equal, less test cases means a more efficient test suite.

Efficiency. Let $size(\Theta_k)$ be the number of test cases contained by test suite Θ_k and let Θ_{min} be the minimal set of test cases that is required to obtain the effectiveness of Θ_k, then the efficiency ϕ of Θ_k can be defined as:

$$\phi(\Theta_k) = \frac{size(\Theta_{min})}{size(\Theta_k)}$$

2.3.2 Measuring efficiency

Counting the number of test cases in a test suite is not very difficult, but how do we determine the minimum amount of test cases? Obviously, this problem can be reduced to just finding *a* minimal set of test cases (Θ_{min}). This problem is generally known as the *test suite minimization* or *test suite reduction* problem [46, 28, 9].

Test suite reduction techniques can be applied to existing test suites, but are also commonly integrated within test case generation tools [33, 34, 10, 35, 1, 30, 8, 38]. Because they are automatic, test case generation tools have the capability of creating huge amounts of test cases, if necessary. However, for reasons already explained, it is generally not desired to maintain and run test suites of tremendous size, even though such test suites might be quite effective. Instead, most test case generation tools try to *evaluate* the test cases they create, so that in each iteration the best test cases are selected and the rest is discarded. The approach that is used to generate new test cases in each iteration varies from random input selection [33] to deterministic calculation of inputs [34, 10] to the use of metaheuristics [35, 1, 8, 38], in which the best test cases are typically used to create new ones.

The basic problem is how to determine which tests of a test suite belong to the minimal test suite and which do not. This problem could be dealt with by systematically comparing two test cases and see if they are *identical*. According to Binder [9, p. 47]:

> *A test case specifies the pretest state of the [tested unit] and its environment, the test inputs and conditions, and the expected result.*

He mentions later (p. 768) that test cases can be considered identical when their input and expected output for a specific interface are identical, but that even such identical test cases are not necessarily *redundant* because:

> *... Test cases are sometimes repeated to perform a necessary setup, or to accomplish a time delay, or because coding test cases is easier than developing iterative control in the test driver of script.*

We can add to this last statement that two seemingly identical test cases may be run on a unit that has been brought into different initial states. Therefore, we may suggest that two test cases are only *truly* identical when their inputs and outputs for a specific interface, *and* the environment in which they are executed are exactly equal. It may be obvious that this situation will not occur very often, which implies that removing a fair amount of test cases generally comes at the cost of some loss in the effectiveness of the test suite. However, the obtained cost-reductions by the size-decrease of the test suite may be worth this loss when it is minimized.

The most accurate way to remove (or turn off) a set of test cases from an existing test suite is by manually inspecting the test cases and let a group of programmers or other experts of the system decide which test cases are similar enough to be able to justify discarding a few of them. Nonetheless, it is, of course, not a very practical or economical solution, for test suites may contain up to tens of thousands of test cases, and sometimes it is even hard to decide which test cases are redundant in a small suite. For this reason, there have been developed several ways to reduce the test suite in a partially or fully automated way.

One way to estimate whether two test cases are identical is by comparing the paths they execute in the tested system. This technique has been explored and Binder makes note of it as a type of *unsafe reduction* of a test suite [9, p. 771]. The basic idea is that the *probability* of two test cases exposing the same fault is correlated with the path it executes in the code,

but it is by no means ensured that both test cases are identical if both executed paths are. For instance, both test cases may be testing on different boundary values, which implies that blindly removing test cases from a test suite that execute the same or similar paths will typically affect the overall effectiveness of a test suite in an unpredictable way.

However, we assume that comparing the executed paths of different test cases may still be quite effective when it comes to determining which test cases are quite similar. We do not aim to actually calculate a minimal test suite, but rather aim to merely *indicate* which groups of test cases are executing similar paths within the same method. This way, a programmer or manager will get some more insight into how efficient certain methods and classes are actually tested, which is our general objective as stated in the previous chapter. Furthermore, we expect, for example, that 'copy-and-paste-test cases' will show high similarity during our measurements, making it easier to detect these practices.

To be able to calculate the path-based similarity of test cases, we need to extract these run-time paths and associate them with the appropriate test cases. In the next chapter, we present our approach to extracting run-time paths from arbitrary Java programs. In chapter 4 we first explain how we associate these paths with the right test cases. We also propose a new way of calculating the similarity of these paths and thus of their associated test cases.

Chapter 3

Extracting Run-time Paths

Extracting or recording run-time paths is known as *path profiling*. Path profiling is, in fact, a form of dynamic code analysis in which we observe which path is being executed in a program or method.

Unfortunately, we failed to find an existing (open-source) tool that could be adopted for our project. However, we did run into some very interesting work by Ball and Larus [4] on path profiling that has enabled us to implement a path profiling algorithm ourselves. Moreover, InsectJ [36], a framework that made it relatively easy to instrument arbitrary Java programs, had just recently been published. For these reasons, we decided to design and implement our own tool, ExsectJ, capable of extracting the run-time paths of Java programs in a fairly efficient manner.

The advantage of creating the tool ourselves was the inherit freedom and flexibility. Naturally, all the source code was available and the extra functionality that was needed later (the implementation of a path-comparison algorithm) could be added with relative ease, since the code was self-developed and fully understood.

The downside of choosing this option was, of course, the extra time we had to spend on design, implementation and testing. Because of limited time constraints, all the extra time spent on developing the tool was subtracted from the time we had to perform the rest of our research.

3.1 The concept of path profiling

So far we have mentioned that test cases execute paths in a system under test and that we may extract these paths using path profiling. In this section, we will look a bit closer at what paths actually are and we will describe the basic theory on how to determine which path is being executed in a running program.

3.1.1 Intra-method versus inter-method profiling

Paths essentially describe the sequence of instructions or statements that are executed in a program. We can discriminate between two types of paths: those that only describe the sequence of instructions within a single method, and those that describe the whole sequence

of instructions that are executed within a single thread. Recording the former type of paths is called ***intra**-method* profiling, whereas recording the latter type is called ***inter**-method* or *inter-procedural* profiling.

The original efficient path profiling algorithm reflects an intra-method path profiling algorithm. Naturally, inter-method path profiling algorithms provide more detailed information about a program's execution, and since the original article was published the algorithm has been extended to support such profiling [24, 31].

However, we found two reasons to stay with the original algorithm. First of all, we doubted the value of the extra information obtained by the inter-method alternative. Since each test case is typically designed to test a single method, we argued that the information provided by intra-method paths would be sufficient. Secondly, regarding the limited time constraints of our thesis, we reasoned that it would simply take too much time to implement the additional functionality.

3.1.2 General aspects of instrumenting a program

If we are to determine the sequence of instructions - the path - that is being executed inside a method, we need to insert some instructions into the observed program before it is run, such that these instructions provide us with the information required to determine this sequence. Inserting code into a program this way is also known as *instrumenting* a program.

Note that the instrumentation of a program itself is a form of static code analysis, including the whole process of calculating where to insert the code into another program. However, the instrumentation of a program enables the observation of a program's behavior during run-time, which is the dynamic code analysis part of the process. This process is visualized in figure 3.1.

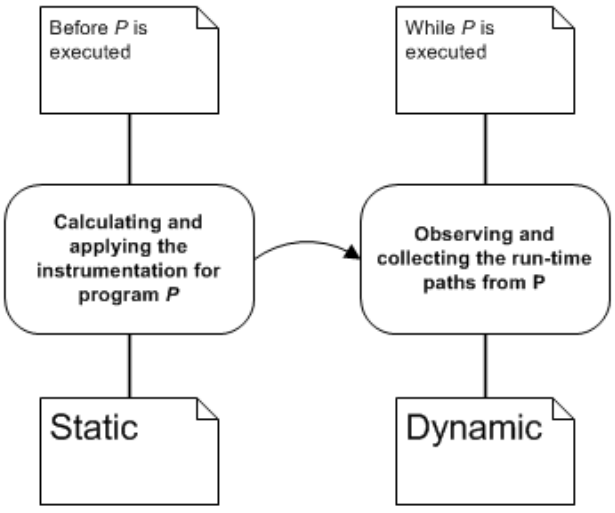


Figure 3.1: Schematic representation of path profiling a program P

If we instrument a program to observe the paths that are executed, we need to make sure the natural sequence of instructions and the values of a program's own variables are not altered as a result of our instrumentation. Hence, we need to make our instrumentation completely *transparent* with respect to these two properties.

At this point, two questions remain: what information do we need to extract from a program to obtain a certain path, and how do we instrument a program such that it meets these requirements? The answer to the first question is provided in the next paragraph. The second question is answered in the following section, in which we describe the path profiling algorithm of Ball and Larus.

Introduction to control-flow

Until now the term *path* has been rather vague. In order to provide a clear answer to the questions above, we need to define exactly what a path is. The sequence in which instructions or statements are executed are often referred to as the *control-flow* of a program. Control-flow analysis is a form of static code analysis in which each possible sequence of instructions in a program or method is considered. The whole collection of possible instruction- or statement-sequences can be captured in a *control-flow graph*, or CFG for short.

Consider, for instance, the code in figure 3.2. By static analysis, the control-flow graph of this method can be constructed as depicted in figure 3.3. This graph immediately gives a good insight into the different ways the statements of the method can be executed.

```
public int indexOf(Object o) {
    for (int index = 0; index < objectArray.length; index++) {
        Object currentObject = objectArray[index];
        if (o == currentObject || (o != null && o.equals(currentObject)) {
            return index;
    }
    return −1;
}
```

Figure 3.2: Returning the index of an Object instance

Each node in this graph is called a *basic block*. In a book on compiler design we found the following definition of a basic block [20, p. 320]:

> *A **basic block** is a part of the control-flow graph that contains no splits (jumps) or combines labels. It is usual to consider only **maximal basic blocks**, basic blocks which cannot be extended by including adjacent nodes without violating the definition of a basic block. . . . the control-flow inside a basic block cannot contain cycles.*

From this definition we can deduce that basic blocks have only a single *entry point*, which is the first instruction or statement of the block, and one or more *exit points*, which can be any

of the items. Note that the blocks A and J do not contain any statements: they reflect basic blocks but are added to denote a single entry- and exit-point of the method. The directed edges between the blocks represent the possible control-flow from one block to another.

At this point, we can define a path within a certain method to be any sequence of basic blocks of its corresponding control-flow graph that could be constructed by starting at the entry-block and ending at the exit-block, and of which any two succeeding blocks b_i and b_{i+1} are connected by a directed edge $\{b_i, b_{i+1}\}$ in the graph. For example, the sequences A-B-C-I-J or A-B-C-D-E-G-C-D-H-J are valid paths for the method of figure 3.2.

Furthermore, the edge connecting node G and C is called a *back-edge* because it introduces a cycle in the graph, which implies a loop is present in the associated method. Cycles or back-edges result in an infinite set of paths, since the number of occurrences of C-D-E-(F-)G within a path is theoretically unlimited in the context of control-flow analysis. In reality, the number of iterations is limited by the size of `objectArray`. Such information can be obtained by performing *data-flow analysis*, but that topic is outside the scope of this paper.

Using the control-flow graph for instrumentation

Since the control-flow graph describes, by definition, any path that can possibly be executed within a certain method, we can use it to determine where to place the instrumentation inside a method. One possibility, for example, is to use the control-flow graph to insert some extra instructions inside each basic block, such that the execution of such a block can be observed during run-time. In a non-parallel setting, the recorded path would simply be the recorded sequence of blocks. Although Ball and Larus propose a more efficient solution, their algorithm still requires the control-flow graph to be constructed before the places of the instrumentation-code can be calculated and applied. For this reason, we will discuss how to construct a control-flow graph in section 3.3, but first we will spend a few words on the framework we have used to instrument Java programs.

3.2 The InsectJ Framework

For instrumenting arbitrary Java programs we use InsectJ [36]. InsectJ is a framework that implements the functionality to easily instrument class-files and a class's methods in particular. The concept of this framework is to take away the burden of having to implement your own module to read the appropriate class-files from disk, convert them to instances that can be instrumented, etc. We refer to the original article for an in-depth description of the framework. Furthermore, appendix A focuses more on how we have extended this framework in our self-developed tool.

The most important motivation to adopt this framework was that it took away the extra work that was otherwise needed to set up an instrumentation framework ourselves, just as it intends to do. This way, we could concentrate on the implementation of the profiling algorithm itself. However, the reason we discuss this here is because InsectJ works with

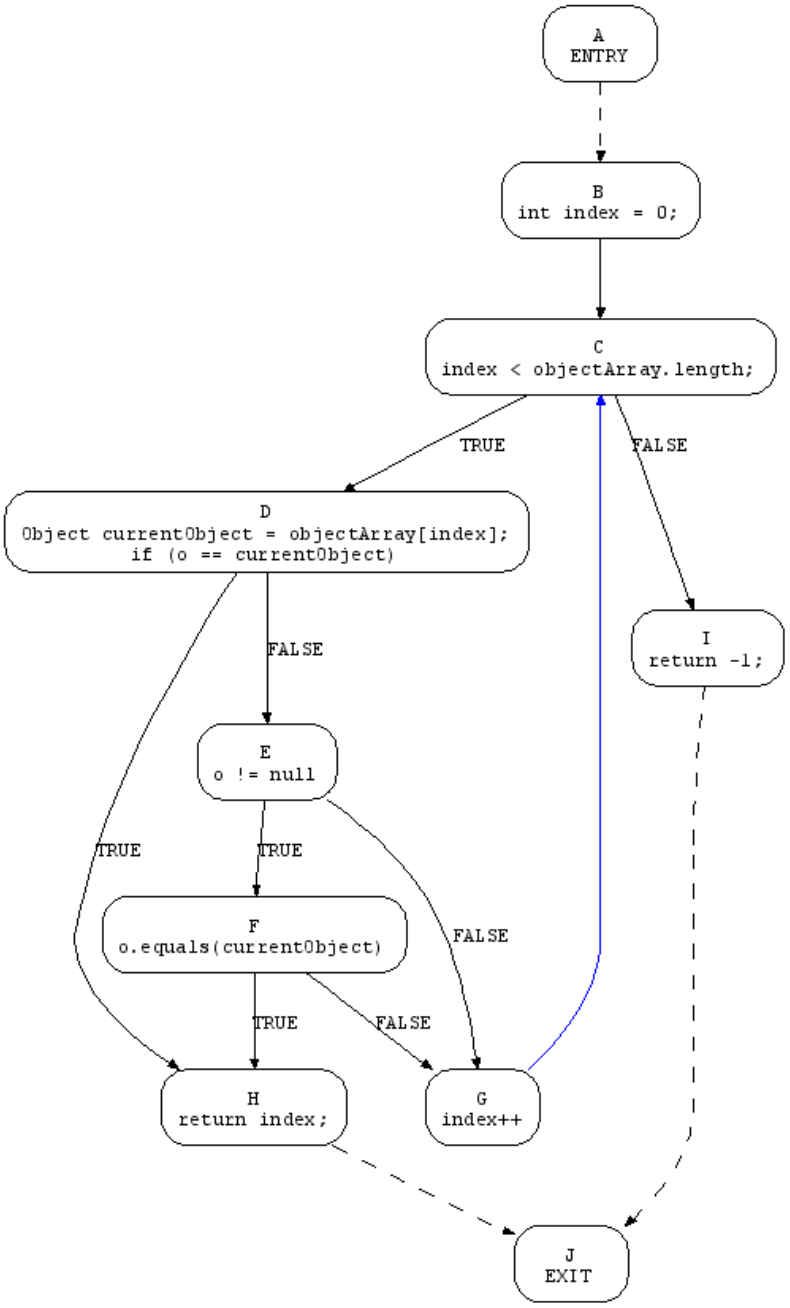


Figure 3.3: High level control-flow graph

the Byte-Code Engineering Library, or BCEL for short[1]. BCEL is a layer between the byte-code representation of a class-file and a Java-program that needs to manipulate this class at the byte-code level. This implies we will be performing the instrumentation - and inheritly observing the path-executions - at the byte-code level, not on the source code level.

The advantage of instrumenting at the byte-code level is that it is more flexible than instrumenting the source code of a program. First of all, since we are not bounded to high level language constructs, it is easier to manipulate the instructions and control-flow where needed. For the same reason it is also usually more efficient. And, quite trivially, the source code does not have to be available, which can be a big advantage when instrumenting third-party libraries.

The disadvantage is that it will be more difficult to retain a mapping to the source code, if it is available. For example, if the generated path profiles are to be used in a development environment, it may be desirable to show which paths have and have not been executed in the form of statements at the source code level. Moreover, problems can occur because the graphs representing control-flow at different levels of a program are not ensured to be equal.

For instance, Java's boolean-invert operator - `!` - is not available at the byte-code level, so it is converted to an `if-else`-construction in the assembly. Optimizations performed by a compiler may cause the divergence of the two control-flow graphs to become even greater. These problems are somewhat mitigated by the fact that when a program is compiled using the `-debug` option, the resulting Java class-files will contain the information about which instructions in each method belong to a particular line of the source code.

3.3 Constructing the control-flow graph

As explained on page 3.1.2 we will have to construct the control-flow graph before we can start instrumenting a method. Because of our choice to instrument Java programs at the byte-code level, it follows naturally that we have to construct the CFG at the byte-code level as well.

This section has been divided up into two parts. In the first part, we will explain how the control-flow has been defined on the byte-code level. The second part embodies a discourse about the actual construction of the graph through a step by step approach.

3.3.1 Control-flow at the byte-code level

To illustrate how methods are structured and how the control-flow is defined at the byte-code level, we will use one simple method as an example, which is depicted in figure 3.4. Assume this is a method that takes the name of a file and a sentence, and attempts to write every word of this sentence as a separate line in the file. If it succeeds, it returns `true` and `false` otherwise.

The control-flow at the byte-code level of a method is determined by two essential pieces: the *instruction-list* and the *exception-table*. This information can be visualized by running

[1] Also see `http://jakarta.apache.org/bcel/`

```
public boolean writeToFile(String fileName, String sentence) {
        StringTokenizer tokenizer = new StringTokenizer(sentence);
        FileWriter writer = null;
        try {
                writer = new FileWriter(fileName);
                while (tokenizer.hasMoreTokens()) {
                        writer.append(tokenizer.nextToken());
                        writer.append(NEW_LINE);
                }
                writer.flush();                                          10
                writer.close();
        } catch (IOException e) {
                return false;
        }
        return true;
}
```

Figure 3.4: Writing the words of a sentence to a file

Java's default disassembler at the command line (`javap -c %classfile%`), and if we apply it to our example method, the result is somewhat similar to what is shown in table 3.1.

The instruction-list is divided into several blocks so it's not too hard to figure out which parts of the source belong to particular parts of the compiled code. When a method is called, the Java Virtual Machine (JVM) initializes a stack and stores the method's arguments, if present, in the method's variable table (not shown in the table). Hereafter, the first instruction is loaded and execution of the method begins.

The instructions are typically processed one after another, from top to bottom. We refer to this type of control-flow as *normal* control-flow. The second type is determined by special jump-instructions, such as `goto`, `if` or `tableswitch`. These are the type of instructions that may target any instruction in the list and may therefore redirect control-flow accordingly. The third type are `return`-instructions, which will, as expected, stop a method's execution by returning the current top-value of the stack.

The last and special type is *exceptional* control-flow, which is determined by the throwing and catching of exceptions[2]. Exceptional control-flow is basically determined by the information from the exception-table. Every row in this table contains the data about a range of instructions for which a handler exists, and for which type of exception this handler is to be executed. In terms of Java source code, the range of instructions is typically the code stemming from the `try`-block, and the handler-code is defined by a `catch`- or `finally`-block. This is confirmed by the code in table 3.1, where the singular row in the table defines a handle at instruction 64 for exceptions of type `java.io.IOException`, should one be

[2] Actually, we mean *throwables*, since Java makes a clear distinction between, for example, the `Error` type and the `Exception` type, which are both subclasses of the `Throwable` type. However, for simplicity, we will solely use the term *exceptions*.

```
Instruction-list
      0  new #29;            StringTokenizer
      3  dup
      4  aload_2
      5  invokespecial #31;  StringTokenizer(String)
      8  astore_3
      9  aconst_null
     10  astore 4

     12  new #34;            FileWriter
     15  dup
     16  aload_1
     17  invokespecial #36;  FileWriter(String)
     20  astore 4
     22  goto 44

     25  aload 4
     27  aload_3
     28  invokevirtual #37;  StringTokenizer.nextToken():String
     31  invokevirtual #41;  FileWriter.append(CharSequence):Writer
     34  pop
     35  aload 4
     37  getstatic #18;      NEW_LINE: String
     40  invokevirtual #41;  FileWriter.append:(CharSequence):Writer
     43  pop

     44  aload_3
     45  invokevirtual #45;  StringTokenizer.hasMoreTokens():boolean
     48  ifne 25

     51  aload 4
     53  invokevirtual #49;  FileWriter.flush()
     56  aload 4
     58  invokevirtual #52;  FileWriter.close()

     61  goto 68

     64  astore 5
     66  iconst_0
     67  ireturn

     68  iconst_1
     69  ireturn

Exception table
  Range  Target              Type
12 - 58  64                  java/io/IOException
```

Table 3.1: Instruction-list and Exception-table

thrown within the range of instructions 12 to 58 (inclusive). Exceptions may be thrown for a number of reasons, which we have summarized as follows:

1. The JVM throws an exception, which can be caused by two different reasons, namely:
   - the current instruction requires one or more values or references of a certain type which are not present on the stack. In other words, the state of the stack is invalid. A good example of such an exception is the `NullPointerException`, which is thrown whenever a reference to an instance is expected but a `null`-value is found.
   - execution cannot continue because of a serious problem, in which case an instance of `Error` is thrown (e.g. `OutOfMemoryError` or `StackOverflowError`).
2. An exception is thrown using the `throw`-instruction.

In any case, execution stops at the point where the exception was thrown, and the reference to the instance of the exception is stored into the variable-table. The JVM is signaled to consult the exception-table next. The rows of this table are enumerated top to bottom, and the target instruction of the first row that results in a match is executed. A match occurs when the instruction where the exception was thrown lies within the covered range and the exception itself is of the type specified by this row.

When no match is found, the JVM stops execution of the current method, stores the exception's reference in the variable-table of the calling method and tries to find a match in the exception-table of this method. This process is recursively iterated until either one method catches and handles the exception or when the program exists because the exception was not caught in the running thread at all.

3.3.2 Inferring the graph from byte-code

In this section, we will infer the CFG of the example method from the previous section (table 3.1). Constructing the CFG from the instruction-list is not a new concept and is widely applied in the area of compiler design [20, 13]. Despite the existing literature on the subject, we found it was not too hard to develop our own approach for creating the CFG from Java byte-code, and we will outline this approach below. Moreover, the graph that we create is slightly different than a typical CFG, and we will do so in several steps.

In the first step, we will build a graph that will represent the control-flow between individual instructions. In the second step, we will merge the blocks containing the instructions into maximal basic blocks where possible, as defined by the definition of a basic block on page 17. We enhance the graph in the third step, in which we add an entry- and an exit-block to ensure the graph has an entry-block with no incoming edges and only a single exit-block. Finally, in the last step, we add a block which is not very common (to our knowledge not present in other graphs at all), which we call the *catch*-block. This block is added to explicitly model the exit of a method because an exception was thrown in a callee (or also remained uncaught there) as a set of separate paths in the graph. The whole concept will be discussed in more detail in step four.

3.3.3 Step one: single-instruction blocks

In the first step, we simply create a separate block for each instruction while traversing through the instruction-list. In a second iteration through this list, we analyze the control-flow between these instructions and create edges between the blocks accordingly.

A part of the resulting graph is shown in figure 3.5. Each block is represented by a rounded box, whereas the normal arrows represent normal control-flow and jumps and the dashed arrows represent exceptional control-flow.

3.3.4 Step two: merging to maximal blocks

The graph created in step one can be simplified by merging some of the blocks into maximal basic blocks. This is highly usual, because a control-flow graph is commonly expected to be build from maximal basic blocks. Moreover, with a few exceptions, the resulting blocks in the graph will reflect the same basic block at the source code level. Paths deduced from this graph, although still reflecting control-flow between instructions, will therefore more naturally reflect paths at the source code level as well.

As we were later pointed out, the process of merging single instructions into (maximal) basic blocks is essentially the same thing as creating a data-flow analysis graph in minimal *static single assignment* form [15]. However, before discovering this literature, we had already re-invented this trick and set up a proof to show there exists only one single control-flow graph containing the maximal basic blocks. Since we do not focus on data-flow analysis here, we will outline this proof and a textual description of the merging process below. If the reader is already familiar with this process, however, he may proceed with **step three**.

