Thesis for the degree of Master of Science

Extending T2 with Prime Path Coverage Exploration

Maaike Gerritsen

October, 2008

INF/SCR-08-14

Universiteit Utrecht

Supervisor: I.S.W.B. Prasetya

Center for Software Technology
Dept. of Information and Computing Sciences
Universiteit Utrecht
Utrecht, The Netherlands

Abstract

There are very few testing tools on the market today that combine testing and coverage. The tools that do combine these two element use a very simple coverage metric. This thesis contributes an extension to an automated testing tool called T2 that enhances it with a more sophisticated coverage measurement based on prime paths. Our solution crucially relies on byte code instrumentation. However, a more elaborate instrumentation scheme is now needed. We solve this by combining it with a programming paradigm called Aspect Oriented Programming. Although a widely accepted paradigm, Aspect Oriented Programming has not been used eminently in the testing field.

T2 generates sequences of tests and randomly creates values as input to these tests. Due to T2's random properties, it cannot guarantee a high code coverage. Another important contribution of our extension is the addition of active strategies to improve T2's coverage. The results so far are quite positive.

Contents

Chapter 1

Introduction

As software programs become larger and larger, it becomes very important to thoroughly test these programs. Since it is not possible to test every situation by hand, many automated testing tools exist. It is important for testing tools to have high code coverage so that thorough tests can be guaranteed. Here, the possibilities of improving the coverage of an automated testing tool called T2 [1] will be explored.

The remainder of this chapter will provide a motivation for this thesis, describe the problems at hand and provide the main contribution.

1.1 Motivation

When a testing tool runs a certain program and finds no errors, does this mean that the program does not contain any faults at all? Probably not. To be able to ensure that the code contains no errors, every part of the program's source code should have been executed with all possible values for its input. Even an automated testing tool can not cover the complete input space and might not be able to cover every existing path in the source code. However, as a user it is important to know how much code has been tested.

This is why the term code coverage is used. Code coverage describes the degree to which source code has been tested, usually expressed in percentages. So now, when a program is tested and no errors show up, we can reason about the chance that undiscovered errors exist. A logical deduction is that one would like to see that code coverage is as high as possible.

There are several methods of describing the coverage of a tool. Two practical ways are statement coverage and branch coverage. Statement coverage is a metric expressing how much, usually expressed as a percentage, of the set of the statements in a piece of code are actually executed during a test. Branch coverage checks what percentage of all possible branches have been tested. These two measurements of coverage might seem similar but are not the same. In the example in Fig. 1.1 a piece of code is shown. The programmer wants to print the result of a simple calculation. If this program is tested once with the value 0, then statement coverage for the method `calc` is 100% because every statement has been explored. However, branch coverage is just 50% because the possibility of skipping the `if` statement has not been tested. Here the danger can also be shown. Maybe the programmer has not taken into account that the value of `i` could be negative. Then `Math.sqrt` will return `NaN` and the addition of `j` and 2 will fail. This is why branch coverage is important; if not all branches are explored then there is the possibility of undetected errors even though statement coverage is complete.

However, if the method is tested with 0 and a positive value for `i`, then branch coverage will be 100% but the error will still not be found. This shows that 100% branch coverage also does not guarantee that all errors are found, but it can improve error detection.

It is critical for an automated testing tool to achieve high coverage. High coverage leads to greater confidence being placed in the testing tool's findings. If the coverage a testing tool

```
1  public void calc(int i){
     if(i==0)
3      i = 1;
     int j = Math.sqrt(i);
5    int k = j+2;
     System.out.println("result: "+k);
7  }
```

Figure 1.1: Example statement and branch coverage

achieves is not satisfactory, it is important to improve the coverage in order to uncover hidden errors or to be able to trust the results given by the testing tool.

Also, a good coverage metric should be used so that it is not possible to place false confidence in a tool's findings. As shown above, 100% statement coverage does not always mean that a method is thoroughly tested. Nevertheless, 100% branch coverage can also not guarantee that the tested method is error free. It could be possible that a certain value is changed in one branch of a method and this change leads to an exception in another branch. This error might not be found if both branches are tested separately.

A stronger coverage metric than branch coverage is path coverage. Path coverage measures if every possible route in a certain piece of code has been executed. This means that it will look at a piece of code as a combination of its branches. 100% path coverage indicates that all possible branch combinations have been tested.

Path coverage will uncover more errors than branch coverage, therefore it should be the coverage metric of choice. However, path coverage is unfeasible when cycles are involved. A path through a graph that contains cycles is possibly an infinite path which means that this path can not be tested.

Intuitively, a route through a certain piece of code is seen as a path taken from the beginning of this code to an endpoint. A route could also be defined as a path that will never come across the same code twice. This kind of route is called a prime path. Another property of a prime path is that it is never a subpath of any other path. This is especially interesting when code contains cycles. Prime paths make it impossible to define infinite paths, but they also help reasoning about paths containing cycles.

In this thesis coverage is defined as prime path coverage. Most coverage analysis tools prefer branch coverage over path coverage argumenting that path coverage has much overhead. This might be a correct statement, but path coverage is so much more powerful than branch coverage. The overhead is a small price to pay for more confidence. Prime path coverage offers a practical coverage metric that will let us handle cycles more appropriately. Therefore it is the coverage metric of choice for this thesis. Several optimisations will be implemented to keep overhead as low as possible.

From this motivation a research question can be formulated that this thesis will answer:

Is it possible to improve prime path coverage of an automated testing tool that uses a random test generation?

1.2 Contribution

This thesis contributes an active prime-path-based coverage tool called T2Extension. The tool records each path in the target program by modelling them in a graph. Using this model, T2Extension keeps track of which paths have been tested and which have not. It uses the information to calculate the prime path coverage of the test. Subsequently, the tool will use this

information to construct new tests that will cover the paths that have not been tested. The tool will be coupled as an extension to an automatic testing tool called T2 [1].

There are certain restrictions that the extension to T2 will have to consider when investigating coverage. T2 receives class files as input, so the extension must also use bytecode instrumentation because it can not be assumed that the source code is also present. This generates some difficulty when trying to identify branches. In Java source code it would be possible to add certain checking code by looking at patterns. A parser could easily be written for this. In this case, an investigation into a manner of reading bytecode and interpreting and modifying this without destroying code written by the user is needed.

Another difficulty that presents itself is the fact that in compiled source code, all nested loops and branched are flattened. This makes the identification of different paths complex. The identification of nested code is important because it could hide faults or it could cause faults later on.

Taking these restrictions into account, the extension to T2 will need a smart solution of modelling all branches and it should be able to update its model whenever a branch has been tested.

T2Extension is the first testing tool that combines testing with a sophisticated coverage metric. There are very few tools in existence that combine testing and coverage. However the coverage metric that these tools use are very simple. Not one testing tool exists that implements prime path coverage.

T2Extension is also the only tool that uses aspect oriented programming for the administration and registration of paths. All existing coverage measurement tools instrument the source code or bytecode directly. This thesis will show the strengths that lie in using aspect oriented programming to this end.

Another innovative property of T2Extension is that it actively tries to improve a testing tool's coverage. Existing testing tools will at best report that their coverage is below an acceptable level.

1.3 Approach

In short, the approach to modelling all paths will be to add code to the class under test (CUT) and let T2 test the new class file. When T2 executes the code, the extension to T2 will register for each method which path is taken. The path that have not been executed yet will be examined after T2 has terminated and, using T2, branches are manipulated in paths that have been tested and execute them again with the goal of also testing paths not tested so far.

A class file consists of byte code. This is not readable or understandable. To be able to add code to a class file, a tool is needed that makes it possible to manipulate byte code. The tool that will make this possible is BCEL [12]. BCEL parses the class file into an object containing methods and their instructions. These instructions are contained in an instruction list for each method. This instruction list can be manipulated, instructions will be added to this list whenever a decision point is found so that each branch can be monitored.

BCEL is a simple but powerful tool that makes it possible to insert almost anything into the instruction list. One could build their own class completely, just using BCEL. However, working with instructions is a very low level of working. Low level instructions are difficult to read and understand, and when manipulating the instruction list, it is not always possible to oversee the consequences. The objective is to manipulate as little as possible with BCEL and choose a high level solution for the monitoring of found branches in a method.

The high level solution is to work with Aspect Oriented Programming (AOP) [7]. How it works and what can be done with AOP is explained in chapter 5. This manner of programming is used to monitor the code inserted into the class file with BCEL and to register which paths T2 has tested so far.

The advantage of using AOP over BCEL for monitoring tested branches is that AOP makes it possible to act upon certain pieces of executed code. For example, it is possible to execute a certain piece of code whenever a method in a certain class is executed. Or, after a method is executed, register the path it followed.

Working with a high level solution has the advantage that a good understanding is easily obtained of the modelling of the branches. However, this type of modelling will also require many heavy and expensive algorithms. These will be discussed later on in this thesis.

1.4 Structure of the Thesis

This thesis starts by describing the architecture of T2Extension in chapter 2. This chapter is meant to give the reader an overview of the tool itself. Chapter 3 provides the theory about code coverage and graphs that is necessary for the explanation of the implementation of T2Extension. The thesis will then continue with a short introduction to T2 in chapter 4. Chapter 5 elaborates on Aspect Oriented Programming and its uses so far in the testing field.

With the theory accounted for and the context set, chapter 6 provides four algorithms crucial for T2Extension. Chapter 7 describes the actual implementation of T2Extension while chapter 8 reveals the test results.

Chapter 9 compares T2Extension to existing testing and coverage tools. Finally, this thesis is concluded in chapter 10 with a reflection on T2Extension's goals. Future work is also discussed in this chapter.

Chapter 2

The T2Extension Architecture

This chapter describes the architecture of the extension to T2, which will be named *T2Extension* from now on. The architecture will provide the reader with a mental picture of T2Extension while delving into some theory and background needed for the discussion of the tool. Chapter 7 explains about the actual tool, its workings and the decisions that were made.

The explanation of the architecture will first define the purpose of T2Extension. It will then go on to the requirements and constraints and discuss external tools that were used. Afterwards, the process T2Extension follows is explained. The last section diverges into the T2Extension package structure.

2.1 Purpose

T2 generates sequences of tests. This causes methods to test each other which might in turn detect more errors than a normal unit testing tool that uses methods as its unit instead of classes. T2 generates the sequences in a random fashion. That is to say, at every step along a test sequence, it will randomly choose a method to test and randomly create arguments for it if necessary.

Although T2 will check every accessible method of the target class (CUT), it might not explore every execution path inside a method because of its random fashion. T2 has no way of knowing if it needs to construct other arguments so that a method is thoroughly tested. This causes its overall path coverage to be relatively low. In section 1.1 the importance of achieving a high path coverage has been discussed.

The T2Extension tool must try to achieve a higher coverage. It must be able to steer traces in the direction of undiscovered paths and make sure that as many paths as possible are executed.

It is assumed that the user of T2Extension has already tested their target class with T2 and solved any errors found by T2, and now wants to search more paths in the class to ensure maximum test coverage. This would increase their confidence as well as potentially uncover hidden errors.

2.2 Requirements and Constraints

There are quite a few important matters to consider when designing the architecture for T2Extension. A noteworthy constraint is the fact that the extension tool must be built on top of T2. The T2 software should not be altered in any way unless it is absolutely necessary. T2 uses Java class files as input with no guarantee of source code being present. A user who knows T2 and is familiar with T2, should not need to learn much for the use of T2Extension. The ramification of these constraints is that the extension tool must also work with class files and keep the architecture of T2 in mind at all times.

T2 has a solid mechanism for testing in the form of traces, or test sequences. It also holds much information about the traces. T2Extension must use these traces and their information when trying to improve path coverage. This will ensure that T2Extension will not alter the behaviour of T2 in the sense that T2Extension can offer the same guarantee of correctness that T2 does.

T2Extension must be a reliable tool. Therefore, a critical requirement is that the path coverage metric must be proven and infallible. When the extension tool states that it has improved coverage, the user must be able to trust this statement. A component to the requirement of a reliable coverage metric, is the parsing of the class to be tested. When methods inside a class are explored, a foolproof way of searching for and registering different paths should exist.

Improving coverage leads to another requirement, namely the requirement that T2Extension must search through as many paths as viable. The search for unexplored paths should be (almost) complete ensuring that as many paths as possible will be found.

Another requirement is that T2Extension should not destroy the class to be tested. The tool will alter the class, but it should return the class to the user in the original state.

A last requirement should be to make T2Extension as fast as possible. It will definitely be a slower tool than T2 by itself because it needs to administrate and search for branches, but T2 is a fast tool and it would be a pity if the extension would slow it down considerably.

Chapter 10 will discuss if all of these requirements are met. Below is a short list that summarises the requirements and constraints.

- Build T2Extension on top of T2
- Java class file as input
- Use existing T2 tools
- Reliable path coverage metric
- Reliable administration of paths
- Search for paths should be complete
- Return original class
- T2Extension should be as fast as possible

2.3 Used Tools

T2 is built in Java and tests Java classes. T2Extension will also be built completely in Java but several tools were used to facilitate the implementation of T2Extension. This is a list of tools used and a short description.

BCEL BCEL stands for Bytecode Engineering Library. It is intended to give users a convenient possibility to analyze, create, and manipulate (binary) Java class files. For more information see [12].

AspectJ As said before, I will use AOP to register branches encountered when a method is executed. AspectJ is an aspect-oriented extension to Java which makes it possible to compile and use aspects. For more information see [8].

XML XML is used to store information about the tested objects. It is easy to read, which facilitates keeping an overview.

Apache Ant Apache Ant is a Java-based build tool. It makes it possible to use ant tasks which facilitates the use of several tools. More information on Ant can be found at [15].

2.4 Process

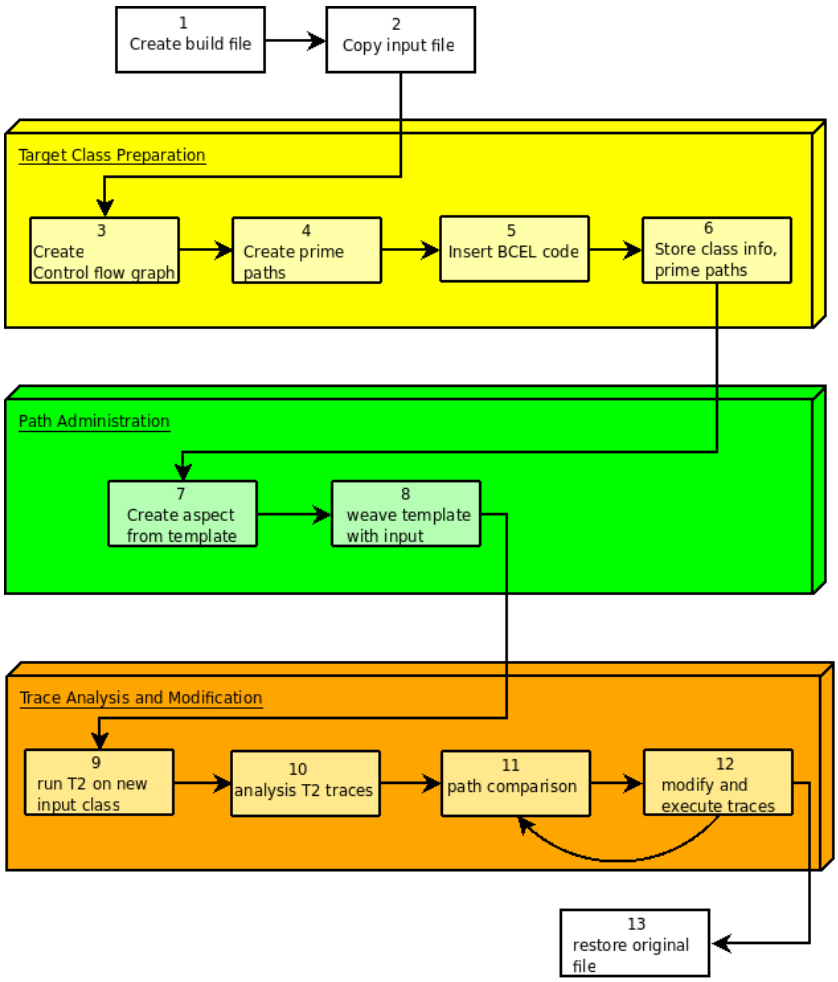


Figure 2.1: Program flow

In this section the flow of T2Extension is described and the data flow of T2Extension is shown. Before the program flow of T2Extension can be explained, the use of Ant must be clarified. Ant is a build tool which makes it possible to compile classes and call programs as one would do when using the command-line. However, instead of using a command-line instruction, Ant uses a XML file, called a build file, that stores all the necessary information.

Ant is mainly used to compile the template aspect and weave it with the input class file. When weaving binary class files, AspectJ expects them to be in a jar file. So Ant will also add files to jars whenever necessary. T2 and its traces can be called from within another program by just calling the main function of T2's main class, but since the class to be tested is now in a jar and needs many auxiliary classes; the class that handles T2 and its traces is executed with Ant. This ensures that T2 knows that the target class is present in a certain jar file.

A general picture of the program flow is given in figure 2.1. The process T2Extension follows can be categorised in three different stages. In figure 2.1, these stages each have their own colour and are boxed in. T2Extension starts with the creation of a build file for Ant. This build file has to be created dynamically because it needs input from the user like the name and location of the class that will be tested. The next step is to copy the original input file so that it is safe and it can be restored later. Since there is a copy of the input, T2Extension can freely alter the class file.

Step 3 creates a control flow graph for each method. When this graph is created, its prime paths (see definition 3.4.3) are calculated. Step 5 entails the insertion of BCEL code. Code is inserted at every decision point in a method. Then some information about the target class and the prime paths of its methods are stored in an XML file.

The reason for storing the prime paths in an XML file has to do with this thesis research. The XML file facilitates the analysis of which paths have been found so far and analysis of the behaviour of T2Extension. Also, in the future, this XML file could be used for easy and rich reporting.

The stored class information is used to build an aspect. This aspect registers each path that has been encountered. When the aspect is created, it can be woven with the modified target class.

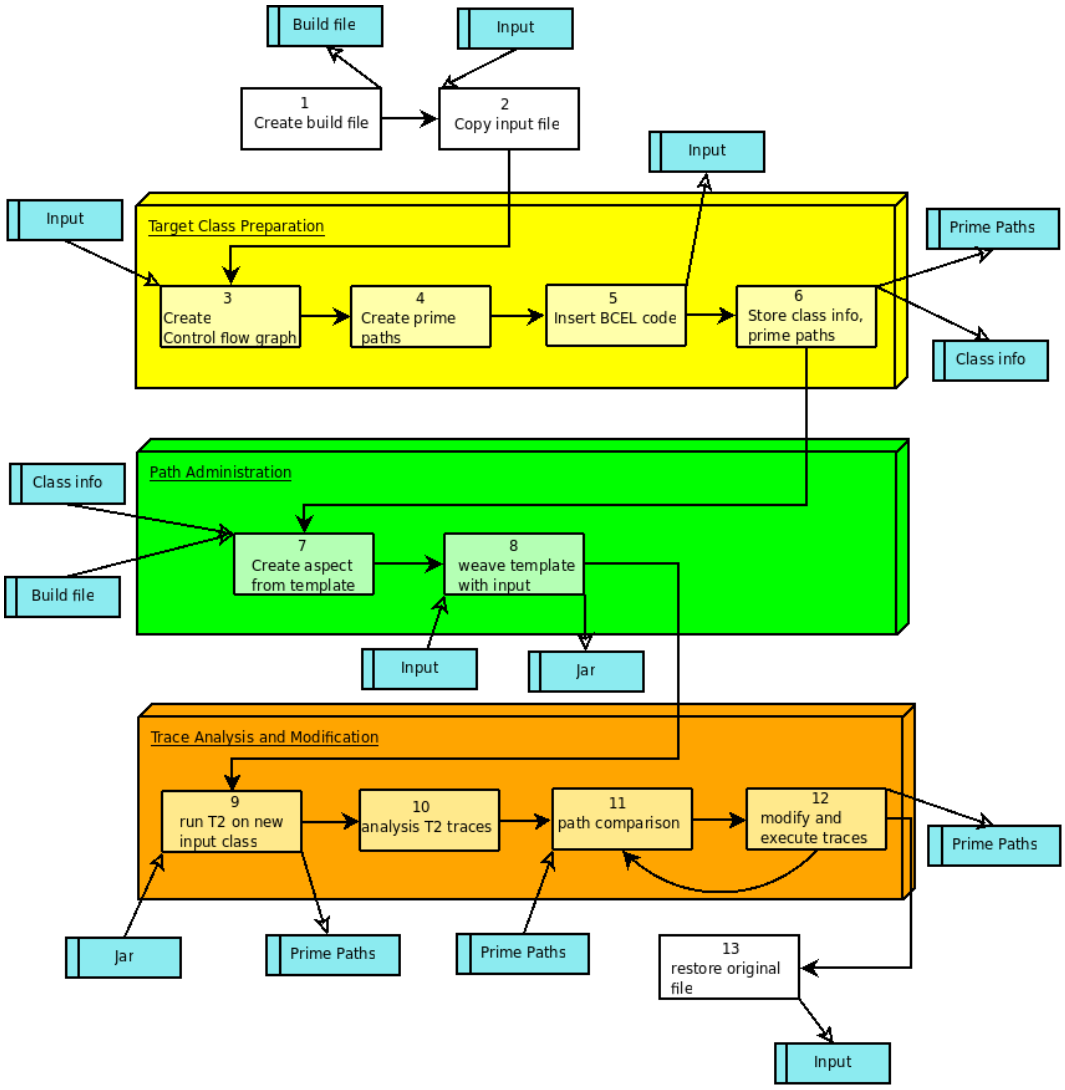


Figure 2.2: Data flow

The target class is now ready to be tested. First T2 runs it and stores all traces that it generates. These traces are then analysed. From this analysis, paths that have not been encountered yet can be compared to found paths. For each path that has not been found yet, traces that contain paths much alike the unfound paths, are modified and executed in the hope to find the path searched for.

Steps 11 and 12 are done several times with different domains for constant values in T2.

This will be explained in chapter 7. When the repetition of step 11 and 12 stops, the program has essentially finished. It prints its findings on the user's screen and restores the original file.

Much data is being passed around within the process of figure 2.1. To clarify how the data is being manipulated, figure 2.2 shows a data flow diagram where the data is represented within blue boxes. There are two main datastructures to follow. The first is the input class and the second is the file holding all prime paths. Figure 2.2 shows the data flow of T2Extension. The processes from figure 2.1 have been used. Arrows with black heads link processes. Arrows with white heads link processes to data. Arrows from a datasource to a process signify that the datasource is used as input to the process. Arrows from a process to a datasource signify that the process returns a modified instance of the datasource.

2.5 Structure

This section will start by studying the package structure of T2Extension and will continue to study the class structure of T2Extension.

T2Extension has a very simple and clean structure. The package *t2ext* is the main package that connects all others. The *util* package holds, as its name suggests, utility classes that every package uses except the *xml* package. Figure 2.3 shows the relation between the different packages. The *xml* package holds classes that process XML. The packages *bcel*, *weaver* and *trace* all have their own *util* packages which are utility packages just for their parent packages.

