UNIVERSITY OF OSLO
Department of Informatics

Instrumentation and transformation of Java source code for automated testing with search-based testing algorithms

Master thesis

Karsten Jansen

August 2010

Abstract

Search-based Software Test Data Generation is a field of research treating test input generation as a search problem. Search algorithms require that an *objective function* computes the quality of solution candidates. When the goal for testing is to achieve code coverage, one type of objective function uses *branch distance*, a heuristic describing how "close" the condition in a control flow statement is from being true.

An attempt is made to define procedures to allow the calculation of branch distance for both predicates in conjunction and disjunction predicates with short-circuiting operators && and `||`, by identifying and avoiding situations causing side effects and exceptions.

This thesis also attempts to give examples of instrumenting and transforming control flow statements, examples which are limited or lacking in the research literature.

A program implementing the defined rules for instrumentation and transformation of control flow statements was developed to validate these rules.

Acknowledgements

I would like to thank my supervisor Andrea Arcuri for valuable and vital help and feedback during the writing of the thesis, and for interesting discussions.

I would also like to thank friends and family for supporting me throughout the years of education, and especially my dear Marita for putting up with me, helping me, and inspiring me throughout writing the thesis.

Contents

1 Introduction

1.1 Automatic testing

1.1.1 Cost of testing

Testing is vital and ubiquitous in all software development. Nearly 50 percent of costs of development and life cycle can be attributed to testing [7]. Done manually, as is the common procedure today, this is a tedious and error-prone procedure. Consequently, huge sums of money and developer time can be saved on any improvement of the level of automation in testing. Direct financial issues are not all that can be improved, customer satisfaction rises when software is shipped with as few faults as possible.

Although complete automation of the testing of software is desirable, it will in practice be impossible. The general test data generation problem is an undecidable problem [18].

1.1.2 Search-based Software Test Data Generation

A promising field of research named "Search-Based Software Test Data Generation" [18] (SBSTDG) treats the test data generation problem as a search problem.

Although not guaranteed to find the optimal solutions, techniques developed and improved in the research done in this field find solutions which are "good enough", i.e. surpassing a certain threshold of quality, or beating the previous best solution. Such a strategy has potentially huge benefits, as even a modest improvement in automation can save millions of dollars in development and support expenses.

1.2 Contribution of the thesis

Typically when searching for test input to a given program, an "objective function" is defined, which computes how well a candidate solution solves the search problem. One search technique uses what is known as the "branch distance" in this objective function, a measure of how far a control flow statement such as an *if-then-else*-statement is from evaluating to either *true* or *false*. The software being tested is typically *instrumented*, meaning code is inserted in order to monitor the execution of the software. To compute the branch distances, some parts of the code may be (temporarily) *transformed* as well.

Although important, instrumentation and transformation preceding calculation of such branch distances is not treated in very much detail in the literature surveyed. This thesis attempts to give such examples for a selection of control flow statements of the Java programming language.

There are certain structures in Java that prevent effective calculation of the branch distance, for example the "short-circuiting" conditional operators "&&" and "||". The situations in which calculations are inhibited will be explained, and an attempt will be made to formulate procedures to circumvent these situations.

An application for the automatic instrumentation and transformation of Java source code will be presented, and will serve as a proof-of-concept and testing grounds for rules regarding branch distance calculations.

1.3 Structure of the thesis

Section 2 contains the theoretical background of the research field. Subjects important to the main contribution of the thesis, such as *instrumentation* and *transformation* of source code, and *branch distance*, are given extended consideration. The section ends with the main challenges in the thesis being fleshed out. This concerns mainly instrumentation, transformation and branch distance calculation for specific, selected control flow statements and predicates containing the conditional operators. Circumventing the limitations of the conditional operators may involve using heuristics instead of normally generated branch distances. These heuristics are discussed, and some overview is given on the consequences of not using them.

Section 3 is an empirical analysis where I investigate the distribution of control flow statements and expression types contained within them.

As the different expression types making up the predicates of control flow statements require different rules for transformation and heuristics, the analysis will serve as a rough guide to prioritizing the development of such rules.

Section 4 states the rules defined for each type of expression, both for calculating branch distance and heuristics. It also gives an overview of the requirements and the implementation of VIns, the application for automatically instrumenting and transforming Java source code in line with these rules.