Property definition of merging blocks. Let b be a basic block, $S(b)$ be the set of blocks targeting b and $T(b)$ the set of blocks that are targeted by b. When a block x is a source-block of y, then it follows that y is a target block of x, and vice versa. Formally we write:

$$x \in S(y) \leftrightarrow y \in T(x) \tag{3.1}$$

Now we define whether block x can merge with block y as follows:

$$mergable(x, y) : \begin{cases} 1 & S(y) = \{x\} \wedge (T(x) - \{y\}) \subset T(y) \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

The first statement of the equation demands that the only block that is targeting y is block x. This ensures that x dominates y and that the merged block will still conform to the definition of a basic block. The second statement requires that every target-block of x with the exception of y is also a target-block of y, which brings us to the following theorem.

Theorem 1. Provided with equation 3.2, a certain block s can only merge with at most one of its target blocks. Formally, we denote this as:

$$(\sum_{i=0}^{n} mergable(s, t_i)) \in \{0, 1\} \qquad t_i \in T(s) \tag{3.3}$$

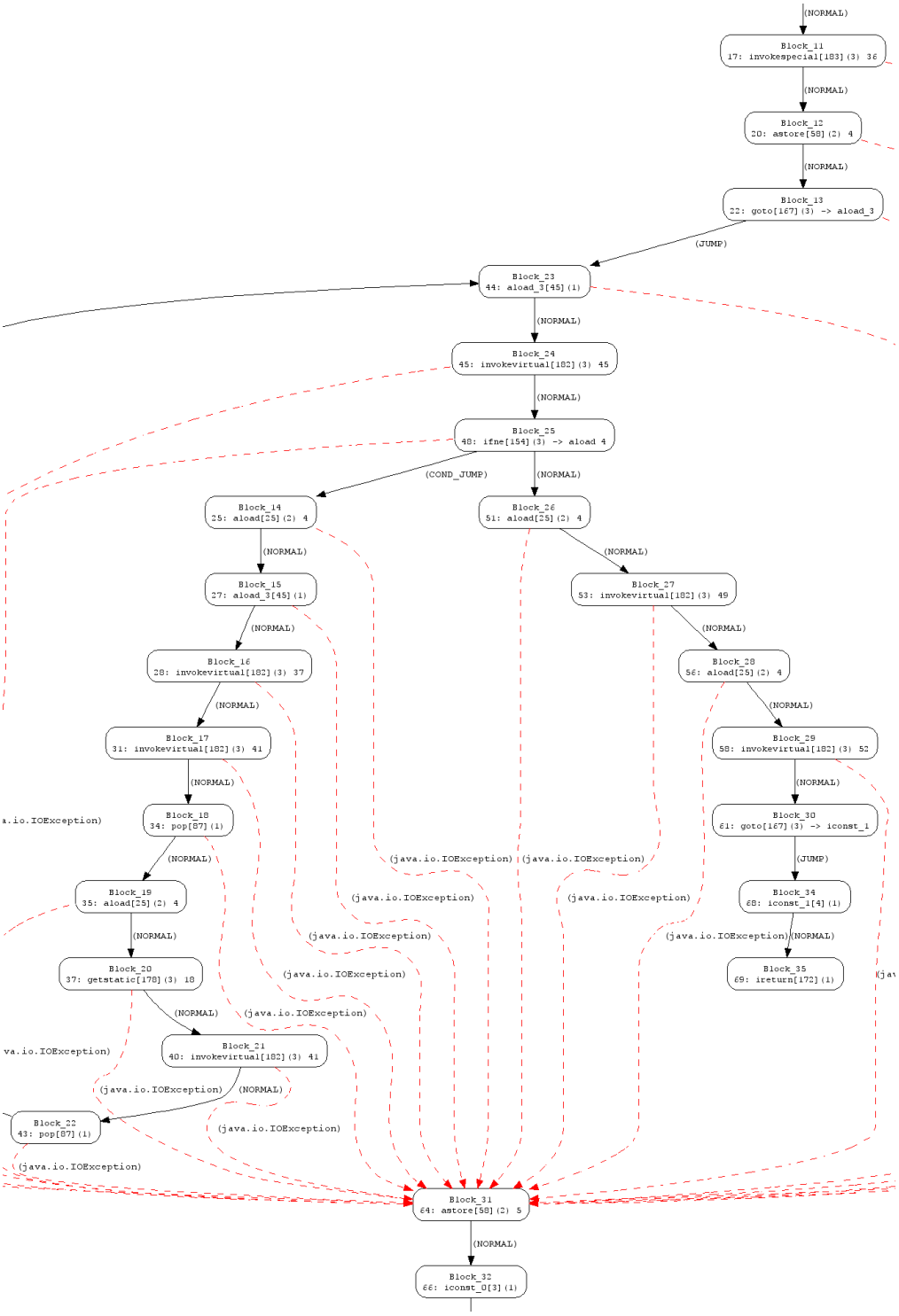


Figure 3.5: (Partial) control-flow graph after initial analysis

Proof. Equation 3.3 can be proven as follows. Assume we have blocks s, t and u such that:

$$s \neq t \neq u \tag{3.4}$$

$$t, u \in T(s) \tag{3.5}$$

Now let us first analyze the first statement of 3.2 which tells us that x is the only element of the set $S(y)$. Naturally, this implies that if we take a block z that is not equal to x that this block is not a member of $S(y)$:

$$(S(y) = \{x\} \wedge x \neq z) \rightarrow z \notin S(y) \tag{3.6}$$

Also, if x is a member of a certain set $T(y)$ and this set is again a subset of a certain set $T(z)$, then it follows intuitively that x is also a member of $T(z)$:

$$(x \in T(y) \wedge T(y) \subset T(z)) \rightarrow x \in T(z) \tag{3.7}$$

Now let us assume that s is found to be mergable with t. From 3.2 we get:

$$(T(s) - \{t\}) \subset T(t) \tag{3.8}$$

but because $mergable(s, t) = 1$ this also implies that s can not be found to be mergable with any other block, thus:

$$(\sum_{i=0}^{n} mergable(s, u_i)) = 0 \qquad u_i \in T(s), u_i \neq t \tag{3.9}$$

Recall that for any $u_i \neq t$ it follows from 3.7 that:

$$u_i \in T(t) \tag{3.10}$$

Then we substitute u for u_i and use 3.1 to get:

$$t \in S(u) \tag{3.11}$$

Now if we try to invalidate theorem 3.9 by assuming that $mergable(s, u) = 1$, then it follows again from 3.2 that $S(u) = \{s\}$ and by using theorems 3.4 and 3.6 we get:

$$t \notin S(u) \tag{3.12}$$

which creates a contradiction with 3.11. □

This proof is important, because it shows that the algorithm for merging blocks is *deterministic*, that is, it will generate the same graph each time it is run (provided that the code has not changed).

When two blocks are found to be mergable, the merging process itself is relatively simple: we create a new block while adding the source block to the top of the list and the target block to the end of the list. Then we redirect all incoming edges of the source block to become the incoming edges of the merged block and its outgoing edges are derived from the set of outgoing edges of the old target block. This process is visualized in figure 3.6.

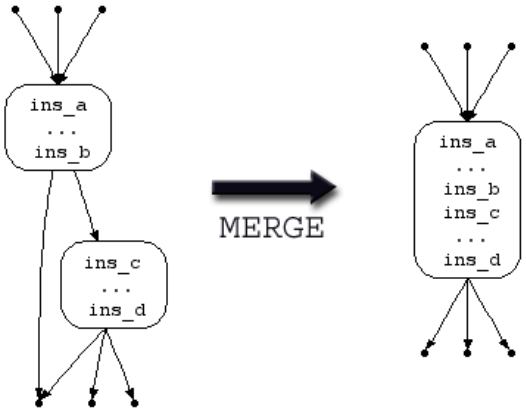


Figure 3.6: Example of two blocks merging into one.

3.3.5 Step three: ensuring a single entry and exit point

The algorithm for recording paths, discussed in the next section, requires that the control-flow graph has a single entry- and a single exit-block. The entry-block is defined as a block without any incoming edges, whereas the exit-block must be a block without any outgoing edges.

We have already seen that a method can have multiple exit-blocks, because of multiple `return`- or `athrow`-instructions. We can solve this by introducing a dummy exit-block which acts as a basic block but does not represent a piece of code. Any instrumentations applied to this block can then be forwarded to the actual exit-blocks.

Similarly, we introduce a dummy entry-block. Although methods have, by default, already a single entry-block, this block may have an incoming edge when the first instruction is targeted by another instruction within the method.

3.3.6 Step four: handling uncaught exceptions

At this point, the CFG is ready to be used for instrumentation purposes as discussed in section 3.1.2. There is, however, one specific situation we did not account for. On page 23 we basically argued that any instruction may cause an exception to be thrown, instead of just `throw`-instructions. Until now, we did not consider this to be part of the control-flow as defined by the CFG.

As will be discussed in more detail in the next section, any unexpected exception may corrupt the recorded path-information, because the algorithm presumes paths to be executed completely, not interrupted unexpectedly. Moreover, it may prevent the path to be recorded at all. Since (unexpected) exceptions play an important role in unit testing, we argued it would be important to be able to handle them and to record the path executed to the point at which the exception was thrown.

The solution for this problem is quite straightforward: we create a dedicated handler which is executed when any exception is thrown that is not caught by existing handlers.

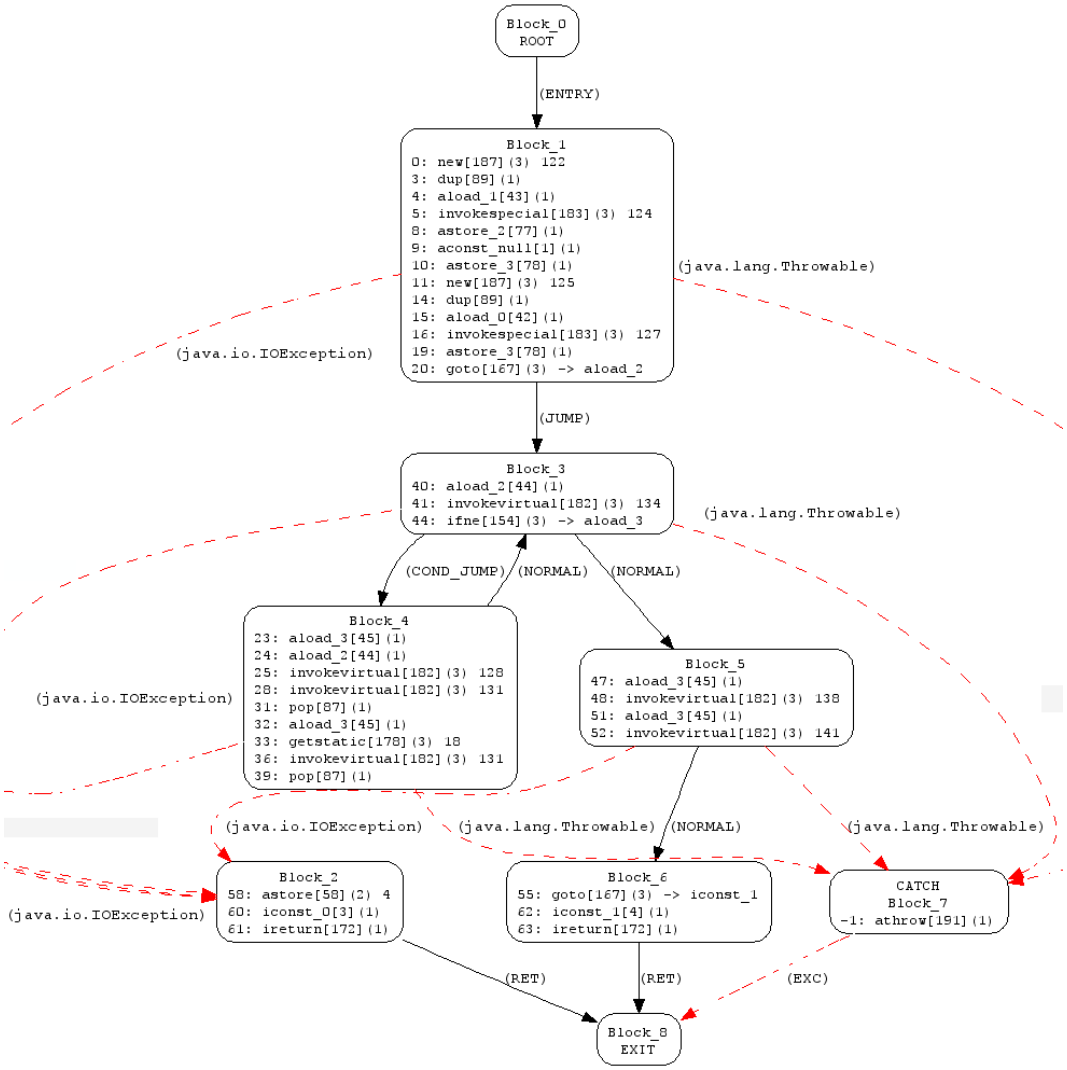


Figure 3.7: Control-flow graph with merged blocks and added entry-, exit- and catch-block. Note that the truncated edges do not target any blocks outside this figure.

Remark that this new handler is only introduced to consider the throwing of such exceptions as to be part of the expected control-flow within a method, not to perform any recovery. In fact, we need to ensure this solution stays transparent, so after catching it, we have to re-throw the exception.[3] We implement the solution as follows.

First, we define a handler consisting of a single `athrow`-instruction. This instruction can simply be added to the end of the method's instruction-list. Hereafter, we create a new row in the exception-table for each range of instructions that is *unprotected*, where each row targets the catch-block. An instruction is called *protected* when *any* type of instruction is caught when thrown at that point. Consequently, a new edge $\{b, b_{catch}\}$ is created for each block b that contains one or more unprotected instructions, where b_{catch} is the *catch*-block, representing the dedicated handler. We are actually introducing new paths that would be executed when at some point an exception is thrown that would otherwise remain uncaught.

In practice, instructions are rarely protected by default, so if no special care is taken, virtually every block in the graph would be connected to the catch-block, and the number of paths in the graph would increase exponentially. To mitigate this problem, we may choose not to protect every single instruction of the method, but to select those instructions for which we estimate the chance of an unexpected exception to be thrown is relatively high, and leave the rest unprotected.

A good strategy is to draw the line between `invoke`-instructions and the rest. The chance of an exception to be thrown when executing `invoke`-instructions is relatively high because the called method may throw an exception, or the call may result in a `NullPointerException`, which is a fairly common occurrence. In contrast, although still possible, unexpected exceptions are seldomly thrown at other points in the code. Our algorithm chooses this strategy by default.

The resulting graph is demonstrated in figure 3.7. Note that only those edges that contain an `invoke`-instruction are connected to the catch-block.

3.4 Ball and Larus's Efficient Path Profiling algorithm

The easiest way to record a path when the control-flow graph has been created is to simply add some logging functionality at the start of each basic block. While working with the InsectJ Framework [36], this functionality would be implemented by means of a call to a monitor-class object instance. However, it is not hard to imagine that this would cause tremendous overhead during run-time, since calls are relatively expensive.

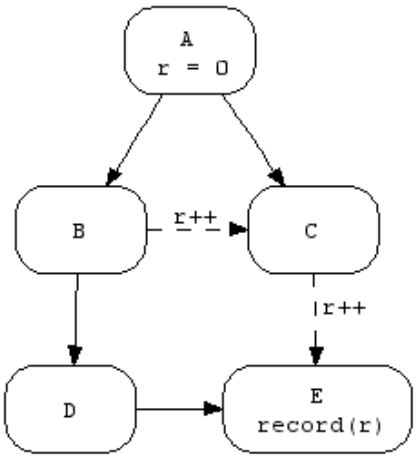
For this reason, Ball and Larus describe a more efficient path profiling algorithm [4]. To improve the readability of this paper, we will introduce the basic concepts of this algorithm below. To be more precise, our goal in this section is to show how the CFG is used to calculate the instrumentation, how this instrumentation is applied, which problems may occur and how we solved them. For more technical details, such as the theory behind the

[3] Actually, this is precisely how `finally`-blocks are implemented in byte-code: they catch any exception thrown in the `try`-block, perform the required operations, and then re-throw the exception as if it were never caught.

algorithm and the exact calculations, we refer to the original article.

Instead of introducing a call at every block, Ball and Larus propose to place a couple of 'smart' instrumentations on the *edges* of the graph. The first part of the algorithm identifies all paths in the graph and assigns a unique integer ps to each path such that $0 \leq ps < \#paths$ where $\#paths$ is the number of paths in the graph.

The idea is that a counter is introduced that is updated during run-time. When the method is exited, this counter will contain a value that identifies the path that was executed. This value is therefore called the *path-sum*. A graph analysis algorithm is applied to determine which edges should be instrumented with an appropriate counter operation to minimize run-time overhead. To visualize this process, consider the graph on the left in table 3.2. We identify three paths as shown in the table on the right. The figure presents one solution for placing the instrumentation. Note that each of the three paths will update the counter such that it will contain the value of the associated path-sum.



Path	Reported value of `r`
A-B-D-E	0
A-B-C-E	2
A-C-E	1

Table 3.2: Example graph with path and path-sums

3.4.1 The directed acyclic graph

The problem with arbitrary control-flow graphs is that they may contain cycles, which produce an infinite amount of possible paths. Recall that the edge that introduces this cycle is called a back-edge. To limit the amount of paths, each back-edge $\{s, t\}$ is replaced by the edges $\{b_{entry}, s\}$ and $\{t, b_{exit}\}$, where b_{entry} and b_{exit} represent the added entry- and exit-blocks respectively. Finally, one more extra edge is created, from exit to root. This edge, too, is required by the graph analysis algorithms. The resulting graph is called the *directed acyclic graph*, or DAG. Using the DAG, four types of *acyclic* paths can be identified, which are defined as follows [4, p. 52]:

Suppose that $\{v, w\}$ and $\{x, y\}$ are back-edges. A general CFG contains four possible types of acyclic (back-edge free) paths:

- A path from b_{entry} to b_{exit}.
- A path from b_{entry} to v, ending with execution of back-edge $\{v, w\}$.
- A path from w to x (after execution of back-edge $\{v, w\}$), ending with execution of back-edge $\{x, y\}$ (note: $\{v, w\}$ and $\{x, y\}$ may be the same edge).
- After executing back-edge $\{v, w\}$, a path from w to b_{exit}.

Figure 3.9 shows the transformed version of figure 3.7. The dashed arrows are the edges that replace the original back-edge $\{5, 3\}$, whereas the bold edges are called *chord*-edges. Chord-edges are the edges that are not a member of the set of edges of a calculated *spanning-tree* of the graph. However, their exact use and how they are calculated is beyond the boundaries of this discussion. Note that some edges are labeled with `r = x` or `count[(r+)y]++`. These labels represent the type of instrumentation that was calculated to be placed on that specific edge. With `r` as counter and `x` as any arbitrary integer, there are essentially four types of instrumentation as shown in figure 3.8.

Counter operations can be divided up into two categories,

- *Initializations* (`r = x`).
- *Updates* (`r += x`).

Path-count increments record or save the path-sum of the executed path. In our implementation, this type of instrumentation corresponds with a call to a monitor-instance. The name and notation are derived from the original article where the counter or path-sum indexes an integer-array, where each slot is associated with a particular path in the graph, and the value in each slot indicates the number of times a certain path has been executed. We distinguish between two types:

- *Immediate*: record the path-sum of which the value is fixed at this point (`count[x]++`).
- *Counter-dependent*: record the path-sum using the run-time value of the counter (`count[r+x]++`).

Figure 3.8: Description of the four possible types of instrumentation

The DAG provides a clear view on how and when each individual acyclic path is recorded during run-time. For example, while executing path 0-1-3-5-6-8 we traverse instrumentations `r = 8` and `count[r+3]++`, thus reporting a path-sum of 11.

Blocks in the graph that target themselves are called *self-loops*. Self-loops are special cases. In contrast to normal loops, their back-edge is not replaced but simply removed from the control-flow graph. Their presence is initially ignored while calculating the amount of acyclic paths and where to place the instrumentation. Once this information for regular acyclic paths is known, each self-loop path gets assigned a path-sum that is one higher than the current highest path-sum. Their instrumentation is of the type immediate path-count increment and will be placed on the back-edge $\{x, x\}$.

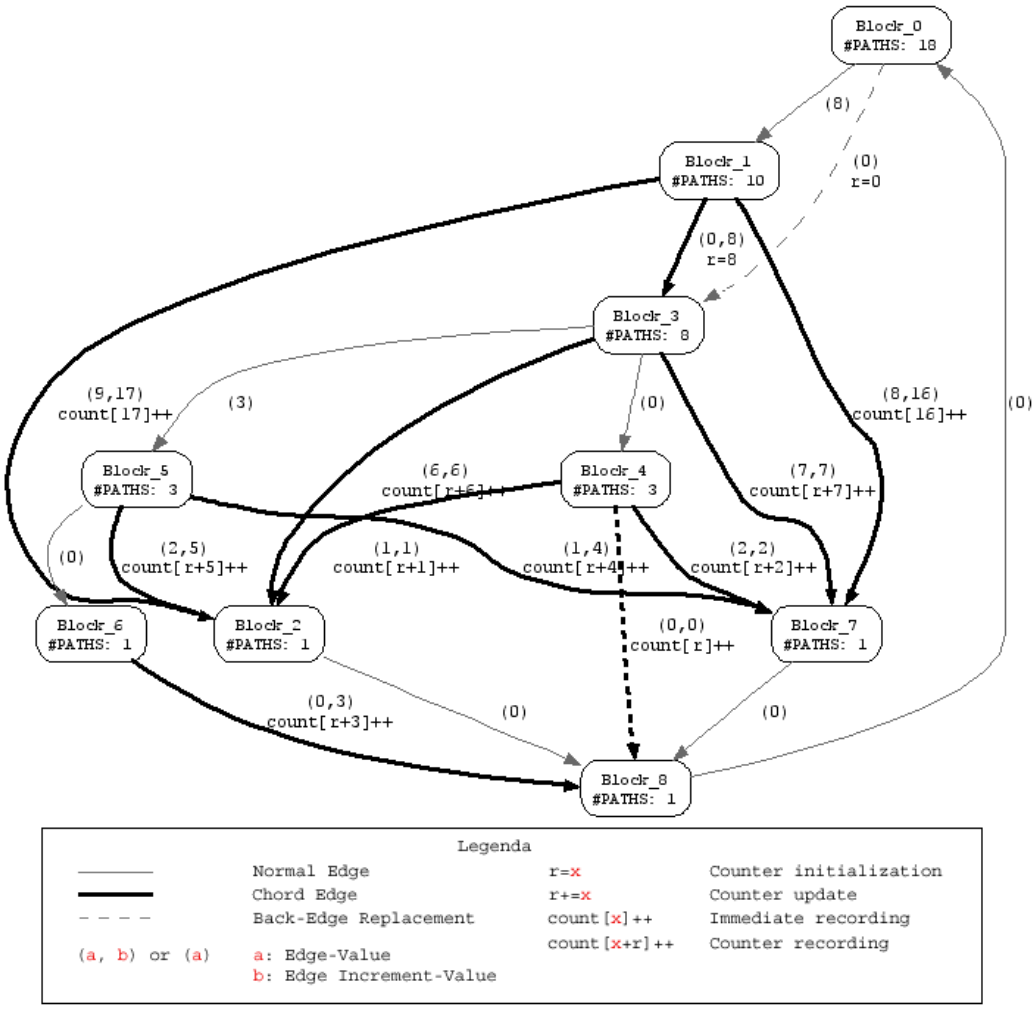


Figure 3.9: DAG with calculated optimal instrumentation

3.4.2 Placing the instrumentation

Once the edges and each type of instrumentation are known, the tool is ready to apply the instrumentation. Because the instrumentation is associated with edges, we have to redirect the control-flow as such that the instrumentation is executed but remains completely transparent with respect to the rest of the behavior of the program. How the byte-code is instrumented depends on the type of control-flow of the edge.

Normal control-flow and jumps

Instrumenting a particular edge $\{s, t\}$ of a control-flow graph basically means that the instrumentation-code should be executed after s has finished executing and before the first instruction of t is executed. In another article [25], Ball and Larus propose to place the

instrumentation for each $\{s, t\} \in S(t)$ right in front of the first instruction of t, and redirect or create jumps to ensure that for each particular edge only the appropriate instrumentation-code, if present, is executed.