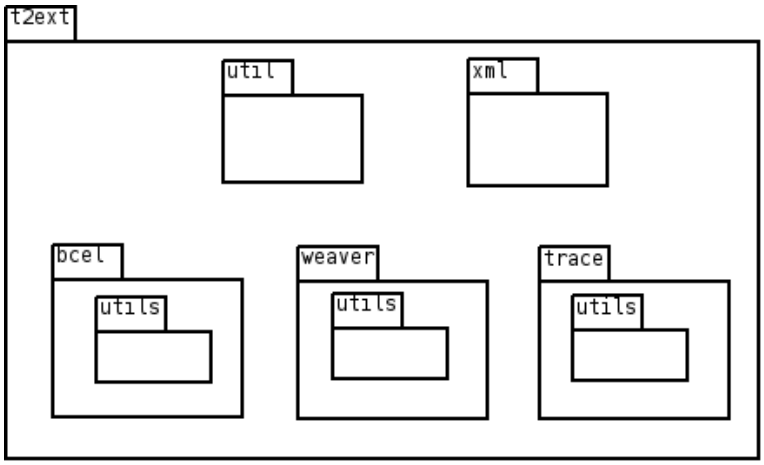


Figure 2.3: Package UML of T2Extension

The *bcel*, *weaver* and *trace* packages are also the worker packages. The construction of the control flow graph from a class file is done in the *bcel* package. This package corresponds to the first stage of the process shown in figure 2.1. The *weaver* package holds an aspect that is later adapted to the class file. This package corresponds to the second stage of the process that T2Extension follows. It also holds the class that handles all the administration of paths that are found so far. The *trace* package holds the classes that analyse and manipulate T2's traces, which corresponds to stage three of the program flow.

Figures 2.4, 2.5 and 2.6 show the class diagrams for the worker packages. Class diagrams for the *util* and *xml* package were not included, as these are less relevant. Keep in mind that almost all classes use these two packages. The class T2Extension is the main class and handles the complete flow of the tool. The class ExecuteRndEngine handles everything to do with T2 and its traces.

In the package *bcel*, the class ControlFlow is the main class. This class takes the target class as input which can be seen in step 3 of figure 2.2. The ControlFlow object constructs the graph

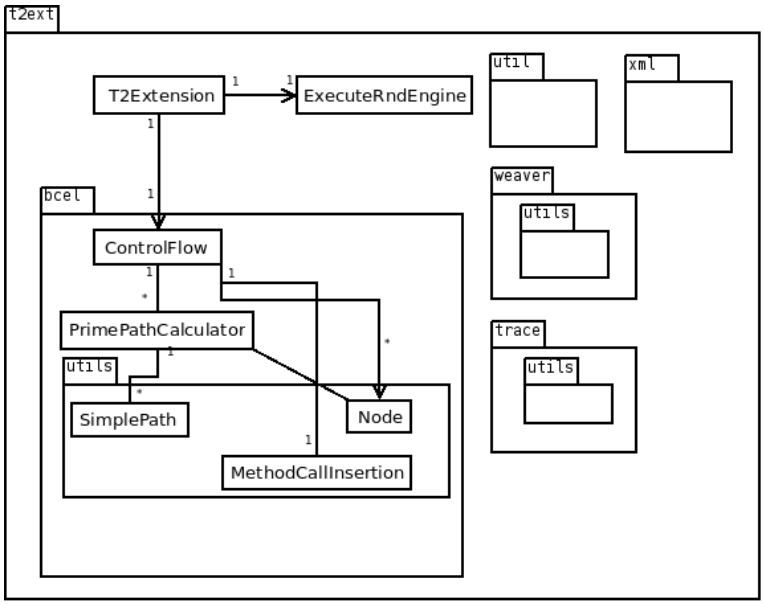


Figure 2.4: UML of bcel package

depicting all paths per method. The object Node represents one node in the graph and holds information about its children, parents and if it is a branching node. The PrimePathCalculator object calculates all prime paths and is therefore linked to SimplePath which is a representation of one reachable path in a graph. In order to register found paths, code is inserted at each decision point in a method. The class MethodCallInsertion returns the code to insert. It ensures that all inserted code in completely correct and that the new class will compile and work properly. The insertion of code corresponds to step 5 in figure 2.2, this step shows that a modified target class is returned and stored.

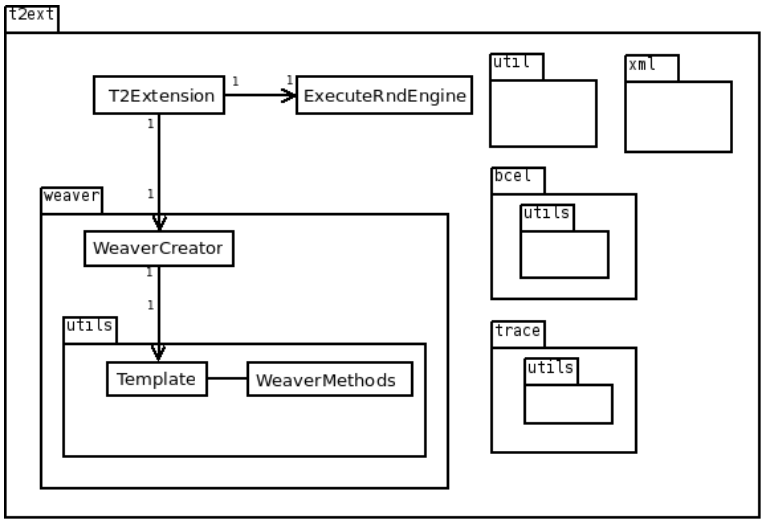


Figure 2.5: UML of weaver package

The package *weaver* corresponds to steps 7 and 8 in figure 2.2. The Template object is an aspect template that is adapted to the class under test, using class information. The class WeaverCreator handles the template modification. Ant will weave the aspect with the class under test. Once woven, the aspect ensures that all found paths are registered. The aspect does

this by calling methods from the class WeaverMethods. This object handles all administration.

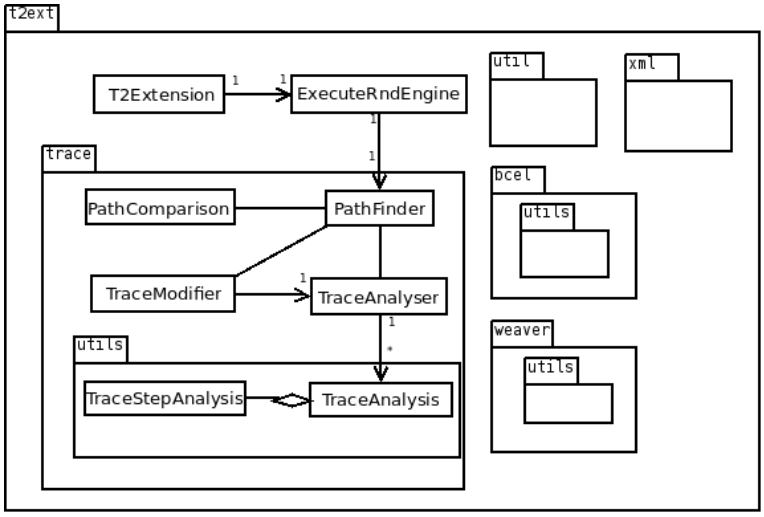


Figure 2.6: UML of trace package

Figure 2.2 shows that the package *trace* takes the woven class, which has been put in a jar file, as input. As new paths are found, it updates the list of prime paths which is shown in step 12.

The PathFinder object in the *trace* package is not called by the class T2Extension but by ExecuteRndEngine. PathFinder is the object that delegates the analysis and manipulation of traces. It passes a collection of traces to TraceAnalyser which gathers information on the traces. The TraceAnalysis object holds this information. It states which methods a certain trace calls and which paths within that method were called. A trace has multiple steps, here represented as TraceStepAnalysis. This object gives more specific information on each step and links it to a method, if possible.

The class PathComparison compares paths that have not been found yet to paths that have been found. For each path that has not been found it returns a path that is most alike. This is later used for trace manipulation, which is what the class TraceModifier does. It uses the analysed traces and takes traces that have executed paths that are most alike the paths that have not been found, and manipulates these traces in the hope of discovering paths that have not been uncovered yet.

Chapter 3

Code Coverage

Before the implementation of T2Extension can be discussed, some formal definitions of code coverage must be explained. These definitions, several of which are taken from [14], will later help to explain the algorithms used in the implementation.

3.1 Coverage

In the field of testing, coverage of source code is a measurement of how much code has been explored by a testing tool. This measurement is normally given in percentages. There are many different coverage criteria, in chapter 1 statement coverage and branch coverage were already mentioned.

Definition 3.1.1 (Code Coverage)**.** The degree to which the source code under scrutiny has been tested.

There are different ways of defining coverage and different interpretations will yield distinct results. Certain criteria should be defined that will box in the definition of coverage.

Definition 3.1.2 (Coverage Criterion)**.** A rule or collection of rules that impose test requirements on a test set.

It is possible to work with different coverage criteria. When working with multiple criteria the possibility exists that a certain criterion is a subset of another criterion. This could be important in proofs about coverage criteria.

Definition 3.1.3 (Criteria Subsumption)**.** A coverage criterion C_1 subsumes C_2 if and only if every test set that satisfies criterion C_1 also satisfies C_2.

As an example, branch coverage subsumes statement coverage. If all branches in a program have been covered, all statements have also been covered. This is not the case the other way around. Look again at Fig. 1.1. If all statements have been covered, then the possibility exists that not all branches are covered. Namely, the branch that skips the *If* clause might not have been found.

Definition 3.1.4 shows an example of a coverage criterion, namely branch coverage.

Definition 3.1.4 (Branch Coverage)**.** The set of test requirements contains each evaluation point (decisions) in the source code under scrutiny.

3.2 Graphs

The easiest way to depict all branches in a program, is to abstract the program as a graph with nodes and edges. Here, each method in a class is abstracted as a different graph. This means

that each graph will have one initial node, because there is always just one entry point to a method. However, it is possible to have more than one final node, because a method can have multiple exit points.

Definition 3.2.1 (Graph)**.** A graph G is represented by the following elements:

- A set N of *nodes*
- One *initial node* N_0, where $N_0 \subseteq N$
- A set N_f of *final nodes*, where $N_f \subseteq N$
- A set E of *edges*, where E is a subset of $N \times N$

It is possible to define a *subgraph* of a graph by taking a subset of the building blocks of a certain graph.

A node can have children and parents.

Definition 3.2.2 (Child Node)**.** A node j is a child node of node i if and only if j has a direct incoming *edge* from i.

Definition 3.2.3 (Parent Node)**.** A node N_i is a parent node of node j if and only if i has a direct outgoing *edge* to j.

In this setup, a node can have 0 or more parents and have at most 2 children. The reason for this is that the graph to be used, will be a *Control Flow Graph*. This is the most common graph used to abstract source code. A Control Flow Graph is a good way to depict all branches in a method, because each node is seen as a decision point. When a decision is to be made, an expression is either evaluated to *true* of $false$. This is why a node can have at most 2 children. However, different branches could come together in the same node, this makes it possible to have multiple parents.

```
1
   public int calc(int y, int x){
3   int i = 0;
    if(x < y){
5     while(i<10){
        x = f(y, i);
7       i++;
      }
9   } else if(x == y)
      x = x + y;
11  else
      x = x - y;
13
    return x;
15 }
```

Figure 3.1: Example Code

Fig. 3.2 is an example of a control flow graph that depicts nodes with multiple children and parents. The graph is based on the code from figure 3.1 Of course, a root node will never have parents, as final nodes will never have children. A node that depicts a line of code where no decision is made, only has one child. A node that depicts a line of code where a decision is made, has exactly two children. Note that in this case, the final node, which is the *return* statement, has three parent nodes.

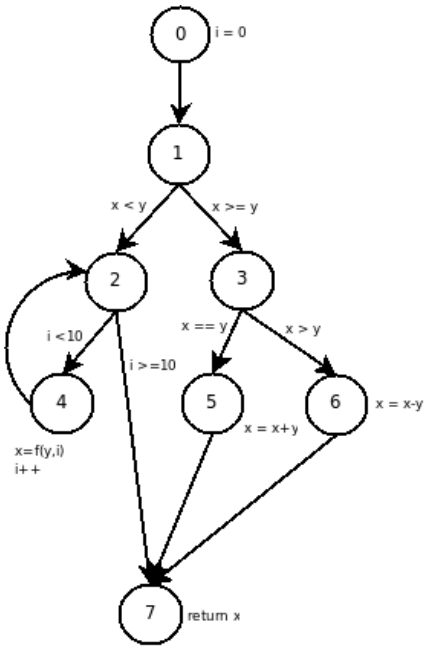


Figure 3.2: Example Control Flow Graph for figure 3.1

3.3 Paths

In a graph there are different paths to take to reach certain nodes. Formally, a path is defined as in definition 3.3.1.

Definition 3.3.1 (Path)**.** A *path* is a sequence $[n_0, n_1, ..., n_M]$ of nodes, where each pair of adjacent nodes, (n_i, n_{i+1}), $0 \leqslant i < M$, is in the set E of edges.

The length of a path is defined as the number of edges it contains. A *subpath* of a path is defined as in definition 3.3.2.

Definition 3.3.2 (Subpath)**.** A path p is a subpath of a path q if and only if q is a concatenation of some path u, p and some path v, where u and/or v can be empty sequences.

It is now possible to construct a set of *test paths*. This set should ideally cover the complete control flow graph so that each path from the initial node to the final nodes is executed at least once. Definition 3.3.3 gives the coverage criterion using paths.

Definition 3.3.3 (Path coverage)**.** The set of test requirements contains each reachable path in a control flow graph.

Definition 3.3.4 (Reachable path)**.** A reachable path in a graph, is a path that can be reached from the initial node.

The definition of path coverage, Def. 3.3.3, expects that all paths from the initial node in a graph to the final nodes are executed. However, it may be possible that certain paths cannot be executed. For example, a valid path in Fig. 3.2 is [0,1,2,7]. This means that the loop would not be executed, this is never possible in this piece of code, so this path can never be executed. It is said that this path is *syntactically reachable* because it is a valid path in the control flow graph, but it is *semantically unreachable* because this path can never be executed. Another problem is that it might be possible to construct infinite paths because of loops. This implicates that, to be able to satisfy the set of test requirements for path coverage, loops will have to be handled differently.

In Fig. 3.2, a valid path is [0,1,2,4,2,8], but also [0,1,2,4,2,4,2,8] and also [0,1,2,4,2,4,2,4,2,8] and so on. It is not possible to construct all these test paths because it is not systematically clear when a loop ends. For example, the loop *while* $(x < y)$, does not state how many cycles this loop contains, it depends on the values of x and y. To still be able to handle paths with loops, paths should be defined in such a way that all the problems stated above are solved. This is described in the next section.

3.4 Cycles

When testing software that contains cycles, one would like to test all possible iterations of a loop, including a test that skips the cycle. However, to expect this from an automated testing tool is infeasible. As explained in the previous section, it is not always statically known how many times a loop executes. It is also possible that a loop is always executed exactly x times which makes it impossible to test what happens if the loop is executed $x - 1$ times.

In order to cover cycles in the best possible way with the information that is statically known about a loop, the following is assumed.

Assumption 3.4.1 (Loops)**.** We only consider test requirements that do not pose a constraint on the amount of times that a loop is executed. Only two cases are relevant: whether the loop is executed, or not.

To be able to handle paths containing loops, test paths are constructed a little differently, using simple paths.

Definition 3.4.2 (Simple path)**.** A path p from node n_i to node n_j is *simple* if no node appears more than once in p, with the exception that the first and last nodes may be identical.

This definition of a path is useful, because it is now possible to define paths with loops, without the path becoming infinite because a simple path will never have an internal loop. However, a simple path can be a loop itself.

Every reachable path through a control flow graph can be composed from simple paths. From this one can deduct that the coverage of all simple paths weakly implies coverage over all reachable paths.

However, not all simple paths are interesting or relevant because they are subpaths of other simple paths. So testing all simple paths would cost very much in memory and time when most paths are actually irrelevant. To this end, the interesting paths are paths that are not a subpath of any other simple path. These are called *prime paths*.

Definition 3.4.3 (Prime Path)**.** A path from n_i to n_j is a prime path if it is a simple path and it does not appear as a proper subpath of any other simple path.

The coverage criterion used in T2Extension is shown below.

Definition 3.4.4 (Prime Path Coverage)**.** The set of test requirements contains each prime path in graph G

The difference between path coverage and prime path coverage is that in prime path coverage the paths are defined a little different in order to abstract away from cycles. This ensures that the test automation does not have to take infinite paths into account when encountering graphs containing loops.

To clarify the above, a small example is given. Again the control flow graph of Fig. 3.2 is used. The first step is to find all simple paths. We do this by starting with simple paths of length 1, then length 2, length 3 and so on until we have reached the longest paths. This is a finite process because of the manner in which loops are treated in simple paths and because there will never be a path longer than the amount of nodes in a graph.

Length 1	**Length 2**	**Length 3**	**Length 4**	**Length 5**
[0]	[0,1]	[0,1,2]	[0,1,2,4]!	[0,1,3,5,7]!
[1]	[1,2]	[0,1,3]	[0,1,2,7]!	[0,1,3,6,7]!
[2]	[1,3]	[1,2,4]	[0,1,3,5]	
[3]	[2,4]	[1,2,7]!	[0,1,3,6]	
[4]	[2,7]!	[1,3,5]	[1,3,5,7]!	
[5]	[3,5]	[1,3,6]	[1,3,6,7]!	
[6]	[3,6]	[2,4,2]*		
[7]!	[4,2]	[3,5,7]!		
	[5,7]!	[3,6,7]!		
	[6,7]!	[4,2,4]*		
		[4,2,7]!		

Figure 3.3: All simple paths for a graph

In Fig. 3.3 all simple paths for the graph from Fig. 3.2 are shown. Paths with a ! behind it, are paths that cannot be extended any further because a final node is reached or the definition of a simple path does not allow it. Paths with a * behind it are loops.

The path [0,1,2,4] cannot be continued any more because the only possibility to extend this path would be [0,1,2,4,2] according to the graph. According to the definition of simple paths, this is not allowed.

From this table of simple paths, it is possible to construct prime paths. Since the paths without a ! or * are paths that can still be extended, these are discarded because they will be subsumed by paths with a ! behind it. Paths with a * behind them can never be a subpath of any other simple path, so these are immediately prime paths. The longest paths are studied first. These are [0,1,3,5,7] and [0,1,3,6,7], they are prime paths because they are not a subpath of any other path. The paths [1,3,5,7] and [1,3,6,7] are subpaths of [0,1,3,5,7] and [0,1,3,6,7], respectively. So those paths are not prime paths. However, [0,1,2,7], [0,1,2,4] and [4,2,7] are prime paths. All other paths with a ! are a subpath of the prime paths found so far. This means that the following paths are prime paths:

[0,1,3,5,7]
[0,1,3,6,7]
[0,1,2,7]
[0,1,2,4]
[4,2,7]
[2,4,2]
[4,2,4]

Note here that there are two representations of the cycle shown in Fig 3.2, namely [2,4,2] and [4,2,4]. This shows that even though most redundancy has been eliminated by using prime paths, there still is some left. Nevertheless, using prime paths instead of simple paths still reduces the amount of paths considerably.

When the prime paths are determined, the set of test paths is determined. The test paths below will cover all prime paths.

[0,1,3,5,7]
[0,1,3,6,7]
[0,1,2,7]
[0,1,2,4,2,4,2,7]

The test path [0,1,2,7] is an example of a path that can never be executed. In the example

code of Fig. 3.1, it is shown that the loop will always be executed and the path [0,1,2,7] is a path that skips the loop. This path will prevent us from meeting the test requirements. This is why the order of the nodes visited in a subpath of a path will be loosened a little. There are several ways to relax this requirement slightly by allowing *sidetrips* or *detours*.

Definition 3.4.5 (Tour)**.** Test path p is said to tour path q if and only if q is a subpath of p.

Definition 3.4.6 (Tour with Sidetrips)**.**
Let p_e be the list of edges of a test path p.
Let q_e be the list of edges of a path q.
Test path p is said to tour path q *with sidetrips* if and only if q_e can be obtained from p_e by deleting elements in p_e.

Definition 3.4.7 (Tour with Detours)**.** Test path p is said to tour path q *with detours* if and only if q can be obtained from p by deleting elements in p.

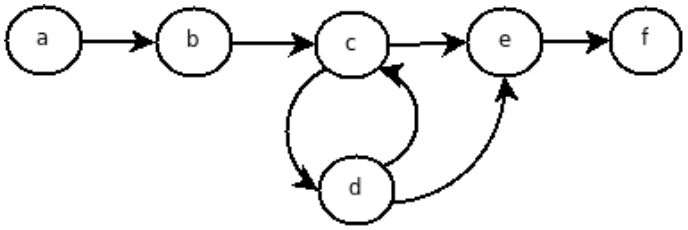


Figure 3.4: Sidetrips and Detours

The difference between sidetrips and detours can be explained with the graph from Fig. 3.4. A subpath p represented as $[b, c, d, c, e]$ is a tour with sidetrips of subpath q represented as $[b, c, e]$. The list of edges for p is represented as $[(b, c), (c, d), (d, c), (c, e)]$. The list of edges for q is represented as $[(b, c), (c, e)]$. The list of edges of q can be obtained from the list of edges of p by deleting the elements (c, d) and (d, c), so p is a tour with sidetrips of q.

The subpath p represented as $[b, c, d, e]$ is a tour with a detour of the subpath q represented as $[b, c, e]$ because q can be obtained from p by removing the element d from p. The subpath p is *not* a tour with sidetrips of q because the edge (c, e) is present in the list of edges of q, but not in the list of edges of p.

Detours have the potential to drastically change the behaviour of a program, not executing an edge could cause undetected faults. Sidetrips are less dangerous, because every edge is still in the same order.

A tour with sidetrips will solve the problem of semantically unreachable paths. But since there is still some danger in allowing sidetrips, a combination will be used. First the graph will be toured without allowing sidetrips. If the test requirements are not met, the graph will be toured again and this time sidetrips will be allowed. Detours are too dangerous and will therefore never be allowed.

To calculate all possible paths from a control flow graph, the calculation of prime paths is used. Since prime paths are slightly different from complete paths in a graph, simple path coverage is not a correct coverage criterion. Below the definition for prime path coverage is given.

The example above shows that prime path coverage covers a determinable set of paths through a control flow graph. In this example path coverage would require to test infinite paths due to the cycle in the CFG. In theory path coverage is a powerful coverage criterion, but in the practice of software testing prime path coverage provides a criterion that is just as powerful but also realistic. Prime path coverage is a criterion that offers a set of requirements that can be met by any given finite test paths (allowing sidetrips).

3.5 Software Controllability

T2 randomly creates values for arguments to a method. This is the reason why it is not possible to guarantee that each branch in a method is executed. Since T2Extension will try to influence the path that are taken in a method, the method's arguments must be controlled. If it is easy to determine which values a method needs to execute a certain branch, it is said that the program's software controllability is high.

Definition 3.5.1 (Software Controllability)**.** How easy is it to provide a program with the needed inputs, in terms of values, operations and behaviours.

The definition above will be used when analysing traces in T2.

Chapter 4

The T2 Tool

The T2 tool is an automatic trace-based unit testing tool that targets Java. It was created by I. S. W. B. Prasetya from Utrecht University and can be found on the following website [6].

Most existing automated testing tools for object oriented programs test methods in isolation. This requires complete specifications of what each method should and should not be able to do to generate meaningful and useful tests. However, this is hardly ever the case. Writing specifications is time consuming and might not be necessary for each method in a program. Especially in large programs it is not worth the time to specify each and every method. When testing methods in isolations, partial specifications are a problem because the possibility exists that errors are not found when they are actually present. Another problem is that many meaningless tests could be generated, which could be costly in time.

T2 solves all these problems because it generates sequences of calls, also known as traces, to methods that are checked as the program runs, or on-the-fly as the creator of T2 calls it. This has a very advantageous side effect, namely that methods check each other. Now, when methods are only partially specified, this will not pose such a problem. It is hoped that if a method does not fail even though it is not correct, later on in the sequence another method will fail due to the faulty method. When an error is produced, the traces can be analysed so that the faulty method can be found.