A discussion of the validity of the program is given in Section 5, and following that, Section 6 has a discussion of the impact of the findings in the empirical analysis, on how the thesis has handled the challenges depicted in the first section, and any limitations in the implementation of the program and the rules for branch distance calculations and heuristics.

2 Theory

2.1 The research field

Search Based Software Engineering (SBSE) is the field of treating software engineering as a search problem, i.e. using various search algorithms to generate the data defining the solution to the engineering problem.

In particular, software testing is an area which has seen much progress using this approach. This special form of SBSE is commonly called *Search-based Software Test Data Generation* (*SBSTDG*).

Test data generation is an undecidable problem. To circumvent this, search techniques which have the property of finding a "good enough" solution are being used.

According to Harman *et al.* [14], there has been an explosion in the number of articles published in the field of SBSE, as well as a gold rush of undirected research typical of a new field.

2.1.1 Metaheuristic search techniques

"Metaheuristic search techniques are high-level frameworks which utilize heuristics in order to find solutions to combinatorial problems at a reasonable computational cost" [18].

Metaheuristic search has been used for generating test data for specific structures in *white-box testing* (see below), to execute a certain structure in a program, finding and triggering flaws in a program, and also to test properties of software, such as execution time [18].

When employing meta-heuristic search techniques, there are certain trade-offs one has to accept [14].

- A global optimum, i.e. the perfect solution, may not be found. However, such search techniques will find a host of solutions better than a given threshold.

- Predictability may be low. With a large enough search space, which will be the case in most of the times such techniques are utilized, one will see different results every time one searches.

- There are also no instant results — search may take a great deal of time.

2.1.2 White-box testing

White-box testing generates tests based on the structure of the software, extracted with an analysis of the code.

A test case in white-box testing is a set of inputs to the code under test, for example consisting of values for local and global variables found in the code [13].

The *control flow graph* ("cfg", see Figure 1) is a representation of the control structures of a program (or other unit of code). In the description taken from Tracey *et al.* [25], the nodes in this directed graph correspond to blocks of code, of which all or none are executed. The edges are possible transfers of control, and in the case of more than one edge out from a node, a *branch predicate* (or just "*predicate*") will decide between them. The branch predicate is a control flow statement, such as "`if (a == 1)`".

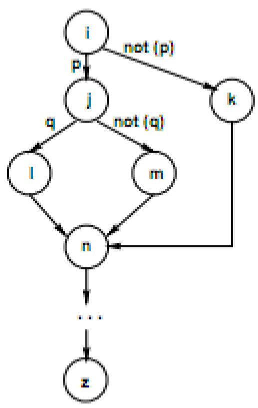

Figure 1: Control Flow Graph (CFG)

A *test adequacy criterion* is some measure or set of measures that a test has to fulfill in order to be deemed successful.

A test set is said to be "adequate for a given criterion if the whole or a subset of the tests satisfy the criterion" [22].

Such a criterion can be for example *code coverage* [28], which is a measure of the structures in a program executed by a particular test input or set of test inputs.

Different code coverage measures exist, for example *branch coverage*, which concerns whether all edges in a program have been executed, *function coverage*, which says whether all functions in the program have been executed, statement coverage, which says whether all nodes in a program have been covered, and others.

Coverage is measured by *instrumenting* the code (see 2.3).

Static analysis consists of analyzing the code without actually executing it. *Symbolic execution* is an example of this. It involves traversing the control flow graph of the program, and analyzing any internal variables in light of the input variables provided. Constraints on these variables are defined in regards to branches of the control flow graph of the program, and solutions to these constraints constitute the test data.

McMinn [18] states that generating and solving such constraints are in the general case an NP-complete problem, but is possible when certain properties fall in to place, such as when the constraints are linear. In other cases, heuristic methods can be used even when these properties are not present. Even so, the relations between variables and inputs may not be analyzable because of loops, arrays and pointers [8]. Loops are problematic because of the possibly large number of paths that have to be analyzed. Arrays and pointers are often assigned dynamically, and this information will not be available to a static analysis. Particularly object-oriented systems, rich in all these structures, may be difficult to handle using symbolic execution [4].

Dynamic analysis is the execution of a program and the subsequent observation of its behavior and/or results by way of instrumentation (see 2.3, and see earlier on test adequacy and coverage).