For example, suppose a certain block t is targeted by three source blocks s_0, s_1 and s_2, and the edges $\{s_0, t\}$ and $\{s_2, t\}$ are to be instrumented. First, each block of instrumentation is inserted in front of block t, such that after execution of any block, control is transferred to t immediately. Now suppose that blocks s_0 and s_2 are targeting t by means of jumps, and s_1 had been adjacent to t, then the jumps of s_0 and s_2 should be redirected to their associated instrumentation-blocks, and a new jump has to be inserted right after s_1 in order to jump over the inserted instrumentation-blocks. This process is visualized in figure 3.10.

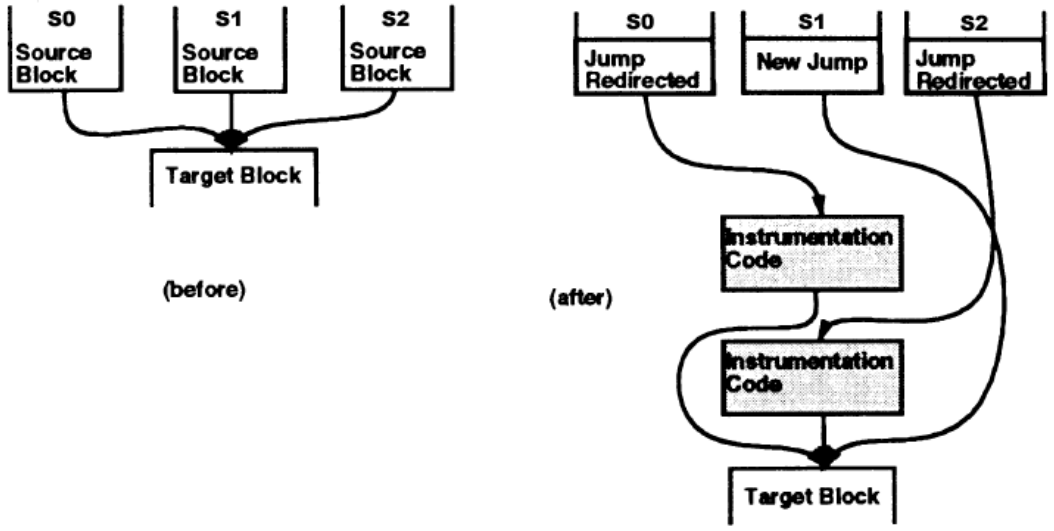


Figure 3.10: Instrumentation of edges (after figure 4, p. 203 from [25])

Exceptional control-flow

The same concept applies to the instrumentation of edges of exceptional control-flow, but it differs in how the control-flow is redirected from a source-block to a particular instrumentation-block. Instead of updating the targets of jump-instructions, we have to update the exception-table.

Since exceptional control-flow edges are actually representing a row in the exception-table that is targeting the first instruction of a handler, we somehow have to update this row to point to the instrumentation-block instead. Note that multiple edges may originate from the same row, because the instruction-range defined in this row may cover instructions of more than one basic block. If we want to ensure that only the control-flow represented by a certain edge $\{s, t\}$ is redirected, we have to split up the row in one that covers only the instructions of s and two other that cover the remainders of the original domain. The target of the former row can then be updated to point to the appropriate instrumentation-block.

	Instruction-range	Target
Before	$[a, d]$	t
After	$[a, b-1]$	t
	$[b, c]$	t_{new}
	$[c+1, d]$	t

Table 3.3: Split-up of a row within the exception-table

Table 3.3 presents the row-split-up process in a formal way. The bracketed part represents the instruction-range defined by the row, where $x-1$ and $x+1$ point to the instructions right in front of and right after x, respectively. Furthermore, $[b, c]$ is the range covered by the source block of the edge that is being redirected and t_{new} is the new target, pointing to the first instruction of the inserted instrumentation-block.

3.5 Restoring paths from the recorded information

The algorithm described in the previous section describes a way to record one integer, called the path-sum, for any acyclic path in an arbitrary method. This implies that we have to restore these paths in a later stage, for example, after the observed program has finished executing. In order to be able to do this, we require a little bit more information instead of just the path-sum.

Naturally, we need to know to which method the recorded path-sum belongs, since path-sum are only unique within the scope of a single method. Fortunately, the InsectJ Framework [36] allows to assign a unique integer to each *probe* that is inserted into the program, where a probe is considered to be one atomic piece of instrumentation code. In our implementation, we consider the entire instrumentation of one method to resemble a single probe, and hence we assign one unique integer, called the *probe-id*, to each method. For the sake of clarity, we will refer to this probe-id as the *method-id*[4].

Conclusively, to be able to regenerate all acyclic paths from a program we need two integers. The procedure that is used to regenerate these paths is still part of the algorithm of Ball and Larus and will be explained in § 3.5.1. The reader may, however, recall from our approach as explained in the previous chapter that we would like to compare the entire paths that are executed by test cases within a method. Therefore, we also need to somehow 'glue' the acyclic paths together to form a set of complete intra-method paths. This is where we extend the original algorithm, and we will discuss this procedure in § 3.5.2.

3.5.1 Regenerating acyclic paths

You may have noticed that the edges of the DAG depicted in figure 3.9 are also labeled with either (a) or (a, b). The value a is called the *edge-value* and is calculated as one of the first steps of the instrumentation phase [4, p. 50]. We can use these values to determine which

[4] Actually, we use this probe-id to obtain the DAG that was constructed and saved earlier, since it's this graph we need to regenerate the acyclic path.

edges of the DAG have been traversed. Thus, an acyclic path is actually regenerated by using the path-sum as a 'guide' that tells us which route to take while traversing the DAG. Section 3.5 of the original article uses an example to explain how this procedure works. Since we use a different example, we will try to put the explanation is our own words.

Let $val(e)$ be the value of edge e, ps the path-sum of the path we restore and $F_{out}(b)$ be a set of outgoing edges of b such that:

$$F_{out}(b) \subseteq E_{out}(b), \qquad \forall e_{out} \in F_{out}(b) : \qquad val(e_{out}) \leq ps$$

Then, starting at $b_{curr} = b_{entry}$, we select edge $e_{sel} = \{b_{curr}, b_{next}\}$ such that

$$\forall (e_{sel} \in F_{out}(b_{curr}), e_{out} \in (F_{out} - \{e_{sel}\}) : \qquad val(e_{sel}) > val(e_{out}))$$

Hereafter, we set $b_{curr} = b_{next}$ and $ps = ps - val(e_{sel})$ and select the next edge until we have reached b_{exit}.

The formal notation may make it seem more difficult than it really is. In simple words, we just start at the entry-block of the DAG. Then, in each iteration, we create a subset of edges of which the values are not greater than the current path-sum ps. Out of this subset we select one edge with the highest value and traverse it to end up at its target block. We then decrement the current path-sum with the value of the selected edge and repeat this procedure until we have reached the exit-block.

For example, let $e_{xy} = \{b_x, b_y\}$ and suppose we have to regenerate the path from the DAG at page 32 with a path-sum of 14. At block b_0, we select edge $\{b_0, b_1\}$ because $val(e_{01}) > val(e_{03})$. In the second iteration, we select $\{b_1, b_3\}$ since $val(e_{17}) > val(e_{13}) > val(e_{12})$ but $val(e_{17}) > ps$. In this fashion, we also select edges e_{34}, e_{42} and e_{28} to arrive at b_8, which is the exit-block. Hence, the path identified by path-sum 14 is 0-1-3-4-2-8.

Traversed in DAG	Restored acyclic path
0-3-5-2-8	3-5-2-8
0-1-3-4-8	0-1-3-4
0-3-4-8	3-4

Table 3.4: Restored acyclic paths after removing back-edge replacement edges

Remember that there are four different types of acyclic paths, of which three types do not start and end at the entry- and exit-block respectively. When the selected edge $\{b_{entry}, b_x\}$ in the first iteration is a back-edge replacement edge (the dashed arrows), we do not consider b_{entry} to be part of the acyclic path. Similarly, when the last selected edge $\{b_y, b_{exit}\}$ replaces a back-edge, the acyclic path ends at b_y. Table 3.4 shows one example for each of the other three types that is restored by traversing the DAG.

The last case are self-loop paths. By definition, they comprise a single block that has a self-reference. Path-sums that reflect a self-loop are always easily identified as such, since their

value is equal or higher than the number of paths in the DAG. We assign a unique identifier $0 \leq id(b_i) < n$ to each self-loop block b_i where n is the number of self-loops in the CFG. Henceforth, the self-loop path with path-sum ps is restored by simply selecting block b_i such that $ps = \#paths + id(b_i)$.

3.5.2 Reconstructing complete intra-method paths

At the start of this section, we explained that we need both the method-id and the path-sum to be able to regenerate all acyclic paths. In this paragraph, we will explain how we can arrange these acyclic paths such that they will form paths as defined in § 3.1.2: a sequence of basic blocks from entry- to exit-block. To do this, we will need to add two more pieces of information in the record of a single acyclic path, namely a *thread-id* and an *execution-id*. Figure 3.11 shows an overview of all the information we store for each acyclic path.

```
class PathSumRecord {

    // Identifies the thread
    Thread thread;

    // Identifies the method
    int probeID;

    // Identifies the acyclic path within the method
    int pathSum;

    /*
     * Identifies the specific call to the method, or, equivalently,
     * the specific path that is executed within a method.
     */
    int executionID;
}
```

Figure 3.11: The information that is stored for each acyclic path during run-time.

Combining acyclic paths

Suppose that in the CFG from figure 3.7 the path 0-1-3-4-3-4-3-5-6-8 is executed. The order in which the acyclic paths that make up this path are recored is 0-1-3-4, 3-4, 3-5-6-8, which is exactly the order in which we require them to be.

Consider again the four types of acyclic paths that appear in the bulleted list in the same order on page 30. We will now denote them as:

$$\begin{aligned} p_{se}: &\quad \{b_{entry}, \cdots, b_{exit}\} \\ p_{s}: &\quad \{b_{entry}, \cdots, b_{v}\} \\ p_{m}: &\quad \{b_{w}, \cdots, b_{x}\} \\ p_{e}: &\quad \{b_{w}, \cdots, b_{exit}\} \end{aligned}$$

Note that the first type, p_{se}, already represents a complete path itself and are recorded when paths do not contain any loop-iterations. If at least one loop was iterated one or more times, a path would be recorded as p_s (head of the path, including one loop-iteration), zero or more loop-iterations[5], with each iteration reported as p_m, and p_e representing the tail or end of a path. Hence, using a standard grammar notation, paths can be reconstructed from their acyclic parts as:

$$complete\_path \quad \rightarrow \quad p_{se} \mid p_s \ (p_m)^* \ p_e$$

However, imagine we store all acyclic paths we record into a list (in the order they are reported), then it is not ensured that the acyclic paths that make up a single path will occur consecutively within this list. First of all, multiple threads may be running, with each thread reporting its own acyclic paths in an unpredictable order. Clearly, acyclic paths of separate threads also belong to separate paths, so we can solve this problem by attaching a thread-id to each acyclic path. We can then use this thread-id to sort the list, such that each part of the list containing the same thread-id resembles a non-parallel setting.

Furthermore, methods may be recursive, either directly or indirectly. If a recursive call is made, the reported path-sums of the nested execution(s) mix with the path-sums reported during the execution in which the call was made. Fortunately, since we know that in the context of a single thread the execution of a certain method is postponed until a nested call has returned, the order of the reported acyclic paths incorporate a predictable, nested structure:

$$method\_path \quad \rightarrow \quad p_{se} \mid p_s \ (\ method\_path \mid p_m \ )^* \ p_e$$

Paths can thus be reconstructed by maintaining a stack of unfinished paths, pushing a new element every time an acyclic path of type p_s is encountered in the list, while popping and completing a path once a type of p_e is encountered.

Inserting self-loop paths

The second and slightly more complex problem is caused by self-loops. To understand this problem, we have to note that during the phase where the location and type of instrumentation-blocks is determined, an optimization step is performed that aims to defer any counter-operations and bring forward any path-count increment operations [4, p. 52]. When a counter-operation is deferred such that it can be placed on the same edge as a path-count increment operation, they can be combined into a more efficient instrumentation block. For example, when a counter-initialization (`r = x`) is placed on the same edge as a counter-dependent increment (`count[r]++`), we might as well combine these steps and place a single immediate increment (`count[x]++`).

Now consider an arbitrary acyclic path $p_x : \{b_i, \cdots, b_n\}$ where $b_k \in p_x$ is a self-loop block. When p_x is executed, b_k will also be executed one or more times, resulting in one

[5]Note that these iterations do not have to be part of the same loop.

occurrence of p_x and zero or more recordings of path $\{b_k\}$. We may reason that since b_k is actually part of p_k, we need to insert the self-loop path occurrences into p_k after restoring them. Obviously, when encountering a self-loop path $\{b_k\}$ in the recorded list, we need to find a way to which recorded path p_x it belongs. To discriminate clearly from self-loop paths, all occurrences of one the four basic types already discussed (p_{se}, p_s, p_m and p_e) will be denoted as *regular* acyclic paths.

Since self-loops are initially ignored while calculating and optimizing the instrumentation for regular acyclic paths, the instrumentation-block reporting on path p_x may be placed on an edge before or after b_k. For example, suppose b_3 were a self-loop in figure 3.9. In this case, any additive iterations of 3 are reported *before* the path 0-1-3-7-8 is reported. However, it may also be the case that if b_7 were a self-loop path, all additive iterations of 7 would be reported *after* 0-1-3-7-8.

Consequently, we do not know whether the execution of self-loop path $\{b_k\}$ or path p_x is reported first. It is not hard to see why this is a problem. Suppose another acyclic path $p_y, b_k \in p_y$ is executed before p_x, and that the path-sums are recorded in the order: $p_y, \{b_k\}^*, p_x$. It is now impossible to determine which occurrences of the self-loop should be inserted into p_y and which in p_x.

We solve part of this problem by preventing that during the instrumentation-optimization phase, the code responsible for reporting a regular path is placed on an edge before any self-loop in that path. In other words, we ensure that self-loop paths are always reported immediately before the regular paths they belong to.

However, the possibility remains that because of recursion, self-loops and acyclic paths of different paths from the method are mixed. This is solved by storing an execution-id, which has been introduced earlier in this section. The execution-id is an integer which allows us to discriminate between different calls to the same method, and it is attached to every reported acyclic path. This means that we can safely insert each reported self-loop $\{b_k\}$ into the first acyclic path that was reported after it, contains b_k *and* has the same execution-id.

3.6 Summary

This chapter was dedicated to a description of how we extract run-time paths from arbitrary Java-programs. Generally, extracting paths from a running program is called path profiling. The general concept of path profiling usually embodies the instrumentation of a program P, whereafter P is run and the inserted code provides the information about which paths are executed within each method. There are basically two types of path profiling: intra-method profiling, which we apply in our project, and inter-method profiling. The former method extracts only the paths that are executed within a method, whereas the latter collects information about the executed path within the whole thread. We chose to limit ourselves to intra-method path profiling because intra-method paths provide enough information to achieve our general objective: comparing test cases based on these paths.

To be able to instrument a program efficiently, we need to construct the control-flow graph for each method that will be observed during run-time. The control-flow graph is a graph that describes any possible path of execution within the corresponding method. The construction of this graph and the instrumentation thereafter is a form of static code analysis, in contrast to the extraction of the path-information later, which is a form of dynamic code analysis (see figure 3.1).

The code analysis required to build the control-flow graph and applying the calculated instrumentation in a later stage is performed through the InsectJ Framework [36]. This framework provides a flexible framework that enables easy instrumentation of Java program using BCEL, which is a layer between the class-files loaded into memory by a class-loader and the application that manipulates these files. However, using BCEL implied we would be analyzing classes at the byte-code level. The most important advantage of this is that working on the byte-code level is more flexible and efficient than instrumenting programs at the source code level. It also means that it would be more difficult to retain a mapping to the source code, if available, to show, for example, the results of path-observations in a source code editor.

As the first step of the path profiling process, we infer the control-flow graph from the byte-code. Although constructing this graph is not particularly new in itself, the graph we create contains, if necessary, a so-called catch-block, which will be executed whenever an exception is thrown other than by a `throw`-instruction within that method. As such, we explicitly model the possible exit of a method which is not defined by the control-flow of the instructions. Not only does this gain more insight into the control-flow of a method, it also solves the problem of not being able to record a path because the method was exited unexpectedly.

In the second and third step, we use Ball and Larus's Efficient Path Profiling algorithm [4] to create a directed acyclic graph or DAG, which is a derivative of the CFG, in which all back-edges (loops) are removed. We use this DAG to calculate a fairly efficient set of instrumentations and apply these instrumentations with the InsectJ Framework. With the completion of this instrumentation, we are ready to extract the executed paths during run-time.

While the observed program is running, we collect information about all *acyclic* paths that are executed. An acyclic path is basically one piece of the whole path that is being recorded within a method. Since we aim to collect complete paths, not acyclic paths, we need four different kinds of data for each acyclic path recorded (figure 3.11): the thread on which the acyclic path was executed, a method-id, a path-sum and an execution-id.

After the observed program P has finished executing, we first regenerate the acyclic paths using the recorded method-id and the recorded path-sum, where the recorded path-sum identifies the acyclic path that was traversed in the DAG. Hereafter, we reconstruct the complete paths from these acyclic paths, primarily based on the order in which the latter were recorded. However, this order is only predictable in a non-parallel setting, which is why we need to attach the thread-information, so that we can sort the acyclic paths and reconstruct all paths on a thread-by-thread basis. A second problem are self-loops, which are one-block paths that are recorded in another way than their regular counter-parts. Since

these blocks are part of at least one regular acyclic path, any self-loop occurrences must be inserted into the appropriate regular path. Because methods can be recursive, which implies the recordings of multiple paths within the same method may be mixed, we identify the appropriate paths by whether they contain the self-loop block to be inserted and the execution-id, which discriminates between different calls to the same method.

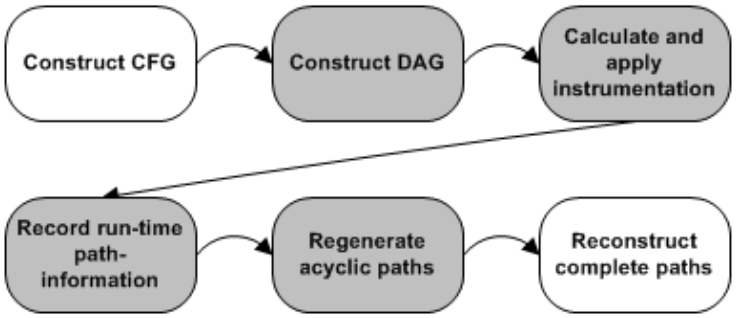


Figure 3.12: Schematic overview of the process of extracting paths from a program

The whole process of this chapter is visualized in figure 3.12. The phases with a darker background are those that are largely inspired by or described within the work of Ball and Larus. The phases with a lighter background represent our extensions to this work.

Chapter 4

Comparing Test Cases

In chapter 2 we explained the primary objective of this thesis was to compare test cases based on the paths they execute in a system under test. To do this, we first we need to know which test cases have executed certain paths in a system. Secondly, we have to define how we will compare these paths. In the previous chapter we have shown how we can extract the executed paths from a running system.

The first section of this chapter explains how we will associate the correct test cases to these paths, or, in other words, how we will determine which test cases are responsible for executing certain paths in the tested system. In the second section, we will outline and motivate the way of how we compare these paths such that we will get an estimation of the similarity of different test cases in a test suite.

4.1 Associating test cases and paths

If we are to match the run-time paths with the test cases that executed them, we need a way to find out when a test case is fired and when it has ended: in the context of a single thread, all the paths that are recorded can be assigned to each test case that has been fired but has not yet returned. In other words, all the recorded path-information can be associated with the *current stack* of active test cases.

The standard Java library provides a way to access the current call-stack of each running thread through `Thread.getAllStackTraces()`[1]. However, this call-stack contains *all* running methods of a single thread, while we are only interested in the currently active test cases, so using this call-stack would require us to search through and identify these test cases within the stack. Moreover, such filtering of the call-stack might incorporate a relatively large overhead.

Partly because of these reasons, we thought it would be a better solution to maintain our own dedicated call-stack of test cases. Maintaining our own stack solves the problems mentioned above, but the most important motivation to do this was probably the gained flexibility and surprisingly little effort it took to implement. With the InsectJ Framework [36] at hand, we could easily instrument each test case with a probe that signaled the entry and

[1] Also see `http://java.sun.com/j2se/1.5.0/docs/api`

a probe that signaled the exit of the method. Although slight adjustments were made, both types of probes are standard functionality in this framework. All we had to do ourselves is define a monitor-class that maintained the test case call-stack and gave access to it such that a path could be associated with a current stack. Naturally, we maintain one stack per active thread. The whole process is depicted in figure 4.1. For more technical details on the actual implementation, we refer to appendix A.

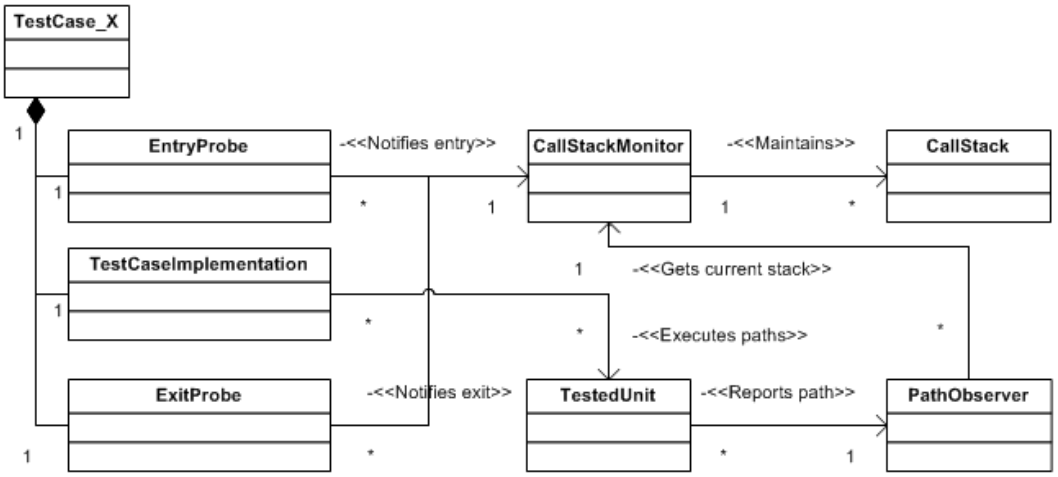


Figure 4.1: UML-representation of associating test case run-info with executed paths in a system

4.2 Defining test case isomorphism

In chapter 3 we motivated our choice to stay with measuring intra-methods paths. This means that we will be calculating a similarity between paths *per method*. The first aspect we have to account for is that each test case may execute multiple paths within the same method, either directly or indirectly. A test case may, for instance, implement a loop that contains a call to a method in the tested class, or the callee may call other methods, including itself, inside the tested system, which implies that all the paths executed in those methods will be associated with the same test case. For these reasons, we will be calculating the similarity between every two test cases that have executed at least one path in a certain method m, and we will base this calculation on the associated *path-sets*.

Test case isomorphism. *Isomorphism* is a kind of mapping between objects to express a certain degree of similarity based on the equality or similarity of particular properties or structure of these objects. This degree can be expressed a value between 0.0 (no similarity) and 1.0 (completely equal). Now let $P_m(\vartheta)$ be the set of paths that are executed by test case ϑ within method m, then the isomorphism of two test cases ϑ_i and ϑ_k is approximated by the isomorphism of their executed path-sets:

$$iso_m(\vartheta_i,\ \vartheta_k) \approx iso_{est}(P_m(\vartheta_i),\ P_m(\vartheta_k))$$

$$iso_{est}(P_m(\vartheta_i),\ P_m(\vartheta_k)) = iso_{est}(P_m(\vartheta_k),\ P_m(\vartheta_i))$$

The second term expresses the fact that comparing ϑ_i with ϑ_k is essentially the same thing as comparing ϑ_k with ϑ_i and should therefore result in the same isomorphism. Note that if two test cases do not call a certain method at all, then their isomorphism remains *undefined*, since we reasoned that such comparison would be meaningless in its mathematical sense, in the same way that $0 \div 0$ remains undefined. Thus, when we calculate the isomorphism of two test cases, we always assume the following:

$$(P_m(\vartheta_i)\ \cup\ P_m(\vartheta_k)) \neq \oslash \tag{4.1}$$

4.3 Calculating test case isomorphism

The reader may recall from § 2.2 that the effectiveness of test cases is essentially determined by the states that are checked while running. Paths may represent a good metric to estimate whether two test cases take a system or unit into similar states, and therefore we suspect that path-similarity may be a good metric for the relative effectiveness of two test cases.