T2 is quite a powerful tool because it combines 2 different manners of testing. First, as explained above, traces are generated with the intent to check programs on-the-fly. This method of testing is then combined with in-code specifications which means that it is not necessary to generate test scripts. Most testing tools first create test scripts which are then executed. Tests with T2 are therefore faster and more interactive than tests done with other automated testing tools. T2's checking on-the-fly can be compared to runtime verification. The only difference is that the traces are not generated by real applications but by models of application in a controlled environment. This is because T2 is a unit testing tool.

Another advantage of the T2 tool is that the specifications are written in Java and are therefore, as said before, in-code. This makes writing the specifications quite manageable.

When no specifications are given, T2 checks methods for internal errors and runtime exceptions. However, T2 can also check Hoare triple specifications, class invariants and temporal properties. A Hoare triple is a combination of a precondition, a certain command (in this case methods) and a postcondition. A class invariant is a property of a certain class that must be maintained by all the methods in that class. Because traces are generated and these can be analysed, temporal properties can be defined.

Application models are used to generate the traces for the execution of the program to be tested. Also, they are used to drive the generation of traces. From the perspective of a given class C, an application is just a program that uses objects from C. This is why an application model is defined: it is used to (abstractly) specify rules for the range of applications that will use C. In the T2 paper [1] this model is described as follows.

We can impose a model M that indirectly describes the range of applications A that can safely use objects from a class C, in such a way that properties expressed in terms of M will be satisfied by A.

The model can be used for property checking. For example, it could specify that a certain application can only use a list if it has been initialised. Also, the model is used to filter traces, so a generated trace is dropped if it violates the application model. Furthermore, M can be viewed as a state automaton. The model M is seen from class C's perspective. This means that a transition in M is either a call to one of C's public methods [1] or a transition is achieved by another step that does not influence the objects of C.

The application model is used to drive generated traces. A trace is in principle an infinite path in the application model. This means that T2 will actually only use a finite prefix of a trace, called a *test*, for testing. Of course, from an application model many different traces can be calculated that respect this model, the test engine generates a set of tests from the model.

When the user does not specify an application model, T2 simply uses a default model. It conservatively models every possible state transition without any assumptions about a class. The consequence of this is that if a user would specify their own application model, they could restrict the amount of generated traces because they could restrict the model.

T2 is a fast and interactive automated testing tool which shows great potential. However, there is still much room for further development.

[1] To simplify matters a method is either public or private. Here, protected methods are not considered.

Chapter 5

Aspect Oriented Programming for Testing

Object oriented programming facilitates separating different program concerns by separating code in packages, classes and methods. This makes it quite straightforward for a programmer to see what certain code does. Also code can be grouped to a certain extent according to its application. Furthermore, object oriented programming provides a tool to re-use code.

The problem with object oriented programming is that it is not always possible to completely separate different concerns. There are often secondary requirements that are scattered across the code. For example, it could be possible that all throughout a program security checks need to be performed. If at some point it is decided that a change should be made to the security considerations, the complete code would need to be run through to change these checks. This is very time consuming and makes evolution difficult.

Concerns like security or logging are called 'cross-cutting concerns'. A cross-cutting concern is an aspect of a program that affects other concerns. The same behaviour happens throughout the program but it is not possible to encapsulate these concerns into one module. Aspect Oriented Programming [7] (AOP) gives us the ability to separate these cross-cutting concerns from the rest of the program. AOP tries to separate code as much as possible into pieces that do not overlap at all. AOP encapsulates cross-cutting concerns into a special class, an *aspect*. An aspect can alter behaviour or add code to a program without being intertwined with the program itself. It does this by applying *advice* at various *join points*. These join points can then be specified in *pointcuts*.

An advice in an aspect adds behaviour to a program. A join point is a point in a program, this can be a method call or a call to a constructor for example. To relate this to the security example above; it is now possible to put advice in an aspect that asks the program user for login information just before a method is called that accesses a part of the program which needs authorisation. Of course this would not solve very much if the same advice has to be declared for each join point in a program where it applies. Pointcuts provide a way to specify or quantify over joint points. It is possible to say that a certain advice should be applied before any method call in a certain package, or after any method is called that starts with 'set'.

The code that is held in aspects is later woven into the non-aspect part of the code. AOP offers a method to separate cross-cutting concerns in such a way that a program becomes easier to maintain and better readable to the programmer.

Aspects are written in AspectJ [8]. This is a Java dialect that has its own compiler which will weave an aspect with the classes it provides advice to and return normal Java classes. This means that the Java Virtual Machine does not need to be altered in order to work with AOP.

AOP is a powerful tool for T2Extension. Pointcuts can be used to quantify over the methods in the target class. The advice linked to the pointcuts can then register found paths after a method has been executed. However, it is not possible to look into a method with AOP. This

makes it difficult to follow which path is actually executed within a method. The next section looks into ideas that might solve this problem.

5.1 Related Work in Aspect Oriented Programming

Aspect oriented programming has already been widely used to increase modularity in program code. However, it has not been used so much yet for testing purposes. Nevertheless a few examples of uses of AOP for testing were found.

Hridesh Rajan and Kevin Sullivan who were based at the University of Virginia at the time extended the idea of aspect oriented programming to make it much more fine-grained. They describe their research in the following paper; [2]. Their aim was to create a language-centric approach to automated test adequacy analysis. Whether or not a test is adequate is determined by code coverage. So a test is presumed to be useful and meaningful if and only if it covers all the properties of the program to be tested thoroughly. This can be determined by setting up test adequacy criteria. Of course not all code has to always be included in coverage information because not all code is important for testing. For example, library code should not be included in coverage information because this would give a wrong image of the actual coverage of a test. According to Rajan and Sullivan there are three problems with most existing tools for test analysis. They require manual selection of included code to be tested which is tedious and error-prone. The code selection is too coarse-grained making it possible to select methods but not code blocks inside methods. Also, existing tools do not have the ability to express the intent of the tester. All these problems are solved by a new language for test adequacy analysis purposes called *Eos-T* and a tool in which this language is used called *AspectCov*.

Rajan and Sullivan recognised the power of aspect oriented programming for their purposes. Using AOP, they could express the code to be included in a test adequacy criterion using pointcut constructs. The aspect language AspectJ is the most widely used language in AOP and this has many powerful patterns that can be used. There is the possibility to express type patterns that express types in a package and all its subpackages. This is used for the automated selection of code to include in the test criterion. Whenever a call is made to a method in the selected code, this is automatically recognised by the aspect. This means that whenever code is extended to make a new call to the included code, there is one central point that registers this call. If the code was selected manually, the programmer would have to keep track himself of all the calls made to the selected code.

AspectJ is however not completely well suited to solve the problem as shown by the authors. Its join point model is not fine-grained enough for Rajan and Sullivan's purposes. Join points can only be exposed as calls to methods. Whatever happens inside methods is therefore unknown to the tester. The authors want to have complete white box testing and decided to create their own extension to AspectJ that generalises the join points. This language is called *Eos-T* and is an aspect oriented version of the language C#. Its join point model can now also expose conditional and iteration statements. This is quite powerful because now code selection for test adequacy is at a much lower level making it possible to test more thoroughly and express more properties about the code.

The framework that Rajan and Sullivan created around *Eos-T* makes it possible to not only weave advice code as AspectJ does, but it is also possible to express other actions that should be performed at certain joint points. This makes it possible to state in a much clearer fashion what the intent of the tester is. The underlying framework and language can be accessed through the tool built by the authors called *AspectCov*. They used this tool to test their work with two C# projects. The coverage improvement that they have made is at best modest. There are still some optimisations to be made to make *AspectCov* use less memory and time. Nonetheless, this project does seem to have a lot of potential with respect to automated testing.

Daniel Hughes and Philip Greenwood, based at Lancaster University at the time, designed an aspect oriented testing framework [5] with the goal to simplify the testing stage in the software engineering process. They are especially interested in testing distributed applications because they find that these are very difficult to test. There are several problems with monitoring distributed software. First, it is complicated to monitor distributed components running simultaneously. Also, in large applications it is very time-consuming and tedious to manually insert custom monitoring code. Another problem for such specific applications is that testing code becomes very specific which makes it impossible to reuse monitoring code. The proposed testing framework solves all these problems. This should create a testing environment that is maintainable and mostly automatic.

Because testing is done at runtime, there is a need for a framework that can extract information about objects in the program at runtime. This requires the program to reflect on itself. Reflection is defined as the capability of a program to reason about and act upon itself. Java implements its own framework for this, called Reflection API. This API makes it possible to extract all the needed information from a class for the testing framework.

The aspect oriented testing framework will use aspects as templates. Tags will be used to give advice for method calls. This makes the testing code reusable in a very efficient way because joint points can be reused for any and every class the tester wishes to use it for. All that needs to be done is to replace the corresponding tags with the correct class and method.

The testing framework would then automatically insert and remove monitoring code. Also it would support automated monitoring of distributed components. These two advantages lead to a rapid deployment of the testing stage and the ability to reuse code. Also the authors guarantee easy customisation of test cases and the testing framework will ensure the correctness of the testing code.

People based at the Software Engineering Lab of the East China Normal University took Aspect Oriented Programming even one step further [4]. They developed a language called AOTDL and a tool called JAOUT, which uses this language. The tool is used to automatically generate aspect oriented unit tests. The language AOTDL is an extension to AspectJ and is meant to group aspects that are application-related in top-level aspects.

The authors of the paper [4] found that unit testing is a widely used manner of testing, but unit tests are tiresome to write and it is quite difficult to write unit tests that model the usage of an application. AOP makes it possible to separate concerns and therefore in AOP unit tests can be broken down into the different concerns they are supposed to address. This makes it much easier to model applications.

The only problem the authors found when using AOP to generate unit tests was that aspects can represent many different crosscutting concerns and it is not clear for a tester how to point out certain aspects meant especially for testing. If this was possible, then these aspects could automatically be recognised and be turned into test oracles. Test oracles are used to check the correctness of a test result. This can be a formal specification, an assertion or program invariants. It would be quite advantageous to have test oracles generated automatically because formal specifications are difficult to write and mistakes are easily made.

To solve the problem above, the authors created their own language AOTDL which stands for Aspect Oriented Test Description Language. As said before, this language makes it possible to group aspects with similar use in top-level aspects. Now all aspects that are meant for program testing are grouped in a *TestingAspect*. The difference here with a normal AspectJ aspect is small but significant. Since aspects are now grouped, the aspect type is shown in the aspect class declaration. Also, advice are structured a little different. There is a utility advice which holds all pointcut declarations. Then specific advice modules are declared which make it possible to group specific advice for certain pointcuts. The whole idea is to group everything that can be

grouped. This is why similar advice is also grouped in advice modules. For the specifics on AOTDL the reader is referred to the papers [4] or [3].

The specific top-level aspects are translated back to AspectJ by the JAOUT/translator. Then after the aspect code is woven in with the normal Java code, the program is compiled and ready to be used for the generation of unit tests. JUnit [10] is used as the unit testing framework. The testing aspects are now seen as test oracles, these hold all the necessary specifications to determine whether a test is correct or not. This is done by checking what kind of exceptions are thrown when a method is called. The specific advice that was declared in the testing aspect throws a certain exception when the specified conditions fail. If one of these exceptions is thrown, a test is considered to fail. If an exception is thrown that was not specified in the testing aspect, the test is considered to succeed because the tested method did not fail. Of course, when no exception is thrown, the test also succeeds.

Unit tests are generated by the JAOUT/generator. To be exact, the generator creates three classes from a Java class; a JUnit test class that tests all methods in the Java class, a test case provider and a test client which in which test cases can be generated automatically by JMLAutoTest [11]. JMLAutoTest is an automated testing framework by two of the authors of the paper [4].

The authors use a special form of testing called double-phase testing. Their argument is that normal automatic testing generates many meaningless tests, using up unnecessary time and memory. Double phase testing should reduce these meaningless tests considerably. The idea is to filter out the meaningless tests by using statistics. The tester is required to make a criterion that divides the complete test space into partitions. Then, during the first phase of testing, a small number of test cases from each partition is used to determine the number of meaningless tests. A meaningless test is defined as a test where the input lies outside of the range necessary for complete and correct testing. During the second phase, a larger portion of test cases is selected from each partition in the test space according to the proportion of meaningless tests in a certain partition. The idea behind this is that in the second test far less meaningless tests will be selected. The authors used double-phase testing on a binary tree. The objective was to find the complete subtree when a node inside the tree was provided. They found that there were considerably more meaningful tests and the total testing time was a little shorter.

Recently a paper [24] has been written by I.S.W.B. Prasetya, based at Utrecht University and T.E.J. Vos, based at the University of Valencia. It proposes a method for clean in-code algebraic testing based on T2 [1]. Many people write the specifications for their software in separate specification languages. The largest drawback of this approach is that the specifications cannot automatically be synchronised with the code that implement them. This makes it difficult to check if the code adheres to its specifications. However, the advantage of a separate specification language allows for abstract and formal specifications.

As explained in chapter 4, T2 favours in-code specifications. The advantage here is that no new language needs to be learned and the specifications are immediately synchronised with the implementing code. In-code specifications are still very declarative, can be expressed in a formal fashion and are very powerful due to the fact that the complete programming language can be used to express the specifications. A disadvantage of in-code specifications is that they are less abstract which makes it difficult to determine if the specifications still reflect what it was intended to state.

The paper proposes an extension to in-code testing, namely to use in-code algebraic specifications, meaning that the specification has the form of an algebra. Given a system under test (SUT), one or more *test interfaces* are constructed. A test interface is an algebra without axioms. It defines a set of operations and observation functions. It is then possible to build specifications which will refine the test interface by adding axioms. T2 will not test the SUT

anymore directly, but it will communicate with the SUT through the test interface and its specifications.

T2 will test the SUT by calling the operations of the test interfaces. The problem here is that T2 randomly generates values for its test sequences, making it difficult to achieve an acceptable coverage of the methods is the SUT because of the extra abstraction of the test interface. The authors' solution is to create an *adaptive data generator* which means that values are generated using knowledge about the state of the SUT. They propose to build the adaptive data generator using AOP. This provides the advantage that the test interface and the data generator are kept completely separate. This is done by setting pointcuts before an operation is called. Each operation is inspected and needed values are generated using the knowledge about the state of the SUT.

The authors give the example of the Reversi game which is a boardgame with two players (for more information on this game the reader is referred to e.g. Wikipedia). This game will probably have an operation 'move' that must ensure that it is not possible to make an invalid move. With a random data generator it would be difficult to test the operation with invalid and valid moves because the state of the game board is not taken into consideration. A random data generator may find it difficult to find either moves. The adaptive generator will inspect the game board before the 'move' operation is called and will find a square on the board that would represent an invalid move and a square that would represent a valid move. This shows that the adaptive data generator is able to steer T2 towards certain values for its test sequences which will ensure a higher coverage.

This paper shows the strength of T2. With some minor adjustments, a new testing system can be plugged into T2. More importantly, it shows the strength of AOP. The test interface is extended with a powerful adaptive data generator without having to touch the test interface itself.

Above are short descriptions of three possibilities of using Aspect Oriented Programming for testing. The first three projects were in their development stages and seem to have never been finished. The project about in-code algebraic testing is in its proposal stage. This shows that there are very few precedences of AOP in testing.

5.2 Aspect Oriented Programming in T2Extension and Generic Aspects

In T2Extension aspects are used to register and administrate found paths. The idea behind this is that the aspect reacts whenever a branch is executed. When the method has executed, the aspect will have a list of branches which together make a path through the called method. The aspect can then handle the administration. The advantage here is that the original code does not need to be altered much, the aspect will handle all the instrumentation.

The problem however is that some sort of a generic aspect, or template if you will, will need to be created. Aspects are quite powerful and it is possible to create pointcuts that cover a broad range of code. Statically, the extension to T2 cannot know what class it will test and to which package the class belongs. This means that a pointcut should be written as

```
pointcut foo(): call(* *(..));
```

The pointcut above says that all methods of all classes, in all packages should be monitored. With a pointcut like that, one would always be able to cover all methods of the class that is tested. Of course, this is no solution, because all classes, including the T2Extension code and all included libraries would respond to this pointcut.

A template of an aspect will solve this problem. Statically, the aspect can be built up as much as possible. Then, while analysing the code to be tested, the blanks can be filled in.

Here the dynamic compilation of the class to be tested and the aspect itself must be considered, because they have to be woven before the class is actually tested by T2.

The paper [5], as explained in section 5.1 use aspects as templates with the goal of easy insertion and removal of code. This is an idea that can definitely be used by T2Extension. Using a template will facilitate the reusability of the aspect for each class to be tested. Also, the aspect will be very accurate because the exact necessary code for the class to be tested will be inserted. Chapter 7 will explain in detail how the template is built up and used.

Chapter 6

Extending Coverage

Before the actual implementation of the extension to T2 can be discussed, some needed algorithms must be explained. First of all, the construction of a Control Flow Graph should be explored. Also, a formal algorithm must be given for the search of all prime paths and a decision must be made on determining how a side trip can be calculated. Finally, an algorithm is shown that compares paths and returns a metric of how alike two paths are.

6.1 Control Flow Graph Construction

The input for the extension to T2 is a Java class file. This means that a Control Flow Graph (CFG) has to be constructed from bytecode instructions instead of the actual source code. As explained before, a CFG is built per method. So, the input to the CFG construction algorithm will be a list of instructions per method.

There are three kinds of instructions. The first is a normal instruction which will only have one child in the CFG. This instruction could be a method call or a write to or read from memory. Branch instructions are the decision points in the CFG. These instructions are conditional statements. Branch instructions will have two children in the CFG. The last kind of instruction is a jump instruction. This instruction also has one child in the CFG and represents a jump over a certain amount of instructions. This means that it represents a jump over a certain branch or a jump back to the start of a loop.

A loop consists out of a branch instruction, the loop code and a jump instruction. The branch instruction's target is the first instruction after the loop. If the branch instruction's condition fails, the loop is not executed and the program jumps to the target instruction. If the condition succeeds, the code inside the loop is executed. The last instruction before the target instruction is found is a jump instruction. The jump instructions' target is an instruction just before the branch instruction. This way the loop can be executed until the condition on which the loop must be executed fails.

To construct the Control Flow Graph, first each instruction is mapped to a node. Then the list of instructions is walked through from beginning to end. A CFG in this case always has one root node; the first instruction in the list, but may have several end nodes, since it is possible to return from a method with multiple *return* statements in different branches.

When the list of instructions is walked through, each node is connected by looking at the kind of instruction. A normal instruction has the next instruction as a child in the CFG. These two nodes are connected in a parent child relation.

A branch instruction has a so-called target. This is the instruction the program should jump to if the condition this instruction holds fails. This means that the target is the first instruction of the next branch. If the condition succeeds, the next instruction in the list is executed, otherwise the program jumps to the target. This means that the left child of a branch instruction will be the next instruction in the list and the right child will be the target instruction.

A jump instruction also has a target, namely the instruction to jump to. This instruction is the jump instruction's child. To clarify everything explained above, a small example is given.

```
1  public int calc(int a, int b){
     int k = 0;
3    if(a<b)
       a++;
5    else
       b++;
7    k = a+b;
     for(int i=0;i<3;i++)
9      k--;
     return k;
11 }
```

Figure 6.1: Control Flow Graph Example

The source code in figure 6.1 shows a simple method. This method is mapped to a list of instructions. The instructions are shown in the table in figure 6.2. They are simplified to make the table easier to read. The first column gives each instruction its own id so that it will be easier to refer to the instructions later. The second column shows the instruction itself. Details about what each instruction means are not given because this is not important at this stage. The last column matches the instructions to each line of code in figure 6.1 to give some idea of what an instruction represents.

Instruction ID	**Instruction**	**Line of Code**
0	CONST 0	2
1	STORE k	2
2	LOAD a	3
3	LOAD b	3
4	IFCOMP → INSTR 7	3
5	INC a	4
6	GOTO → INSTR 8	4
7	INC b	6
8	LOAD a	7
9	LOAD b	7
10	ADD	7
11	STORE k	7
12	CONST 0	8
13	STORE i	8
14	LOAD i	8
15	CONST 3	8
16	IFCOMP → INSTR 20	8
17	SUBT k	9
18	INC i	8
19	GOTO → INSTR 14	8
20	LOAD k	10
21	RETURN k	10

Figure 6.2: Instruction list

Now that a list of instructions is defined, the Control Flow Graph can be constructed.

Remember that each instruction is first mapped to a node. Later, each node is connected to create the graph. Figure 6.3 shows the CFG that matches the instruction list. Each node number corresponds to the instruction ID from the instruction list table.

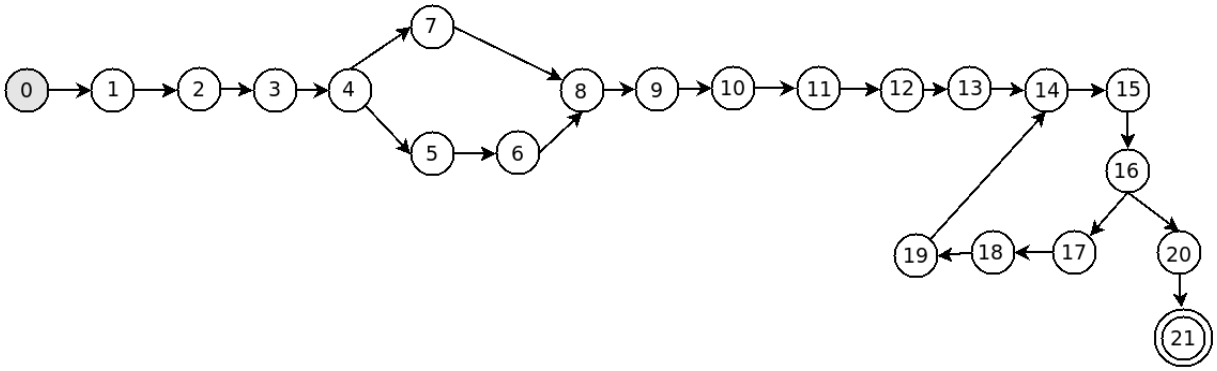


Figure 6.3: Control Flow Graph

In figure 6.3 the nodes [4,5,6] represent the `if` branch from figure 6.1. The nodes [4,7] represent the *Else* branch. The nodes [14,15,16,17,18,19,14] represent the *for* loop on line 8 and 9. Node 21 is the only node without children because there is only one *return* statement.

The branch instruction, which is the instruction with id 4, has the instruction with id 7 as its target. It is possible to deduct that this is the **else** branch because if the condition of the branch instruction fails, the **else** is executed.

The jump instruction with id 6 has instruction 8 as its target. Instruction 6 is the last instruction in the `if` branch and the program will need to jump over the *Else* branch. The *Else* branch ends with instruction 7, so instruction 6 must jump to instruction 8, which the control flow graph depicts.

As explained in section 3.4, it is quite costly to calculate all prime paths in a control flow graph. The more nodes a CFG has, the costlier it is. Figure 6.3 shows that there are many nodes that will not influence the amount of prime paths. Namely, almost all nodes with only one child. For example, if node 1 would be removed and node 0 would be connected to node 2, the CFG will not be changed significantly and it will not result in the addition or removal of a path. If nodes can be removed, the prime path calculation can be optimised. The only rule here is that we do not alter the prime paths in any way, apart from the fact that they will be shortened.

The CFG is optimised as follows. The CFG needs an entrance point, leading to the conclusion that the root node cannot be removed. A branch node and its direct children are also never removed. This means that in our example only nodes 0, 4, 5, 7, 16, 17 and 20 will be saved. These nodes will be connected properly in a manner that will remove unnecessary nodes systematically, but keep all the connections intact.

Definition 6.1.1 (Unnecessary nodes)**.** A node n is unnecessary if and only if it is not a root node (a node with no parents), or a branching node (a node with exactly two children) or a direct descendant of a branching node.