Loops, pointers and arrays are a much smaller problem in dynamic analysis, since the number of iterations and the values are known at runtime [18].

2.1.3 Search algorithms

Normally in a test input search, the algorithms start with random input. This input, and every subsequent proposed solution, are compared to the test adequacy criterion, in order to see if the solution is good enough. If the criterion is not met, for example if the criterion is 100% branch coverage and a branch is not executed, search is continued until it is.

An *objective function* guides this search, by informing the algorithm in question how well the attempt fared. The objective function is often called the *cost function* if the goal is to minimize the value of the function.

A few common search algorithms are discussed; Random search, *Genetic Algorithm*, which is an example of global search, and *Hill climbing*, which is an example of local search.

Random search Random Search (RS, see Listing 1 [2]) is the simplest search algorithm. It just samples search points at random, and stops when a global optimum is found (i.e., when the target branch is covered).

It not really an example of meta-heuristic search as it does not use any information gathered to guide the search. However, it is widely used as a benchmark for comparing the performance of other algorithms.

Although simple to implement, for any non-trivial program it is inefficient and "unlikely to exercise deeper features of software that are not exercised by mere chance" [18].

Listing 1: Pseudo code for the Random Search (RS) algorithm

```
while termination criterion not met
        Choose I uniformly from S.

(where I is a test input)
```

Genetic Algorithms Genetic Algorithms (GA, see Listing 2 [14]), the variant probably best known [18] out of the larger class of algorithms evolutionary algorithms, use mechanisms of pseudo-natural selection to *evolve* solutions. A solution is often called an *individual*, with the parts making up this solution called *genes*. If the individual is a vector of inputs, one of those inputs would be a gene.

One mechanism borrowed from evolution in nature is the recombination of two individuals' genes, i.e. the two individuals are *breeding*. Which genes pass on to offsprings can be chosen at random, or some rule may be defined, depending on design and need. Another mechanism is mutation, where a gene has a certain chance of being changed. As the genes consist of inputs to the program, care must be taken to avoid going outside of boundaries the inputs can have in the program.

GAs keep track of many individuals, and the group is called a population. The strongest individuals, meaning those scoring best on the objective function, have the best chance to survive to the next generation and/or be parents of the next generation.

Different selection and ranking strategies exist, and whereas the strongest have the best chance of surviving, an element of randomness may be involved.

Hill climbing *Hill Climbing* (HC, see Listing 3 [2]) is a local search algorithm often used in the research litterature. From the search space, a starting point is chosen, and the *neighborhood* in the search space is investigated. If the neighborhood contains a better solution, this is chosen. Then the neighborhood search continues, until no better

Listing 2: Pseudo code for the Genetic Algorithm (GA)

```

Set generation number, m:= 0
Choose the initial population of candidate solutions, P(0)
Evaluate the fitness for each individual of P(0), F(Pi(0))
loop
  Recombine: P(m):= R(P(m))
  Mutate : P(m) := M(P(m))
  Evaluate: F(P(m))
  Select: P(m+1) := S(P(m))
  m := m +1
  exit when goal or stopping condition is satisfied
end loop;
```

Listing 3: Pseudo code for the Hill Climbing (HC) algorithm

```

while termination criterion not met
        Choose I uniformly at random from S.
        while I not a local optimum in N(I),
                Choose I2 from N(I) according to strategy S
                if f(I2) < f(I),
                        then I := I2.
```

solution can be found. The solution is then an *optimum*, either local or global.

The name "Hill climbing" comes from visualizing the search space as a landscape, where the peaks represent good values for the objective function for the particular input in the search space, and the valleys represent bad values. The neighborhood search thus represents climbing these hills.

Depending on the implementation of the algorithm, the search can have one or more *restarts*, in which more points in the search space are chosen as starting points, and the highest peak climbed is the solution.

Hill climbing is simple and gives fast results, but there is a danger of never finding the global optimum [18].

Algorithms can be combined, to have the best of both local and global search. For example, memetic algorithms combine a GA with a HC. Whenever an offspring is created, local seach is applied to find a local optimum [3].