However, in order to exploit this expected correlation, the executed path-sets of different test cases should be compared in such a way that this state-similarity property is accounted for. In other words, when defining the calculation of the similarity of path-sets, we need to think about how certain differences in these sets manifest themselves in differences between the encountered state-sets.

Suppose, for instance, we are comparing two separate paths that have been executed in the sample method of figure 3.4. Imagine both paths are the same, except that one contains only a few loop-executions and the other many, like a thousand. We could simply calculate the number of equal elements of these paths, such as the number of basic blocks present in both paths, and dividing this number by the total number of elements. Using the simple approach, the isomorphism of these two paths would be almost zero. Although it is clear that both paths are not equal and the encountered state-sets of these paths may have differed, we may argue that these paths are still similar to a high degree. In the next few sections, we propose a more sophisticated approach to compare two path-sets that takes these cases into account. For simplicity and readability, we will outline our approach by assuming we are comparing two single, separate paths. We will later extend the algorithm to also support comparing two sets of paths.

4.3.1 Levels and sub-paths

We learn from the previous example that a lot of extra iterations through loops may not necessarily significantly change the similarity of two paths and that when two paths are structurally equivalent and differ only because one path contains more repetitive parts, we would not expect a very low isomorphism.

One approach to tackle this issue is to split up paths into loop-less *sub-paths*. The similarity of these sub-paths can then be analyzed in isolation and the results can be combined to obtain the desired path-isomorphism value.

We may subdivide paths into different sub-paths by embodying loop-structures in the CFG in so-called loop-blocks. Let t be the target block of a particular set of back-edges, then a loop-block consists of all blocks that are a member of a cycle path $\{t, \cdots, t\}$. This encapsulation of blocks by loop-blocks is a recursive process, so nested loops are captured into nested loop-blocks. Figure 4.3.1 shows a transformed version of an arbitrary CFG where two nested loops have been captured into loop-blocks.

We call the graph in this figure a *path-block graph*, where a *path-block* represents one item of a sub-path. Its primary function is to facilitate the analysis of recorded paths, where in this case we use it to calculate the isomorphism. The path-block graph enables us to analyze paths at different *levels*.

Level. A level is defined as a sub-graph with a single and unique entry-block where each immediate embodied loop-block is considered an elementary block with no internal structure and of which the outgoing back-edges are considered invisible. Each level in the path-block graph with entry-block b_x will be denoted as l_x. Level l_0 is the level with entry-block $b_0 = b_{entry}$ and is therefore always the top-level.

The graph depicted on page 45 contains three levels: l_0, l_2 and l_5. Also note that the figure shows small dots interrupting the edges connecting blocks on the in- and outside of loop-blocks. These dots act as *border-blocks* between two levels and can be viewed as the *exit-blocks* of paths inside a loop-block, but they are not part of the recorded intra-method paths. Each border-block will be denoted as e_{st} where the border-block is targeted by s and itself targets t. A sub-path can now be defined as follows.

Sub-path. A sub-path is defined as any (acyclic) path within a certain level from its entry-block to one of its exit-blocks.

We would like to emphasize here that these sub-paths are not the same sort of acyclic paths discussed in the previous chapter. Table 4.1 shows all sub-paths that are specified by the sample graph.

Now suppose we would have recorded the paths shown in table 4.2. In order to compare them at each sub-path level, we determine which parts of the paths belong to which level. In the third column, the braces indicate that a particular part belongs to the same level. The semi-colon represents a back-edge, or the start of a new iteration in the same level.

For each level we store the information about which paths have executed particular sub-paths into a *sub-path table*. Additionally, each row-entry keeps track of how many times a particular path has entered a certain level. We will refer to this as the number of iterations, but there is a slight difference with normal loop-iterations. The number of loop-iterations indicates how many times the body of a loop is *sequentially* executed, whereas the number of iterations in the sub-path table merely shows the *total* amount of times that a particular path has executed a sub-path on that level.

Suppose a certain path p_x enters l_2 n times, and in each iteration, it enters the inner-loop (l_5) k times, then the sub-path table of l_5 will show $n * k$ iterations for path p_x. Moreover,

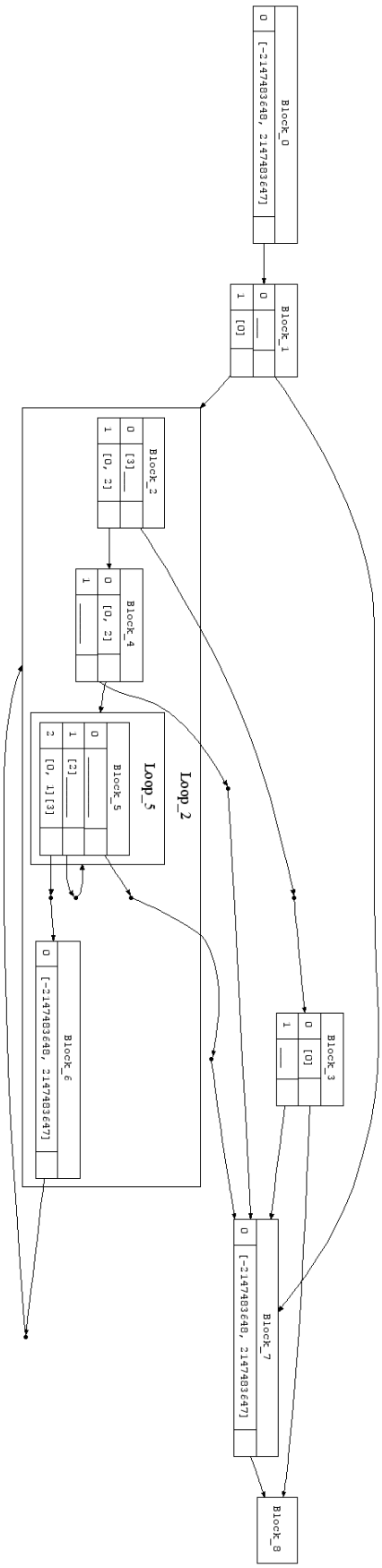


Figure 4.2: Path-block graph with two nested loops

Level	**Sub-paths**
l_0	$\{l_{0a} : b_0, b_1, b_7, b_8\}$ $\{l_{0b} : b_0, b_1, l_2, b_3, b_7, b_8\}$ $\{l_{0c} : b_0, b_1, l_2, b_3, b_8\}$ $\{l_{0d} : b_0, b_1, l_2, b_7, b_8\}$
l_2	$\{l_{2a} : b_2, e_{23}\}$ $\{l_{2b} : b_2, b_4, e_{47}\}$ $\{l_{2c} : b_2, b_4, l_5, e_{57a}\}$ $\{l_{2d} : b_2, b_4, l_5, b_6, e_{62}\}$
l_5	$\{l_{5a} : b_5, e_{57b}\}$ $\{l_{5b} : b_5, e_{55}\}$ $\{l_{5c} : b_5, e_{56}\}$

Table 4.1: Sub-paths specified from figure 4.3.1

entering and executing a sub-path on a certain level $l_i,\ i > 0$ does not imply that the body of a loop is executed, since some blocks, including entry-block b_i, may be part of the loop-condition, which may of course evaluate to `false` before the loop-body has been executed at all. Table 4.3 shows the sub-path tables with the path-information of p_0 and p_1.

Path	**Regular notation**	**Divided into sub-paths**
p_0	0-1-2-4-5-5-6-2-3-8	$\{\ b_0,\ b_1,\ (\ b_2,\ b_4,\ (\ b_5;\ b_5\ ),\ b_6;\ b_2\ ),\ b_3,\ b_8\ \}$
p_1	0-1-2-4-5-6-2-4-7-8	$\{\ b_0,\ b_1,\ (\ b_2,\ b_4,\ (\ b_5\ ),\ b_6,\ b_2,\ b_4\ ),\ b_7,\ b_8\ \}$

Table 4.2: Paths divided into sub-paths

l_0:

	l_{0a}	l_{0b}	l_{0c}	l_{0d}	Iterations
$\{\ p_0\ \}$		X			1
$\{\ p_1\ \}$				X	1

l_2:

	l_{2a}	l_{2b}	l_{2c}	l_{2d}	Iterations
$\{\ p_0\ \}$	X			X	2
$\{\ p_1\ \}$		X		X	2

l_5:

	l_{5a}	l_{5b}	l_{5c}	Iterations
$\{\ p_0\ \}$		X	X	2
$\{\ p_1\ \}$			X	1

Table 4.3: Sub-path tables contain the information about which sub-paths have been executed by different paths at a certain level.

	l_{0a}	l_{0b}	l_{0c}	l_{0d}	Iterations
$\vartheta_x : \{ p_0, p_1 \}$		X		X	2

Table 4.4: Combining the path-information of p_0 and p_1 into a single row.

4.3.2 Isomorphism on a single level

At this point we are ready to start comparing sub-paths. In fact, we are not limited to comparing individual sub-paths: we can calculate the isomorphism of sets of sub-paths.

The structure of the sub-path table makes it easy to combine the sub-path-information of a single path at the same level, because the rows containing the sub-path-information can easily be merged. In the same fashion, the path-information of multiple paths that belong to same test case can be merged into a single row of path-data. This is why we are capable of comparing two path-sets instead of just two separate paths. Table 4.4 shows the result for l_0 in case that some test case ϑ_x has executed both p_0 and p_1.

We realize that there exists an infinite amount of possible equations we may use for calculating the isomorphism, and many of them could be backed up by good argumentation. In the next few sections, we infer one possible solution to this problem. In the examples we always use $P_m(\vartheta_0) = \{ p_0 \}$ and $P_m(\vartheta_1) = \{ p_1 \}$. Furthermore, along the same lines as equation 4.1, we do not consider the case in which two test cases have not executed any sub-paths on a certain level:

The isomorphism of two test cases with empty (sub-)path-sets remains undefined. Let $P_{sub}(l_i, \vartheta_i)$ denote the set of sub-paths that are executed by test case ϑ_i on level l_i, then we always assume:

$$P_{sub}(l_i, \vartheta_x) \cup P_{sub}(l_i, \vartheta_y) \neq \oslash \tag{4.2}$$

Factor 1: Executed sub-paths

For the first part of the equation, we consider the set sub-paths that are executed. We assume that each sub-path has its own encountered state-set character, so in that sense, every sub-path is unique. Furthermore, we have no strong argument to expect that one sub-path is more important than another. For this reason, we simply divide the number of sub-paths executed by both test cases by the total number of sub-paths executed by either one of them.

Sub-path factor. Let $size(P)$ be the number of elements within the set P, then we define the isomorphism of ϑ_x and ϑ_y on l_i with respect to the executed sub-paths as:

$$iso_{sub}(l_i, \vartheta_x, \vartheta_y) = \frac{size(P_{sub}(l_i, \vartheta_x) \cap P_{sub}(l_i, \vartheta_y))}{size(P_{sub}(l_i, \vartheta_x) \cup P_{sub}(l_i, \vartheta_y))} \tag{4.3}$$

For instance, on the method-level l_0 for ϑ_0 and ϑ_1 this would give us:

$$iso_{sub}(l_0, \vartheta_0, \vartheta_1) = \frac{size(\{ M_2 \} \cap \{ M_3 \})}{size(\{ M_2 \} \cup \{ M_3 \})}$$

$$= \quad 0 \div 2 = 0$$

Factor 2: Executed edges

The example equation shows that if two paths share no sub-path on a particular level, this isomorphism-factor becomes zero. However, when certain sub-paths share many edges, we may argue that it is still likely that their corresponding state-sets still overlap to some extent. We introduce a second term in which we divide the number of shared edges by the total number edges executed by either one of them, in the same fashion as the first factor.

Edge factor. Let $E(P_{sub}(l_i, \vartheta_i))$ denote the aggregated set of edges that are executed by test case ϑ_i on a certain level l_i, then we define the isomorphism of ϑ_x and ϑ_y on l_i with respect to the executed edges as:

$$iso_{edg}(l_i, \vartheta_x, \vartheta_y) = \frac{size(E(P_{sub}(l_i, \vartheta_x)) \ \cap \ E(P_{sub}(l_i, \vartheta_y)))}{size(E(P_{sub}(l_i, \vartheta_x)) \ \cup \ E(P_{sub}(l_i, \vartheta_y)))} \tag{4.4}$$

Resolving this term for the Loop_2-level l_2 would result in:

$$\begin{aligned} iso_{edg}(l_2, \vartheta_0, \vartheta_1) &= \frac{size(\{\ \{b_2,\ b_4\}, \{b_4,\ l_5\}, \{l_5,\ b_6\}, \{b_2,\ e_{62}\}\ \})}{size(\{\ \{b_2, e_{23}\}, \{b_4, b_7\}, \{b_2,\ b_4\}, \{b_4,\ l_5\}, \{l_5,\ b_6\}, \{b_2,\ e_{62}\}\ \})} \\ &= \frac{4}{6} = \frac{2}{3} \end{aligned}$$

Factor 3: Iterations

When two test cases have executed exactly the same sub-paths, and consequently the same edges, within a certain level, the previous two terms calculate a hundred percent score on isomorphism. In spite of the fact that both state-sets are presumably relatively equivalent, a difference in the number of iterations (as defined in the sub-path table) may still indicate a small shift in isomorphism. It is not hard to imagine that if, for example, a loop is iterated only one time in one test case and five times in a second test case, those test cases check upon slightly different states, even if the sub-paths they execute are exactly the same. We therefore introduce a third and last term, in which the number of iterations are compared.

If we zoom in on loop-iterations for a moment, we see that in practice most loops are iterated only a handful of times: somewhere between zero and twenty times. This is what we will call the *critical area*. *Boundary values* are values against which a variable is checked in a certain decision point in the code. In most loops we find at least one boundary value, which controls the number of iterations (e.g. `array.length`). Other boundary values may be used inside the loop which determine the control-flow inside the loop-body. Based on our own experience, we assume that most boundary values within a program also fall into the critical area. Because mistakes using the comparison-operators are easily made (e.g. using $<$ where $\leq$ is required), testing on the correctness of these operators is heavily encouraged by Binder [9].

Furthermore, many boundary values are typically compared with a variable that depends on the iteration-index. This is why the number of loop-iterations (indirectly) play an important role in the fault-revealingness of test cases, especially within the critical area: we expect that a difference in the number of iterations in a low range causes more fluctuations in the similarity of the corresponding state-sets than a difference in higher ranges. In other words, the difference between 2 and 5 iterations is more significant than the difference between 1000 and 1003 iterations. This means that the isomorphism of two test cases is lower in the former case than in the latter.

Iteration factor. Let $x \in \mathbb{N}$ and $y \in \mathbb{N}$ be the number of iterations executed by two test cases on a certain level l_i, then we define the following equation:

$$f_a(x, y) = 1 - \left(\frac{\mid x - y \mid}{xy} \right)^{\delta} \tag{4.5}$$

The part between the parentheses is the most important thing to notice. By dividing the absolute difference of x and y by their product, the enumerator is relatively significant in the lower values of x and y, but when both variables have high values, this term goes down to zero. Subtracting this term from 1 gives us the desired proportionate isomorphism values as described in the previous paragraph. Note that no matter what the exact values are, if the number of iterations are equal, $f(x, y)$ is equal to 1. The last variable δ can be used to control the significance of the difference relative to the exact values of x and y. By trial and error, we have found the best values for δ are in the range $[0.2, 0.4]$. Figure 4.3 shows the graph $z = f_{0.2}(x, y)$.

You may have noticed that the given function is not defined for all values of x and y. Although the case $x + y = 0$ is not considered (that case is undefined, see equation 4.2), we do have to account for $x = 0$ or $y = 0$. In that particular case, we define the isomorphism of two test cases to be 0. Consequently, the iteration-factor becomes:

$$iso_{itr}(l_i, \vartheta_x, \vartheta_y) = \begin{cases} f_\delta(x, y) & x > 0 \wedge y > 0 \\ 0 & x = 0 \vee y = 0 \end{cases} \tag{4.6}$$

Using $\delta = 0.2$ as in the figure, the iteration-factor for l_5 would be:

$$\begin{aligned} iso_{itr}(l_5, \vartheta_0, \vartheta_1) &= f_{0.2}(2, 1) \\ &= 1 - \left(\frac{\mid 2 - 1 \mid}{2 * 1} \right)^{0.2} \\ &\approx 0.13 \end{aligned}$$

Combining all factors

All three factors defined above represent the core of the isomorphism-calculation. We are now able to combine these factors into one single equation that calculates the isomorphism of two test cases on a single level l_i. Equation 4.7 shows our solution. The specified weights

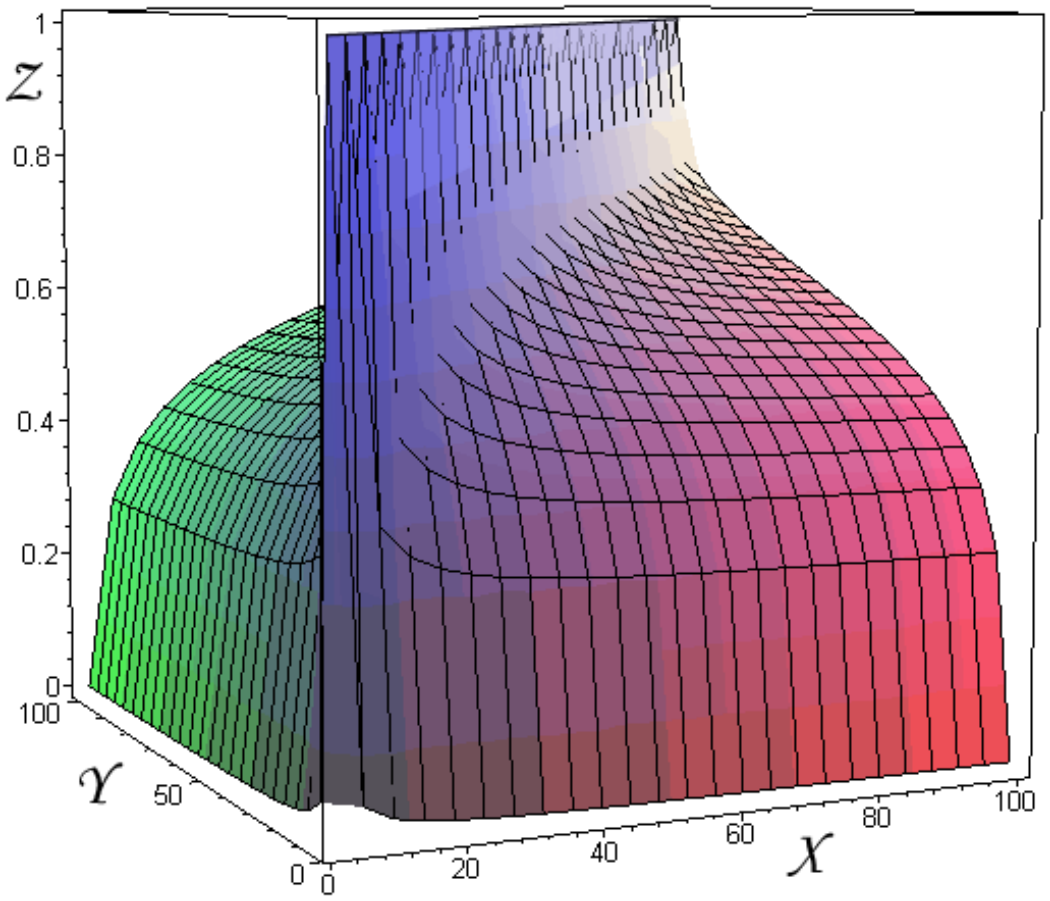


Figure 4.3: 3D-perspective of the isomorphism (`z`) between two paths with respect to the number of iterations `x` and `y` at α `= 0.2`

α, β and γ can be used to increase or decrease the importance of a certain factor, but unless stated explicitly otherwise, we will assume $\alpha = \beta = \gamma = 1$.

$$iso_{lev}(l_i,\ \vartheta_x,\ \vartheta_y) = \frac{\alpha\ iso_{sub}(l_i,\ \vartheta_x,\ \vartheta_y) + \beta\ iso_{edg}(l_i,\ \vartheta_x,\ \vartheta_y) + \gamma\ iso_{itr}(l_i,\ \vartheta_x,\ \vartheta_y)}{\alpha + \beta + \gamma} \tag{4.7}$$

4.3.3 Combining all levels

The final step is to combine the results of all different levels at which a certain path was compared. During our research, we questioned ourselves whether it would be a good approach to look at the degree of nesting of levels and assign weights to them accordingly. One possible solution, for example, would be to assign lower weights to deeper nested lev-

els. After several debates, however, we could not clearly see how the degree of nesting is correlated with the state-set corresponding to a path or test case.

Hence, we simply calculate the average result of all levels. Let n be the number of levels for which a certain isomorphism of two test cases was calculated, then we define the overall estimated isomorphism of ϑ_x and ϑ_y as follows:

$$iso_{est}(P_m(\vartheta_x),\ P_m(\vartheta_y)) = \frac{\sum_{i=0}^{n-1}\ iso_{lev}(l_i,\ \vartheta_x,\ \vartheta_y)}{n} \tag{4.8}$$

Completing our example, the estimated isomorphism for ϑ_0 and ϑ_1 is:

$$\begin{aligned}
iso_{lev}(l_0,\ \vartheta_0,\ \vartheta_1) &= \frac{0+1+1/2}{3} = \frac{1}{2} \\
iso_{lev}(l_2,\ \vartheta_0,\ \vartheta_1) &= \frac{1/3+2/3+1}{3} = \frac{2}{3} \\
iso_{lev}(l_5,\ \vartheta_0,\ \vartheta_1) &= \frac{1/2+2/3+0.3764\ldots}{3} = 0.4588\ldots \\
iso_{est}(P_m(\vartheta_0),\ P_m(\vartheta_1)) &= \frac{1/2+2/3+0.4588\ldots}{3} \approx 0.54
\end{aligned}$$

4.3.4 Combining scores of different methods

So far we have only discussed the isomorphism of two test cases on a single method. Naturally, test cases may execute paths in more than one method, so to calculate a general similarity between two test cases, we have to define how we combine the separate isomorphism-scores on these methods for each pair $(\vartheta_i, \vartheta_k)$. We have chosen to simply take the average score of each method where either ϑ_i or ϑ_k has executed at least one path:

$$iso_{est}(\vartheta_i, \vartheta_k) = \frac{\sum_{m=0}^{n-1}\ iso_{est}(P_m(\vartheta_i), P_m(\vartheta_k))}{n} \tag{4.9}$$

Here, n is the number of methods where $(P_m(\vartheta_i) \cup P_m(\vartheta_k)) \neq \oslash$. For any pair $(\vartheta_i, \vartheta_k)$ where $n = 0$, we let $iso_{est}(\vartheta_i, \vartheta_k) = 0$.

Chapter 5

Evaluation

At the time of writing, ExsectJ contains the functionality to extract paths from arbitrary Java programs as presented in chapter 3 and supports the test case comparison technique as explained in chapter 4.

The primary objective in this evaluation is to verify that the output of ExsectJ - the calculated isomorphism scores - correspond with our intuition, with what we would have expected. In other words, we would like to see if ExsectJ calculates high scores for test cases that an expert would consider to be very similar (or at least closely related), and low scores for test cases that an expert would consider relatively unique or irreplaceable.

We will perform the evaluation with two case studies. In first case study, we have examined the output of ExsectJ on a small, self-created test suite. This enabled us to compare the output with results we had expected. In the second case study, we have analyzed the test suite of CheckStyle and tried to make sense of the output produced by our tool.

5.1 ModuleTest

In order to thoroughly examine the produced output, we chose to create a simple class, `Module`, containing three methods, for which we have written a set of test cases that are implemented in class `ModuleTest`. Both classes have been depicted in figures 5.1 and 5.2 respectively.

It may be clear from the implementations that we did not concern ourselves with designing a real-world class example with respect to functionality. Instead, we tried to implement three methods such that we could clearly evaluate the isomorphism-scores that were calculated by ExsectJ.

You may notice we only explicitly test the method `weirdMethod`, whereas the other two (private) methods are implicitly tested. Furthermore, we simulate a fault in `doSomethingElse` at `x = 7`, which can be triggered by executing `weirdMethod` with the same value of `x`.