When unnecessary nodes are removed, their parents are connected to their children. Hence, if node 1 is removed, node 0 is connected to node 2. When doing this one node at a time it is ensured that all connections are replaced in a correct fashion. In order to optimise the example CFG, unnecessary nodes are removed. Nodes 1,2 and 3 are unnecessary according to definition 6.1.1. These will be removed and node 0 will become the parent to node 4. Node 6 is also unnecessary, it is deleted and node 5 will point to node 8 at first. Nodes 8 to 13 can also be removed, node 5 and 7 will then point to node 14. Node 14 is the start node of a cycle. If it

is removed, information about the cycle might be lost. This shows the importance of removing one node at a time; to keep cycles intact. At this point, node 5, 7 and 19 are pointing to node 14. When this node is removed, nodes 5, 7 and 19 must point to node 15. This way the cycle stays intact and no information or path is lost. The same trick can be applied to node 15, and now nodes 5, 7 and 19 will point to node 16, which is a branch node. Nodes 17 and 20 are direct descendants to a branch node, these will not be removed. Node 21 will be removed which means that node 20 will become the last node in this CFG.

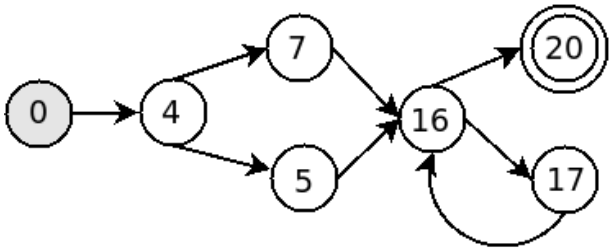


Figure 6.4: Optimised Control Flow Graph

The loop can also be optimised. According to definition 6.1.1, nodes 18 and 19 are unnecessary nodes. This means that these will be removed and node 17 will point to node 16. The optimised CFG is shown in figure 6.4. The CFG is converted from a graph with 21 nodes to a graph with just 7 nodes. Section 6.2 will show how important this is in saving time when calculating prime paths. All other algorithms will work with the optimised CFG.

Summarising; a list of instructions is mapped to nodes in a control flow graph. The nodes are connected by walking through the instruction list and pointing an instruction's node to the instruction's child. The graph is then optimised in order to keep the expenses of the prime path calculation algorithm to a minimum.

6.2 Prime Path Calculation

The prime paths of the optimised control flow graph can be deducted according to definition 3.4.3. Section 3.4 explains the general idea behind this algorithm. Here, pseudocode is given to show what is needed to make sure that all prime paths are found.

To systematically find all prime paths in a control flow graph, three steps are needed. The general idea is to start with all paths of size 1, this means that all single nodes are recorded. Then each path is extended according to the rules of simple paths, see definition 3.4.2. The algorithm looks at the last node in the path recorded so far and the path is extended with the last node's child. If a node has two children, the path is copied and the original path and its copy are extended with the left child and right child respectively.

Figure 6.5 shows the prime path calculation algorithm. The input to this algorithm is the control flow graph G. The output of this algorithm will be a list all prime paths. Prime paths are represented as a list of its nodes.

Before the searching for prime paths is commenced, two auxiliary lists are created. These are named *unfinishedPaths* and *finishedPaths*. The list *unfinishedPaths* is the list with paths that can still be extended. In this initialisation step *unfinishedPaths* holds lists of single nodes $[v]$ where v has to be present in graph G and v should have children so that the path can actually be extended. The list *finishedPaths* holds all simple paths that cannot be extended any further. To this end, it holds lists of single nodes $[v]$ where v of course also has to be present in G but v cannot have any children.

The next step is to extend all paths in the list *unfinishedPaths* until they cannot be extended anymore. When a path is a maximal simple path, it is moved to the list *finishedPaths*. The step

of extending lists is repeated until the list *unfinishedPaths* is empty. When *unfinishedPaths* is empty, all maximal length simple paths have been found.

To effectively remove and add paths consistently to *unfinishedPaths*, the algorithm does not loop over *unfinishedPaths* itself, but over a new list, *unfinishedPaths0*, that holds all the path that can still be extended in the current *while* loop. For each path *l1* in the list *unfinishedPaths0*, the last node, y, is taken from this path. This node will always have at least one child because *l1* is an unfinished path.

A node has either just one child, a left child, or it has two children in which case it will also have a right child. If a node has one child, *l1* can be extended. However, when a node has two children, a copy of *l1* must be made for the right child. The reason for this is that from the current path, two paths can be constructed. One extended with the left child and one extended with the right child. Note here that instead of actually working with *l1*, a path *l2* is created.

If z is a left child, the path *l2* is removed from *unfinishedPaths*. When *l2* is extended with z and it cannot be extended anymore, it will need to be removed from *unfinishedPaths* anyway and must be added to *finishedPaths*. When *l2* is extended with z and it can still be extended, it will be put back into *unfinishedPaths* but with a new node extended to it. This step ensures that at one time *unfinishedPaths* will be empty. of course, if z is a right child, *l2* does not need to be removed from *unfinishedPaths* because it is a copy of *l1* and does not exist in *unfinishedPaths*.

There are three constraints that dictate when a simple path is maximal length. The first one is a logical one, the last node in the path has no children. The second constraint is that the path represents a loop, which is another way of saying that the first and last node in a path are the same node. The third constraint states that a simple path cannot hold internal loops.

If for a child of y one of the first two constraints hold, the child is added to *l2* and then added to *finishedPaths*. If the third constraint holds, then the child will not be added to *l2* but *l2* will be added to *finishedPaths* as is. If none of the three constraints hold, the path can still be extended and is returned to *unfinishedPaths* so that it can be extended again in the next *while* loop.

The last step in this algorithm filters *finishedPaths*. It makes sure that all subpaths of other paths in this list are removed so that only the prime paths of graph G are returned.

The prime path calculation is quite expensive because it is required to iterate over the control flow graph many times. For each node n all maximal paths are calculated. Of course nodes nearer to nodes without children will not have such long paths, but it could be the case that there are many nodes whose paths are also of length n. This means that if each node has just one path that originates from it, the algorithm's complexity in the worst case is already n^2, assuming that an operation to add a node to a path has a cost of 1.

There are many nodes which have two children. All these nodes add an extra path to the list of finished paths. A control flow graph with just three different **if** branches has 2^3 paths starting from the root. So, in a CFG with k branches, the complexity of the prime path calculation could be, in the worst case, $O(n^2 \times 2^k)$. However, in practice methods do not typically have lots of (sequenced) branches. This makes n^2 the more dominant factor.

6.3 Tour with Side Trips

It often happens that the prime path calculation algorithm finds paths that are semantically unreachable, that is to say; they can never be executed, as explained in section 3.4. The consequence of these paths being present is that the test requirements for prime path coverage are not met. To solve this problem, tours with sidetrips are allowed, see definition 3.4.6. The consequence is that after everything has been tried to find a certain path p_1 and it is still not found, an algorithm will test if it would be found when sidetrips are allowed.

To decide whether p_1 is found with sidetrips, one must look at all test paths found so far and compare these to p_1. If a path p_2 is found that tours p_1 with sidetrips, it is allowed to say

that p_1 has been tested. Note here that p_2 is a *test* path and not a simple path. So it is a tour of the control flow graph.

If the definition of tours with sidetrips is applied to paths p_1 and p_2, we say that p_2 tours p_1 with sidetrips if and only if every edge in p_1 is also in p_2 in the same order.

The prime path algorithm represents paths as a list nodes. An edge between two nodes in a path is then the connection between a node and its direct neighbour to the right. To illustrate, we have a path of nodes represented as their ids [0,1,2,3,4], the edges in this path would be represented as [(0,1),(1,2),(2,3),(3,4)].

The algorithm that checks whether a certain test path is a tour with sidetrips of a certain path will first represent the paths as lists of edges. Then the edges are compared. Figure 6.6 shows the algorithm for the comparison in pseudocode. Input to the algorithm are two paths represented as their edges. Output is a boolean value that is set to true if *P2* is a tour with sidetrips of *P1*.

The list *P1* is the path to find. Each of its edges must be found in *P2* in exactly that order. Accordingly, for each edge of *P1* the algorithm must walk through *P2* looking for that edge. If the edge is found, its position is stored in *counter* and the next edge is taken from *P1*. For this edge, the algorithm starts looking in *P2* where it left off and continues. If the end of *P2* is reached and there are still some edges left unexplored in *P1*, then *P2* is not a tour with sidetrips of *P1*.

To clarify the workings of this algorithm with an example, the control flow graph from figure 6.4 is used. The path that must be found is [0,4,5,16,20]. The test path to examine is [0,4,5,16,17,16,17,16,20]. The first thing to do is to convert these two paths into their edges.

When converted into edges, *P1* will become [(0,4),(4,5),(5,16),(16,20)] and P2 will become [(0,4),(4,5),(5,16),(16,17),(17,16),(16,17),(17,16),(16,20)]. The first edge of *P1* is (0,4), this matches the first edge of *P2*, so the algorithm continues. The second edge of *P1* also matches the second edge of *P2*, hence the algorithm continues again. The same holds for the third edge in *P1*. However, the fourth edge in *P2* is (16,17) and the edge to find is (16,20). The program continues along *P2* looking for edge (16,20). The last edge in *P2* is the edge (16,20), so it can now be said that *P2* is indeed a tour with sidetrips of *P1*.

If *P2* had been represented as [(0,4),(4,7),(7,16),(16,17),(17,16),(16,17),(17,16),(16,20)], it would not have been a tour with sidetrips of *P1* because the edges (4,5) and (5,16) are not present in *P2*. Also, if it was possible to escape the loop in figure 6.4 and *P2* was represented as [(0,4),(4,7),(7,16),(16,17),(17,20)] it would still not be a tour with sidetrips because *P1*'s edge (16,20) would not be found in *P2*.

This proves that the sidetrips algorithm is sound. It will only detect sidetrips if the test path holds complete loops.

The complexity of the sidetrip calculation algorithm is not as bad as the complexity of the prime path calculation. The test path *P2* is walked through once in total. Assuming each step in the walk-through has a cost of 1, then the cost of the walkthrough is n, the length of *P2*. The length of prime path *P1* is also around n. Mostly it will actually be slightly shorter. The algorithm walks through *P1* once as well, making the complexity of the complete algorithm O(2n), thus linear.

6.4 Path Comparison

There are many ways to compare paths but it is very difficult to find a way to compare paths that will actually give a good metric when it comes to comparing paths in a control flow graph. The extension to T2 will need a methodology that gives a metric of how similar two paths are. One must think about the different possible structures of a graph and wonder how much influence the structure of an **if** branch with many **else if** branches has on comparing paths. Also, what influence does a graph with many decision points have? Let us look at different comparison

methodologies and see if they are suited for the problem.

One way to compare paths would be to examine the difference in suffixes in two paths. Two paths, p_1 and p_2 are compared and given a metric of how similar p_1 is to p_2. The metric counts the nodes in the suffix of $p1$ from the point that a node in $p2$ has been found that is not present in $p1$ or not present in the correct order. For example, if p_1 is represented as [0,1,2,3,4,5] and p_2 as [0,1,2,3,6,7], then two nodes of p_1 have not been covered, namely node 4 and node 5. The metric would be 2 out of 6 nodes that have not been covered. This metric works well for a control flow graph with $if - then - else$ branches. Figure 6.7 shows such a graph.

Take any path from the graph in figure 6.7 and the calculation of similarities gives a good representation of how alike paths are. For example, if one wants to know which path is most similar to the path [0,1,3,4,6,7,9], the suffix metric would state that path [0,1,3,4,6,7,8] is the most similar, differing in only one node suffix. If we would like to know which path is most similar to path [0,1,2], all other paths are evenly similar to that path, which makes sense because the path breaks off at the first decision point and of all other paths there is not one more or less similar to that path as another one.

However figure 6.8 shows a graph for which the suffix metric will not work. If path [0,1,2,4,6] is examined, the suffix metric would say that path [0,1,3,4,6] is very different from the path we are examining with a suffix of 3 nodes out of 5 that is different from path [0,1,3,4,6]. Actually these two paths are quite similar, showing that this metric will not work properly in all cases. Note here that this is a small graph; when looking at larger graphs there will be more paths to consider.

Figure 6.8 makes one wonder if it might be better to use a metric that compares paths according to their nodes and the order in which the nodes are found. In the case of figure 6.7 this would mean that the metric would be the same as the suffix metric. In the case of figure 6.8 this metric would work better than the suffix metric does. The difference between path [0,1,2,4,6] and [0,1,3,4,6] is now only 1 node, which gives a better representation of how similar they are.

Nevertheless, this metric comparing nodes works too well. Figure 6.9 shows a graph that depicts a method with two `if` branches without `else` branches. Now, if path [0,1,3,4,6] is examined and compared to path [0,1,2,3,4,6], the metric comparing nodes would say that path [0,1,3,4,6] has no nodes different from [0,1,2,3,4,6] but of course, this is not the case.

From studying both these metrics, it is concluded that it is best to use a metric of comparison that compares edges and their order. Looking at figure 6.9 again, path [0,1,3,4,6] has 1 edge different from path [0,1,2,3,4,6], namely edge (1,3) is not present in the other path. This gives a better representation of the similarity of the two paths.

The metric that compares edges calculates a similarity metric between path p_1 and path p_2 by counting how many edges can be found in p_2 in exactly the right order and subtracts this amount from the total amount of edges. This leaves the different edges as the metric which is exactly what is required. The algorithm is much like the algorithm that checks whether a test path is a tour with sidetrips of a certain path, depicted in fig 6.6. Instead of returning a boolean value, this algorithm counts the edges of p_1 not found in p_2 and returns this number. This metric will work on all control flow graphs with all different types of branches, which makes it versatile and useful.

To prove that the algorithm for path comparison is correct, a proof of induction is given in figures 6.10, 6.11 and 6.12. The idea behind this proof is to prove that if a certain path covers another path, the distance between the two paths should be 0. This should also hold the other way around. In other words, if path p is covered by path q, then all edges in p can be found in q in the correct order, making the distance between p and q 0, and if the distance is 0, then p is covered by q. In this proof, paths are seen as a list of its edges.

Figure 6.10 gives the recursive definitions of distance and 'covered by'. In this proof, path p is compared to path q. The distance is expressed as a difference in edges. There are two base

cases. The first case states that when all edges in p have been found, the distance between p and q is 0. The second base case covers the opposite. If all edges in q have been inspected but some edges in p are uninspected, the distance between p and q will be the amount of edges left in p. Statement C in figure 6.10 covers all other possibilities for p and q. It is a recursion down the two paths comparing edges. If e_1 is equal to e_2, the distance between the paths stays the same and continues with the tails of p and q. If e_1 is not equal to e_2, there are two options. If e_1 is not present in q, there is an edge difference and the difference is added to the distance. However, if e_1 is present in q at a larger index in the list, the comparison algorithm must go there and continue its search for edges from there. To give the reader a picture; let p be represented by [(0,1),(1,2),(2,3),(3,4)] and q by [(1,2),(2,3),(3,0),(0,1),(1,3),(3,4)]. The first edge in p is (0,1) and it occurs as the fourth edge in q. So the algorithm must jump to that point. The second edge in p is (1,2), which is not present in q after the fourth edge, so 1 is added to the distance. The same holds for the edge (2,3), making the distance 2. The edge (3,4) however can still be found in q, making the total difference between p and q 2.

The definition of 'covered by' is expressed as a boolean value. Again, this definition has two base cases. If p is empty, then all of its edges have been found in the correct order in q which means that p is covered by q, which is the first base case. The second base case states that if all edges in q have been inspected, but there are still edges left uninspected in p, p is not covered by q. Statement F handles all other cases of p and q and recurses down the paths. If e_1 is equal to e_2, the tails of the two path are checked. If the edges are not equal, we try to find e_1 further along q.

The base steps will be proven first. This is depicted in figure 6.11. It is fairly straightforward. If p is an empty list, then according to the definitions, if p is covered by q then the distance between p and q is 0 and vice versa. If q is empty, then according to the definitions, the distance must be greater than 0 because there is always at least 1 edge left in p which should lead to the fact that p is not covered by q, which is correct because the second base case in the definition of 'coveredBy' returns *false*.

Figure 6.12 shows the proof for all other cases of p and q. The first case in this proof is when e_1 and e_2 are equal. If the distance between the complete paths is 0, then the distance between their tails must be 0 according to the definition of distance. Using base case (1), we can then say that the tail of p is covered by the tail of q. Since e_1 and e_2 are equal, then we can also say that the complete path p is covered by the complete path q which shows that this case is also correct.

The second case to prove is when e_1 and e_2 are not equal and e_1 is not present in q. This should lead to p is not covered by q which would make the distance larger than 0. Which is the case, using base case (2) and definition C. The last case to prove is when e_1 and e_2 are not equal but e_1 is present in the remainder of q. This proof follows the same structure as proof (3) (1).

Figures 6.10, 6.11 and 6.12 prove that the algorithm for path comparison is correct. The extension to T2 will use this algorithm to manipulate traces in T2. More on this in chapter 7.

```
Algorithm primePathCalc (G) :
  // initialisation :
  unfinishedPaths ← {[v] | v ∈ nodes (G), v ∉ end (G)}
  finishedPaths ← {[v] | v ∈ nodes (G), v ∈ end (G)}

  while unfinishedPaths != empty do
    unfinishedPaths0 ← unfinishedPaths

    for (l1 : unfinishedPaths0) do
      y ← last (l1)
      for (z : children (y)) do
        if z = leftchild (y) then
          l2 ← l1
        else
          l2 ← copy l1

        if z = leftchild (y) then
          unfinishedPaths ← unfinishedPaths / {l2}
        if z ∈ end (G) ∨ z = first (l2) then
          finishedPaths ← finishedPaths ++ [l2 ++ [z]]
        else if z ∈ l2 then
          finishedPaths ← finishedPaths ++ [l2]
        else
          unfinishedPaths ← unfinishedPaths ++ [l2 ++ [z]]
      endfor
    endfor
  endwhile

  // Filter :
  result = {t | t ∈ finishedPaths, ∀ u ∈ finishedPaths : t subpath u → u = t}
  return result
```

Figure 6.5: Pseudocode prime path calculation

```
Algorithm sideTrips (P1, P2) :
  n ← length (P2)
  result ← true
  counter ← 0
  for (edge1 : P1) do
    if counter < n then
      edge2 ← P2 [counter]
      while edge1 ≠ edge2 ∧ counter < n do
        counter ← counter + 1
        edge2 ← P2 [counter]
      endwhile
    else
      result ← false
  endfor


  return result
```

Figure 6.6: Pseudocode tour with sidetrips

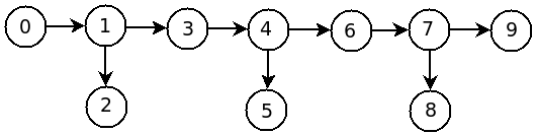


Figure 6.7: Control Flow Graph with else if branches

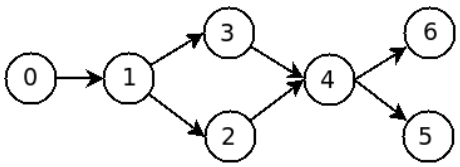


Figure 6.8: Control Flow Graph with different if branches

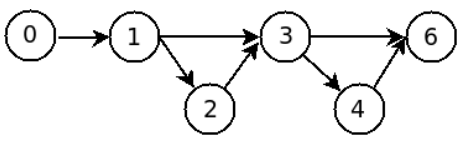


Figure 6.9: Control Flow Graph with different if branches without else branches

// Definition of distance between path p and path q

$$
\begin{array}{lllll}
A) & \triangle\ [\,]\ q & = & 0 & \\
B) & \triangle\ (e_1 : p)\ [\,] & = & \#(e_1 : p) & \\
C) & \triangle\ (e_1 : p)\ (e_2 : q) & = & \textbf{if}\ \ e_1 = e_2 & \rightarrow\ \ \triangle\ \ p\ q \\
 & & \square & e_1 \neq e_2\ \ \wedge\ \ e_1 \notin q & \rightarrow\ \ \triangle\ \ p\ (e_2 : q)\ \ +\ \ 1 \\
 & & \square & e_1 \neq e_2\ \ \wedge\ \ e_1 \in q & \rightarrow\ \ \triangle\ \ (e_1 : p)\ q
\end{array}
$$

// Definition of path p covered by path q

$$
\begin{array}{lllll}
D) & [\,]\ \ coveredBy\ \ q & = & true & \\
E) & (e_1 : p)\ \ coveredBy\ \ [\,] & = & false & \\
F) & (e_1 : p)\ \ coveredBy\ \ (e_2 : q) & = & \textbf{if}\ \ e_1 = e_2\ \rightarrow\ \ p\ \ coveredBy\ \ q & \\
 & & \square & e_1 \neq e_2\ \ \rightarrow\ \ (e_1 : p)\ \ coveredBy\ \ q &
\end{array}
$$

$$
TO\ PROVE : \forall\ \ p, q : p\ \ coveredBy\ \ q \iff \triangle\ \ p\ q = 0
$$

Figure 6.10: Proof path comparison definitions

(1) $p\ =\ [\,]$

$\triangle\ \ [\,]\ \ q = 0$
$\quad \Leftrightarrow\ \ \{\, definition\ \triangle, see\ A \,\}$
$true$
$\quad \Leftrightarrow\ \ \{\, definition\ coveredBy, see\ D \,\}$
$[\,]\ \ coveredBy\ \ q$

// Proven : $[\,]\ \ coveredBy\ \ q \iff \triangle\ \ [\,]\ \ q = 0$
DONE

(2) $p \neq [\,]\ \ \wedge\ \ q = [\,]$

$\triangle\ \ p\ \ [\,]\ =\ 0$
$\quad \Leftrightarrow\ \ \{\, definition\ \triangle, see\ A \,\}$
$\#\, p = 0$
$\quad \Leftrightarrow\ \ \{\, p\ is\ not\ empty \,\}$
$false$
$\quad \Leftrightarrow\ \ \{\, definition\ coveredBy, see\ E \,\}$
$p\ \ coveredBy\ \ q$

DONE

Figure 6.11: Proof path comparison, base step

(3) *The first argument has the form* $(e_1 : p)$
The second argument has the form $(e_2 : q)$
There are three cases to consider :
(1) $e_1 = e_2$
(2) $e_1 \neq e_2 \quad \wedge \quad e_1 \notin q$
(3) $e_1 \neq e_2 \quad \wedge \quad e_1 \in q$

(3.1) **case** $e_1 = e_2$

$\triangle \quad (e_1 : p) \quad (e_2 : q) \quad = \quad 0$
$\quad \Leftrightarrow \{ \textit{definition } \triangle, \textit{see } C \}$
$\triangle \quad p \quad q \quad = \quad 0$
$\quad \Leftrightarrow \quad \{ \textit{Induction Hypothesis} \}$
$p \quad \textit{coveredBy} \quad q$
$\quad \Leftrightarrow \quad \{ \textit{definition coveredBy}, F \}$
$(e_1 : p) \quad \textit{coveredBy} \quad (e_2 : q)$

DONE

(3.2) **case** $e_1 \neq e_2 \quad \wedge \quad e_1 \notin q$

$\triangle \quad (e_1 : p) \quad (e_2 : q) \quad = \quad 0$
$\quad \Leftrightarrow \quad \{ \textit{definition distance}, C \}$
$\triangle \quad p \quad (e_2 : q) \quad + \quad 1 \quad = \quad 0$
$\quad \Leftrightarrow$
false
$\quad \Leftrightarrow \quad \{ e_1 \notin q, 2 \}$
$(e_1 : p) \quad \textit{coveredBy} \quad q$
$\quad \Leftrightarrow \quad \{ \textit{definition coveredBy}, \textit{see } F \}$
$(e_1 : p) \quad \textit{coveredBy} \quad (e_2 : q)$

DONE

(3.3) **case** $e_1 \neq e_2 \quad \wedge \quad e_1 \in q$

$\triangle \quad (e_1 : p) \quad (e_2 : q) \quad = \quad 0$
$\quad \Leftrightarrow \quad \{ \textit{definition } \triangle, \textit{see } C \}$
$\triangle \quad (e_1 : s) \quad q \quad = \quad 0$
$\quad \Leftrightarrow \quad \{ \textit{Induction Hypothesis} \}$
$(e_1 : p) \quad \textit{coveredBy} \quad q$
$\quad \Leftrightarrow \quad \{ \textit{definition coveredBy}, \textit{see } F \}$
$(e_1 : p) \quad \textit{coveredBy} \quad (e_2 : q)$

DONE

Figure 6.12: Proof path comparison, induction step

Chapter 7

The T2Extension Tool

This chapter describes the implementation of the T2Extension tool. All the necessary theory has been discussed in the previous chapters. Chapter 2 sketches an architectural overview of T2Extension. A short manual on T2Extension can be found in appendix A.