2.2 Fitness function and branch distance

2.2.1 Role of the fitness function

The *fitness function* (in general optimization problems called "objective function") is what informs the search algorithm of the quality of a solution, and hence possible improvement or worsening of this quality in successive proposed solutions [14]. This also holds true when searching for and generating test input data. The fitness function computes a value representing how well the solution is fulfilling the test adequacy criterion.

In the example of genetic algorithms, an individual that has a good fitness value has a better chance of prevailing to the next generation. When branch coverage is used as the test adequacy criterion, what may be used as a fitness function is the *approach level* and the *branch distance*, combined as " Fitness = approach level + branch distance" [19].

2.2.2 Approach level

The *approximation level*, also called *approach level* [17], is a measure of how far from executing the target branch the current solution candidate came. To execute a target branch, the program must execute a certain set of branches leading up to it. Whenever the test inputs make the flow of control divert from these branches, the target cannot be reached. The number of predicates in this set still not executed is the approach level.

The value 1.0 is added to the fitness function value for each such predicate. This is an approximation representing the potential value of each branch distance in the path to the target. The maximum value for a branch distance (1.0) is added, since the actual value is unknown.

A slightly different way of explaining the approach level is that it is the number of potential problem nodes that lay on the shortest path from the actual problem node to the targeted test goal [27], the actual problem node being the one containing the predicate that caused the control flow to divert from the desired path.

The problem node is called a critical, or decisive, branch. Since this branch has been chosen, the "failure to reach the target has been 'decided' "[20].

2.2.3 Branch distance

Branch distance is a measure of how far a predicate is from evaluating to a specified boolean value, normally "true".

For example, if a predicate states "`if a == 1`" and "a" has the value 2, then the branch distance in this case could be assigned as "`distance d = abs(2−1) = 1`". For other comparison operators, other functions decide, which can be seen in Table 1, as reported by Tracey *et al.*[25]. *K* refers to a constant called a "penalty constant", which is used to increase the cost of predicates being false.

Table 1: Branch distance calculation from Tracey *et al.*

Element	Value
Boolean	if TRUE then 0 else K
a = b	if abs(a-b) = 0 then 0 else abs(a-b) + K
a $\neq$ b	if abs(a-b) $\neq$ 0 then 0 else K
a < b	if a-b < 0 then 0 else (a-b) + K
a $\leq$ b	if a-b $\leq$ 0 then 0 else (a-b) + K
a > b	if b-a > 0 then 0 else (b-a) + K
a $\geq$ b	if b-a $\geq$ 0 then 0 else (b-a) + K
a $\vee$ b	min (cost (a), cost (b))
a $\wedge$ b	cost (a) + cost (b)
$\neg$ a	Negation is moved inwards and propagated over a

The resulting branch distance is conventionally normalized to a value between 0.0 and 1.0, where 0.0 signifies true. The larger the gap between the values compared, the closer the branch distance is to 1.0. This is often

achieved with a formula such as
"$normalize(branchdistance) = 1 - 1.001^{-branchdistance}$" [20].
However, Arcuri [1] points out flaws in this formula which may actually have an impact on the testing effort. He proposes another, less computationally expensive function (x / x + beta), where beta is a constant value.

The branch distance for a specified branch is 0.0 (i.e. there is no distance) when the predicate has a boolean value that causes that branch to be executed.

For example, in "(x < 2)", if x has the value 1, the branch distance for the **true** branch is 0.0, because the predicate evaluated to **true**.

Fitness landscape is a term describing a mapping between the values of the search space and the values of the fitness function. Values or ranges of values in the search space causing poor values in the fitness function will look like valleys, and those causing good values will look like hills (see Hill Climbing, 2.1.3).

2.2.4 Calculating the branch distance

The procedure used in Tracey *et al.* [25], is to calculate the branch distance from an instrumented source by inserting "Cost_N(...)" calls to replace branch predicates in the source. During execution, these calls evaluate the original predicates and register the branch distance. "N" refers to the indexed number of the predicate. The return value from a call is a boolean, namely the result of evaluating the predicate. In this way, the execution flows through the correct path even in the instrumented version of the code.

If the target node lies in the true-branch, the cost of the branch predicate should be added to the total of the fitness function value. If the target node lies in the false-branch however, the cost of the logical negation of the branch predicate is added, i.e. $distance(\neg(branchpredicate))$.

Rival methods exist. Bottaci [8] comes up with another variation of the formula than Tracey *et al.* (Table 2).