Before we turn to the output of our tool, let's first look at these methods more closely. Method `weirdMethod` has basically two important decision points: `if (x > 10)` and `else if (x > 5)`. These decision points cut the relevant input space into three domains:

```
package testsystem.module;

public class Module {

        private Object[] objArray;

        public Module(int size) {
                if (size < 0) {
                        throw new IllegalArgumentException();
                }                                                          10
                objArray = new Object[size];
        }

        public int weirdMethod(int x) {
                if (x > 10) {
                        int result = 0;
                        for (int i = 0; i < x; i++) {
                                result += doSomething(i);
                        }
                        return result;                                     20
                }
                else if (x > 5) {
                        return doSomethingElse(x);
                }
                return -1;
        }

        private int doSomething(int i) {
                if (i > 40) {
                        return 20;                                         30
                }
                return i + 2 * objArray.length;
        }

        private int doSomethingElse(int x) {
                if (x == 7) {
                        throw new RuntimeException();
                }
                return 2 * x;
        }                                                                  40
}
```

Figure 5.1: Module: a simple case study class

```
package testsystem.module;

import org.junit.*;
import static org.junit.Assert.*;

public class ModuleTest {

        private Module module;

        @Before
        public void setUp() {
                module = new Module(4);
        }
        @Test
        public void weirdMethodTest_Negative() {
                assertEquals(-1, module.weirdMethod(-6));
        }
        @Test
        public void weirdMethodTest_Zero() {
                assertEquals(-1, module.weirdMethod(0));
        }
        @Test
        public void weirdMethodTest_Three() {
                assertEquals(-1, module.weirdMethod(3));
        }
        @Test
        public void weirdMethodTest_Five() {
                assertEquals(-1, module.weirdMethod(5));
        }
        @Test
        public void weirdMethodTest_Six() {
                assertEquals(12, module.weirdMethod(6));
        }
        @Test(expected=RuntimeException.class)
        public void weirdMethodTest_Seven() {
                module.weirdMethod(7);
        }
        @Test
        public void weirdMethodTest_Eight() {
                assertEquals(16, module.weirdMethod(8));
        }
        @Test
        public void weirdMethodTest_Ten() {
                assertEquals(20, module.weirdMethod(10));
        }
        @Test
        public void weirdMethodTest_Eleven() {
                assertEquals(143, module.weirdMethod(11));
        }
        @Test
        public void weirdMethodTest_Thirty() {
                assertEquals(675, module.weirdMethod(30));
        }
        @Test
        public void weirdMethodTest_OneThousand() {
                assertEquals(20328, module.weirdMethod(1000));
        }
}
```

Figure 5.2: ModuleTest: a simple case study test class, containing some redundant test

$[-\infty, 5]$, $[6, 10]$ and $[11, \infty]$. Therefore, it would be a good plan to test on the boundaries of these domains, and perhaps add a couple of test cases to test for one value somewhere in the middle of each domain. Hence, a good test suite would contain tests with the inputs `5`, `6`, `10` and `11` as boundary values, and perhaps three more test cases with inputs $a < 5$, $5 < b < 10$ and $11 < c$.

Now consider figure 5.2 where all implemented test cases are shown. Besides all the boundary values, multiple values for a (`-6`, `0` and `3`), b (`7` and `8`) and c (`30` and `1000`) were tested. Generally, we would consider this to be a reasonably effective test suite, but it does not necessarily have to be a very efficient test suite.

	t_{-6}	t_0	t_3	t_5	t_6	t_7	t_8	t_{10}	t_{11}	t_{30}	t_{1000}	μ: 0.93
t_{-6}		1.00	1.00	1.00	0.44	0.43	0.44	0.44	0.19	0.19	0.19	μ_{-6}: 1.00
t_0			1.00	1.00	0.44	0.43	0.44	0.44	0.19	0.19	0.19	μ_0: 1.00
t_3				1.00	0.44	0.43	0.44	0.44	0.19	0.19	0.19	μ_3: 1.00
t_5					0.44	0.43	0.44	0.44	0.19	0.19	0.19	μ_5: 1.00
t_6						0.50	1.00	1.00	0.19	0.19	0.19	μ_6: 1.00
t_7							0.50	0.50	0.19	0.19	0.19	μ_7: 0.50
t_8								1.00	0.19	0.19	0.19	μ_8: 1.00
t_{10}									0.19	0.19	0.19	μ_{10}: 1.00
t_{11}										0.91	0.90	μ_{11}: 0.91
t_{30}											0.92	μ_{30}: 0.92
t_{1000}												μ_{1000}: 0.92

Table 5.1: Test case isomorphism scores for method `weirdMethod`

	t_{11}	t_{30}	t_{1000}	μ: 0.80
t_{11}		1.00	0.41	μ_{11}: 1.00
t_{30}			0.41	μ_{30}: 1.00
t_{1000}				μ_{1000}: 0.41

Table 5.2: Test case isomorphism scores for method `doSomething`

	t_6	t_7	t_8	t_{10}	μ: 0.85
t_6		0.40	1.00	1.00	μ_6: 1.00
t_7			0.40	0.40	μ_7: 0.40
t_8				1.00	μ_8: 1.00
t_{10}					μ_{10}: 1.00

Table 5.3: Test case isomorphism scores for method `doSomethingElse`

Running ExsectJ produces the outputs shown in tables 5.1, 5.2 and 5.3, which are the results of our test case isomorphism-formula of chapter 4. Note that a score of 1.00 indicates that these test cases are probably very similar. The right-most column of this table

contains the highest similarity-score that occurred for each test case, which can be used as an indicator for how unique a specific test case is a certain test suite is.

With t_x representing a test case using x as input-value, the scores in table 5.1 clearly show that the sets $S_0 : \{t_{-6}, t_0, t_3, t_5\}$, $S_1 : \{t_6, t_8, t_{10}\}$ and $S_2 : \{t_{11}, t_{30}, t_{1000}\}$ present highly similar test cases, which is what we would expect concerning our input-domain analysis earlier. Furthermore, we can see that t_7 has been marked as relatively unique, *even though it basically executes the same path as the test cases of set* S_1.

The reason for this is that this value triggers the fault in method `doSomethingElse`, resulting in an exception that is not caught in `weirdMethod`. Recall that this unexpected exit has been modeled as a separate path in the CFG by adding the catch-block, and thus this test case is the only one that 'executes' this path, resulting in a relatively low isomorphism score. Also note that the test cases from S_2 are considered slightly less similar than the members of S_0 and S_1. This is caused by the fact that the test cases from S_2 all execute the loop a different amount of times (equal to their inputs). However, even though the difference in iterations between t_{11} or t_{30} and t_{1000} is considerate, the calculated isomorphism stays relatively high, because this number of iterations is also the *only* difference between the paths that are executed.

5.1.1 Reducing the test suite

If we were to remove some test cases to reduce the size of this test suite, we would be guided to eliminate some test cases from S_0, S_1 and S_2 and to certainly leave t_7 untouched. However, to make this decision, the scores for the other methods are also relevant. If we consider the scores of table 5.3, we quickly see that test cases t_6, t_8 and t_{10} are, again, highly similar, whereas t_7 stands out because it triggers the fault, so this information does not influence our options. If we consider table 5.2, we can observe that t_{11} and t_{30} are still equal but t_{1000} executes a different path. Because, in this case, t_{1000} is a relatively unique test case, we had better keep this test case as well.

Conclusively, it would be best to remove some test cases from S_0, S_1 and $S_{2a} : \{t_{11}, t_{30}\}$. This conclusion can also be based on the results that are shown in table 5.4, which contains the average isomorphism scores for each pair of test cases in the test suite as defined by equation 4.9. This table clearly shows the uniqueness of test cases t_7 and t_{10} and the higher similarity of the rest.

5.1.2 Estimating the efficiency

The reader may have noticed that the tables contain some more information in the right-most column of the tables. The number at the top of this column, μ, expresses the average of all maximum scores. Therefore, if many test cases execute similar paths, μ will be high, whereas μ will be lower when lots of test cases are unique. For this reason, we feel that $1 - \mu$ might be a good metric or indicator of how efficient a certain method has been tested and, using the average score table, how efficient the whole test suite is.

The efficiency-score for the original test suite, represented by `ModuleTest`, would then be calculated at 0.07, which doesn't seem to reflect a very efficient test suite at all. Suppose

	t_{-6}	t_0	t_3	t_5	t_6	t_7	t_8	t_{10}	t_{11}	t_{30}	t_{1000}	μ: 0.93
t_{-6}		1.00	1.00	1.00	0.22	0.21	0.22	0.22	0.09	0.09	0.09	μ_{-6}: 1.00
t_0			1.00	1.00	0.22	0.21	0.22	0.22	0.09	0.09	0.09	μ_0: 1.00
t_3				1.00	0.22	0.21	0.22	0.22	0.09	0.09	0.09	μ_3: 1.00
t_5					0.22	0.21	0.22	0.22	0.09	0.09	0.09	μ_5: 1.00
t_6						0.45	1.00	1.00	0.06	0.06	0.06	μ_6: 1.00
t_7							0.45	0.45	0.06	0.06	0.06	μ_7: 0.45
t_8								1.00	0.06	0.06	0.06	μ_8: 1.00
t_{10}									0.06	0.06	0.06	μ_{10}: 1.00
t_{11}										0.95	0.65	μ_{11}: 0.95
t_{30}											0.66	μ_{30}: 0.95
t_{1000}												μ_{1000}: 0.66

Table 5.4: Average test case isomorphism scores for test suite class ModuleTest

we would remove some test cases that have been found to be highly similar, such as t_{-6}, t_0, t_3, t_8, t_{10} and t_{30}. Recalculating the efficiency would result in a score of 0.51, which is significantly better.

However, even after removing almost half of the test cases, we still get a score that we consider reasonably low. It is arguable that taking the average of the *maximum* isomorphism scores is too strict, and other calculations may be suggested that provide results that are closer to our intuitively expected efficiency scores. For example, we may as well use the average isomorphism score of the whole table to determine μ. Unfortunately, we had no time left to investigate what would be the best way to calculate the efficiency of a test suite, so this subject is still open for debate.

5.2 CheckStyle

In our second case study, we ran ExsectJ on CheckStyle 4.4, an open source plug-in for Eclipse, that helps a project to maintain a single coding standard.[1] CheckStyle contains about 250 classes divided amongst 18 packages and has a test suite of 19 packages, 156 classes and a total of 546 test case methods. It might be important to mention here that all test cases pass, so we don't expect any test cases to stand out because they trigger a certain fault.

The following section explains how ExsectJ was configured and how it performed over several executions of CheckStyle's test suite. Section 5.2.2 discusses the output that was generated.

5.2.1 General usage and performance

Although we are still dealing with a prototype, we thought it would be nice to provide some information about the current use, performance and overhead of ExsectJ. All analysis was

[1] More information about CheckStyle can be found at `http://checkstyle.sourceforge.net`.

performed on a private laptop using an Intel Pentium M at 1.7 Ghz with 1.0 GB of RAM, running MS Windows XP SP 2 and Eclipse Platform version 3.3.0.
CheckStyle uses ANT to build the application and run the test suite. The `build.xml` file contains one target where JUnit (version 3.x) is launched. To analyze the test suite, we had to add three lines of XML as depicted in figure 5.3.

```
<jvmarg value="-Xmx768m" />
<jvmarg value="-javaagent:${insectj_runtime.jar}=${testclasses.xml}" />
<jvmarg value="-javaagent:${insectj_runtime.jar}=${systemclasses.xml}" />
```

Figure 5.3: Required JVM-arguments to run ExsectJ on CheckStyle

Note that ExsectJ, that acts as a java agent (see appendix A), is packed into a single jar-file. The first line in the figure simply allows a maximum of 768 MB of internal memory to be used by the JVM. The second and third argument call on ExsectJ to instrument all the test cases and all the system classes. We have to specify two different xml-files because the test cases need to be instrumented in a different way than the system classes, and InsectJ currently allows only one set of classes to be specified for one or more types of instrumentation per config-file.

Run-time overhead

Using the default time-measurements of JUnit, the test cases of CheckStyle take up a total of about 17 seconds to complete. In contrast, when ExsectJ is activated, execution of these test cases takes about 84 seconds to complete, which means a factor of 5 slower. Recall that while the test cases are being executed, the only thing that ExsectJ does is maintaining a custom call-stack of active test cases and recording the path-sum records as depicted in figure 3.11. We believe this overhead is primarily caused by copying and attaching the current test case stack to each recorded acyclic path.

After processing

There were basically two ways to execute and analyze the test suite, because ANT provides two ways to execute JUnit. The first option is to create a single JVM in which all test cases are run. CheckStyle uses this option by default, and it is also the only option we have if we want to compare all test cases in the whole test suite. The second option is to create a separate JVM for each test case class. If we choose this type of execution, we can only compare the test cases of a single test case class with each other. However, we have noticed that the peak memory use is in this case considerably lower, which enabled us to analyze more test cases in a single run than while using a single JVM.

In the first few runs, we tried to analyze the whole test suite using a single JVM. Unfortunately, ExsectJ required too much time and memory to analyze and compare all test cases of the entire test suite at once. With a trial and error approach, we got it to calculate the

similarity between 250 test cases, slightly less than half of the whole test suite, taking about 15 minutes, with a peak memory consumption of about 700 MB.

Since we did want to gather data on the whole test suite, we also ran the test suite using a separate JVM for each test case class. Although we now had the advantage of less memory consumption, we were still forced to exclude two (important) system classes, `StringArrayReader` and `DetailAST`, simply because too many paths were executed in these classes, even by test cases that did not test these classes explicitly. However, when these classes were excluded, ExsectJ ran stable and smoothly. It took about 32 minutes to run and calculate the similarity of all test cases.

5.2.2 Interpreting the data

In each run, ExsectJ produces a set of html-pages containing the similarity scores. One of these pages contains the table with the average similarity-scores (like table 5.4). In this section, we will show and review the data of these tables.

After the first run, in which we analyze all test cases at once, an average-score table with $250^2 = 62500$ scores is produced. Taken the number of entries, we chose to convert this table into a color-map, which is shown in figure 5.4. Low isomorphism-scores are represented by 'cold' colors (blue, green) whereas high scores are represented by 'hot' colors (orange, red).

The color-map provides a nice overview of the test suite as a whole (or at least of the part we were able to analyze). The first thing that catches your eye in the figure is, of course, the large blue area, representing the isomorphism scores of all test cases in the range [0, 45], which is in shear contrast with the rest of the test suite.

Furthermore, the figure clearly shows that test cases in the ranges [155, 180], [185, 210] and [215, 230] show high similarity with other test cases, especially within there own range. However, for practical reasons, it would too much work to analyze all this data by hand.

To evaluate the scores from figure 5.4 we ran ExsectJ a second time, in this case with every test case class in its own JVM, each producing one table of (average) scores. Hereafter we manually went through a series of 155 score-tables and selected some test case classes for closer inspection of the scores.[2]

As could have been expected by having observed the color-map in figure 5.4, many test case classes contain test cases that are marked as highly similar. To be more precise, more than 25 test case classes contain at least four test cases that show isomorphism-scores between 0.95 and 1.00. Based on the exact scores and the names of the test cases, we have carefully analyzed a small number of test case classes. Figures 5.5, 5.6 and 5.7[3] show the implementations of several test cases that have obtained high similarity scores when compared to each other. Furthermore, we have also added some test cases with a very low

[2] We mentioned earlier that there are 156 test case classes, but we excluded class `DetailASTTest`, since we also excluded class `DetailAST` for reasons explained earlier, and we figured that any measurements on this class would therefore be meaningless.

[3] To save some space, we have replaced some lines of code in method `testIt()` with dots.

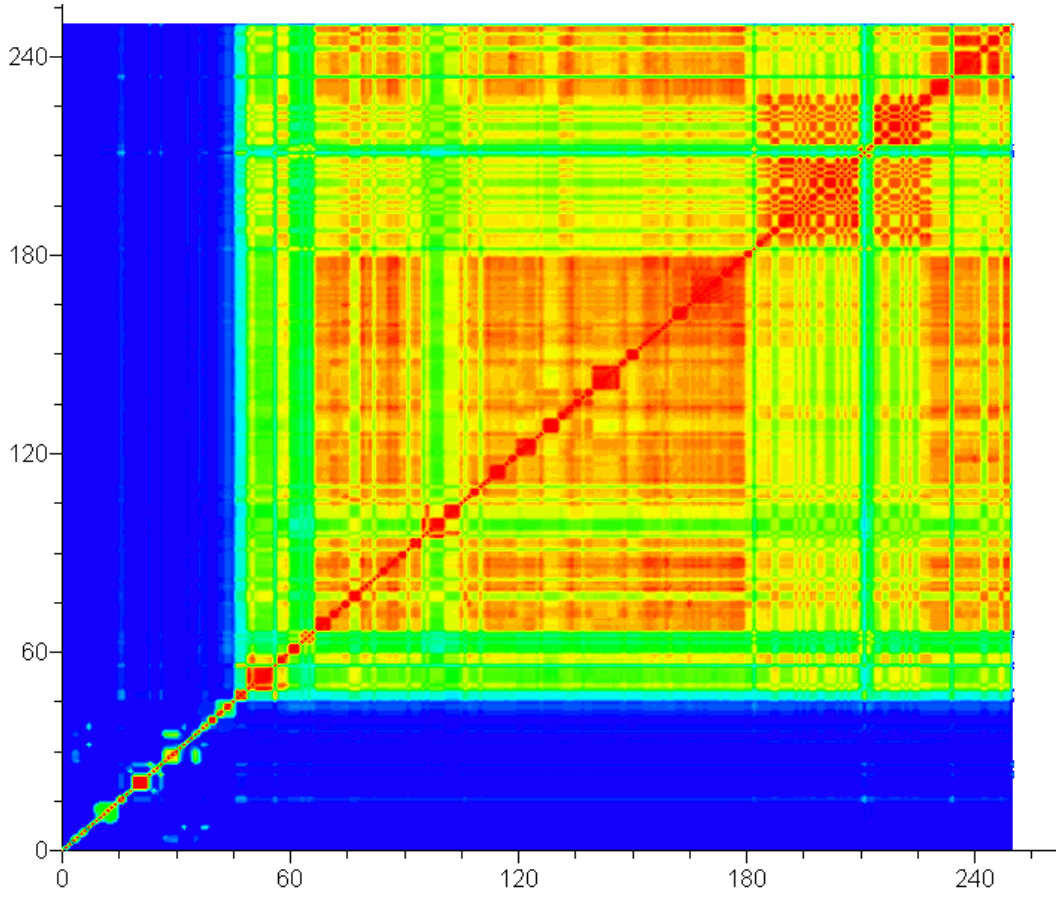


Figure 5.4: CheckStyle's test case similarity visualized in a color map. Each test case is represented by a unique number along the horizontal and vertical axis. Red areas reveal test cases that execute highly similar paths in the system.

similarity (which were identified as cases 12, 13 and 14 in figure 5.4). These cases are only exemplary, but they are a good representation of most of CheckStyle's test cases.

Needless to say, the three sets of test cases displayed in the figures immediately appear similar because of their equivalent coding structure and setup. If we take the methods of class `ConfigurationLoaderTest` (figure 5.5), for example, we can see that the environment of their execution and the methods they invoke are exactly equal: they only use different arguments. Apparently, the change in arguments does not cause much difference in the way the code in the system classes are executed.

In contrast, the test cases from class `JavadocMethodCheckTest` only differ in their setup: the instance represented by `mCheckConfig` is initialized differently each time, but the calls and used arguments remain the same. Again, this seems to have very little effect on the executed paths in this case, and the same holds for the displayed test cases of class `WhiteSpaceAroundTest`.

```
public void testMissingPropertyName()
{
    try {
        loadConfiguration("missing_property_name.xml");
        fail("missing property name");
    }
    catch (CheckstyleException ex) {
        assertTrue(
            ex.getMessage().endsWith(
                "Attribute \"name\" is required and must be specified "
                    + "for element type \"property\".:8:41"));
    }
}

public void testMissingPropertyValue()
{
    try {
        loadConfiguration("missing_property_value.xml");
        fail("missing property value");
    }
    catch (CheckstyleException ex) {
        assertTrue(
            ex.getMessage().endsWith(
                "Attribute \"value\" is required and must be specified "
                    + "for element type \"property\".:8:41"));
    }
}

public void testMissingConfigName()
{
    try {
        loadConfiguration("missing_config_name.xml");
        fail("missing module name");
    }
    catch (CheckstyleException ex) {
        assertTrue(
            ex.getMessage().endsWith(
                "Attribute \"name\" is required and must be specified "
                    + "for element type \"module\".:7:23"));
    }
}

public void testMissingConfigParent()
{
    try {
        loadConfiguration("missing_config_parent.xml");
        fail("missing module parent");
    }
    catch (CheckstyleException ex) {
        assertTrue(
            ex.getMessage().endsWith(
                "Document root element \"property\", must match DOCTYPE "
                    + "root \"module\".:7:38"));
    }
}
```

Figure 5.5: Four similar test cases in class ConfigurationLoaderTest

```
public void test_generics_1() throws Exception
{
    mCheckConfig.addAttribute("allowThrowsTagsForSubclasses", "true");
    mCheckConfig.addAttribute("allowUndeclaredRTE", "true");
    final String[] expected = {
        "15:34: Expected @throws tag for 'RE'.",
        "23:37: Expected @throws tag for 'RE'.",
        "31:13: Expected @param tag for '<NPE>'.",
        "38:12: Unused @throws tag for 'E'.",
        "41:38: Expected @throws tag for 'RuntimeException'.",
        "42:13: Expected @throws tag for 'java.lang.RuntimeException'.",
    };
    verify(mCheckConfig, getPath("javadoc/TestGenerics.java"), expected);
}

public void test_generics_2() throws Exception
{
    mCheckConfig.addAttribute("allowThrowsTagsForSubclasses", "true");
    final String[] expected = {
        "15:34: Expected @throws tag for 'RE'.",
        "23:37: Expected @throws tag for 'RE'.",
        "31:13: Expected @param tag for '<NPE>'.",
        "38:12: Unused @throws tag for 'E'.",
        "41:38: Expected @throws tag for 'RuntimeException'.",
        "42:13: Expected @throws tag for 'java.lang.RuntimeException'.",
    };
    verify(mCheckConfig, getPath("javadoc/TestGenerics.java"), expected);
}

public void test_generics_3() throws Exception
{
    final String[] expected = {
        "6:8: Unused @throws tag for 'RE'.",
        "15:34: Expected @throws tag for 'RE'.",
        "31:13: Expected @param tag for '<NPE>'.",
        "38:12: Unused @throws tag for 'E'.",
        "41:38: Expected @throws tag for 'RuntimeException'.",
        "42:13: Expected @throws tag for 'java.lang.RuntimeException'.",
    };
    verify(mCheckConfig, getPath("javadoc/TestGenerics.java"), expected);
}
```

Figure 5.6: Three similar test cases in class JavadocMethodCheckTest

```
public void testIt()
    throws Exception
{
    final String[] expected = {
        "16:22: '=' is not preceded with whitespace.",
        "16:23: '=' is not followed by whitespace.",
        ...
        "156:20: ':' is not preceded with whitespace.",
        "156:21: ':' is not followed by whitespace.",
    };
    verify(checkConfig, getPath("InputWhitespace.java"), expected);
}

public void testIt2()
    throws Exception
{
    final String[] expected = {
        "153:27: '=' is not followed by whitespace.",
        "154:27: '=' is not followed by whitespace.",
        "155:27: '=' is not followed by whitespace.",
        "156:27: '=' is not followed by whitespace.",
        "157:27: '=' is not followed by whitespace.",
        "158:27: '=' is not followed by whitespace.",
    };
    verify(checkConfig, getPath("InputSimple.java"), expected);
}

public void testIt3()
    throws Exception
{
    final String[] expected = {
        "41:14: 'while' is not followed by whitespace.",
        "58:12: 'for' is not followed by whitespace.",
        // + ":58:23: ';' is not followed by whitespace.",
        //   + ":58:29: ';' is not followed by whitespace.",
        "115:27: '{' is not followed by whitespace.",
        "115:27: '}' is not preceded with whitespace.",
        "118:40: '{' is not followed by whitespace.",
        "118:40: '}' is not preceded with whitespace.",
    };
    verify(checkConfig, getPath("InputBraces.java"), expected);
}

public void testIt4()
    throws Exception
{
    checkConfig.addAttribute("allowEmptyMethods", "true");
    checkConfig.addAttribute("allowEmptyConstructors", "true");
    final String[] expected = {
        "41:14: 'while' is not followed by whitespace.",
        "58:12: 'for' is not followed by whitespace.",
    };
    verify(checkConfig, getPath("InputBraces.java"), expected);
}
```

Figure 5.7: Four similar test cases in class WhiteSpaceAroundTest

```
public void testBlockOption()
{
    BlockOption stmtOpt = BlockOption.STMT;
    assertEquals("STMT", "stmt", stmtOpt.toString());
    BlockOption textOpt = BlockOption.TEXT;
    assertEquals("TEXT", "text", textOpt.toString());
    BlockOption stmtDecode = (BlockOption)(stmtOpt.decode("stmt"));
    assertTrue("STMT decode", stmtOpt == stmtDecode);
    BlockOption textDecode = (BlockOption)(stmtOpt.decode("text"));
    assertTrue("TEXT decode", textOpt == textDecode);                    10
}

public void testLeftCurlyOption()
{
    LeftCurlyOption eolOpt = LeftCurlyOption.EOL;
    assertEquals("EOL", "eol", eolOpt.toString());
    LeftCurlyOption nlOpt = LeftCurlyOption.NL;
    assertEquals("NL", "nl", nlOpt.toString());
    LeftCurlyOption nlowOpt = LeftCurlyOption.NLOW;
    assertEquals("NLOW", "nlow", nlowOpt.toString());                    20
    LeftCurlyOption eolDecode = (LeftCurlyOption)(eolOpt.decode("eol"));
    assertTrue("EOL decode", eolOpt == eolDecode);
    LeftCurlyOption nlDecode = (LeftCurlyOption)(nlOpt.decode("nl"));
    assertTrue("NL decode", nlOpt == nlDecode);
    LeftCurlyOption nlowDecode = (LeftCurlyOption)(nlowOpt.decode("nlow"));
    assertTrue("NL decode", nlowOpt == nlowDecode);
}

public void testOperatorWrapOption()
{                                                                        30
    OperatorWrapOption eolOpt = OperatorWrapOption.EOL;
    assertEquals("EOL", "eol", eolOpt.toString());
    OperatorWrapOption nlOpt = OperatorWrapOption.NL;
    assertEquals("NL", "nl", nlOpt.toString());
    OperatorWrapOption eolDecode = (OperatorWrapOption)(eolOpt.decode("eol"));
    assertTrue("EOL decode", eolOpt == eolDecode);
    OperatorWrapOption nlDecode = (OperatorWrapOption)(nlOpt.decode("nl"));
    assertTrue("NL decode", nlOpt == nlDecode);
}
```

Figure 5.8: Three test cases in class OptionTest with very low similarity

It is also not hard to see why the test cases from figure 5.8 are marked as very unique: each test case seems to be responsible for testing a different class. Consequently, the paths they execute do not overlap at all.