T2Extension instruments bytecode in order to keep track of prime path coverage. Bytecode instrumentation is the adaptation of class files in contrast to source code instrumentation. Each method in a class has bytecode inserted that models the method's paths. An aspect will then respond to the inserted bytecode and register paths that it finds. The aspect responds to the instrumented bytecode when traces generated by T2 execute methods in the class under test (CUT). The traces are manipulated in order to search for paths. When T2Extension has finished searching for paths, it reports on the prime path coverage.

Here the three most important components of T2Extension are explored. First, the modelling of branches is discussed. Then the use of aspect oriented programming is shown. Finally, the use of T2 and its traces are investigated. Chapter 8 will reveal if T2Extension has actually improved T2's prime path coverage.

7.1 Modelling Paths

A model is a good abstraction and consistent manner of creating an overview of a method's paths. In order to be able to calculate a method's prime paths, a control graph is first built. Several examples have been given already. Figure 3.2 shows a control flow graph based on source code. However, T2Extension does not work on (java) source code (this was part of our requirement), but on the byte code class file instead. The algorithm for the construction of this graph is described in section 6.1.

BCEL is used to parse bytecode and will provide instruction lists per method. Figure 7.1 shows the general steps of how BCEL is used and what is actually needed to implement the algorithm for the construction of a control flow graph.

T2Extension first creates a class object from the input class file so that BCEL can instrument the bytecode, this is shown in lines 2 and 4. Then for each method in scope (line 13) the instruction list is retrieved, depicted on line 18. Each instruction in the instruction list is mapped to a node (line 20). Afterwards the control flow graph is created as described in section 6.1 which is shown on line 25.

When the control flow graph is created, it is first optimised and then the prime paths are computed. The control flow graph is also used as information for the bytecode instrumentation. Once this is done for all methods, the new CUT is saved.

```
public void createControlFlow() {
  JavaClass clazz = Repository.lookupClass(byteClass); //BCEL class object
  //A classGen makes it possible to alter the BCEL class object
  ClassGen classGen = new ClassGen(clazz);
  Method[] methods = classGen.getJavaClass().getMethods();
  for (int i = 0; i < methods.length; i++) {
    Method method = methods[i];
    //The LineNumberTable maps instructions to their source code line numbers
    LineNumberTable lines = method.getLineNumberTable();
    String methodName = method.getName();
    //Check if T2 will actually test this method
    boolean allowed = checkIfMethodIsInScope(method);
    if (allowed) {
      ConstantPoolGen pgen = classGen.getConstantPool();
      String cname = classGen.getClassName();
      //A MethodGen makes it possible to alter the BCEL class object
      MethodGen methodGen = new MethodGen(method, cname, pgen);
      InstructionList list = methodGen.getInstructionList();
      //Map each instruction to a node
      setInstructionToNode(list);
      //Gather information for the classInfo.xml file
      _builder.registerMethodsForClassInfo(methodName, methodGen
          .getArgumentTypes());
      //Connect all the nodes by walking through the instruction list
      createCFG(list, methodName, lines);
      //Insert BCEL code at decision points
      InstructionList newList = createNewList(list);
      methodGen.setInstructionList(newList);
      classGen.removeMethod(method);
      classGen.addMethod(methodGen.getMethod()); //Save the new method
      list.dispose();
    }
  }
  //Store the new bytecode class
  writeToFile();
  _builder.createClassInfoFile();
  _builder.createPathFile(getPaths());
}
```

Figure 7.1: Flow of method modelling

7.1.1 Administration of Paths

In order to keep track of the prime path coverage, a method must be monitored. When a method has executed, it must be possible to determine which of the method's paths were executed, but how can T2Extension know which path was actually followed? The use of an aspect was mentioned before. In general, an aspect can monitor pieces of code inside a target class using pointcuts and joinpoints (see chapter 5). Advice can be executed before, after or during the execution of a pointcut. T2Extension can exploit aspects to check which paths has been followed after a method has been executed. The problem is that most aspect weavers, like AspectJ, can only monitor methods. It is not possible to weave advice around statement constructs inside a method. The consequence is that an aspect in AspectJ cannot monitor the decision points within a method. As discussed in section 5.1, some have tried to build a compiler based on AspectJ that can weave advice around lines of code inside a method. This is a sizable amount of work and needs much knowledge of the inner workings of AspectJ.

A better technique might be to work with AspectJ instead of modifying it. If AspectJ's lowest level of units are methods, then branches will be registered using methods. The idea is

to insert a method call to a unique method at each decision point. An aspect can then weave advice around the unique method and register which branch was taken at each decision point. The branches together will form the followed path. A small example is given in figure 7.2. It is based on the method from figure 6.1. At each branch leading out from a decision point a call to a fresh auxiliary method is inserted. The method's name always starts with 'aux'. The auxiliary method itself is empty. It is solely used as a target for advice weaving. The method name is suffixed with a number; this number corresponds to the ID of the corresponding node in the method's CFG; in this case it is the graph as shown in Figure 6.4. Since advice is woven around the auxiliary methods, the advice can work together in order to construct the path that was followed. If the order of methods called is aux_calc_5() and then aux_calc_17(), the aspect will deduct that path [0,4,5,16,17,16,20] was followed for this particular method and register that this path was found. Actually it is a little more complicated due to loops, but this will be covered later. This example gives the general idea.

```
public int calc(int a, int b){
  int k = 0;
  if(a<b){
    aux_calc_5();
    a++;
  }
  else{
    aux_calc_7();
    b++;
  }
  k = a+b;
  for(int i=0;i<3;i++){
    aux_calc_17();
    k--;
  }
  return k;
}
```

Figure 7.2: Example path registration

A tool named BCEL is used to insert the method calls. BCEL represents classes as objects containing all the information about the class, methods, fields and bytecode instructions. It almost works like Java reflection, but it can do more because the class object can be manipulated and stored. An idea of how BCEL works can be seen in figure 7.1.

T2 normally just generates test sequences for public methods. It is possible for the user to pass arguments to T2 requesting that T2 includes or excludes private methods, or static methods, etc. For more information on these arguments, check the T2 manual [1]. T2Extension takes the user's requests into account and will only use BCEL to manipulate the methods that are in scope. In principle, T2Extension has the same behaviour as T2 when deciding which methods are in scope.

7.1.2 Nested Branches

When source code is compiled, some information is lost. Of course, no information is lost that is crucial to the behaviour of the source code, but a high level language is compiled to bytecode instructions, so some of Java's expressiveness will be lost. This is a disadvantage of bytecode instrumentation. For example, it is not possible to distinguish between decision points with more than one condition to test, or nested branches. BCEL is a powerful and expressive tool, but it also cannot distinguish between these different types of branches. This is a problem encountered during the implementation of T2Extension.

```
1 public int calc(int a, int b){
    if(a<b && b>10)
3     a = a+b;
    else
5     a = a-b;
    return a;
7 }
```

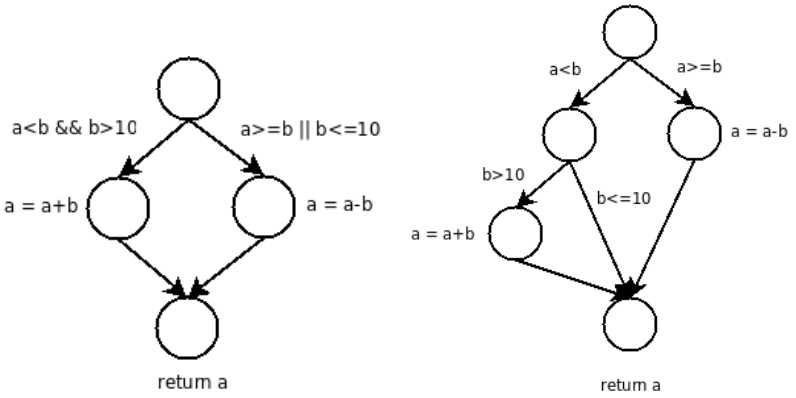
(a) Multipe conditions

```
1 public int calc(int a, int b){
    if(a<b)
3     if(b>10)
        a = a+b;
5   else
      a = a-b;
7   return a;
  }
```

(b) Nested branch

Figure 7.3: Different branches with the same representation

Figure 7.3 shows two methods with different branches, but, due to the way Java source code is compiled, the instruction lists produced by BCEL for these methods are exactly the same. This is understandable because the methods are semantically the same. The bytecode instruction lists are exactly the same because all the conditions for a decision point are split up and turned it into several decision points. For example, the decision point on line 2 in figure 7.3(a) is split into two branch instructions which would result in a control flow graph as represented by figure 7.4(b). Arguably one would prefer to represent the code in figure 7.3(a) as the CFG in figure 7.4(a). The effect of this graph representation for the method in figure 7.3(a) is that an extra path is created when this is not necessary and not entirely correct. The extra node that is created results in extra prime path calculation time which is already so expensive. One could ignore this, but the difference in syntax should result in different graphs for the code in figures 7.3(a) and 7.3(b).



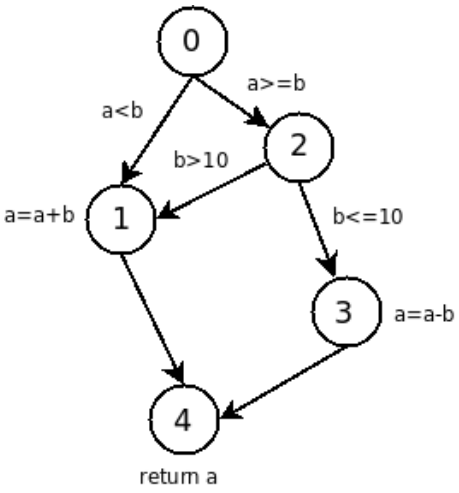
(a) Graph for fig. 7.3(a)

(b) Graph for fig. 7.3(b)

Figure 7.4: Control Flow Graphs for fig. 7.3

A bigger issue lies in the representation of branches for which one of many conditions must succeed instead of all conditions having to succeed as in figure 7.3(a). BCEL also represents this using multiple decision points. If one condition fails, the next is tried. If the next condition succeeds, the instruction list jumps to the code inside the branch body. Apart from the fact that a control flow graph based on this list would create nodes for each condition which again means that the prime path calculation time would grow, paths are created that might never be executed which violates the coverage criterion.

Figure 7.5 shows a piece of code with a branch with 2 conditions. For the code inside of the branch to execute, just one of these conditions has to succeed. Next to the code resides the control flow graph based on the method's instruction list. This shows that if the first condition




```
  public int calc(int a, int b){
2   if(a<b || b>10)
      a = a+b;
4   else
      a = a-b;
6   return a;
  }
```

Figure 7.5: Incorrect graph representation

succeeds, the list will immediately jump to the code inside the branch. If the first condition fails, the second condition is checked and if this succeeds, the code inside the branch is executed after this check.

The possibility exists that the first condition always succeeds or never succeeds. If it would always succeed, then path [0,2,1,4] would never be executed, making it an unfeasible path. This would be a violation of the test requirements for prime path coverage. Also, a false image of the coverage percentage is created. Redundant paths can never be found, will lower the coverage percentage drastically when actually the redundant paths have been checked. This is a problem that cannot be solved by allowing side trips or detours. If detours were used to solve this problem, there would be no way to distinguish between a branch with multiple conditions or nested branches.

Another problem with representing one branch as multiple paths, is the fact that another redundant path must be checked, slowing T2Extension down with every extra condition.

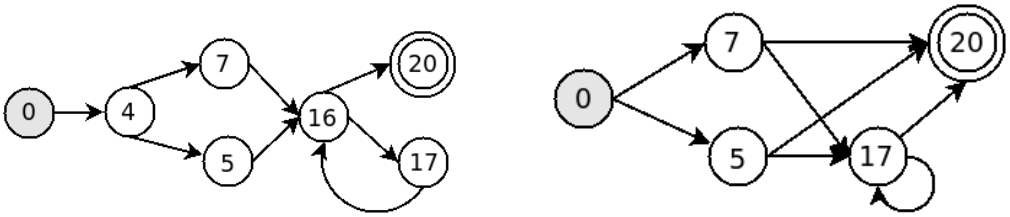
The solution to this problem is to make an assumption on java coding. Using BCEL it is possible to determine to which line of source code an instruction maps. BCEL uses a line numbers table as can be seen in figure 7.1 on line 9. Using Java coding rules, a programmer will almost always write a complete decision point on one line and the code inside the decision point's branch on the following line. One can deduct that nested decision points will never be written on the same line of code.

The algorithm that creates the control flow graph from an instruction list will group decision points. All branching instructions that are on the same line will be represented as one branching node in the CFG. It does this by checking if the branch instruction's left child is one the same line as the instruction itself. If this is the case, the next branch instruction will still belong to the same decision point. The consequence is that a programmer must remember to always write the code that belongs inside a branch on a different line than the conditions belonging to one decision point.

7.1.3 Cycles and Sidetrips

The algorithm for the construction of a control flow graph from section 6.1 optimises the CFG. When the CFG is optimised, the algorithm throws away nodes that are unnecessary. Unnecessary

nodes were defined in definition 6.1.1. A node that is not unnecessary is a branching node. One could wonder why this node is left in the graph. The optimised graph from figure 6.4 could be even more optimised if nodes 4 and 16 were also removed. As long as all other nodes still point to the correct children, no information is lost. The graph from figure 6.4 is shown in figure 7.6(a). If branching nodes would also be optimised, the graph would look like the graph in figure 7.6(b).



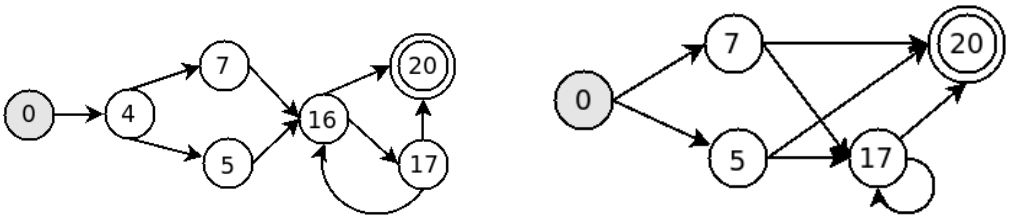
(a) Graph from figure 6.4 (b) Incorrect Optimisation

Figure 7.6: Incorrect graph optimisation

Unfortunately it is not possible to optimise the graph as shown in figure 7.6 due to cycles. The optimisation in figure 7.6(b) causes problems when checking for tours with sidetrips. It could be possible that, in practice, the loop [16,17,16] is always executed. This makes the path p_1 represented as [0,4,5,16,20] a semantically unreachable path. As explained in chapter 3, in order to adhere to the prime path coverage criterion, p_1 is considered as 'tested' if a test path t_1 exists that tours p_1 with sidetrips.

Definition 3.4.6 states that t_1 is a tour with sidetrips of p_1 if and only if the list of edges of p_1 can be obtained from the list of edges of t_1 by deleting elements in the list of edges of t_1. However, the optimisation in figure 7.6(b) does not allow for tours with sidetrips. In the non-optimised graph, t_1 could be represented as [0,4,5,16,17,16,20] which is a tour with sidetrips of p_1. In the optimised graph, t_1 would be represented as [0,5,17,20]. Its list of edges t_{e1} is represented as [(0,5),(5,17),(17,20)], while the list of edges p_{e1} for the optimised version of p_1 is represented as [(0,5),(5,20)]. It is not possible to obtain p_{e1} from t_{e1} according to definition 3.4.6. This shows that in the optimised graph as shown in figure 7.6(b) there is no way to find a tour with sidetrips of certain paths when in actuality they do exist.

Another problem that occurs when branching nodes are left out of the CFG is shown in figure 7.7. Figure 7.7(a) shows a graph where a loop can be escaped without having to complete all the code inside the loop. This is shown by the arrow from node 17 to node 20.



(a) Graph depicting a loop with an extra escape (b) Incorrect Optimisation

Figure 7.7: Incorrect graph optimisation

If branching nodes are left out, the graph from figure 7.7(a) would be optimised as the graph in figure 7.7(b). Note that this is exactly the same graph as in figure 7.6(b). A test path t_2 that escapes the loop in figure 7.7(a) is represented as [0,4,5,16,17,20]. Using the graph in figure

7.6(b), t_2 would be represented as [0,5,17,20]. A test path t_3 that does not escape the loop is represented as [0,4,5,16,17,16,20], the optimised representation is also [0,5,17,20]. The test paths t_2 and t_3 are very different paths, but the optimisation represents them as the same path making it impossible to distinguish which path has actually been walked. It is very important that both t_2 and t_3 are tested because both might hold errors which would otherwise remain undetected. An optimisation that discards branching nodes will not show which of the two paths has actually been executed, giving a false sense of security.

Since it is not always clear straightaway from the BCEL instruction list that the algorithm is dealing with a cycle, all branch instructions also get a call to a method inserted so that no information is lost.

An optimisation has been made in the context of cycles. The prime path calculation algorithm will find several representations of a cycle. For the cycle in figure 6.4, the algorithm will find two representations, namely [16,17,16] and [17,16,17]. The second representation is redundant and will never be found if the loop is only executed once. Also, the amount of representations of a cycle is the length of that cycle subtracted by 1. The result of this is that the larger the cycle, the more redundant representations are present. All these redundant prime paths lead to a larger search area to cover which will slow T2Extension down extensively. Because of all these disadvantages, the redundant prime paths are filtered. Each cycle will have one representation, namely the representation that is found if the loop is executed once.

It is possible to filter the prime paths of redundant paths by looking at the structure of the control flow graph. The correct representation of a loop is the representation that starts with the id with the lowest value. When the prime paths are filtered and a loop representation is encountered, it is analysed in order to check if it is a redundant representation or a correct representation.

Let l_1 be a loop representation holding the following id's; [2,3,4,1,2]. Because this loop contains four different nodes, there are four different loop representations. Namely:

[1,2,3,4,1]
[2,3,4,1,2]
[3,4,1,2,3]
[4,1,2,3,4]

The representation [1,2,3,4,1] is the correct one, all others are redundant. All representations can be computed from l_1. This is done, and the correct representation is added to the collection of prime paths, all others are ignored.

7.1.4 Code Insertion

The last subject to discuss in modelling paths is the code insertion itself. For each node in an optimised control flow graph, a call to a method is inserted directly before its corresponding instruction. As said before, the method that is called starts with 'aux'. This prefix has been chosen because a standard had to be chosen that T2 can ignore. These methods are empty, so T2 has no interest in them. If they are included in the methods to be tested, the auxiliary methods will pollute the testing scope and may wrongly influence the random generation of testing traces. Also, T2 would generate method calls for these auxiliary methods which would be unnecessary.

The auxiliary methods also contain the method name of the method to test, then two delimiters with the node id in between. The auxiliary method's name ends with five randomly generated numbers to ensure its absolute uniqueness. Looking at figure 7.2; the name of the first auxiliary method would be aux_calc_id5_id89345(). Due to overloading, this name for the auxiliary method might match multiple methods, so actually the method's argument types are also added to the name. Making the method name of this auxiliary method aux_calc_intint_id5_id89345 (), but to keep things clear, the method's arguments are left out here.

It is needed to be so specific when building a name for the inserted methods because it must be possible to connect them to the method that calls the auxiliary methods. This manner of naming ensures that the aspect will register the correct branch for the correct method. For example, if the aspect responds to a call to aux_calc_intint_id5_id89345 (), it knows that it must add node 5 to the path found so far for the method calc that takes two arguments of type **int**.

```
2  public int calc(int a, int b){
       aux_calc_id0_id23456();
4      int k = 0;
       aux_calc_id4_id45672();
6      if(a<b){
         aux_calc_id5_id78234();
8        a++;
       }
10     else{
         aux_calc_id7_id16548();
12       b++;
       }
14     k = a+b;
       aux_calc_id16_id45862();
16     for(int i=0;i<3;i++){
         aux_calc_id17_id23476();
18       k--;
       }
20     aux_calc_id20_id45234();
       return k;
22 }
```

Figure 7.8: Example complete path registration

Following everything discussed so far on modelling paths, when the calls to auxiliary methods are inserted, the method from figure 7.2 is transformed to the method in figure 7.8. As discussed before, branch instructions and jump instructions have targets. If these targets corresponds to an instruction that has a method call inserted in front of it, the targets must be redirected to the instruction that starts the method call. This redirection has to be done explicitly, but not by hand. BCEL can take care of replacing targets. Note that even though the control flow graph is always optimised, instructions are never removed from an instruction list because this would alter the behaviour of the method. Table 7.9 holds the instructions corresponding to the original method of figure 7.8, so the instructions for the method call insertions are not shown. The loop in this instruction list starts with instruction 14 and ends with instruction 19. The instructions that will call the auxiliary method aux_calc_id16_id45862() will be inserted just before instruction 14 and instruction 19 will point to the first instruction of the call to this auxiliary method.

The instructions to be inserted in front of the instruction corresponding to node 4 cannot just be inserted immediately before instruction 4. If instructions were inserted there, a runtime error would be produced because instruction 4 is part of a larger whole that covers the complete `if` statement. Instruction 2, *LOAD a*, is the first instruction for the if statement on line 3, this ca be deducted from the fact that it is the first instruction on line 3. The instructions to be inserted will be inserted just before instruction 2 instead of instruction 4.

The modelling of paths within a method is now complete and T2Extension can continue with the aspect template.

Instruction ID	Instruction	Line of Code
0	CONST 0	2
1	STORE k	2
2	LOAD a	3
3	LOAD b	3
4	IFCOMP → INSTR 7	3
5	INC a	4
6	GOTO → INSTR 8	4
7	INC b	6
8	LOAD a	7
9	LOAD b	7
10	ADD	7
11	STORE k	7
12	CONST 0	8
13	STORE i	8
14	LOAD i	8
15	CONST 3	8
16	IFCOMP → INSTR 20	8
17	SUBT k	9
18	INC i	8
19	GOTO → INSTR 14	8
20	LOAD k	10
21	RETURN k	10

Figure 7.9: Instruction list

7.2 The Administration of Paths

In order to register found paths, the CUT is woven with an aspect. This aspect then monitors the class and registers each path it finds. In order to register paths, the aspect will monitor all methods, also the auxiliary methods. In this section, when an *original method* (OM) is mentioned, it refers to a method declared in the CUT which was not created by BCEL. An *auxiliary method* (AM) is a method created by BCEL and is used to follow paths. The CUT will be referred to as *C*.

Statically, T2Extension cannot know which class will be tested, making it impossible to statically create an aspect for *C*. However, pointcuts in an aspect can be defined to monitor any method in any class. Section 5.2 explained that this is not a good solution due to overmonitoring. Another problem with such broad pointcuts is that it would be difficult to distinguish between an original method and an auxiliary method. The solution is to create a template aspect that is filled at runtime when all the needed information is available.