Table 2: Branch distance calculation from Bottaci. ϵ is the penalty constant.

Predicate expression (p.e.)	cost of not satisfying p.e.
$a \leq b$	a-b
$a < b$	a-b + ϵ
$a \geq b$	b-a
$a > b$	b-a + ϵ
$a = b$	abs(a-b)
$a \neq b$	ϵ - abs(a-b)

The calculations of single predicates are basically equivalent, but the authors differ when it comes to compound predicates. For "Predicate A AND predicate B" ("A && B"), Tracey *et al.* use the sum of the two predicates, while Bottaci uses the highest value of the two. For the calculation of "Predicate A OR predicate B" ("A || B"), both Tracey *et al.* and Bottaci use the smallest value, but Bottaci makes the case that in some situations the sum should be used here as well.

It is worth noting that some logical inconsistencies may arise, depending on the choice of calculation rules. As the sum of two values is not the inverse of the minimum of two values, two logically equivalent predicates written in different ways may not have the same distance value calculated. This is presumably not impacting testing negatively.

Liu *et al.* [17] have an alternative formula for compound predicates:

d(P1 AND P2) = d(P1) + d(P2)

d(P1 OR P2) = (d(P1) * d(P2)) / (d(P1) + d(P2))

2.2.5 Distances of non-numerical values

The calculations above are made with numerical values. However, other values are also compared in predicates, such as boolean values, enumerations and objects.

Boolean values are often compared, or even evaluated by themselves. As the boolean values are only true or false, the distance becomes either 0.0 (if the boolean values compared are equal, or the lone boolean value is true), or else 1.0.

Since there are only two possible values, multiple iterations or generations will not provide any guidance through the fitness function in the same way numerical values do. In other words, comparisons of the branch distances for the predicates will not show an improvement or worsening from candidate solution to candidate solution. The fitness landscape is totally flat, and finding the the desired test input becomes equal to finding a needle in a haystack [15].

Comparisons between Enumerations [12] *can* show a better fitness landscape than that of the boolean values, provided there is an ordering of the values to help guide the search. Otherwise, the landscape is as flat as for the boolean values.

When comparing object references with the *equals ("==")* operator, either the references point to the same object, and the branch distance is 0.0 — or they point to different objects, and the branch distance is 1.0. The fitness landscape is thus no better than for boolean values.

Another comparison between objects is done with the *equals()* method. This method finds out if two objects represent the same, i.e. are of the same type and have the same state. In these cases, there might be a fitness landscape with more of a search-friendly gradient, as the differences in states between two objects conceivably could be captured in a cost function. However, method calls are generally not transformed when calculating branch distances, and no general procedure for a cost function comparing two objects has been seen in the literature.

2.3 Instrumenting code

Instrumentation is the process of inserting custom code into an existing program, which allows information from the execution of the program to be collected. Code is instrumented to, among other things, be able to see the paths the program takes through the code given a set of input variables. The paths become visible through inserted *probes* [23], which are calls to functions in the search framework that keep track of which parts of the code have been executed.

The code can be instrumented on the fly when it is being executed, or the modified source can be saved to another file, which is then executed. The instrumented code will now run the inserted code whenever these structures are called in the running program.

2.3.1 Source code versus byte code

Both Java source code and byte code can be instrumented, depending on preference. One advantage argued for using byte code is that one does not have to make type inferences, as this is done by the compiler at compile-time [21]. Another advantage is the possibility of testing third-party code for which the source is unavailable. An advantage with the instrumentation of source code, however, is that the instrumented source code is much easier to understand when inspecting it, so that development and debugging is easier.

2.3.2 Instrumentation for control flow analysis

For the trace of a program to give sufficient information about the flow of control in a program's execution, it is interesting to register both which methods are executed, as well as any blocks of code executed within these methods. A probe is thus inserted in both cases. It is desirable to have the probe as the first call in any block of code, in order to correctly register the control flow sequence. Having the probe later in the code block could mean that other methods and branch predicates were executed first, thus executing their own probe calls.