In the start of this chapter we mentioned that our main objective was to evaluate whether ExsectJ's output conforms to an expert's opinion - ourselves in this case - about which test cases are similar and which are not. In our first case study, simulating a very basic unit testing environment, ExsectJ performed exactly as expected. However, after applying it to CheckStyle's test suite, this main question is hard to answer.

First of all, an expert's opinion is quite subjective: there is no clear consensus on when test cases are similar and when they are not. Secondly, even when, for instance, someone would think the test cases are truly similar, he or she must still admit these test cases are not completely redundant, because they do seem to test different key functionality.

Another remarkable result was that, completely against our own expectations, about 75% of all test cases execute paths in the system that reasonably to highly overlap. The most likely reason for this is that many test case classes extend class `BasicCheckTestCase`, which is an abstract class that implements basic functionality to easily run specific checks using a specific configuration and some external file. Consequently, every concrete subclass uses the same core system-classes to execute their specific checks (the actual test cases), causing an almost default overlap of many paths in these core classes.

If we were to take a final stand in our view of ExsectJ's performance, we would say that it proves to be a nice prototype tool that can create a useful overview of the efficiency of the implemented test suite (through color-maps as in figure 5.4) and consequently acts as a good guide to specific test case classes that may contain some redundant test cases.

Chapter 6

Related Work

Since our work is essentially three-fold, we will compare it on three different levels with existing techniques and solutions. First, we will examine some topics related to path profiling. Secondly, we will review our work on path-comparison techniques. Finally, we will outline some existing work related to test suite reduction and test efficiency calculation.

6.1 Path profiling

A common way to extract run-time information from a Java program is by using the standard Java Virtual Machine Profiler Interface, or simply JVMPI [44]. This profiler interface framework allows to extract many different kinds of run-time information, such as execution times, memory use and information concerning the garbage collector, and for this kind of data, the JVMPI may be a better choice then instrumenting a program and implementing the whole profiling framework yourself. However, the JVMPI is quite intrusive if compared to observing a running program by instrumenting it. Moreover, the JVMPI does not allow to observe the execution of run-time paths, which was a deliberate choice concerning the performance cost. In this respect, instrumenting a program is also more precise.

In the area of program-instrumentation to extract run-time paths, the work of Ball and Larus [4] cannot remain unnoticed. Their algorithm has been used extensively for many profiling techniques [2, 40] and has been extended to support inter-procedural profiling [24, 31, 39]. Therefore, applying or implementing their algorithm is hardly new. However, our work is somewhat different in two ways.

First of all, by extending the InsectJ framework [36], we enable flexible profiling of arbitrary Java programs with minimal run-time overhead. By maintaining customized xml-configuration-files, the profiling of different paths of execution remains limited to the classes of choice, reducing the overhead and resource-consumption of profiling an entire program, while there may only be interest in the analysis of small parts of a program or test suite. Furthermore, ExsectJ allows custom implementations for the processing of the recorded path-data by extending the base path-monitor class (also see appendix A).

Close to this concept is the online path profiling framework for Just-In-Time compilers, proposed by Toshiaki Yasue *et al* [47]. This framework basically instruments methods during the compilation phase to retrieve so-called *hot paths*, which are in this case the acyclic paths described in chapter 3 that are executed very frequently. By doing so, methods are scheduled for better optimization for the compiler. The essential differences with our framework are that ExsectJ has not been developed or optimized for one specific goal, such as code-optimization, and we also support the extraction of complete paths instead of acyclic paths only.

Secondly, we have proposed and implemented an enhanced control-flow graph that models not only the visible, basic control-flow of all the instructions, but also the *invisible* control-flow such as abnormal exits of a method or system. This solution allows a path to be recorded even if it is interrupted by an exception unexpectedly. Anticipation of this occurrence in itself is not new. In an earlier article [3], Thomas Ball describes the event-counting algorithm which later evolved to the efficient path profiling we used, and here he describes a way to adjust the event-counter such that incomplete paths can be regenerated. Without going into the details of this procedure, it is obviously a different approach than instrumenting a method such that all exceptions are caught and control-flow is directed to a catch-block. Moreover, by modeling unexpected exits as new paths, we can easily discriminate between test cases that triggered these exits to occur.

6.2 Path-comparison techniques

Because we knew we were to compare the extracted paths and calculate their similarity, we tried to find existing work on the subject of comparing path-like data and grouping them based on similarity-analysis. The literature we found in the area of computer science was minimal, and shifting to a broader scope, we encountered several articles on comparing DNA-strings and data-clustering techniques [29, 23]. With respect to similarity-analysis, the closest thing to our purpose was an article on general similarity-coefficients [16].

However, none of these general similarity-analysis techniques accounted for the properties of run-time paths from a computer-program, which is why we chose to design our own comparison-technique. Consequently, to our knowledge, we are the first to compare path-sets such as described in chapter 4.

6.3 Test suite reduction and efficiency

Because designing, implementing and running test suites is a very costly activity, limiting the growth of test suite size may be lucrative. For this reason, some techniques have been developed to help reduce the size of test suites.

Most articles we have encountered propose or discuss methods that will automatically calculate a more efficient or smaller test suite by selecting a subset of test cases which still satisfy one or more test adequacy criteria, such as a minimally suggested code coverage score [21]. This is different from our approach, since we do not aim to *automatically* reduce the size of a test suite: so far, we only calculate the similarity of different test cases on

certain methods and classes to indicate where a programmer or tester may have over-tested and there's an opportunity to manually reduce the test set himself.

The advantage of this is that a programmer can usually decide best which test cases are more valuable than another, while this is very hard to implement in automatic reduction. Automatic test suite reduction techniques are usually *unsafe*, which means they may remove many key test cases that are highly valuable, not only in revealing faults but also for regression testing purposes. However, Binder also talks about *safe* reduction of test suites [9]. A safe reduction essentially means cutting away those test cases that do not touch code that may or will be changed. Keeping those test cases is pointless since they will never reveal regression-faults in the future, and removing them during the development process will therefore not affect the effectiveness of a test suite[1].

Leon and Podgurski have summarized and compared the effectiveness of several test suite reduction techniques [28]. They explain there are basically two types of test suite reduction techniques: one type uses coverage criteria to select or filter a subset of test cases, whereas the other type divides a certain system into several clusters and selects a small set of test cases from each other, which is called *distribution-based filtering*.

The former type can be applied to existing test suites but is generally also used in test case generation tools to limit the amount of test cases generated. For example, one tool called Eclat [34] aims to generate a set of test cases that are fault-revealing. The fault-revealing property is determined by checking whether a test case triggers a certain exception in the code under test. Although the generated set of test cases might contain some test cases that actually reveal some faults (note that not every triggered exception is a fault-indication), it will not find faults related to wrong output, and it will not generate test cases that might be good regression tests. In other words: this approach may lead to a fairly ineffective test suite, because it may discard valuable test cases. Some tools try to overcome this problem by generating test cases from formal specifications [10, 32] such as the Java Modeling Language [26, 27, 11]. Formal specifications enable checks on test case input (pre-conditions) as well as system output (post-conditions), but formal specifications are still not very common and require a lot of extra time and effort to implement.

Other generators evaluate their test cases based on some form of code coverage [35, 1]. These tools may obtain great coverage scores and may therefore generate very effective test suites. However, even for the most sophisticated tools, the problem remains that some test cases may be thrown away which an expert would have considered highly valuable for reasons other than its code coverage score.

A remarkable outcome of the investigation of Leon and Podgurski is that test suite reduction using distribution-based filtering can be as effective or even more effective than coverage-based techniques. They also show that increasing the granularity of the measured code coverage (from basic block to branch coverage) had only a minimal increase in the resulting effectiveness of the reduced test suite. However, the difference between basic block and branch coverage may be a lot smaller than the difference between branch coverage and path coverage, which makes it hard to predict whether the same reduction technique based

[1] That is, of course, only true if all these test cases are always executed in the same environment, with no randomness included.

on path coverage would have shown the same minimal increase in effectiveness of the reduced test suite. We think it may therefore be interesting to see how a test case reduction or a test case generation tool, based on ExsectJ's output to determine the (relative) value of test cases, would perform.

Chapter 7

Concluding Remarks

7.1 Contributions

The work described in this thesis makes the following contributions:

- We have extended the work of Ball and Larus [4] such that it is possible to extract intra-method paths from arbitrary Java programs. This extension consists of two parts:
  1. We explicitly model exceptional control-flow in the CFG as separate paths and also take into account exceptions that are not thrown or caught inside a certain method.
  2. All regenerated acyclic paths are re-aligned to form complete intra-method paths, while supporting the use of threads and recursive calls.
- To our knowledge, we have come up with a new approach for calculating the similarity between two test cases by
  1. encapsulating loops into separate *levels*, and
  2. defining a formula that approximates the similarity of two test cases by comparing the (sets of) paths that have been executed.
- We have designed and implemented ExsectJ, a generic, open source prototype tool that extends the InsectJ framework and contains all the functionality to calculate the efficiency of a test suite using both techniques described above.
- We have shown the results of our approach by applying our tool within two case studies and concluded that the similarity-scores can provide a nice overview of how efficient a test suite has been implemented and reveal test cases that might be very similar.

We achieved these contributions as follows. At first, we rephrased our main objective to the problem of finding test cases that are highly similar, for we reasoned that large areas of similar test cases might indicate inefficient testing or deliberate thorough testing. In any

case, we reckoned the acquired information might be helpful to point to parts of a test suite where there is a possibility to improve or reduce the size of a test suite.

Consequently, we needed a way to measure the similarity between test cases. Since there is no clear consensus on how to determine to what degree two test cases are similar, the solution to this problem was not straightforward. As outlined in the previous chapter, we have looked at the techniques that are used by existing, automated test suite reduction and test case generation tools and found that most tools use code coverage measurements to evaluate the *effectiveness* of a test suite, but not the *efficiency*. For these reasons, we wanted to suggest a different approach and proposed to estimate the similarity between test cases by extracting and comparing the run-time paths that they execute.

To evaluate this approach, we required a tool that could measure the paths that were executed by each test case. Since we could not find any existing tools that were suited enough to be adopted or extended for this cause, we decided to implement our own tool, using an existing and efficient path profiling algorithm [4].

Furthermore, we needed an algorithm that would give us a good estimation of the similarity of two test cases by analyzing their paths. Again, we were unable to find existing work that would serve this purpose, and therefore we designed and implemented a new approach.

Luckily, we did have the opportunity to extend the InsectJ framework, which saved us a great deal of time in the setup of our own tool. However, we then decided to extend this framework the way it was meant to be extended: in addition to implementing our own functionality to perform our case studies, we also tried to design ExsectJ to allow others to use, change or extend this functionality without much difficulty.

7.2 Future work

In the current situation, ExsectJ produces a set of html-pages containing similarity-scores of (paths of) test cases and one table where the scores for all test cases are averaged when compared to each other. This means that all this data has to be analyzed by hand, which is, as we have encountered in our evaluation on CheckStyle's test suite, not very practical for test suites with hundreds of test cases. One important improvement would therefore be to extend the tool such that parts with a lot of similar test cases or other areas that contain interesting information can be analyzed and inspected in a more automated and apprehensible way.

Another option is to integrate the path-comparison algorithm with existing test suite reduction or test case generation tools such as EvoTest [35] and see if it performs well in this area when compared to existing test suite evaluation techniques. Applying our tool in this area also provides the opportunity to test and experiment with different path-comparison algorithms.

Moreover, instead of trying to find an optimal path-comparison formula, one could also extract more run-time information to determine the (relative) value of test cases. One could, for instance, try and detect if test cases are testing on boundary values and mark them as valuable if they do. Such information would be useful when trying to reduce a test suite and one wants to keep only the most valuable test cases.

In a broader sense, ExsectJ could be used or extended to support any research where extracting run-time (path-)information is required. However, since our tool is only a prototype, we think there is an opportunity to improve its performance in both speed and memory consumption and encourage others to do so in the future.

Bibliography

[1] B T Abreu de, E Martins, and F L Sousa de. Automatic test data generation for path testing using a new stochastic algorithm. `http://www.sbbd-sbes2005.ufu.br/arquivos/16-\%209523.pdf`, 2004. Universidade Estadual de Campinas, Instituto de Computacao (Brasil).

[2] Taweesup Apiwattanapong and Mary Jean Harrold. Selective Path Profiling. In *Program Analysis for Software Tools and Engineering*, pages 35–42, 2002.

[3] Thomas Ball. Efficiently Counting Program Events with Support for on-Line Queries. *ACM Transactions on Programming Languages and Systems*, 16(5):1399–1410, September 1994.

[4] Thomas Ball and James R. Larus. Efficient Path Profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[5] M Barnett, BY E Chang, R DeLine, B Jacobs, and R M Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. `http://www.cs.berkeley.edu/~bec/2006-boogie/fmco05-boogie.pdf`, 2006. Microsoft Research, Redmond.

[6] M Barnett, R M Leino, and W Schulte. *The Spec# Programming System: An Overview*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004. ISBN: 3-540-24287-2.

[7] Joachim Berg, van den and Bart Jacobs. *The LOOP Compiler for Java and JML*, volume 2031 of *Lecture Notes in Computer Science*, pages 299+. Springer Berlin / Heidelberg, 2001. ISSN 0302-9743.

[8] D Berndt, J Fisher, Johnson L, J Pinglikar, and A Watkins. Breeding Software Test Cases with Genetic Algorithms. The Computer Society, 2003. `http://www.coba.usf.edu/berndt/research/papers/hicss2003ga.pdf`.

[9] R.V. Binder. *Testing Object-Oriented Systems*. Addison Wesley, 4th edition, 2003.

[10] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA), Rome, Italy, July 22–24*, Cambridge, MA, USA, 2002. MIT Laboratory for Computer Science. `citeseer.ist.psu.edu/boyapati02korat.html`.

[11] L Burdy, Cheon Y, D R Cok, M D Ernst, J R Kiniry, G T Leavens, K R M Leino, and E Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3), 2005. `http://www.cs.ru.nl/E.Poll/publications/`.

[12] David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML - Progress and issues in building and using ESC/Java2. Submitted for publication, 2004. `citeseer.ist.psu.edu/cok04escjava.html`.

[13] K. Cooper, T. Harvey, and T. Waterman. Building a Control-flow Graph from Scheduled Assembly Code, 2002. Technical Report TR02-399, Rice University.

[14] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[15] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[16] Andria da Silva Meyer, Antonio Augusto Franco Garcia, Anete Pereira de Souza3, and Cludio Lopes de Souza Jr. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (Zea mays L). *Genetics and Molecular Biology*, 27(1):83–91, 2004.

[17] James L Dalley. The Art of Software Testing. In *Aerospace and Electronics Conference, 1991. NAECON 1991., Proceedings of the IEEE 1991 National*, pages 757–760, 1991.

[18] Eleftherios Koutsofios and Stephen C. North. *Drawing graphs with dot*. Murray Hill, NJ, January 2006.

[19] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, volume 37, pages 234–245, June 2002. `citeseer.ist.psu.edu/flanagan02extended.html`.

[20] Dick Grune, Henri E Bal, Ceriel J H Jacobs, and Koen G Langendoen. *Modern Compiler Design*. Wiley, 2000. ISBN: 0-471-97697-0.

[21] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

[22] B Jacobs and E Poll. *Java Program Verification at Nijmegen: Developments and Perspective*, volume 3233 of *Lecture Notes in Computer Science*, pages 134–153. Springer Berlin / Heidelberg, 2004. ISBN 978-3-540-23635-1.

[23] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[24] James R Larus. Whole Program Paths. In *Conference on Programming Language Design and Implementation*, pages 259–269, 1999.

[25] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Technical Report CS-TR-92-1083, Madison, WI, USA, March 1992.

[26] G. Leavens and Y. Cheon. Design by Contract with JML. `citeseer.ist.psu.edu/leavens04design.html`, 2003.

[27] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Jml: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999. `citeseer.ist.psu.edu/leavens99jml.html`.

[28] David Leon and Andy Podgurski. A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases.

[29] Anna M. Manning, Andy Brass, Carole A. Goble, and John A. Keane. Clustering Techniques in Biological Sequence Analysis. In *Principles of Data Mining and Knowledge Discovery*, pages 315–322, 1997.

[30] N Mansour and M Salame. Data Generation for Path Testing. *Software Quality Journal*, 12(2):121–136, 2004. ISSN 0963-9314.

[31] David Melski and Thomas W. Reps. Interprocedural Path Profiling. In *Computational Complexity*, pages 47–62, 1999.

[32] Jeff Offutt and Aynur Abdurazik. Generating Tests from UML Specifications. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723, pages 416–429. Springer, 1999.

[33] C Oriat. *Jartege: A Tool for Random Generation of Unit Tests for Java Classes*, volume 3712 of *Lecture Notes in Computer Science*, pages 242–256. 2005. ISBN: 978-3-540-29033-9.

[34] Carlos Pacheco and Michael D. Ernst. *Eclat: Automatic Generation and Classification of Test Inputs*, volume 3586 of *Lecture Notes in Computer Science*, pages 504–527. 2005. ISBN 978-3-540-27992-1.

[35] A Seesing. EvoTest: TestCase Generation using Genetic Programming and Software Analysis. Master's thesis, TU Delft, Mekelweg 4, June 2006.

[36] Arjan Seesing and Alessandro Orso. InsECTJ: A Generic Instrumentation Framework for Collecting Dynamic Information within Eclipse. Master's thesis, Georgia Institute of Technology, July 2006. `http://insectj.sourceforge.net`.

[37] M Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997. ISBN 0-534-94728-X.

[38] Jim Smith and Terence C. Fogarty. *Evolving Software Test Data - GA's learn Self Expression*, volume 1143 of *Lecture Notes in Computer Science*, pages 137–146. 1996. Evolutionary Computing, ISBN 978-3-540-61749-5.

[39] Sriraman Tallam, Xiangyu Zhang, and Rajiv Gupta. Extending Path Profiling across Loop Backedges And Procedure Boundaries. In *Code Generation and Optimization*, page 251, 2004.

[40] M. Tikir and J. Hollingsworth. Efficient instrumentation for code coverage testing, July 2002. ISSTA.

[41] W van Gool. A survey on various verification and test-generation techniques, January 2007. TU Delft, Mekelweg 4.

[42] W Visser, K Havelund, G Brat, S Park, and F Lerda. Model Checking Programs. Technical Report 10, Automated Software Engineering, 2003.

[43] Willem Visser, Klaus Havelund, Guillaume Brat, and Seung-Joon Park. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003. ISSN 0928-8910.

[44] D Viswanathan and S Liang. JVM Profiler Interface. *IBM Systems Journal*, 39(2), 2000.

[45] Jeffrey Voas. Fault Injection for the masses. *Computer archive*, 30(12):129–130, December 1997.

[46] W. Eric Wong, Joseph R. Horgan, Aditya P. Mathur, and Alberto Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. *The Journal of Systems and Software*, 48(2):79–89, 1999.

[47] Toshiaki Yasue, Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. An efficient online path profiling framework for java just-in-time compilers. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 148, Washington, DC, USA, 2003. IEEE Computer Society.

[48] A Zanderink. Measuring the multiple-condition coverage with test suites for AspectJ programs, June 2006. Twente Student Conference on IT.

[49] Hong Zhu, Patrick A.V. Hall, and John H.R. May. Software Unit Test Coverage and Adequacy. *ACM Computing surveys*, 29(4), December 1997.

Appendix A

Extending InsectJ

In this chapter, we will provide a more in-depth description of the InsectJ framework and how we extended it.

A.1 The InsectJ Framework

Arjan Seesing and Alex Orso developed InsectJ at the Georgia Institute of Technology and published this framework is an earlier article [36]. To improve the readability of this paper, we will give a brief summary of this article here, but for more information we refer to the original article.

The core of the framework acts as a *Java agent*. Java agents are libraries that are used to perform special operations before a regular java application is run, typically to instrument the program. These libraries are activated on the command-line using the `-javaagent` option. When running Java's help in the console it shows the following use of these agents:[1]

```
-javaagent:<jarpath>[=<options>]
```

Here, `<jarpath>` should point to the location of the agent library, whereas the use of `[=<options>]` is optional. These options are comparable to regular command-line options for a main-method of a Java application, but is parsed as a single `String`. Note that multiple agents can be specified on the command-line.

The InsectJ framework and inheritly any extensions to it such as ExsectJ are packed into a single jar-file and are run as:

```
java -javaagent:exsectj.jar=config.xml MyObservedApp
```

The `config.xml` is an xml-file where the type of instrumentation, the type of after-processing of run-time information and the classes and methods to instrument are specified. Figure A.1 shows an example of such a config-file. As is shown in the example, the xml-file actually defines which classes to use for instrumenting methods and monitoring their run-time behavior. This works as follows.