Needed information is the class name and package it resides in and the names of the OM. While the control flow graph is built, this information is saved in an XML file holding all the information on *C*, including arguments that must be passed to methods. These are necessary due to overloading. Defining arguments in a pointcut ensures the uniqueness of the method called. The names of the AM are not required because these can be generalised. The decision has been made that an AM's name starts with 'aux_' followed by the method's name and the method's arguments. This is enough to build up the template because the behaviour for all auxiliary methods belonging to a certain OM will be the same.

7.2.1 The Template

Information that is statically known is how each advice in the aspect should behave. A tour of an OM's control flow graph is extracted when the OM is executed. Within the OM's execution, some of its AMs will also be executed. The tour will be composed out of the executed auxiliary methods and is represented as a list of id's, where the id's represent the nodes in the OM's graph.

Before an OM is executed, the list should be initialised. When an AM is called that belongs to this particular OM, the id from the AM's name is extracted and added to the list. After the OM has been executed, the list is examined and compared to the prime paths stored for the OM. If the test path contains certain prime paths, they are marked as *found*.

```
package t2ext.weaver;

import java.util.List;
import t2ext.weaver.utils.WeaverMethods;

public aspect Weaver {
  private WeaverMethods _auxMethods;

  pointcut initialisation() :
    initialization(<packageName.ClassName>.new(..));

  before() : initialisation(){
    if (_auxMethods == null)
      _auxMethods = new WeaverMethods();
  }

  pointcut weave<uniqueID>() :
    execution(* <packageName.ClassName.methodName(arguments)>);

  before() : weave<uniqueID>() {
     if (_auxMethods == null)
       _auxMethods = new WeaverMethods();
     _auxMethods.startPathList("<auxMethodName>");
   }

   after() : weave<uniqueID>() {
     List<Integer> list = _auxMethods.getCompletePathList("<auxMethodName>");
     if(list!=null)
       WeaverMethods.findPath(thisJoinPoint.toLongString(),list);
   }

  pointcut weave<uniqueID>() :
    execution(* <packageName.ClassName.auxMethodName>*());

   before() : weave<uniqueID>() {
     int id = WeaverMethods.getIdFromMethodName(thisJoinPoint.toLongString());
     if(id!=(-1))
       _auxMethods.addToList("<auxMethodName>",id);
   }
}
```

Figure 7.10: Aspect template

With the information that is statically known, the template is set up. Figure 7.10 shows the template aspect class, its package and its imports. All the text between < and > are names or objects that will be replaced at runtime. A field _auxMethods is declared, this object holds all the methods to register and find prime paths. These methods are the same for all original

methods in the CUT. The object _auxMethods is discussed later. The first declared pointcut is the *initialisation* pointcut. It responds when an object is created, hence the name. The part of the pointcut that reads **initialization**(packageName.ClassName.new(..)) is called a joinpoint. The code on lines 12 to 15 is an advice. The advice states that exactly before C is initialised, _auxMethods is initialised. It is recommended to declare all pointcuts at the top of the aspect, but here pointcuts and their advice are grouped so that the reader has an idea of which pointcut is linked to which advice.

The pointcut on line 17 is a pointcut that is generated for each original method. It must have a unique name; at runtime the part of the name that reads uniqueID is replaced with a number. This pointcut reacts when an OM is executed. There are two pieces of advice for this pointcut, before the OM is executed and after it has finished. The before advice starts on line 20 and checks if the _auxMethods object has been initialised, this is necessary because if the method is static, the initialisation advice will not have been executed. Before an OM is executed, a list is initialised for the execution of this method.

The pointcut on line 32 monitors the auxiliary methods. The * indicates that the method name auxMethodName can have any suffix, which in this case will be the AM's id and a random 5 digit number. The * makes it possible to declare this pointcut once for each OM and not for all the auxiliary methods present. The pointcut on line 32 is linked to before advice. Before an AM is executed, its id is extracted from the method name and added to the list which was initialised in the before advice of the OM.

When an OM has been executed, the complete test path has been created and it can be used to check which prime paths have been found. This is exactly what the after advice on line 26 does.

The template is adapted to C and then compiled. The class produced from the aspect is then woven with C.

7.2.2 Aspect Auxiliary Methods

The object named _auxMethods in figure 7.10 holds the methods the aspect needs for the registration of paths, these are not written in the template itself because they do not need any modifications at runtime. The object _auxMethods holds several interesting methods that will be discussed here.

The first method is the method that is called in the before advice on line 23 in figure 7.10, named startPathList. This method initialises the test path for a particular OM. Meaning that it stores the test path in a Hashtable with the method name as its key. Unfortunately it is not that simple due to recursion. It is possible for a method to call itself. If a test path T_1 is initialised before a method m_1 is started and halfway through m_1 calls itself again, T_1 will not be finished for the original call of m_1, but it will be initialised again for the second call of m_1. In other words, T_1 is emptied before it is finished. The solution is to store T_1 in a Stack. A stack works according to the principle of *last-in-first-out*. Every time m_1 is called, a new test path is put onto the stack. When an auxiliary method has been called, it adds its id to the list on the top of the stack. When the OM has been executed, the test path is popped from the stack and analysed. This ensures that all test paths are stored.

The most interesting method inside _auxMethods is the method named findPath which is called on line 29 in figure 7.10. This method takes the joinPoint's name and the test path as its arguments. It extracts the method name for the OM from the joinpoint's name. Remember that all prime paths were stored in an XML file and they are stored per method. The method findPath reads the XML file holding the prime paths and finds the corresponding method and its paths. If a method has no branches, it has just one path and by executing that method that path has been found, so it immediately checks the method's path as found. If a method has more than one path, each path is compared in its entirety to the test path to see if it is present. A test path t might be longer than the prime path p, so t and p are not compared as equals, but

a check is done to see if p is a subpath (definition 3.3.2) of t. If the path is present, it is again checked as found.

If p is not present in t, T2Extension checks if t is a tour with sidetrips of p according to the algorithm described in section 6.3. If this is the case, then p is marked as *found with sidetrips*. This does not mean that it is not checked again later. It is safer to find a path as is, than having found it with sidetrips. During the entire execution of T2Extension it will still be checked and effort will be taken to find it. If p is not found when all options have been exhausted, only then do we consider it found because p was found with sidetrips. The reason that it is checked at such an early stage is because it saves runtime. If all of T2's traces have to be executed again for paths that have not been found yet to discover if they would have been found with sidetrips, T2Extension would become a great deal slower.

7.2.3 Optimisation

T2Extension performs an exhaustive search when trying to find a certain path. This search can be quite slow, especially when there are many paths to find. If a method is executing it might be desirable to check during this execution if the method is on-track with respect to the path that is looked for. If this is not the case, it might be possible to end the method as soon as the discovery is made, saving runtime and possibly making T2Extension much faster.

This optimisation is implemented, however it is not a default optimisation, it has to be switched on by the user. The idea is to know which prime path p the program is searching for. When an AM is called, a check is made to see if p is still being followed. For example, p is [0,1,2,3,4,5]. When an AM is called, the test path t so far might be [0,1,2,3]. This will let the OM continue its execution because t is still on track with p. If t would be extended with 6, making t [0,1,2,3,6], t is off track and will probably not contain p. This leads to the conclusion that there is no point in finishing the execution of the OM because it will not find p anyway and the OM is cut short and not continued. There is of course a danger here that p is actually present but not found because the method is cut short. An example of this is a recursive method. The path T2Extension is trying to find might only be found in the second or third recursive call. If a method is cut short in its first call because the method is not on track, the path will never be found while actually it could have been found. Therefore this is an optimisation that has to be switched on by the user of T2Extension.

The initial idea was to let an OM end with a special exception if t is not on track. This would complicate the collaboration between T2Extension and T2 because T2 sees every exception as an error and will report this. An optimisation should of course not be reported. A solution would be to alter T2 to ignore this special exception. However, T2 should not be altered in any way if possible as stated in the requirements in chapter 2.

A better idea is to again work with AspectJ instead of working around it. AspectJ has a special piece of advice called **around**. Around advice can modify the execution of a method, it can replace code within a method, or it can even bypass code within a method. We want to bypass the rest of the OM if it is not following the correct path. Figure 7.11 shows what around advice looks like.

If the user has turned the optimisation on, the pointcut on line 1 is created for each OM. This pointcut ensures that the around advice will only respond to method calls done within the code of the OM, so it will only respond to method calls to the OM's auxiliary methods. The around advice itself is a special kind of advice and must always return something. Here, the object corresponding to C is returned. The object is first gathered from the current joinpoint, this ensures that the object's state is correct. It is shown on line 5. Line 6 checks if this particular OM should continue to execute. If it should continue, the object is modified by calling **proceed**(). This method will let the OM execute as is. This step is skipped if the OM should not continue and the object is returned as is. The result of this is that the program will immediately jump to the after advice for this particular OM.

```
pointcut aroundMethod<uniqueID>() :
  withincode(* <packageName.ClassName.methodName(arguments)>);

Object around() : aroundMethod<uniqueID>() {
  Object obj = thisJoinPoint.getTarget();
  if(_auxMethods.findPath("<packageName.ClassName.methodName(arguments)>"))
    obj = proceed();
  return obj;
}
```

Figure 7.11: Around advice

For this optimisation to work, the advice in figure 7.10 has to be altered slightly. The 'before' advice for the OM will store which path T2Extension is looking for. The advice for the auxiliary methods will check if this path is still being followed, if this is not the case, a boolean is set to *false* for this particular OM. The around advice then checks if the boolean is set to false for this particular OM, which is shown in figure 7.11 on line 6. The after advice also uses this boolean. If it is set to true, then it will compare the prime paths to the found test path, otherwise it does nothing.

The around advice is a beautiful part of AspectJ, but should be used with caution since it can alter program behaviour. There is a possibility that this optimisation lowers the prime path coverage (due to the issue with recursion as discussed before) which is something the user of T2Extension should keep in mind.

7.3 Trace Execution

This section describes the last step in improving prime path coverage for T2. Traces are first modified and then executed. The manipulation of traces is done in three phases. The first phase is to run T2. The second phase is the analysis of the traces T2 produces, the last phase is the actual manipulation of the traces.

The user of T2Extension can pass any argument that is allowed by T2 to T2Extension and T2Extension will take them into account. When T2 is run, arguments given by the user are passed to it and T2Extension adds another. It needs the traces T2 produces for the manipulation, so T2Extension passes an argument to T2, asking it to save all the traces it produces. The CUT has been modified by BCEL and has been woven with the aspect. The new CUT is first tested once using T2. Since the class is woven, found paths are registered. When T2 has finished its run, a check is done to see if all paths were found. If the coverage is 100%, T2Extension reports this and stops. If the coverage is below 100%, T2Extension continues with the analysis of traces.

Figure 7.12 gives a general idea of the control flow of the manipulation of traces. The following sections will delve deeper into different components in order to reveal more details. The diagram shows that T2 is run once followed by the analysis of traces. When the analysis is completed, a base domain is set for values used in traces. Then, a collection of methods with unfound paths is created, for each of these methods a list of traces is produced that is adapted to that method. Each unfound path inside the method is compared to a prime path that has been found and this information is used to execute and manipulate traces. There are four different manners of executing traces, if one of these manners finds the path that is searched for, then we can continue with the next unfound path. The above steps are then repeated for this new path. When all paths for a certain method have been processed, the next method from the collection is taken and the steps are repeated from that point. When all methods have been processed but there are still path left unfound, the base domain is changed. So the control flow repeats itself until it has tried several different options which in the case of T2Extension is seen as a full

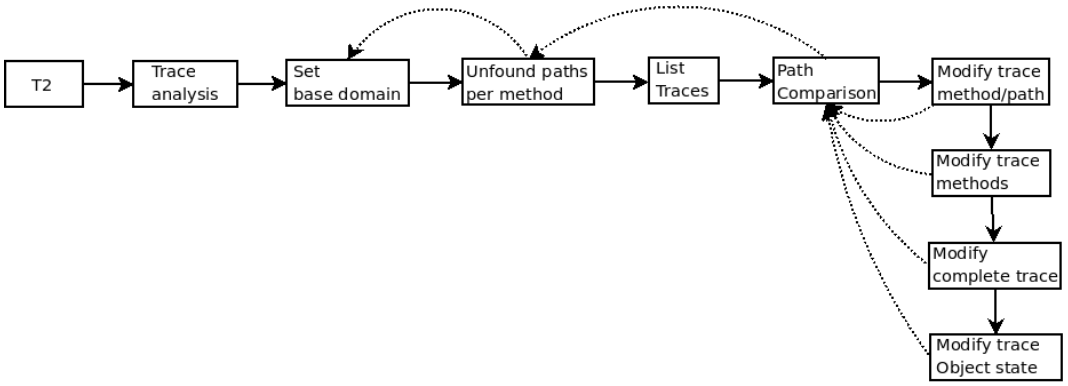


Figure 7.12: Flow of trace manipulation

exhaustive search.

7.3.1 Traces

Before a description of the analysis of traces is given, the trace internals must be discussed. T2 saves the traces in an object TrFile which is written to a file. This file can be loaded and will hold the traces in a LinkedList. A trace is a test sequence and one step in the sequence is called a TraceStep, so a trace is actually a LinkedList of steps. One step either calls a method to test it, or it updates one of the fields of the target object. The values in a certain step have a meta representation called MkValStep. For a method call this means that there is a MkValStep for the method itself, but also for each of its arguments. To clarify, a TraceStep is a step in the test sequence and a MkValStep is the value representation of the step in the test sequence. A diagram showing the relation between all these objects is shown in figure 7.13.

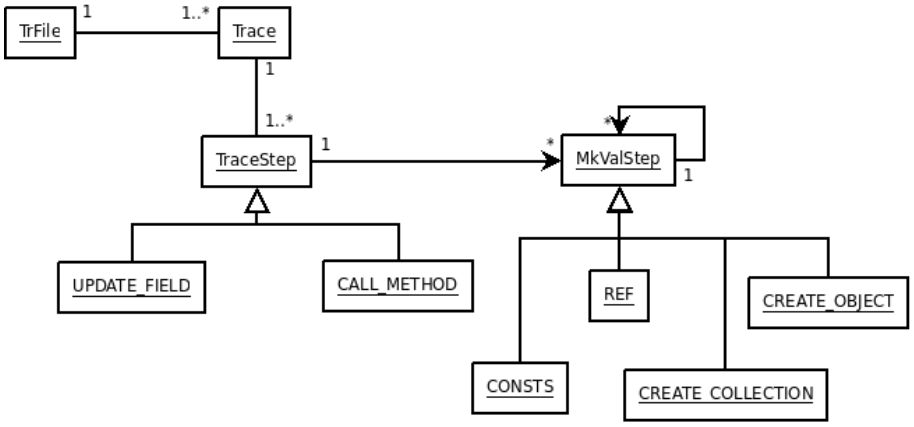


Figure 7.13: Trace UML for T2

When traces are executed, their steps are actually executed one after the other. When a step needs a value, for example an argument to a method, two possibilities exist. T2 has a pool of object from which it can get a value. The pool of objects is created by T2 and its objects can be reused. A value obtained from this pool will become a REF, otherwise the value is created and becomes one of the other children of a MkValStep.

To execute a step, quite some information is needed. Part of this information can be found in the trace or the TrFile. Other information can be found in the BaseEngine. The BaseEngine randomly generates and executes traces. When T2Extension executes traces, it needs this

BaseEngine for the extra information, but also for the creation of values. The user can set certain probabilities in the BaseEngine, for example the probability with which a value is taken from the pool of objects.

After T2 has finished its run, T2Extension will create a new BaseEngine with the parameters set exactly as in the run of T2. When the BaseEngine has been created, the second phase of improving path coverage can be started; the analysis of traces.

7.3.2 Trace Analysis

One approach to improving prime path coverage would be to randomly modify the traces generated by T2 and then see if the coverage increases. This is quite a simple approach and it does not yield enough improvement. Due to BCEL and AOP, T2Extension has an abundance of information and this can be used to pick out traces that might yield a better result in finding a certain path. In order to make a selection of traces, they must be analysed.

The collection of traces is sorted according to which methods they call and which paths they execute. A trace will call a certain method multiple times with different input values. This leads to the conclusion that different steps in a trace might call the same method, but they might execute different paths.

A step holds information about which method it calls with which input values. However, an examination of the TraceStep object will of course not reveal which path has been executed since this is not a functionality in T2. This means that for this analysis, all steps within all traces must be executed.

Each trace has its own TraceAnalysis object (see chapter 2) that stores the trace, information about the object pool and the state of the target object. The target object is the object representation of the CUT. It also holds a list of TraceStepAnalysis which holds the analysis information about one step in a trace.

A step either calls a method or updates a field. If it updates a field, the object TraceStepAnalysis stores a list of fields that were updated so far in this trace. Why the list of fields is stored, is explained in section 7.3.3. If the step calls a method, the method that was called is stored and the paths that were found in this step are saved.

Specifications need some extra information. It is possible to write a specification method *spec* for a certain method m that holds assertions about that method. The idea behind this is that *spec* calls m and then checks if m executed according to certain specifications. Or it might check if the target object is in the correct state before m is executed. When a method has a specification method, T2 never calls the method itself, but always via the specification method. The consequence is that a trace will never contain a step executing m, only steps executing *spec*. When analysing a step that calls *spec*, the paths belonging to *spec* must be saved, but also the paths belonging to m. Later, when a trace is modified to find certain paths for m, the program must search for steps calling *spec*.

The sorted traces are stored in a table. Actually, the sorted TraceAnalysis objects are stored in table. Each method is mapped to a list of these objects that call this method. Each TraceAnalysis object maps methods to their paths. This makes it relatively cheap to sort traces according to T2Extension's needs. When T2Extension is trying to find a certain path, it will obtain the collection of traces from the table that call the method that the path belongs to. From this collection, it will only manipulate and execute the traces that adhere to the criteria of finding this path.

The traces are now ready to be manipulated. Each trace has been executed once by executing all of its steps. At each step as much information as possible is stored. Some handy tricks are applied to make the search for a trace easier. The manipulation of traces relies on the analysis that has been explained in this section.

7.3.3 Trace Manipulation

Traces are modified and executed in order to improve prime path coverage. It is the execution of these modified traces that will yield the result. The execution of the traces is done in several steps and in several different manners to ensure that an exhaustive search is done for all paths that have not been found after the first run of T2.

The first option for an exhaustive search is to set the amount of times the trace manipulation algorithm is executed. By default, T2Extension will execute the algorithm just once because this yields good results and executing the algorithm multiple times does not always add to the result which would slow T2Extension down unnecessarily. This is why the amount of times the algorithm is executed can be set by the user because sometimes it may result in finding a few more paths.

The second part of the exhaustive search is the changing of domains for constants. T2 uses a base domain for constant values. To be specific, if a value is needed and it is not taken from the pool of objects, if it is a primitive type, enumeration type, or a string, it is taken from the base domain. The default base domain offers constants in a certain range. For example, if an **int** is requested, the default base domain will return a random integer value between -3 and 3. One can imagine that tweaking this base domain could result in finding more paths. It is possible for the user of T2 to specify their own base domain, but the existing domains offered by T2 can also be tweaked. T2Extension uses the existing domains and alters them. T2Extension will first try to find all paths by using the default domain. If this yields no result, then the default domain is extended with a larger range of integers and the search for paths is done again. If this will still not yield any results, T2Extension uses another base domain that has a different range for all constant types. It would also be possible to create one large custom base domain as is suggested in section 10.1, but for this thesis the combination of three different domains is used for two reasons. The first reason is that when testing T2Extension, one does want to work with existing standards. However, out of the three base domains provided by T2, there was not one that would always yield better results than the others. Different classes expect different input values and all three domain cover different values. Therefore the combination of the three was chosen, which works exceptionally well. Another reason is that the use of the three domains showed us during testing which class responded best to which domain. This information can later be used to create a custom base domain.

The traces are modified by changing the values passed to the steps. So the order of the steps is not altered and no new steps are created. All values are randomly created. This means that when a method is called, new values are created for its arguments and when a field is updated, a new value is created for this field.

The concept behind the trace modification is that by altering the state of the target object or by passing different arguments to a method, paths that remained unfound before will be found by the modified trace. A trace is only modified if it adheres to certain criteria. This results in a search that is controlled by the analysis done before. A controlled search should be more efficient and yield better results than a random search. The criteria to which a trace must adhere are explained in the paragraphs below.

The actual trace modification algorithm works per method. Each method that has paths that have not been found yet, is examined. Let us call the collection of unfound paths in a certain method $C_{unfound}$ and the collection of found paths in the same method C_{found}. Each path in $C_{unfound}$ is checked against each path in C_{found} with the path comparison algorithm described in section 6.4. This algorithm returns a metric that states how alike two paths are. The path p_{found} from C_{found} that is most similar to the path $p_{unfound}$ from $C_{unfound}$, is linked to $p_{unfound}$ in a table. The idea behind this comparison is that if a trace is modified that contains a step that found p_{found}, there is a better chance of finding $p_{unfound}$ when altering the step that found p_{found} than if steps were modified that found other paths which are less similar to $p_{unfound}$. Another advantage is that the trace will probably not need to be modified much in

order to find $p_{unfound}$.

```
  private void modifyTraces(String methodName, TraceAnalyser analyser,
2         Hashtable<Integer, Integer> bestPaths) {
    //get collection of traces that call this method
4   List<TraceAnalysis> analyserList =
          analyser.getAnalysisForMethod(methodName);
6   Enumeration<Integer> unfoundPaths = bestPaths.keys();
    //find paths as long as there are unfound paths
8   while (unfoundPaths.hasMoreElements()) {
      Integer id = unfoundPaths.nextElement(); //id of path to find
10    Integer bestId = bestPaths.get(id); //id of most similar path
      //if the path has not been found in previous iterations
12    if (!foundPath(id, methodName)) {
        //manipulate traces that execute the most similar path
14      boolean found =
          execBestPathForMethod(name, analyserList, bestId, id);
16      if (!found)
          //manipulate traces that execute this method
18        found = execAllPathsForMethod(name, analyserList, id);
        if (!found)
20        //manipulate the complete trace
          found = execCompleteTraceForMethod(analyserList, id);
22      if (!found)
          //alter the state of the target object
24        found = execTraceWithFieldUpdates(analyserList, name);
      }
26  }
  }
```

Figure 7.14: Modifying traces

When it is determined which path is most similar to the unfound path $p_{unfound}$, the traces can be modified. A simplified version of the method that alters the traces is shown in figure 7.14. This method takes three arguments; the first one is the name of the method that the algorithm is searching for. The second argument is the analyser that holds all analysed traces sorted per method. The Hashtable argument maps the paths from $C_{unfound}$ to the paths that are most similar from C_{found}. If the user has the optimisation from section 7.2.3 switched on, this method will also register which path it is looking for so that the aspect can respond accordingly.

Line 3 in figure 7.14 shows that only the collection C_{trace} of traces that call the method worked with at this time are used. For each path $p_{unfound}$ in $C_{unfound}$ (line 8), the algorithm will try to find it using four tactics. Each subsequent tatic is more coarse grained than the previous one. Notice that on line 12 a check is done to see if $p_{unfound}$ has really not been found. This check is performed because there is a very real possibility that the path might have accidentally been found when the algorithm was actually looking for another path.

As explained before, the concept behind this manipulation is that the chance of finding $p_{unfound}$ is (assumed to be) highest when a trace that calls a path p_{found} most similar to $p_{unfound}$ is executed. This is the first tactic shown in lines 14 and 15. The method execBestPathForMethod called on line 15 reduces the collection C_{trace} to a collection of traces that call p_{found}. The method is shown in detail (although simplified) in figure 7.15. Each trace in this collection is completely executed. Steps within these traces that call other methods or update fields are left unmodified before their execution (line 20). Steps that call the method that $p_{unfound}$ belongs to, are only modified if this particular step finds p_{found}. These steps are modified by randomly creating new arguments for the method call, only after the step has been modified, it is executed (line 18).