In certain cases this is a challenge, such as in constructors. In Java, constructors can be overloaded, meaning a class can have several constructors, of which one is selected when creating an object through the "*new*" keyword. However, a constructor can as part of the object construction call other constructors in the same class or in the superclass[12]. These recursive constructor calls must be the first calls of the constructor. As the other constructors may have probes of their own if they are part of the test search, and any statements including method calls and branch predicates in these other constructors are executed before the first constructor, the proper sequence of control flow might be misrepresented based on the probe calls.

Instrumenting branches consists of inserting a probe as the first statement in the code block representing that branch.

In *if-then-else predicates*, the else-part is not mandatory and will be omitted in the code when it is not needed, i.e. there is no code to be executed. Depending on the implementation of the instrumenting program, an else-part can be made explicit and a probe inserted, without disturbing the logical structure of the code. This way, the previously implicit execution of the branch can be registered.

In loops, the first statement inside the loop body is a probe. A lack of call from this probe means the loop condition was not fulfilled. The procedure used to signify the execution of the post-loop code is to insert a probe immediately after the loop body.

Switch-statements can be instrumented by inserting probes in each of the "case"-blocks.

Conditional expressions, of the type "$\text{Predicate ? then}-\text{expression : else}-\text{expression}$", also known as *ternary expressions*, are essentially in-line if-then-else statements. In these expressions, the branches consist of the then-expression and the else-expression. Since these branches contain one and only one expression each, inserting probes become a bit more challenging (see 2.6.2).

2.4 Transforming code

Transformation of code means, in the most simple sense, modifying existing code, either manually or through the help of a script or an application. Transformation could be anything from inserting probes, as in instrumentation, to completely rewriting the source and getting essentially another program.

In the context of search-based test data generation, transformation is used in order to make generation of test input possible, easier and / or more precise.

2.4.1 Semantic equivalence

Semantic equivalence can be defined to mean that two programs or units of code produce the same output when given the same input. This must hold for the lifetime of the units since the two programs may develop different internal states which may alter any following output.

When generating test inputs using transformed versions of the programs, one needs to ensure that the tests produced are applicable to the original programs.

This can be achieved for example through ensuring semantic equivalence of the branch predicates between the original and the transformed versions.

One way of achieving this is to logically analyze the transformations. Another approach is an empirical investigation. An original and a transformed version of a program could be subjected to the same test inputs, and the outputs be checked to see if they are identical. Any exceptions or errors incurred in one must be matched in the other, given that no programming errors were present in the original.

If done over enough programs and enough inputs, our faith in the semantic equivalence of the original and transformed versions would be strong.

Testability transformation (*TeTra*), coined by Harman *et al.* [15] is another strategy. It is a program transformation designed to make testing more efficient, or enable testing altogether for a given program. Testability is defined as "the ease with which test data can be generated". One example of this is to remove code that has no influence on the given branch to cover with the test.

TeTra does not intend to preserve the original program or its functionality , but it does require that the tests generated for the transformed version are *adequate* for the original. If the test adequacy criterion is a certain branch coverage, the generated test values must fulfill this when given as input to the original, but the transformed version does not even have to contain the same branches. When removing code or restructuring it for example, some edges in the control structure may be removed and others may be added.

The rationale for not demanding full functional or semantic equivalence is that it can be counter-productive when generating tests. Perhaps not all structures have a sufficiently good transformed version, or the transformations are complex and / or resource-consuming.

The transformed program is discarded generating the tests, and is thus never used for the actual testing itself. So if the transformations are incorrect, the test data just fail to give full coverage or, put in general terms, fail to fulfill the test criterion. In the case of code coverage, this is easily verified through analyzing the calls from inserted probes, or by using commonly available code coverage tools. On the other hand, should the transformed program be used for testing, i.e. replacing the original, one could not know whether the transformation was correct.

To illustrate with an extreme example, a transformation could indescriminately delete all but one block of code from a program. Generating test input to satisfy full coverage of this one code block would be easy, and if using the transformed version of code for testing, you would call it a success. It would of course not be good enough for the original.

A challenge to the procedure of TeTra is among others in proving that the transformations are adequate, i.e. that 100% test coverage in the transformed version is guaranteed to be 100% in the original when using the same input. Currently, this is an empirical assessment, as no formal analysis techniques are found.

Also, taking internal state of a program in consideration may be a problem when using Testability Transformation, as certain transformation procedures may remove or change code structures that ultimately influence the state. As such, testing for example object-oriented programs, where encapsulation and internal state may be often used, may be a problem.