[1] More information can be found on `http://javahowto.blogspot.com/2006/07/javaagent-option.html`

The InsectJ core contains an abstract structure as depicted in figure A.2. The `AbstractProbeInserter` class is to be extended by any class that implements a concrete type of instrumentation of a method. As shown in the figure, this class contains a method `instrumentMethod(mgen:MethodGen)` which is called for every method that has been signed up for instrumentation in the config-file. Note that `MethodGen` is the instance of a method provided by BCEL which allows manipulation of a method on the byte-code level.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<insectconfig>
<settings verbose="false" dump="false" />
<probe
    class="IntraMethodPathProfilingProbeInserter"
    monitor="TestCaseAnalyzer" arguments="" />
<instrument>
testsystem.car.Car
testsystem.car.Roof
testsystem.car.Door
testsystem.car.Window
</instrument>
</insectconfig>
```

Figure A.1: Example config-file for ExsectJ to observe and analyze paths in four classes

Every concrete implementation of the `AbstractProbeInserter` class is associated with a certain monitor-interface which contains the methods that are called by the instrumented methods to store or process the run-time information. For instance, the concrete class `BranchProbeInserter`, which instruments the code to observe which branches of the control-flow graph are executed inside a method, is associated with the `BranchMonitorInterface` (not shown), which contains the methods to store the covered branches during run-time. Therefore, any class that implements this interface is capable of processing this type of run-time information as desired.

It is important to mention here that exactly one monitor-instance is created automatically for each class that is instrumented. Also note that each monitor-class contains the `processData()`-method, inherited from `MonitorInterface`. Every monitor-instance is attached to a run-time handle which calls this method when the observed program has finished executing. This allows to postpone the processing of gathered data to minimize run-time overhead when the observed program is running.

A.2 The ExsectJ extension

We have implemented class `IntraMethodPathProfilingProbeInserter`[2] to extract run-time path-information. This sub-class of `AbstractProbeInserter` is a facade which

[2] We were always taught to use descriptive class-names

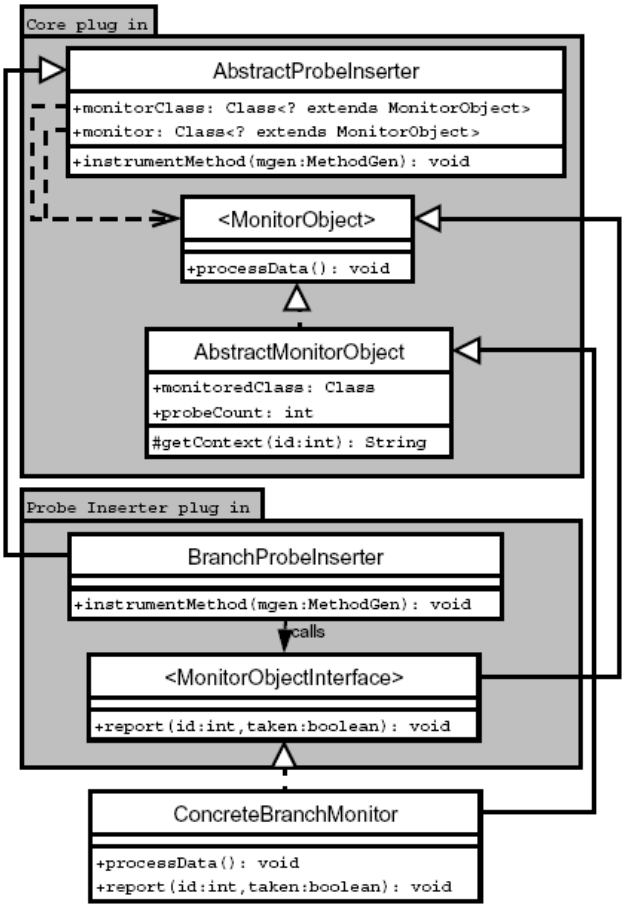


Figure A.2: Class diagram of core-classes and example extension (from [36, fig. 1, p. 3])

guides the whole process from analyzing the control-flow within the method to applying the calculated instrumentation. This process is visualized in figure A.3.

From top to bottom, we can see that this process is basically cut into six phases. Right after `instrumentMethod(mgen:MethodGen)` is called, the following steps are taken:

1. The control-flow graph (CFG) is created.
2. The directed acyclic graph (DAG) is created.
3. The DAG is used to assign different forms of instrumentation to some of the edges of this graph.
4. The calculated instrumentation is converted to instrumentation of the edges in the CFG.
5. The instrumentation is applied to the method-instance (`mgen`).

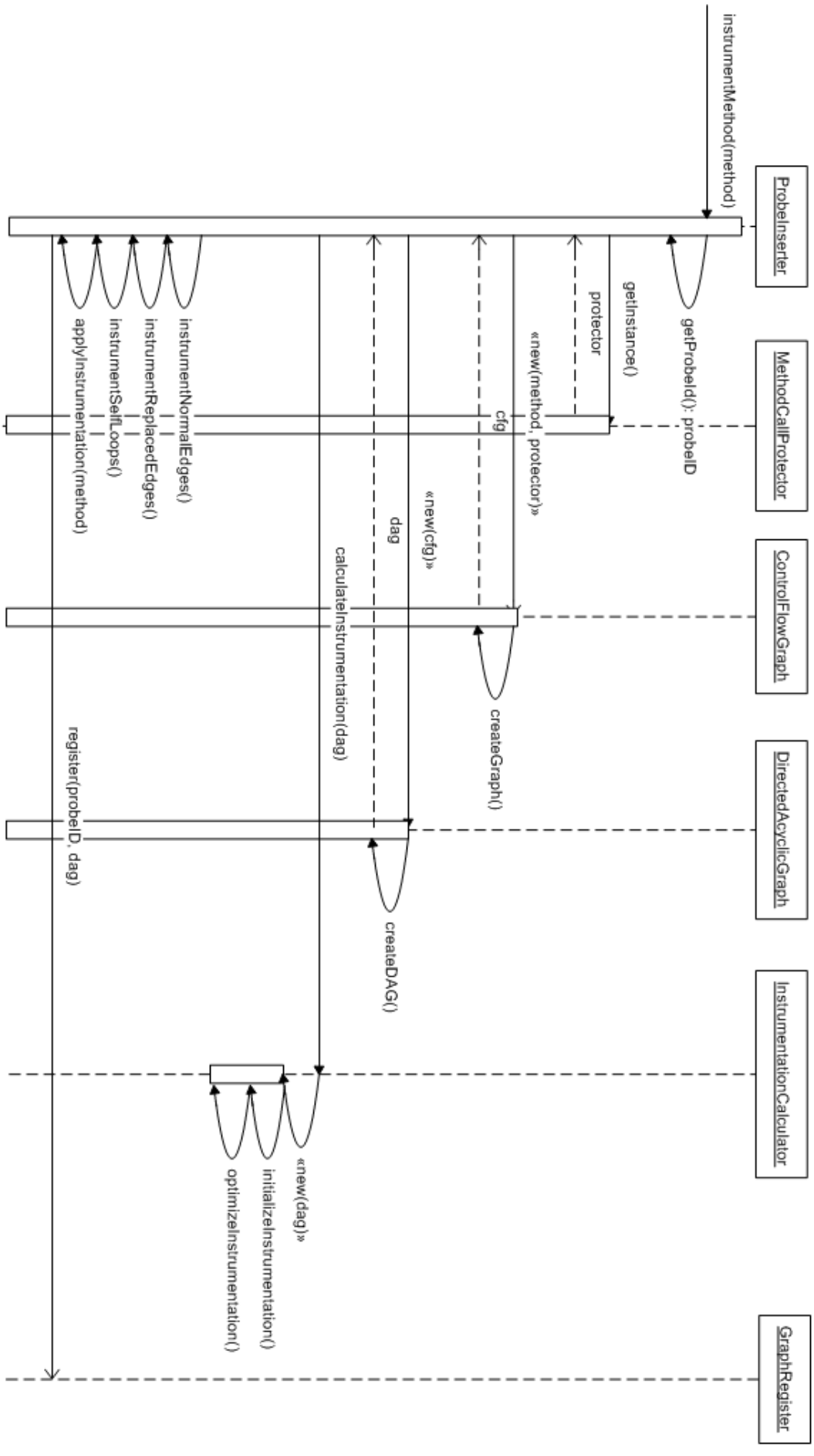


Figure A.3: Rough overview of the instrumentation process

6. The DAG is mapped to the `probeID` that is unique for this method and stored in the `GraphRegister` so that the monitor that regenerates the paths later can access the right DAG.

Note that the figure also shows the creation of a `MethodCallProtector`, which is a singleton-class. This protector is responsible for locating a specific set of instructions - in this case all invoke-instructions - for which new entries in the method's exception-table need to be created (see 3.3.6 for details about why this is desirable). The `ControlFlowGraph` uses the protector to correctly initialize the graph with the added catch-block. The use of different protectors is explained on page 88.

We mentioned earlier that each sub-class of the `AbstractProbeInserter` is typically associated with a special monitor-interface that contains the methods to store and process the gathered run-time data. In figure 3.11 we showed the path-information that is required to regenerate complete paths. Therefore, we defined a special `PathMonitorInterface` that contains three methods; two of which are used to store the path-information and one special method that is used to retrieve a unique `executionID`. Additionally, we have implemented an abstract monitor-class that implements all the basic functionality to record the path-information and regenerate these paths when the `processData()`-method is called. The complete structure is shown in figure A.4. Note that these classes are not all part of the same package.

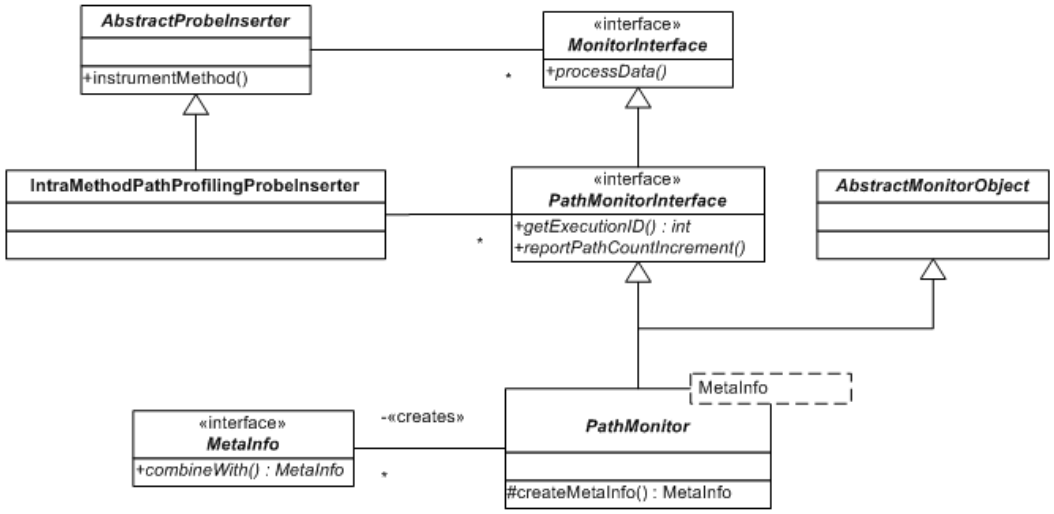


Figure A.4: Extension of InsectJ core classes to extract paths

A.2.1 Adding meta-data to acyclic paths

Recall from § 4.1 that we are able to associate an active stack of test cases to the paths that are being recorded at a certain point in time. We have implemented this functionality by adding a field of type `MetaInfo` to the `PathSumRecord`-class. `MetaInfo` is an interface that can be implemented by any class that contains some data that could be associated with

certain paths. Class `PathMonitor` contains an abstract method `createMetaInfo()` which is called automatically each time a new acyclic path is recorded. Therefore, this mechanism allows for any sort of meta-data to be stored along with certain path-data.

There is, however, one catch. While one `MetaInfo`-instance is associated with a single *acyclic* path, many acyclic paths are merged together into a single path in the regeneration-process. For this reason, the `MetaInfo`-interface contains one method, `combineWith(info:MetaInfo):MetaInfo`, that should implement the functionality to merge two different instances into one. It is important to note that this method should be implemented completely *symmetric*, which means that combining A with B is the same thing as combining B with A. This is because the order in which the different `MetaInfo`-instances are combined with each other is not defined.

A.2.2 Regenerating and processing of paths

Because we assume that many paths can be executed, stored, regenerated and processed, we made two design-decisions that help to minimize memory-use. First of all, all paths that are regenerated are immediately *encoded*. This encoding is mostly based on sharing data of sub-paths using a fly-weight pattern, since many paths may contain the same or structurally equivalent sub-paths. We use the `PathBlockGraph` discussed in chapter 4 to enable this encoding.

Secondly, we have implemented the regeneration of paths through a *pipeline*-structure. This means that each path is regenerated on request rather than regenerating all paths at once. This pipeline structure avoids the peak-memory use that would otherwise be needed to store all path-data.

Appendix B

Technical Documentation

Because so much work has been put into ExsectJ, we will provide a structural overview of its packages here. We will zoom in on some packages that we consider to be relevant.

B.1 General package overview

ExsectJ contains a total of 27 packages and 14344 lines of code.[1] Table B.1 shows the distribution of packages and lines of code added to the original InsectJ framework.

	InsectJ	ExsectJ (added)	Total
Packages	8	19	27
LOC	3901	10443	14344

Table B.1: Main statistics

Furthermore, figure B.1 provides an overview of the most important packages that were added and their dependencies. The BCEL-package represents the Byte-Code Engineering Library as discussed in section 3.2. The JDSL-package contains several data-structure classes that are primarily useful when building graphs and performing graph-operations.[2] We use it for several operations on the DAG (in package `epp.dag`).

On the right side of the figure, three packages are shown that implement the basic functionality to easily export any graph to a file that is formatted using the dot-language [18]: `util.dot.core`, `util.dot.creation` and `util.dot.export`. All graphs in this report (except for the UML-figures) were rendered using these packages through `epp.cfg.dot` and `epp.dag.dot`. Although this functionality was not strictly required for our thesis, we have implemented these packages for testing purposes (e.g. to see if graphs were created correctly) and, needless to say, they provided a comfortable way to visualize the whole project in this thesis-report.

Package `epp.cfg.algorithms` contains classes that enable different traversals of the CFG, whereas the classes in package `epp.cfg.instrumentation` are used to schedule

[1] As counted by the Metrics plug-in for Eclipse, also see `http://metrics.sourceforge.net/`

[2] Also see `http://www.cs.brown.edu/cgc/jdsl/doc/index.html`

and apply the instrumentation of a particular method. The remaining packages will be handled in more detail in the following sections. Each section contains a class-diagram with all relevant classes. However, because of limited space we had to omit the operation-details.

B.2 Packages `epp.cfg` and `epp.cfg.protection`

Figure B.2 contains the relevant classes of packages `epp.cfg` and `epp.cfg.protection`. An instance of class `ControlFlowGraph` can be created by providing a method and an instance of `ProtectionManager`. The first argument is used to build the basic control-flow inside the method through its instructions, which can be accessed using BCEL. Note that the graph contains three types of basic blocks. The main type is a `CodeBlock`, which represents a set of instructions, whereas the graph contains one `EmptyRootBlock` and one `EmptyExitBlock` to ensure a single entry- en exit-point of control-flow in a method.