When this first tactic does not find $p_{unfound}$, it may mean that even though p_{found} is very

```
  private boolean execBestPathForMethod(String name,
2       List<TraceAnalysis> analyserList,
        Integer bestId, Integer idToFind){
4   List<TraceAnalysis> reducedList = reduceTraces(analyserList, bestId);
    boolean foundPath = false;
6   for (Iterator<TraceAnalysis> iterator = reducedList.iterator();
      iterator.hasNext();) {
8       TraceAnalysis traceAnalysis = iterator.next();
        List<TraceStepAnalysis> steps = traceAnalysis.getSteps();
10      for (Iterator<TraceStepAnalysis> iterator2 = steps.iterator();
          iterator2.hasNext();) {
12          TraceStepAnalysis step = iterator2.next();
            //if step executes the path with id bestId
14          //modify it
            if(step.isCallMethod()
16              && step.getMethodName.equals(name)
                && step.foundPaths.contains(bestId))
18            foundPath = modifyAndExecuteStep(step, idToFind);
            else
20            foundPath = executeStep(step, idToFind);
        }
22      if(foundPath)
          break;
24  }
    return foundPath;
26 }
```

Figure 7.15: First tactic: modify method calls that execute a certain path

similar, it is not possible to find $p_{unfound}$ based on p_{found}. So the second tactic is tried, which is the method call on line 18 in figure 7.14. This strategy will take C_{trace} and modify all traces in this collection. The steps within these traces that call the method that $p_{unfound}$ belongs to are modified by providing new arguments to their method calls. All other steps are executed unchanged. The method execAllPathsForMethod is shown simplified in figure 7.16. This figure is different from the method shown in figure 7.15 in two ways. First of all, the collection C_{trace} is not reduced and the condition on which a step should be modified is different, which can be seen on lines 13 and 14 in figure 7.16.

If this strategy does not yield any result, a coarser strategy is needed. The third step in trying to find $p_{unfound}$ is to completely modify all traces in C_{trace}. This tactic is shown on line 21 in figure 7.14. To be more precise, all steps that call a method are modified before they are executed. Steps that update fields are left untouched before their execution.

When all these tactics are to no avail, it indicates that manipulating method calls has no influence on this path. This could indicate that the method in question can only be altered by changing the state of the target object. This last tactic is shown on line 24 in figure 7.14. The state of the target object is altered by updating its fields. This last tactic works as follows. In order to try and find $p_{unfound}$, certain fields are updated just before a step is executed that finds p_{found}. The idea behind this field update is that the method might use some of the target object's fields. A change in these fields might lead to another path being chosen and hopefully this causes $p_{unfound}$ to be found.

Remember that in the analysis phase, each step stores a list of field that were updated so far. This list is compared to the complete list of fields declared in the target object. The fields that were not altered during the execution of the trace the algorithm is working with at the moment, are the fields that are updated. So, before each step that finds p_{found}, some fields are altered. Not all the fields are altered because changing all of them would probably be redundant.

```
private boolean execAllPathsForMethod(String name,
      List<TraceAnalysis> analyserList,
      Integer idToFind){
  boolean foundPath = false;
  for (Iterator<TraceAnalysis> iterator = analyserList.iterator();
    iterator.hasNext();) {
      TraceAnalysis traceAnalysis = iterator.next();
      List<TraceStepAnalysis> steps = traceAnalysis.getSteps();
      for (Iterator<TraceStepAnalysis> iterator2 = steps.iterator();
        iterator2.hasNext();) {
          TraceStepAnalysis step = iterator2.next();
          //if step executes the method 'name'
          if(step.isCallMethod()
              && step.getMethodName.equals(name))
            foundPath = modifyAndExecuteStep(step, idToFind);
          else
            foundPath = executeStep(step, idToFind);
      }
      if(foundPath)
        break;
  }
  return foundPath;
}
```

Figure 7.16: Second tactic: modify steps that call a certain method

If all these tactics do not result in finding $p_{unfound}$, then it might have been found with sidetrips. If this is not the case, it is registered as 'not found'.

The method shown in figure 7.14 does not show how methods are handled that are connected to specification methods. As said before, these methods do not have their own trace step. It is handled by checking if a method m has a specification method m_{spec}. If this is the case, the algorithm continues with m_{spec}, knowing that it is actually looking for paths inside m. This is possible because m_{spec} takes the same arguments as m. To clarify, the algorithm will look for steps that call m_{spec}, but the analysis had also stored the paths each step finds for m, so the algorithm will use these paths instead of the paths inside m_{spec} and will modify the arguments passed to m_{spec}.

If m_{spec} itself has paths that have not been found yet, it is also handled as a normal method. That is to say, m_{spec} will be called whenever m should be investigated, but it is also called when m_{spec} itself needs to be investigated. The list of methods with unfound paths will contain m_{spec} and it will be executed separately from m in order to find its own paths.

The possibility exists that a modified step will result in an error. T2 has reporters that can print the complete exception on the screen but T2Extension cannot access these. T2Extension will report that a faulty trace has been found and it will save this trace. If the user wants to see the complete error message, the user can rerun T2 with the regression option switched on.

The prime path coverage is measured after T2Extension has finished its exhaustive search. It looks at all prime paths, so not per method any more and counts which have been found and which have not. If a path has not been found, T2Extension checks if it has been found with sidetrips. Remember that the aspect also checks if a path would be found if sidetrips were allowed. The percentage of found paths is reported on the user's screen.

The algorithm for the modification of traces is fairly simple but is quite heavy in runtime due to all the traces being examined and then executed multiple times. The complete extension is definitely slower that T2 on its own but if one realises how much it does for the administration of branches and how many traces it modifies and executes time after time, the relatively slow process is not so disturbing.

Chapter 8

Test Results

This chapter reveals the results for T2Extension. The goal for T2Extension is to improve the prime path coverage of T2. The test results will be demonstrated with some example Java classes. The examples have been selected because their methods contain interesting branches. This chapter will study methods containing multiple loops and nested loops, methods containing branches in several combinations and the bubble sort algorithm. Finally, a real world example is shown.

The test results will show that T2Extension yields very good results and will always improve T2's coverage if this is possible. The improvement is substantial, considering that T2Extension also generates values in a random fashion.

It is often said that testing tools with random searches can only be used as a complement to other testing tools due to their low coverage. This chapter will prove that T2Extension makes T2 a testing tool that can be used standalone and that T2's users can place confidence in teh results provided by T2.

8.1 Detailed Test Run

The first example is very detailed. It is used to show how T2Extension calculates its results. This example shows a method containing two loops. Both loops contain several branches and the first loop has a nested loop. The code is shown in figure 8.1. The method itself does not do very much, but it is interesting to see how T2 and T2Extension handle it.

Figure 8.2 shows the control flow graph, as created by T2Extension. Note that the CFG is based on bytecode instructions, but it is difficult to deduct which node corresponds to what part of the source code. This is why source code and line numbers were added to figure 8.2.

The graph depicted in figure 8.2 has 16 nodes and 31 prime paths. The graph actually has 48 prime paths, but redundant cycles were filtered. However, the amount of paths is quite large for such a small method. It has such a large amount of paths due to the many loops that it contains. From the fact that there are many cycles, one can deduct that many path will be found by tours allowing sidetrips.

When the control flow graph is created and the prime paths are calculated for the code in figure 8.1, the template aspect is adapted to the CUT and woven with it. After the weaving, the CUT is ready to be tested for the first time by T2.

When the code is tested, T2 finds that there are no violations; the code has been approved by T2. Figure 8.3 shows the results relevant to T2Extension. T2 finds 10 paths out of the 31 on its own. This is very little, just 32%, but it is justifiable. As said before, many paths will need tours with sidetrips to be found, T2 will first run without taking sidetrips into account. When T2 has terminated, a check is done to see how many paths have been found when sidetrips are allowed. Figure 8.3 shows the results allowing sidetrips. As expected, many paths needed

```
    public int calc(int a, int b) {
2     for (int i = 0; i < 10; i++) {
        b = b + i;
4       if (b > 2)
          b++;
6       else
          b--;
8       a++;
        for (int m = 0; m < 10; m++)
10        b += 2;
      }
12    for (int j = 0; j < 10; j++) {
        a++;
14      if (j < 2)
          b+=a;
16      else
          b-=a;
18    }
      return b;
20  }
```

Figure 8.1: Example with loops

sidetrips; 17. In total T2 found 87% of the paths. In other words; T2 has a prime path coverage of 87% for the code depicted in figure 8.1.

A coverage percentage of 87% is quite reasonable. It shows that T2, despite of its random properties, will still covers most of the code. Nevertheless, a score above 90% would be much better. The next phase is therefore the analysis and modification of traces.

Figure 8.4 shows the results achieved by T2Extension. The first four lines show the results when sidetrips are not allowed. T2 found 32% of the paths in this case. T2Extension does quite a bit better, finding 39% of the paths. T2Extension has found 2 paths more that T2.

The bottom part of figure 8.4 depicts the results when sidetrips are allowed. This time T2Extension has found 18 sidetrips. The difference between the amount of sidetrips T2 found and the sidetrips T2Extension found can be explained by the amount of generated tours. Apparently, T2Extension was able to generate more tours and these tours satisfied 1 more path.

In conclusion, T2Extension has a 97% coverage for the code from figure 8.1, versus 87% achieved by T2. This is a great result. Nevertheless, there is one path that T2Extension did not find. This is path [37,30,31,34,35] which corresponds to the second loop that starts on line 12. This is an example of a semantically unreachable path. The **if** branch on line 14 is only entered the first two times that the loop is executed. All other times, the **else** branch is chosen. This makes it semantically impossible to find a path that will first pass through the **else** branch and that will pass through the **if** branch in the next execution of the loop, which is what the unfound path depicts. This is extremely difficult to detect automatically, for this reason it is left as an unfound path.

Considering that there is no possibility to execute path [37,30,31,34,35], T2Extension has checked every single possible path inside the method and has found no violations. Due to T2Extension, the method calc has been thoroughly tested.

8.2 Results

This section will use six classes to show T2Extension's behaviour. The class DoubleLoopTest is the class used in the previous section. The code for all the other classes can be found in appendix B. All tests have been performed ten times in order to get a good reading of T2 and T2Extension's

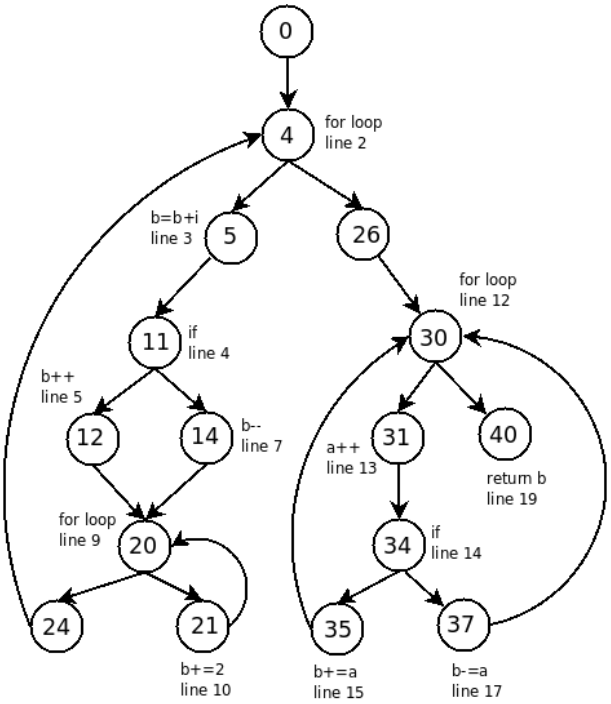


Figure 8.2: CFG for fig. 8.1

```
sidetrips: 17
amount of paths found:27
amount of paths not found: 4
foundPercentage T2: 87%
```

Figure 8.3: Prime path coverage T2 for fig. 8.1

results.

Table 8.5 shows T2's prime path coverage without the help of T2Extension. The second column shows T2's prime path coverage when sidetrips are not allowed. The third column shows T2's coverage when sidetrips are allowed. The column named 'Total Prime Paths' shows the total amount of prime paths for each class. The column named 'Unfeasible Prime Paths' shows the paths that are semantically unreachable, e.g. in the case of DoubleLoopTest there are 19 unfeasible paths, however, 18 of these paths can be found by allowing sidetrips. Only one path is absolutely unfeasible.

Table 8.5 shows that T2's prime path coverage is at best reasonable, but it also shows that T2's coverage is quite poor for ElseIfBranchTest and Rot13. Additionally, the table shows how fast T2 is, it finishes in a matter of seconds. The results for BinarySearchTree were the only results that varied with an average result of 68%, or, on average 26 out of 38 prime paths were found. The results for the class BubbleSort show that it is possible for T2 to achieve a high coverage.

Table 8.6 shows the results for T2Extension. This table shows that, overall, T2Extension achieves a good prime path coverage percentage.

When comparing the results in table 8.6 to the results in table 8.5, one can conclude that T2Extension considerably improves coverage results for T2. For the class ElseIfBranchTest, T2Extension improves coverage on average by 52%. It may take some more runtime than a run of T2, but the results show that this extra time is worth the wait. T2Extension varied in its results for this class, the deviations for the test runs can be seen in figure 8.7. However,

```
amount of paths found: 12
amount of paths not found: 19
foundPercentage: 39%
T2 percentage: 32%
ALLOW SIDETRIPS
sidetrips: 18
amount of paths found: 30
amount of paths not found: 1
foundPercentage: 97%
T2 percentage: 87%
```

Figure 8.4: Prime path coverage T2Extension for fig. 8.1

Class Name	**T2 No Sidetrips**	**T2 With Sidetrips**	**Total Prime Paths**	**Unfeasible Prime Paths**	**Runtime (Seconds)**
DoubleLoopTest	10 32%	27 87%	31	1 ST: 18	6
ElseIfBranchTest	5 25%	5 25%	20	4	4
ManyBranchTest	4 67%	4 67%	6	0	4
BubbleSort	12 80%	15 100%	15	0 ST: 3	5
Rot13	4 25%	4 25%	16	9	5
BinarySearchTree	avg: 26 68%	avg: 26 68%	38	0	7

Figure 8.5: Results with T2

T2Extension is reasonably stable with deviations of just 2% or 3%.

The reason for this substantial improvement for ElseIfBranchTest is twofold. First of all, T2Extension uses several different base domains that are larger than the default domain used by T2. Secondly and most important, the coverage improves so much due to the exhaustive search strategy used by T2Extension.

The table shows that the prime path calculator found 4 prime paths that are semantically unreachable when executing ElseIfBranchTest. In this piece of code, the first **if** statement has two **else if** branches (see figure B.1). In theory it could be possible to completely skip the first **if** statement and its branches if all the conditions for each branch fail. In practice, this is not possible because the conditions for this statement cover all the possibilities. So, paths that skip the first **if** statement are unfeasible. There are exactly four paths that skip this statement and T2Extension will either find 15 or 16 paths (see figure 8.7). This shows that T2Extension actually achieves (almost) full coverage.

The worst result was achieved when testing the class Rot13. Rot13 is a simple substitution cipher that rotates letters in the alphabet by 13 places. T2Extension still improved T2's coverage slightly, but its results are not as impressive as for the other classes. This can be explained by the amount of unfeasible paths. There are 9 unfeasible paths that can be explained as follows. The code in figure B.2 shows that there are two **if** statements that each have nested **if** statements. Characters are compared according to the ASCII table. In this table, symbols and numbers come before capital letters. The capital letters sit before the normal letters. This means that

Class Name	T2Extension No Sidetrips	T2Extension With Sidetrips	Total Prime Paths	Unfeasible Prime Paths	Runtime (Seconds)
DoubleLoopTest	12 39%	30 97%	31	1 ST: 18	1172
ElseIfBranchTest	avg: 15 77%	avg: 15 77%	20	4	102
ManyBranchTest	6 100%	6 100%	6	0	36
BubbleSort	12 80%	15 100%	15	0 ST: 3	2
Rot13	6 38%	6 38%	16	9	128
BinarySearchTree	avg: 35 92%	avg: 35 92%	38	0	34

Figure 8.6: Results with T2Extension

in the code for Rot13 it is never possible to execute a path that will enter both branches.

The coverage percentage for Rot13 shows that T2Extension will sometimes show a much lower result than is really the case in practice. The fact that there are 9 unfeasible paths indicate that T2Extension found all paths but one. This is still quite notable. of course, it would be better to show the user of T2Extension a coverage percentage that would match the good result, but it is not possible to automatically detect when a path is infeasible.

The prime path coverage achieved by T2Extension for the class ManyBranchTest is 100%. This is another very impressive result, considering that T2 only achieves 67%. This result shows that if T2 achieves a reasonable coverage result, T2Extension will do incredibly well.

The result for the bubble sort algorithm proves that it is possible for T2 to achieve complete coverage. T2Extension will try to search for paths that have only been found with sidetrips, but as can be seen, it could not find any extra paths.

The class BinarySearchTree is a real world example. It is a class containing several public and private methods. This class is tested by putting all methods in scope because the private methods are actually the most interesting methods. T2 accomplishes, on average, 68%. This is mediocre, but considering that this class holds many methods this is a good example of a case where the user will apply T2Extension to place more confidence in the result. T2Extension does very well with an average result of 92%. T2 varies quite strongly in its results with a lowest coverage of 58% and a highest coverage of 71%. T2Extension is more stable, varying between 89% and 95%. The deviations in coverage for this class are shown in 8.7(b).

The class BinarySearchTree is an example of T2's strength. Figures B.5, B.6 and B.7in appendix B show the code for this class. T2 will sometimes update the field root which is used in several methods and this will let methods follow different paths. There are also methods that alter the field root, which again will cause more diversity. T2Extension uses these advantages and expands them. It will execute traces several times with several modifications, this will cause the object state to be even more diverse which will lead to growth in path coverage. This shows another advantage of T2Extension; its exhaustive search leads to much diversity in the object state which will lead to a higher path coverage.

One slight disappointment is T2Extension's runtime. As can be seen in figure 8.5 T2 runs the test classes in a matter of seconds. T2Extension takes some more time. The runtime shown in figure 8.6 is the runtime that is needed for the search for the paths. Figure 8.8 shows the complete runtime for T2Extension. This includes all the overhead of path registration and administration and the run of T2. Although T2Extension is much slower than T2, it has little

ElseIfBranchTest			
T2	**Dev.**	**T2Extension**	**Dev.**
25%	0%	75%	2%
25%	0%	75%	2%
25%	0%	80%	3%
25%	0%	75%	2%
25%	0%	80%	3%
25%	0%	80%	3%
25%	0%	75%	2%
25%	0%	80%	3%
25%	0%	75%	2%
25%	0%	75%	2%

(a) ElseIfBranchTest

BinarySearchTree			
T2	**Dev.**	**T2Extension**	**Dev.**
58%	10%	92%	0%
71%	3%	92%	0%
71%	3%	95%	3%
68%	0%	89%	3%
74%	6%	92%	0%
68%	0%	92%	0%
76%	8%	95%	3%
66%	2%	92%	0%
71%	3%	92%	0%
61%	3%	89%	3%

(b) BinarySearchTree

Figure 8.7: Deviations in prime path coverage

overhead, as can be seen in table 8.8. One might have expected that the prime path calculation algorithm would create much overhead. Due to the graph optimisations this is not the case. The runtime for DoubleLoopTest shows that T2Extension is probably searching for many paths that are semantically unreachable. Otherwise, T2Extension has a reasonable runtime. Section 10.1 provides some suggestions for the improvement of T2Extension's runtime.

Figure 8.9 shows a summarising table of the results. The last column states if the results depicted varied in the 10 test runs (indicated by Y) or not. This section has proven that T2Extension is a very useful extension to T2 and that it improves T2's prime path coverage substantially.

Class Name	Runtime in Seconds
DoubleLoopTest	1381
ElseIfBranchTest	137
ManyBranchTest	52
BubbleSort	15
Rot13	163
BinarySearchTree	63

Figure 8.8: Runtime T2Extension

Class Name	T2 No Sidetrips	T2Extension No Sidetrips	T2 With Sidetrips	T2Extension With Sidetrip	Average
DoubleLoopTest	32%	39%	87%	97%	N
ElseIfBranchTest	25%	77%	25%	77%	Y
ManyBranchTest	67%	100%	67%	100%	N
BubbleSort	80%	80%	100%	100%	N
Rot13	25%	38%	25%	38%	N
BinarySearchTree	68%	92%	68%	92%	Y

Figure 8.9: Test results

Chapter 9

Related Work

Testing tools are becoming more and more important as software grows. Consequently, there are many automated unit testing tools. Many coverage measurement tools also exist since their importance also grows. However, there are few tools that combine unit testing and coverage measurements. This chapter discusses some tools that have properties that can be compared to T2Extension.

There is no tool at the moment that has all the properties that T2Extension has. T2Extension is unique due to its coverage criterion. Almost all existing coverage tools measure block coverage or branch coverage, none measure prime path coverage. Tools that combine testing and coverage measurements use very simple coverage metrics and they do not use coverage data to steer testing towards untested code in order to improve coverage.

All coverage measurement tools must instrument the class that is to be tested somehow in order to count how much code has been covered, but none use aspects. The reason for this is probably that the measurement tools do not need such a sophisticated method because the coverage data is not used to improve coverage of another testing tool.

9.1 Clover

Clover [17] is a proprietary coverage analysis tool built by Atlassian, a multinational company. The beauty of this coverage analysis tool is that it can be integrated with many IDE's. To explore Clover's options, it was tested with Eclipse. When integrated with an IDE, Clover can analyse system tests, functional tests and unit tests. It has many reporting options and colours source code green when it has been executed in any way and red when it has not.

This coverage analysis tool gives the user the choice of three coverage criteria; statement coverage, branch coverage and method coverage. Method coverage just checks whether a method has been executed. Clover keeps track of covered code by instrumenting the source code. The reason the Clover site gives for this is that source code instrumentation provides the most accurate coverage measurement with very little runtime overhead. This is true, T2Extension makes assumptions about branches because the bytecode has flattened nested branches and has separated conditional statements that contain more than one condition. However, the disadvantage of source code instrumentation is that the source code must be recompiled when Clover has inserted its statements. When using Clover in combination with an IDE, the user does not notice this, but Clover can also be used in combination with Ant and then the source code has to be rebuilt before it can be executed. T2Extension lets BCEL handle the compilation after it has inserted code and T2Extension weaves the input class without needing any input from the user.

Another disadvantage of source code instrumentation is that Clover cannot handle any special Java dialects. It does not support AspectJ and cannot handle assertions. If a class is woven with an aspect, T2 and T2Extension will not notice the difference because the compiled code

has already transformed the methods that are woven so that they can be executed by the Java Virtual Machine as normal methods. T2Extension ignores assertions because they are a special kind of condition and are not seen as a separate path.

An additional drawback of source code instrumentation is that there is no option for black-box testing. The source code of a program must always be present.

The danger of being complacent with branch coverage is that errors may still be present in the code, but are not recognised. It could be possible that a certain value is altered in one branch and usage in the next branch will result in an exception. If the testing tool has tested both branches separately, the branch coverage criterion has been fulfilled, but the error is not discovered. T2Extension will find this error due to its path coverage criterion, which gives it a large advantage over other tools.

Clover's forté is its reporting. Any report in any format can be generated and source code is highlighted. T2 and T2Extension do not have a GUI yet for reporting. This will be future work because such rich reporting makes any tool more user-friendly.

A freeware tool that is much alike Clover is CodeCover [22]. It can also be used as a plugin to IDE's and has the same working as Clover.