2.5 Identified research problems

2.5.1 Lack of common benchmark

Arcuri and Yao report on the general lack of a common benchmark cluster to evaluate new techniques against preexisting, already tested techniques. The only commonly used method is to compare novel techniques against random search [4].

2.5.2 Scalability

Arcuri and Yao [4] say that scalability is a factor that has not been suffiencently considered. It is unknown whether the algorithms described in the literature will be sufficient to use on industrial-size software.

2.5.3 Flag problem

A flag variable is a boolean variable, and consequently either *true* or *false*. Flag variables are problematic because of a poor guidance of the search at predicates containing flags.

Flags could be assigned as true or false as a result of the outcome of a predicate. These predicates, instead of the flags themselves, have in some attempts at solving the flag problem been used in determining the branch distance, resulting in a smooth guidance. However, according to McMinn *et al.*, constant true or false values, rather than assignments, are more common for flags [18].

2.5.4 Nested predicates

Predicates in control flow statements that are nested inside other control flow statements will not be satisfied until after the enclosing predicates are satisfied. In other words, in order to preserve the execution path to the inner predicate, the inputs that were found to satisfy the outer predicate must be preserved [19]. This makes finding inputs to satisfy inner predicates harder for each level of nesting.

McMinn *et al.* [19] attempted solving a case of the nested predicates problem using testability transformation, whereby the predicates were transformed in a way that evaluated all at the same time. The procedure was promising for predicates where the variables were independent of each other.

2.5.5 Features and problems with object oriented programming

Encapsulation involves hiding internal methods and state, and only making some methods accessible. All branches, even in private methods, must then be covered by using the public methods, making search much harder.

Some branches may only be covered if the program is in a certain internal *state*. Often the variables in question are not accessible, and the state must be manipulated inderectly, perhaps by using only public methods.

Polymorphism is widely used in OO software. It allows an object to be referred to as any of its inherited or implemented classes. This potentially makes the search space much larger, as a Java method that takes an object of a given class as input will also accept any object of a subclass [4].

A class under test will often *extend* another. Methods and fields inherited from the superclass will be implicitly available, but not necessarily in source code. The automated search of the class will be more difficult [4].

2.5.6 Open problems and challenges for SBSE

Harman [14] identifies a small number of challenges for general SBSE.

Stopping criteria. Some of the algorithms require some criterion to be met in order to halt the search. More effectice criteria would increase the power of searches. Where the usual candidates are reaching some goal such as surpassing a certain value, or using up some allocated time or computation resource, Harman sugguests a third alternative in the case of genetic algorithms. When the population has become homogenous, there is little opportunity for improvement. Of course, how to measure homogeneity and define acceptable ranges is open for debate.

Visualization. Mapping the fitness landscape in a problem with more than two dimensions (i.e. more than two genes) can be a challenge. A possible procedure is to map all dimensions into a flat plane, in order to see clusterings of peaks, for example.

Characterizing search spaces. There has been little attempt to map a problem class, a generalization of a problem, to a fitting algorithm. Each author has used a "favorite" algorithm, or tried out a new, and possibly compared it to random search. Harman wants more analysis of fitting algorithms for various software engineering problems.

Human competive results. The "Humies" are held annually by the The Genetic and Evolutionary Computation Conference (GECCO). It awards mechanically evolved programs that equal or surpass results from humans. This has not been attempted by an SBSE project, and Harman thinks it might be because the field handles problems with no defined best solution. He does think that such a prize would have great benefit for the field, and that it will happen as a result of the growing interest and activity.

2.6 Problems addressed in the thesis

The following selection of problems and challenges, and the attempt to answer them, will be the main focus for the remainder of the thesis.

2.6.1 Treatment of instrumentation in litterature

In the research litterature, there is little if any detailed examples on how to perform instrumentation and / or transformation for branch distance calculations.

There is also little to no instructions on how to carry out effective instrumentation on the different control flow statements - as each type of control flow statement requires a somewhat different approach.

2.6.2 Instrumentation of conditional expressions

Conditional expressions, of the type
"$\text{Predicate ? then}-\text{expression : else}-\text{expression}$", is as stated a more challenging control flow statement than the simple if-statement, although they share an important attribute. They both have a branch predicate, a then-part and an else-part. However, the conditional can be used in-line, and can only contain expressions [12]. This makes one more bound in the treatment and instrumentation of the source, as for instance probe calls or distance calculations cannot be inserted in the same manner they can be inserted in a regular if-then-else statement.