The abstract class `ProtectionManager` is responsible for specifying which instructions in the method should be *protected*. A protected instruction is an instruction for which a handler exists that catches any `Throwable`. Hence, this class returns a set of instructions for which a handler must be created. If this set is not empty, then one `CatchBlock`-instance is created and added to the graph, while one new entry for each range of instructions is created in the exception-table, all pointing to the first instruction of this `CatchBlock`.

By default, three concrete `ProtectionManager`-classes are implemented: one that requires every instruction to be protected (the safest option), one that requires only invoke-instructions to be protected (the default option for ExsectJ), and one that requires no instructions to be protected, thereby effectively preventing a `CatchBlock` to be created.

Furthermore, since the `CatchBlock` actually resembles a custom handler, it can be filled with custom code to execute when an exception was thrown in a method that was not caught there by default. Since we did not use this functionality, the `CatchBlock` currently acts as an empty `finally`-clause by only inserting a `throw`-instruction.

B.3 Package `epp.dag`

Figure B.3 shows an overview of package `epp.dag`. The main class here is, of course, class `DirectedAcyclicGraph`. An instance of this class is created as a derivative of a `ControlFlowGraph`-instance and always keeps a reference to it. As explained in chapter 3, the DAG replaces every back-edge in the CFG with two other edges. Every such replacement is embodied by an instance of the `BackEdgeReplacement` class. The only exception are selfloop back-edges, which are removed from the CFG but not replaced by other edges.

Class `InstrumentationCalculator` analyzes the DAG and assigns the four types of instrumentation to the appropriate edges (see § 3.4.1) in the form of different subclasses of type `EdgeInstrumentation`. Later, these types are used to assign the appropriate instruction-blocks to the edges of the CFG through a visitor-pattern.

Internally, the DAG is created through JDSL, which is a library with specialized functionality for creating graphs and performing graph-operations. Using JDSL it was quite

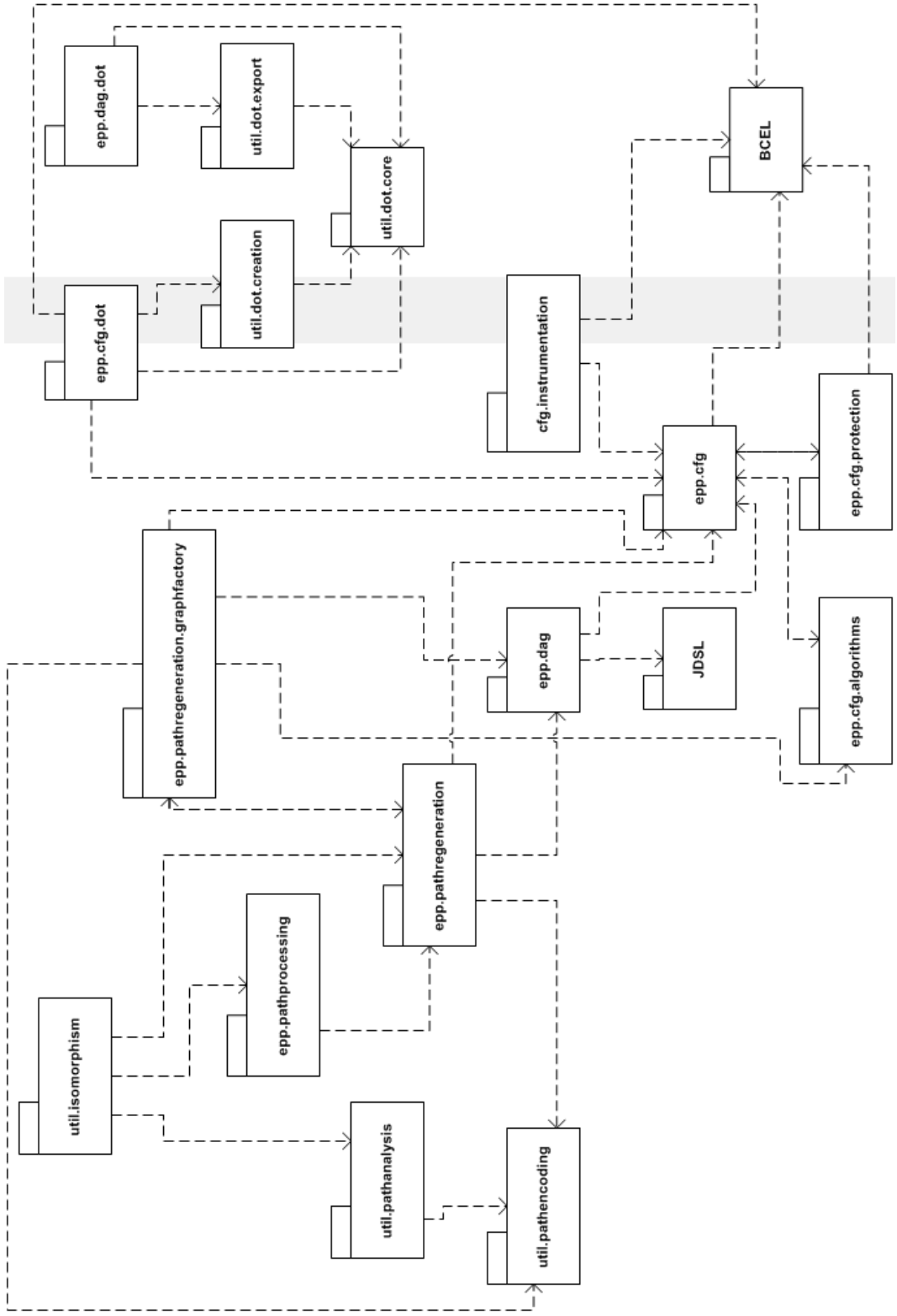


Figure B.1: Overview of the relevant packages of ExsectJ

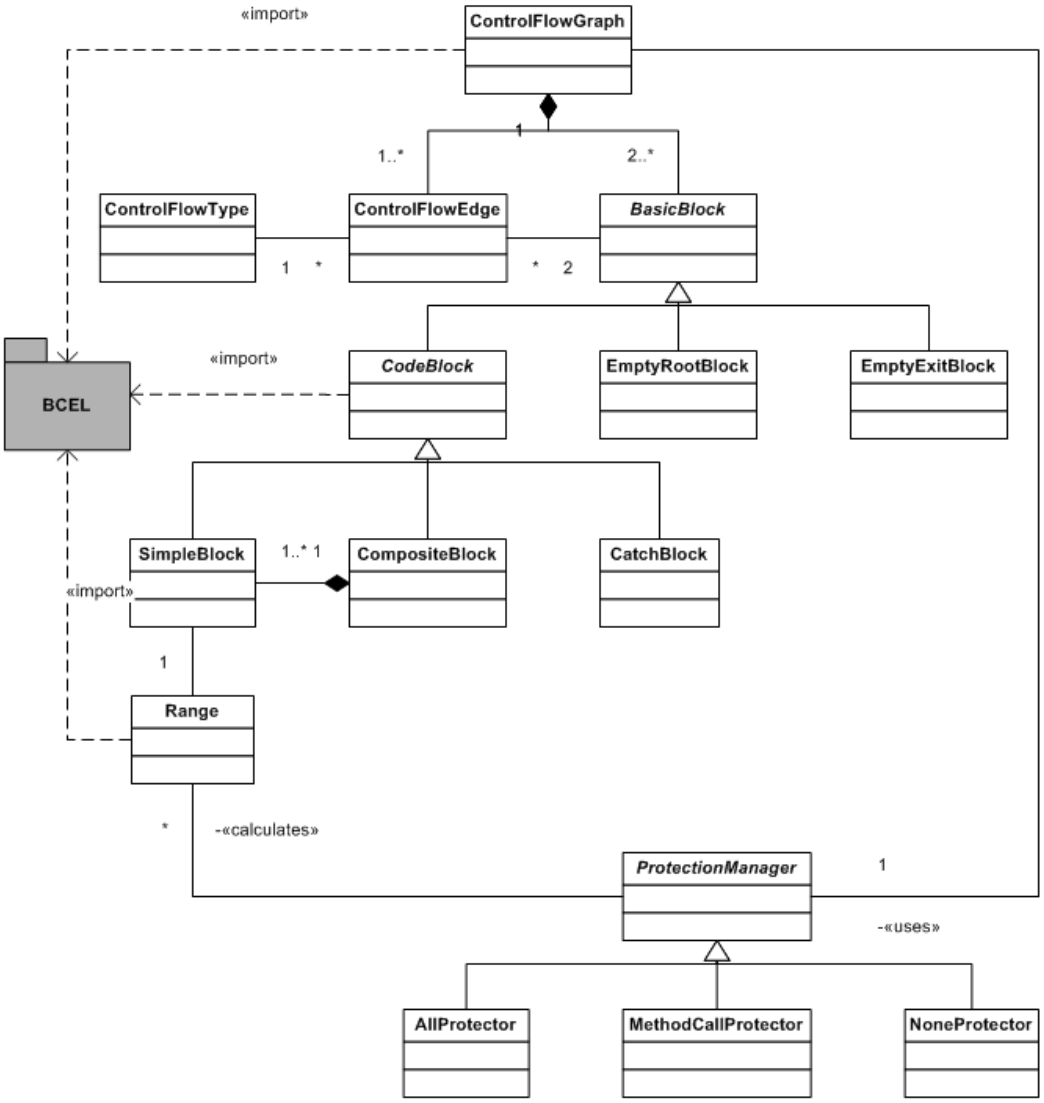


Figure B.2: Relevant classes of packages `epp.cfg` and `epp.cfg.protection`

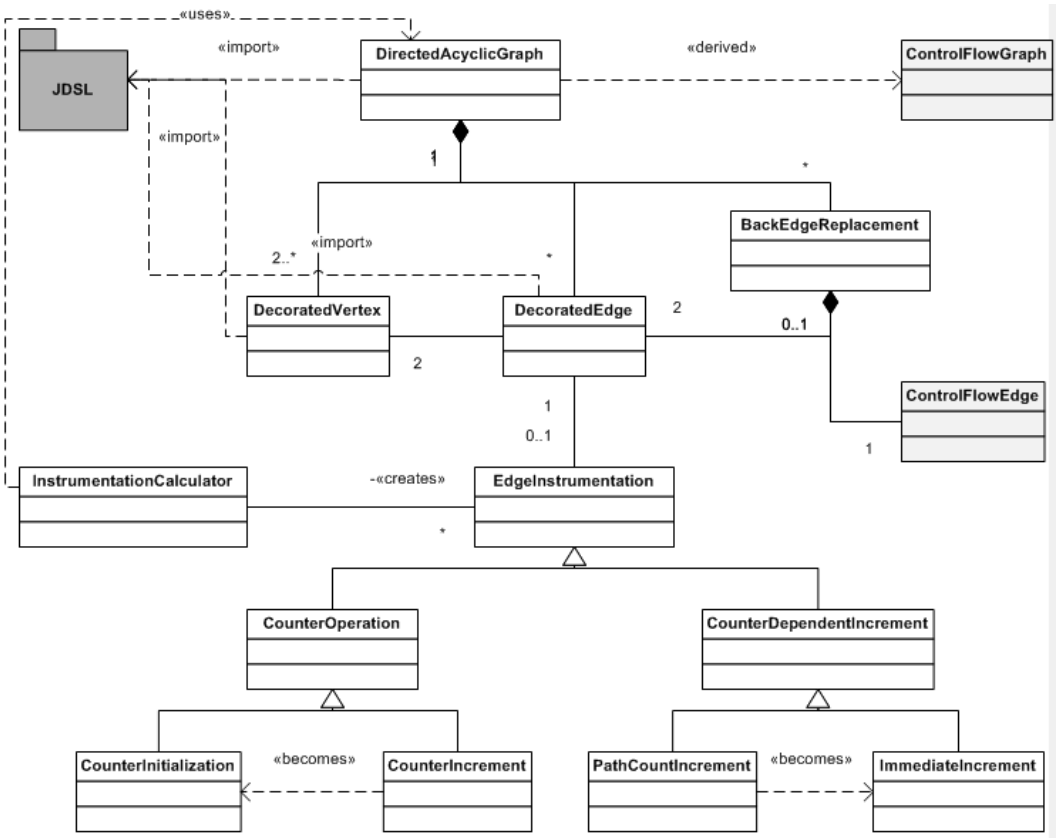


Figure B.3: Relevant classes of package `epp.dag`

easy to locate any back-edges or to 'decorate' nodes and edges in the DAG with arbitrary objects, such as edge-values and the instrumentation-classes.

B.4 Package `epp.pathregeneration`

Figure B.4 shows the relevant classes of package `epp.pathregeneration`. As the name suggests, these classes are responsible for regenerating the paths that were recorded as a set of path-sum records. This process is explained at the end of chapter 3.

In short, every instance of `PathSumRecord` is converted into an instance of `AcyclicPath` by traversing the associated `DirectedAcyclicGraph`-instance. An `AcyclicPath` can be either a `SelfLoopPath` or a `RegularAcyclicPath`. During the regeneration-process, all instances of `SelfLoopPath` are inserted into the appropriate `RegularAcyclicPaths`. Hereafter, these instances can be combined to become instances of type `CompletePath`.

However, since paths can become very large, we decided to encode these paths into instances of `CompressedPath` while they are being regenerated. The encoding itself is delegated to package `util.pathencoding`, which will be described in the next section. Fi-

nally, all paths of the same method are 'packed' into an instance of `CompressedPathPack`, which contains all paths from a single method, including a reference to this method.

In addition, we have implemented the whole regeneration process through a pipeline structure, which is visualized in the figure by all the `Enumeration`-classes. The advantage of implementing a pipeline structure is that paths can be processed one at a time, while effectively preventing all paths to be stored into internal memory at once.

B.5 Package `util.pathencoding`

Figure B.5 shows the classes of package `util.pathencoding`. This is a generic package that can be used to encode any paths of which the shapes can be modeled as a graph with a single entry- and exit-point. It is, however, required that every block in the graph is identified by a unique, positive integer.

The main class in this package is class `PathBlockGraph`, which is used to guide the encoding process. The primary goal of encoding paths is to limit memory-consumption by storing equal parts of a set of encoded lists using the flyweight-pattern. In this case, all equivalent instances of `EncodedSubPath` are stored only once. Note that the data structure that stores the flyweights uses weak references to these instances. This way, the garbage collector can remove any instances that are not used by other objects any longer.

To make use of this package, one must create their own class that represents an encoded path by subclassing the abstract class `EncodedPath`. Class `EncodedPath` contains all the functionality to encode itself; all it requires is a instance of `PathBlockGraph` and one instance of an existing path. `EncodedPath` allows the path to be traversed by exposing an `Enumeration` instance. This `Enumeration` can be used to implement the abstract `decode()`-method, which is to return the decoded version of that path (presumably by just wrapping a list of items of some type).

B.6 Package `util.pathanalysis`

Figure B.6 shows the classes of package `util.pathanalysis`. These classes implement the core functionality to analyze and compare paths on different sub-path levels. As explained in chapter 4, every (nested) loop in the control-flow is captured as a single level, containing a limited amount of sub-paths and edges.

In fact, every `SubPathLevel`-instance corresponds with exactly one `LoopBlock` from figure B.5. Furthermore, every level is associated with exactly one instance of `SubPathTable` (see figure 4.3). Class `TopLevel` is a special type of `SubPathLevel`, which represents the outer-most or main level of the graph. When this class is instantiated, it takes a `PathBlockGraph`, to instantiate the correct nested levels, and a list of encoded paths that are associated with this graph. During the initialization process, every path is traversed using a `TableFiller`-instance, which inserts the path-data for every path in the correct `SubPathTable`.

Two other important classes to mention here are `EdgeSet` and `SubPathSet`, which are both subclasses of type `FiniteSet`, because each instance is always associated with a

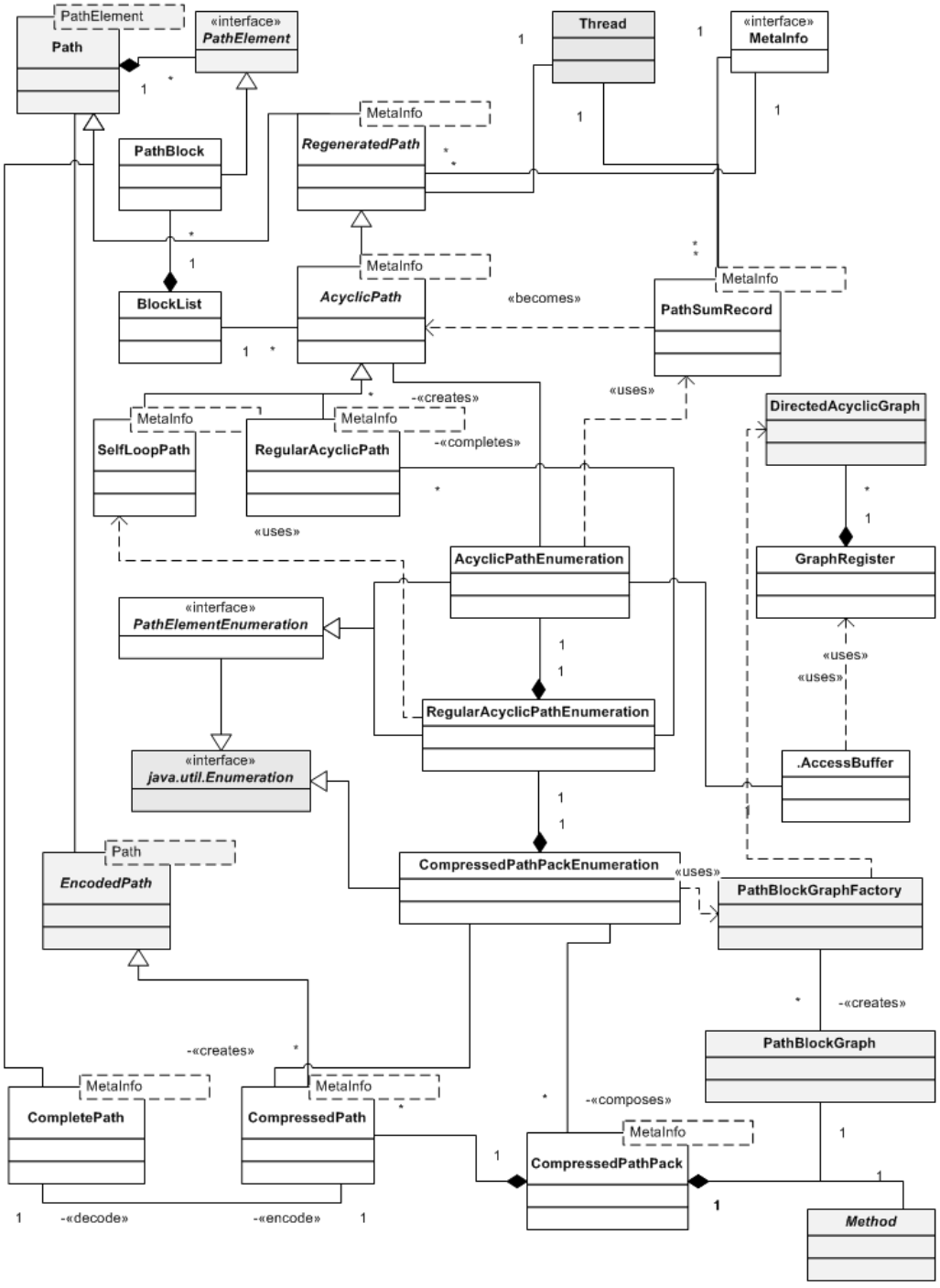


Figure B.4: Relevant classes of package `epp.pathregeneration`

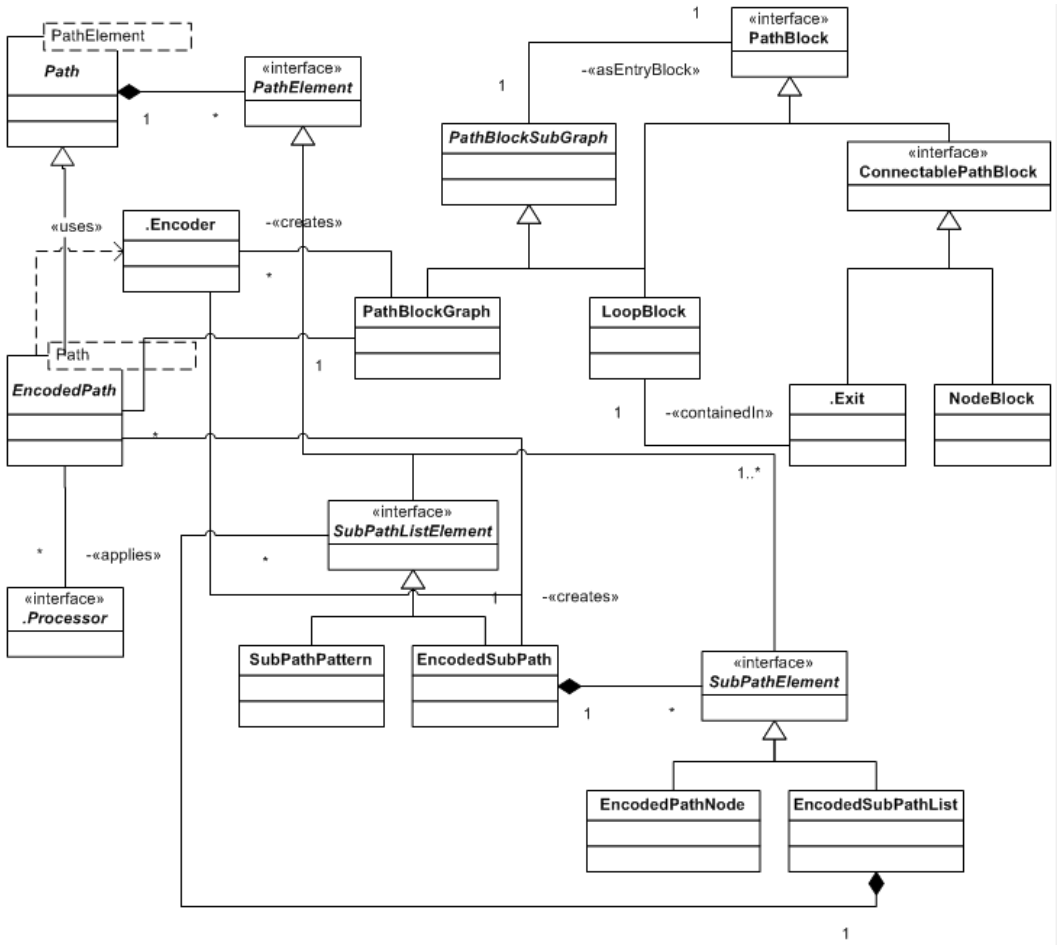


Figure B.5: Relevant classes of package `util.pathencoding`

certain level, and, as mentioned earlier, each `SubPathLevel` only contains a limited or finite set of sub-paths and edges. These two classes are actually wrapping regular `BitSet`-instances, where each bit corresponds to a certain edge or sub-path. A zero in the bit-set means that this item is not present in the set, whereas a one indicates that it is. By using bit-sets, these sets can not only be stored efficiently, but also efficiently used in regular set-operations (or-ing, and-ing and subtracting), which is convenient when implementing a path-comparison formula.

B.7 Package `epp.pathprocessing`

Package `epp.pathprocessing` is depicted in figure B.7. The central class in this package is class `PathMonitor`. Class `PathMonitor` is an abstract class that forms the base class for any concrete monitor class that implements the after-processing of paths that were executed during run-time of the observed class or application (also see section A.2). It im-

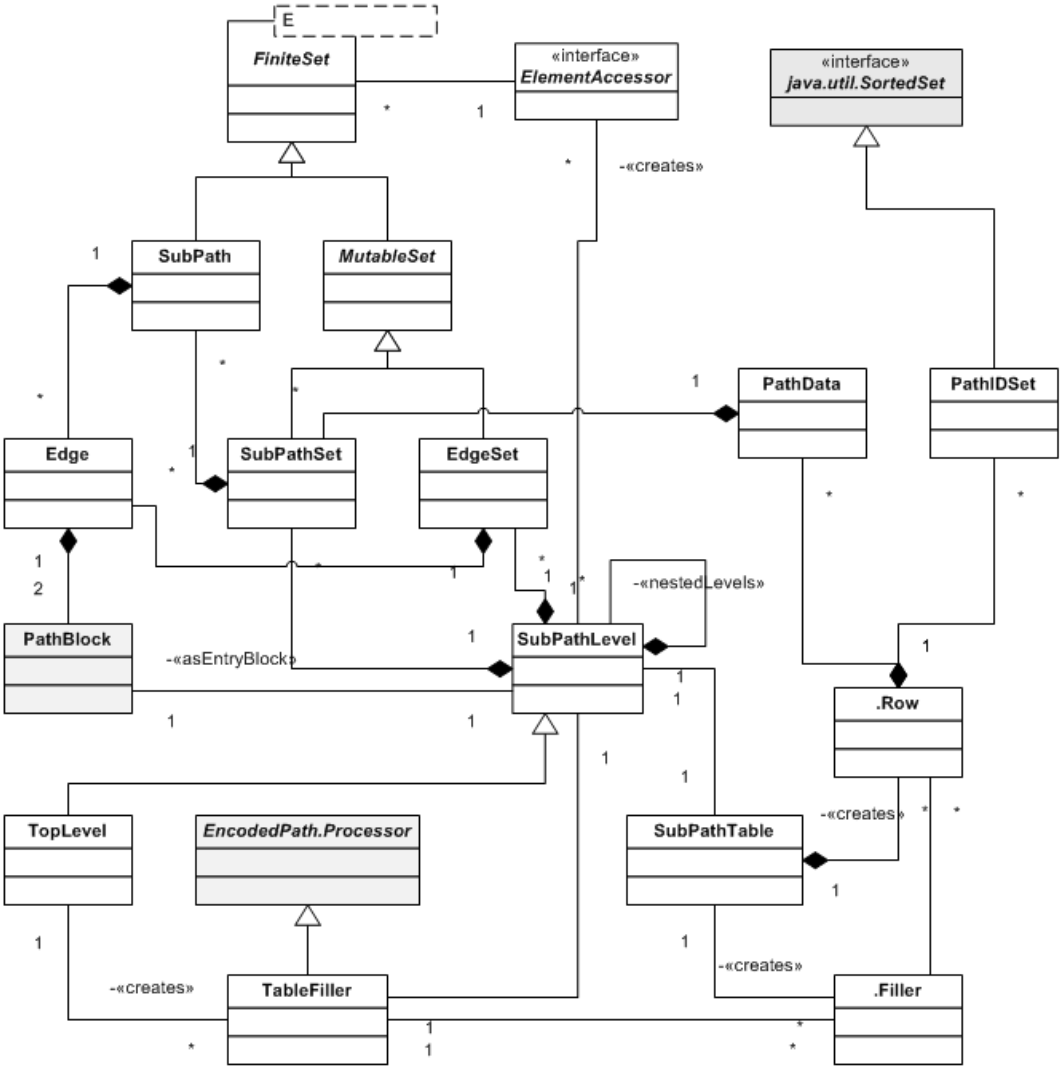


Figure B.6: Relevant classes of package `util.pathanalysis`

plements all the functionality required to store the path-data as `PathSumRecord`-instances and automatically calls the (abstract) method `processPathData(...)` while providing an `Enumeration` that returns instances of type `CompressedPathPack`, which was discussed earlier. Hence, any concrete sub-class can use this `Enumeration` to process all regenerated paths as desired.

B.8 Package `util.isomorphism`

Package `util.isomorphism` contains the actual after-processing of all path-data in ExsectJ. We have implemented class `TestCaseAnalyzer`, which is a indirect subclass of class `PathMonitor`. Each instance of class `TestCaseAnalyzer` creates one table of isomorphism-scores for each method of the class it has been observing and stores them in a static list. These score-tables are represented by instances of class `IsoScoreTable`. When all instances have added their score-tables all scores are used to create one single table of average score. Hereafter, all tables are exported to a series on html-pages.

Furthermore, each table requires an instance of type `FormulaFactory` which creates a new instance of type `Formula`. In turn, class `IsoScoreTable` calls method `calculateIsomorphism(...)` for each pair of test cases. Hence, it is quite straightforward to implement a custom formula.

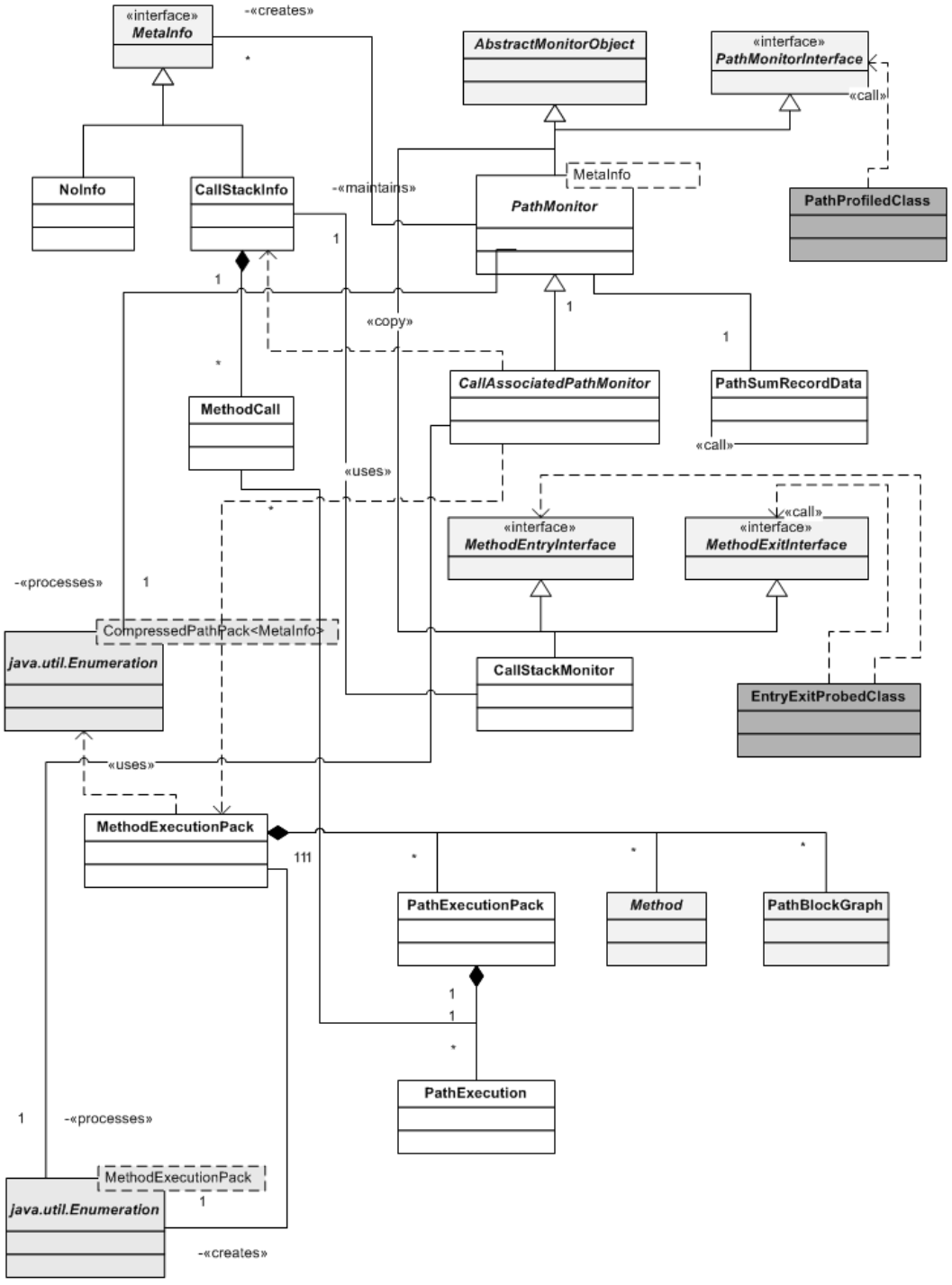


Figure B.7: Relevant classes of package `epp.pathprocessing`

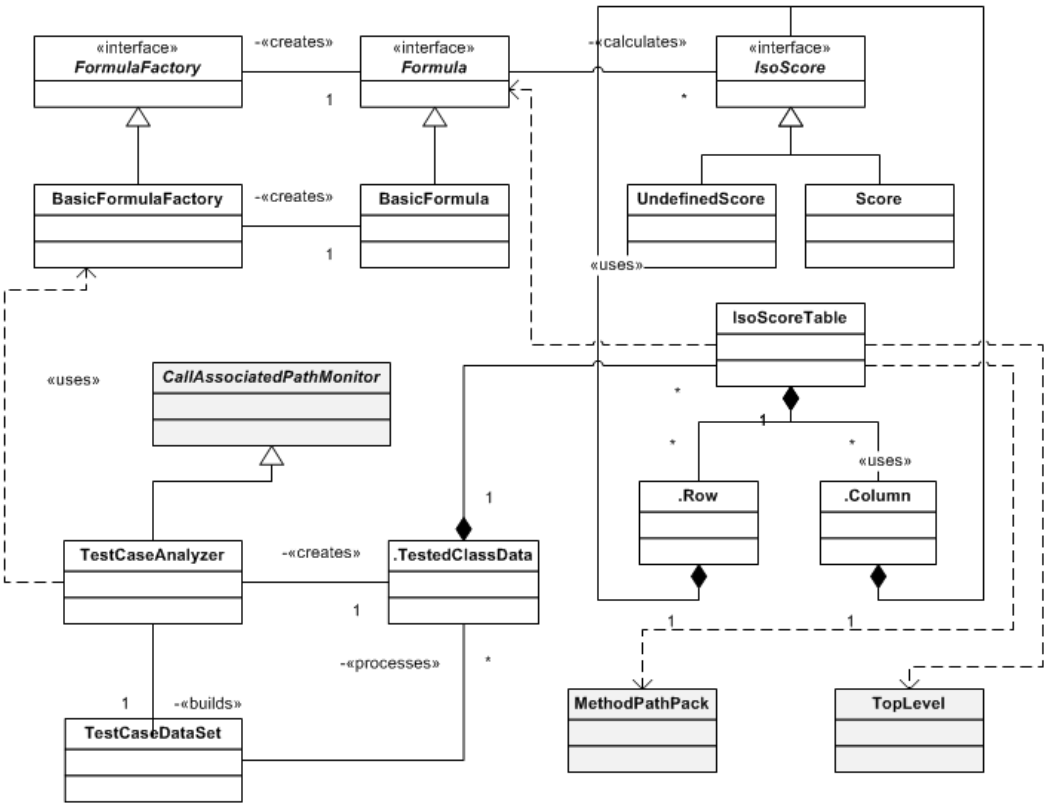


Figure B.8: Relevant classes of package `util.isomorphism`