9.2 Cobertura

Cobertura [18] is also a coverage analysis tool. It is freeware written by Mark Doliner. Cobertura measures line coverage and branch coverage. It has rich reporting like Clover and can generate HTML or XML. Unlike Clover, Cobertura uses bytecode instrumentation. Mark Doliner's argumentation for this is that bytecode instrumentation is faster than source code instrumentation because there is no need to compile twice.

Cobertura works in combination with Ant or from the command line. It is easy to use an can be combined with any testing tool. This analysis tool works by keeping track of various counters that measure how many times which lines of code are executed.

Cobertura has one special property. It has the ability to show the McCabe cyclomatic code complexity. The cyclomatic complexity is a software metric, used to measure the complexity of a certain program. Cobertura shows the cyclomatic complexity per class. The metric for this complexity are the number of linearly independent paths. In other words, this gives an idea about the relation between the covered percentage and the amount of possible paths.

The linearly independent paths are not the same as prime paths. Prime paths are developed in order to properly test cycles and there are many more prime paths in a method than there are linearly independent paths. However, the amount of linearly independent paths are a lower bound for the amount of tours needed to achieve complete path coverage of a certain method. This is very useful when it is possible to instruct a testing tool with certain values so that a certain path is executed. The difference with T2Extension is that it is not relevant for T2Extension to know the cyclomatic complexity of a certain method. T2Extension relies on the traces generated by T2 and just modifies the tours the traces create to achieve its goal.

Cobertura generates report per package, it is then possible to look into classes and study how much code has been covered. T2 works per class and therefore T2Extension also works per class. There is no advantage or disadvantage to this, it is a choice that is made. Cobertura has the possibility to show coverage results per package because it does not test any code. T2Extension does not just show a coverage percentage, it also improves coverage.

One issue with Cobertura, just as with Clover is the fact that it does not measure beyond branch coverage.

9.3 EMMA

EMMA [19] is another code coverage tool. It is an open source project, created by Vlad Roubtsov. It is not as powerful as the two tools discussed above because it does not even measure branch coverage. EMMA can measure class, method, line and basic block coverage. Basic block coverage takes a group of statements without any branches as its unit of measurement. EMMA can put weights on blocks by for example weighing blocks in the coverage metric according to the amount of instructions they hold. Reports can be generated in several forms.

EMMA also uses bytecode instrumentation in order to measure coverage and can instrument the bytecode on-the-fly. T2Extension does this as well. It alters the bytecode while it is running, when T2Extension is finished, it returns the bytecode in the original state.

Like Clover, EMMA does not depend on any testing methodologies, it can be used in combination with any testing tool. EMMA promotes that is has little overhead, which is also one of the fortés of T2. Because it is so fast, EMMA is interactive and can be used for quick testing.

The creator of EMMA reasons that basic block coverage is virtually the same as branch coverage because one branch can be seen as one block of code. This is not entirely true. An example of this oversight has been given in section 1.1 and figure 1.1. 100% block coverage does not guarantee that the branch that skips the **if** branch has also been tested. The creator sees the use of block coverage as an advantage, I perceive this as a dangerous disadvantage. Roubtsov also argues that block coverage is more desirable than path coverage due to the fact that is has less overhead because there is less administration involved. This is correct, path coverage required much overhead, but it is a much safer manner of testing, especially when cycles are involved.

EMMA is a good tool for simple classes that do not contain too many branches or cycles. It greatest advantage is that it is fast and works on the fly. T2Extension is slower because of the administration needed for path coverage. However, its results are much more reliable.

9.4 JWalk

JWalk [20] is an automated testing tool that is quite different from other testing tools. Created by Anthony Simons, it is a lazy systematic class unit tester. Like T2, specifications can be written, but they are not necessary, JWalk also uses classes as its unit. The goal for JWalk is to be a fully automated testing tool, meaning that it learns from the tests that it generates because the human tester provides input. The tool will have learnt enough after several analysis cycles and it can then automatically test the complete class without the need for human interaction or specifications written by hand.

JWalk achieves this automation by focusing on the object's state space. With an abstract state-based model of the class to be tested, it can perform automated tests. T2 also uses this idea by generating test sequences, it modifies the object's state and saves these states. An exhaustive test in JWalk is a test that has covered the complete state model. This is an interesting idea, because a change in an object's state can cause its state to become illegal and therefore generate exceptions.

T2 does not necessarily check if the complete state model has been covered. Nevertheless, this can be achieved by T2Extension with some help from the programmer. The creator of T2 has written a paper [24] on this subject which has been discussed in chapter 5. An object's state is measured by the values of its fields. By building observer functions for these fields, they can be monitored. An overall method can then check whether a field has a certain value and if this is a correct state for the object that is tested to be in. If this is not the case, T2Extension can pass this information to T2 which can then report the error in state space.

JWalk is a very different testing tool from T2, but it is interesting to mention here due to its state-based model. This shows how versatile T2Extension is.

9.5 GrandTestAuto

GrandTestAuto [21] is also a unit testing tool where the units are methods. It was created by Tim Lavers and can be compared to JUnit. It expects the tester to write testing methods and then GrandTestAuto uses these to test methods in a class. What makes GrandTestAuto special is that is checks whether the unit tests provide sufficient coverage.

GrandTestAuto measures coverage on the method level. The testing tool checks whether all the methods in a class are tested, including inherited methods. This is a very simple coverage metric, but GrandTestAuto is one of the few testing tools that actually keeps track of some kind of coverage. T2Extension has of course a much more complex coverage metric that does not compare to the coverage metric of GrandTestAuto, but it is interesting to see that some tools do take coverage into account.

9.6 Evolutionary Testing

An evolutionary testing method is described in the paper [23] by Paolo Tonella. The testing method is implemented in a testing tool called *eToc*. The idea is to generate test cases that will cover all branches in a method. The test cases are generated in an evolutionary fashion. It is first decided which targets should be covered. A target can be a certain branch or path within a method. A pool of input data is created for the method to test and the pool is used to execute the method. From this test cases are deducted. The test cases are used to find targets. If a target is found with a certain test case, the test case is stored to form a pool of test cases later that will cover the complete method.

If a target is not found, a proximity measurement is computed for each test case. A new pool of test cases is created by randomly extracting from the previous population of test cases. Test cases with a closer proximity to the target have a higher probability of surviving and being passed to the next pool of test cases. This is repeated until all targets are found, or until a maximum execution time has passed.

This idea is very similar to what T2Extension does. T2Extension compares paths and manipulates traces that have the best chance of finding a certain path. Tonella's evolutionary testing method works quite well, achieving a branch coverage of over 90%. T2Extension will also often yield such results. However, T2Extension works with a much simpler algorithm of generating test cases. This shows that a random approach works better than one might think and its results are close to a directed search of test cases as the evolutionary algorithm does.

Chapter 10

Conclusion

This thesis ends with an answer to the research question posed in section 1.2. Also, some suggestions are made for future work.

The question this thesis was meant to answer is shown below.

Is it possible to improve prime path coverage of an automated testing tool with a random test generation?

The automated testing tool used here is T2. It has been extended with a tool that was placed on top of it named T2Extension. T2Extension used many techniques and tools like BCEL and AspectJ to achieve its goal. BCEL was used to insert monitoring code into methods and an aspect responded to the monitoring code by administrating all found paths.

T2Extension uses the traces generated by T2 and manipulates them in order to find paths that were unfound. Chapter 3 provides the theoretical basis for T2Extension. Paths in a method are depicted in a control flow graph. Prime paths are computed from the graph and T2Extension performs an exhaustive search for all paths.

The search for all paths in a class is supported by several algorithms, described in chapter 6. Although T2Extension tries to steer traces in the direction of a certain path, all input values for executing steps in a trace are randomly generated.

Chapter 8 has shown that T2Extension achieves incredible results, considering the random fashion in which it creates values. Whenever possible, it will improve the prime path coverage of T2 substantially. In several cases T2Extension was able to completely satisfy the test requirements of prime path coverage when T2 was not able to achieve this.

T2Extension achieves such good results due to its searching strategy. It uses several domains for constant values, increasing the options of input values. T2Extension also searches through traces extensively in four stages as described in chapter 7.

The coverage of testing tools using random searches is often quite low and it is recommended to use these tools as a complement to testing tools with a more direct approach to the search. T2Extension makes it possible for T2 to also be used as a standalone tool providing sufficient coverage.

The answer to the research question is definitely *yes*, with better results than expected. T2Extension does slow T2 down considerably, but its results make the slowdown worthwile.

Another matter that must be studied are the requirements for T2Extension listed in chapter 2. Below all the requirements are discussed.

Requirements	Description
Build T2Extension on top of T2	T2Extension is built completely on top of T2 with virtually no changes made to T2
Java class file as input	T2Extension uses bytecode instrumentation with the help of BCEL
Use existing T2 tools	T2Extension uses T2 itself and T2's traces in order to achieve its goal, it also uses T2 to generate values for method arguments
Reliable path coverage metric	Chapters 3, 6 and 7 show that a reliable coverage metric is used
Reliable path administration	The algorithms in chapter 6 and the template explained in chapter 7 show that if a paths is encountered, it will be registered.
Search for paths should be complete	T2Extension uses three different base domains for the generation of its constant values. It uses four techniques for the manipulation of traces. This is an exhaustive search which the test results prove.
Return original class	This is achieved by copying the original class before modifying it
T2Extension should be as fast as possible	T2Extension incorporates several optimisations which make it reasonably fast, however there exist possibilities for more optimisations

The table above shows that T2Extension meets all of its requirements. However, future work might involve some optimisations.

10.1 Future Work

T2Extension is a stable framework for future work. New classes can be added to it and they can be integrated into the T2Extension project by letting the main class call the additions, or the additions can be executed using Ant. This makes T2Extension easy to extend and facilitates integration with other applications.

The use of AOP generates many possibilities for monitoring and altering the class to be tested. The template aspect can easily be augmented with extra pointcuts and advice.

Here, several suggestions are given for future work.

10.1.1 Application Models

A formal definition of T2's application model has been given in chapter 4. Application models are used to generate traces but could also be used to check certain properties. An application model M is always seen from the perspective of a class C. This means that properties can be defined in M about fields or methods in C. However, T2 does not yet check if a call to a method m_1 that has been declared in C violates the properties stipulated in M if m_1 is called from a different class B. It is of great importance that B respects M when calling m_1. A trace coming from class B that does not respect M makes it impossible to determine whether C still satisfies its own properties.

The satisfaction of a property by C is defined as follows.

C satisfies (a property) ϕ if for all traces π in M we have $\pi \vdash \phi$.

The class invariants for the target object are checked after each step in a trace. The application model is specified in terms of these class invariants. This means that a trace with class B as its target object will never be able to check C's application model, even if B calls a method belonging to C.

With aspect weaving it is very simple to state that after all methods in a certain class, its application model should be checked by checking its class invariants. However, the method that checks the class invariants is a private boolean method because M is seen from the perspective of the class that it belongs to. Since it is a private method, it cannot be called from an aspect. The solution is to insert a public auxiliary method m_{aux} in C with BCEL. It will call the private method that checks the class invariants. This method m_{aux} can then be called from the aspect after every method call. T2 will handle the steps to take if a call to m_1 violates the statements in m_{aux}.

The idea of checking an application model leads to many other properties that can be checked. A user could specify that certain specifications should be be checked before every method in the CUT is called or after each method is called. The aspect provides much potential for the arrangement of a diversity in specification and property checking.

10.1.2 Custom Base Domain

T2 retrieves constant values from a base domain that holds a certain range of primitive values, enumeration values and strings. Chapter 8 has shown that one of the reasons T2Extension yields a higher path coverage than T2 is because it adapts the used base domain.

A custom base domain might speed T2Extension up. Now, T2Extension uses three base domains in order to cover as many values as possible. This custom base domain could be much larger than the default base domain. With some research it could probably be well adapted to a wide range of primitive values, giving T2Extension a much larger scope of values to choose from.

10.1.3 Runtime Optimisations

Apart from a custom base domain, it might be possible to steer traces even more into a certain direction in order to find a path. This might result in faster searches that need less time to achieve results. In order to achieve this optimisation, more analysis is needed. This in turn might require the adaptation of T2 so that its traces hold more information. On the other hand, aspects could also be used for the gathering of information.

Also, a study could be performed that analyses which of the four techniques used to find a certain path is most successful. If one technique achieves better results than the others, this technique might be improved and used by itself instead of in a combination with the other techniques. Nevertheless, this could be dangerous because the search might not be as exhaustive anymore as it is at the moment.

Although several runtime optimisations are plausible, thorough tests should be done to ensure that T2Extension's overall prime path coverage is not compromised.

10.1.4 Reporting

T2 and T2Extension currently have simple reporting shown in the command line. To make the tools more user-friendly a GUI should be built. This can easily be done by plugging in a GUI to T2Extension. The GUI can then be used to provide extensive reporting. T2 could use the reporting for its error messages, and showing the execution of traces that lead to a certain result.

T2Extension can use the GUI for coverage reports. Coverage percentages should be shown per CUT, but also per method. The T2 user might also want to know which paths were not covered. Since T2Extension works with bytecode instrumentation, it is not possible to highlight

source code. Nevertheless, it could be possible to show the control flow graphs for each method in the CUT with source code explanations next to its nodes, as in done in figure 8.2. The source code explanations can be deduced from BCEL instructions. It is then possible to highlight paths in the graph that were not found by T2Extension.

Appendix A

Running T2Extension

This appendix shows how T2Extension is used.

A.1 Necessities

T2Extension is implemented in Java 1.5 and is distributed as a jar, making it platform independent. The user will need a Java Runtime Environment. T2Extension uses Apache Ant and AspectJ, the jars for these tools are also needed and are therefore included in the distribution of T2Extension.

To run T2Extension, the command line is used as follows for Windows users:

```
java -ea -cp .;T2Extension.jar;aspectjrt.jar;ant.jar;ant-launcher.jar t2ext.T2Extension
<input class> <directory T2Extension.jar>
```

Linux users call T2Extension by passing the following to the command line:

```
java -ea -cp .:T2Extension.jar:aspectjrt.jar:ant.jar:ant-launcher.jar t2ext.T2Extension
<input class> <directory T2Extension.jar>
```

An example of how to run T2Extension:

```
java -ea -cp .;C:\t2ext\T2Extension.jar;C:\t2ext\aspectjrt.jar;C:\t2ext\ant.jar
t2ext.T2Extension example.BinarySearchTree C:\t2ext\T2Extension.jar
```

The program needs the directory and name of the jar holding T2Extension because it must copy the jar. The copy jar can then be used to insert certain files into it for the compilation of the template aspect and for the necessary weaving.

A.2 Options for T2Extension

T2Extension accepts all arguments that can be passed to T2 and will handle them as T2 does. T2Extension has two extra options.

--maxattempts=int
: This is the amount of times T2Extension should run the algorithm that manipulates traces in order to find paths. By default it is set to 1. If this amount is passed to T2Extension, it

will either execute the algorithm until all paths are found or until maxattempts is reached, whichever comes first.

−−pathoptm
This arguments switches on an optimisation. With this option turned on, T2Extension will quit executing methods if they are not on course of the path that T2Extension is trying to find.

Appendix B

Code for Test Results

This appendix contains all the code for the test classes used in chapter 8.

```
public int calc(int a, int b){
  if(a<b){
    a += 13;
    if(a > 15)
      a -= 10;
  }else if(a==b)
    a += b;
  else if(a>b)
    a -= b;
  if(a > 10 && b < 20){
    a += 13;
    if(a > 25)
      a -= 5;
  }
  return a;
}
```

Figure B.1: ElseIfBranchTest

```
/**
 * @author David Flanagan
*/
public char rot13(char c) {
  if ((c >= 'A') && (c <= 'Z')) {
    c += 13;
    if (c > 'Z')
      c -= 26;
  }
  if ((c >= 'a') && (c <= 'z')) {
    c += 13;
    if (c > 'z')
      c -= 26;
  }
  return c;
}
```

Figure B.2: Rot13

```
public int calc(int a, int b){
  if(b+a>5){
    a = b;
    return a;
  }
  if(a-b==1){
    a = a+b+2;
    return a;
  }
  if(b-a>2){
    a = a*b+10;
    return a;
  }
  if(a*b<10){
    a = a*b+2;
    return a;
  }
  else
    a = b+2;
  if(a<=0)
    a = 1;
  return a;
}
```

Figure B.3: ManyBranchTest

```
public List<Integer> bubblesort(List<Integer> list){
  if(list!=null){
    for (int i = 1; i < list.size(); i++)
      for (int j = 0; j < list.size() - i; j++)
        if (list.get(j).intValue() > list.get(j+1).intValue())
          swap(list, j, j + 1);
  }
  return list;
}

private void swap(List<Integer> list, int i, int j) {
  int a = list.get(i);
  int b = list.get(j);
  list.set(i, b);
  list.set(j, a);
}
```

Figure B.4: BubbleSort

```
/**
 * A class implementing unbalanced binary search tree.  Note that all
 * "matching" is based on the compareTo method.
 *
 * Adapted from original code by Mark Weiss.
 */

// ******************PUBLIC OPERATIONS*********************
// void insert( x )   --> Insert x
// void remove( x )   --> Remove x
// Comparable find( x )   --> Return item that matches x
// Comparable findMin( )  --> Return smallest item
// Comparable findMax( )  --> Return largest item
// boolean isEmpty( ) --> Return true if empty; else false
// void makeEmpty( )  --> Remove all items
// void printTree( )  --> Print tree in sorted order

public class BinarySearchTree {

  /**
   * Construct the tree.
   */
  public BinarySearchTree( ) {
    root = null;
  }

  public void insert( Comparable x ) {
    // System.out.println(".") ;
    root = insert(x,root );
  }

  /**
   * A specification for insert, saying that after insert(x) x
   * should be in the tree.
   */
  public void insert_spec(Comparable x) {
    insert(x) ;
    assert (find(x) != null) : "POST" ;
  }

  public void remove( Comparable x ) {
    root = remove( x, root );
  }

  /**
   * a specification for remove, saying that after the remove x
   * should no longer be in the tree.
   */
  public void remove_spec(Comparable x) {
    remove(x) ;
    assert (find(x) == null) : "POST" ;
  }

  public Comparable findMin( ) {
    return elementAt( findMin( root ) );
  }
```

Figure B.5: BinarySearchTree, part 1

```
/**
 * The spec of findMin. It says that the returned value, if not
 * null, should be an element of the tree.
 */
public void findMin_spec() {
  boolean wasEmpty = isEmpty() ;
  Comparable x = findMin() ;
  assert (wasEmpty || find(x) == x) : "POST" ;
}

public Comparable findMax( ) {
  return elementAt( findMax( root ) );
}

public Comparable find( Comparable x ) {
  return elementAt( find( x, root ) );
}

public void makeEmpty( ) {
  root = null;
}

public boolean isEmpty( ) {
  return root == null;
}

private Comparable elementAt( BinaryNode t )  {
  return t == null ? null : t.element;
}

private BinaryNode insert( Comparable x, BinaryNode t ) {
  /* 1*/  if( t == null )
    /* 2*/  t = new BinaryNode( x, null, null );
  /* 3*/  else if( x.compareTo( t.element ) < 0 )
    /* 4*/  t.left = insert( x, t.left );
  /* 5*/  else if( x.compareTo( t.element ) > 0 )
    /* 6*/  t.right = insert( x, t.right );
  /* 7*/  else
    /* 8*/  ;  // Duplicate; do nothing
  /* 9*/  return t;
}

class BinaryNode {

  Comparable element ;
  BinaryNode left ;
  BinaryNode right ;

  BinaryNode(Comparable x, BinaryNode u, BinaryNode v) {
    element = x ;
    left = u ;
    right = v ;
  }

}
```

Figure B.6: BinarySearchTree, part 2

```
1   private BinaryNode remove( Comparable x, BinaryNode t ) {
      if( t == null )
3       return t;   // Item not found; do nothing
      if( x.compareTo( t.element ) < 0 )
5       t.left = remove( x, t.left );
      else if( x.compareTo( t.element ) > 0 )
7       t.right = remove( x, t.right );
      else if( t.left != null && t.right != null ) // Two children
9     {
        t.element = findMin( t.right ).element;
11      t.right = remove( t.element, t.right );
      }
13    else
        t = ( t.left != null ) ? t.left : t.right;
15    return t;
    }
17
    private BinaryNode findMin( BinaryNode t ) {
19    if( t == null )
        return null;
21    else if( t.left == null )
        return t;
23    return findMin( t.left );
    }
25
    private BinaryNode findMax( BinaryNode t ) {
27    if( t != null )
        while( t.right != null )
29        t = t.right;

31    return t;
    }
33
    private BinaryNode find( Comparable x, BinaryNode t )
35  {
      if( t == null )
37      return null;
      if( x.compareTo( t.element ) < 0 )
39      return find( x, t.left );
      else if( x.compareTo( t.element ) > 0 )
41      return find( x, t.right );
      else
43      return t;// Match
    }
45
    /** The tree root. */
47  private BinaryNode root;
  }
```

Figure B.7: BinarySearchTree, part 3

Bibliography

[1] I. S. W. B. Prasetya, T. E. J. Vos, A. Baars, *Trace-based Reflexive Testing of OO Programs with T2*, 2007

[2] Hridesh Rajan, Kevin Sullivan, *Generalizing AOP for Aspect-Oriented Testing*, Conference 2005

[3] Guoqing Xu, Zongyuan Yang, *A Novel Approach to Unit Testing: The Aspect-Oriented Way*, 2004

[4] Guoqing Xu, Zongyuan Yang, Haitao Huang, Qian Chen, Ling Chen, Fengbin Xu, *JAOUT: Automated Generation of Aspect-Oriented Unit Test*, APSEC 2004

[5] Daniel Hughes, Philip Greenwood, *Aspect Testing Framework*

[6] The T2 website, *http://www.cs.uu.nl/wiki/bin/view/WP/T2Framework*

[7] Wikipedia explanation on AOP, *http://en.wikipedia.org/wiki/Aspect_Oriented_Programming*

[8] AspectJ, *http://www.eclipse.org/aspectj/*

[9] Daniel Hughes, Philip Greenwood, Geoff Coulson, *A Framework for Testing Distributed Systems*, IEEE International Conference of Peer-to-Peer Computing 2004

[10] JUnit website, *http://www.junit.org*

[11] Guoqing Xu, Zongyuan Yang, *JMLAutoTest: A Novel Automated Testing Framework Based on JML and JUnit*, In Proc. 3rd International Workshop on Formal Approaches to Testing of Software (FATES '03), pp. 118-127, Montreal Canada, October 2003

[12] BCEL website, *http://jakarta.apache.org/bcel/*

[13] Mark Harman, *Open Problems in Testability Transformation*, 1st IEEE International Conference on Software Testing, Verification and Validation, 2008

[14] Paul Ammann, Jeff Offutt, *Introduction to Software Testing*, ISBN: 978-0-521-88038-1, 2008

[15] Apache Ant, *http://ant.apache.org/*

[16] Narsingh Deo, *Graph Theory with Applications to Engineering and Computer Science*, ISBN: 0-13-363473-6, 1974

[17] Clover, *http://www.atlassian.com/software/clover/*

[18] Cobertura, *http://cobertura.sourceforge.net/*

[19] EMMA, *http://emma.sourceforge.net/*

[20] A.J.H Simons, *JWalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction*, Automated Software Engineering, Volume 14 (4), 369 - 418, 2007

[21] GrandTestAuto, *http://grandtestauto.org/*

[22] CodeCover, *http://codecover.org/*

[23] Paolo Tonella, *Evolutionary Testing of Classes*, ISSTA 2004, July 11-14, Boston, Massachusetts, USA

[24] I. S. W. B. Prasetya, T.E.J. Vos, *Patterns for In-code Algebraic Testing*, 2008