A discussion of the instrumentation of conditional expressions has not been seen in the literature.

2.6.3 Branch distance calculations

Determining how to instrument and transform predicates in order to make the necessary run-time branch distance calls is a key focus of this thesis.

Whether contained in an if-then-else construct, a while- or for-loop, or a ternary expression, the predicate is an expression that evaluates to true or false, i.e. a boolean expression.

It is critical that any transformation in these cases preserve the semantics of the predicate completely, so that the original and the transformed versions are logically the same. Otherwise, the tests generated with the help of the transformed version may cause different execution paths to be taken in the original version.

The predicates can be composite and nested, with an arbitrary number of sub-predicates. A challenge lies in finding general rules that can cope with any number of sub-predicates.

The predicates and sub-predicates are made up of a number of building blocks, namely Java expressions. While some expressions evaluate to boolean values, others require combinations to produce boolean values, and thereby be considered a predicate. Such combinations are possible through operators, either logical or arithmetic. Logical operators can be '==', '&&' or '<' or similar, while arithmetic are the adding, multiplication, division and subtraction symbols. Again, rules for every case should be found.

Although tables of branch distance calculations have been published, no detailed recipe for automatically or manually transforming predicates to calculate such distances have surfaced. An attempt to show such a procedure will be made in this thesis.

As an additional demand when transforming the predicates, it is desirable to calculate the distance for *both* branch distances in a predicate — not only the distance the predicate is from being **true**, but also the distance the predicate is from being **false**. The reason is that the target node for the search may reside in either branch, and having both values for the branch distances calculated at the same time can be beneficial when using more sophisticated search strategies. These calculations will both have to be made using the same call, as more than one call may induce state changes and side effects that were not intended in the original code.

The various control flow statements require a somewhat specialized approach when instrumenting and transforming. For example, in an if-then-else statement, the predicate would be replaced with a distance calculation compared to zero. If the branch distance is zero, it means that the predicate is true, and the comparison returns the boolean value **true**. The important objectives are met, in that the predicates are acting the same as the original predicates while at the same time reporting the branch distance.

Loops are somewhat different from the if-then-else statement. The condition is, as stated, of the same format as in the other control flow statements. However, in the case of for- and while-loops, the condition is checked a minimum of one time, and potentially many times. A choice must be made of when to calculate and report the branch distance — either the first time it is encountered, the last time or every time.

The conditional expression is another control flow statement that to my knowledge is relatively untreated in the litterature with regards to instrumentation and transformation. Although these expressions differ from other control flow statements in that they can be nested in other predicates and / or be used in assignments, they should still be treated as a source of information to inform the fitness function and be instrumented in order to generate traces and testinputs.

The challenge lies in the fact that only one expression can exist in either of the "then" and "else" branches. A solution must be found for instrumentation and branch distance call generation which preserves the property of being in one line only.

2.6.4 Compound predicates

Compound predicates are conjunctions or disjunctions (P AND Q, P OR Q) of predicates. Compound predicates consisting of more than two "subpredicates" are possible. These are treated as nested within each other, such that no more than two predicates are compared at a time. In "(P AND Q) AND R", for example, P and Q are first compared, and then R is compared to the boolean value of the first comparison.

In terms of calculating the branch distance, compound predicates are a bit more tricky than singular predicates. In the case of the conjunction, if the first predicate is evaluated to **false**, the second will normally not be evaluated at all, since the boolean value of the compound is already decided. Similarly, if the first predicate in a disjunction is evaluated as **true**, the last predicate in the compound will not be evaluated. The AND and OR operators ("&&" and "||") can be said to be "short-circuiting". This is in contrast to the boolean logical operators "&" and "|" [12], which are not short-circuiting, and will always evaluate both predicates. The very first (sub-)predicate in a compound predicate will always be evaluated. For example, in the predicate "P AND Q", only P is guaranteed to be evaluated. In "(P AND Q) OR R", P is still the only one guaranteed to be evaluated.

The predicate guaranteed to be evaluated will from now on be referred to as being or being in the *early / first part of the predicate*, while the remaining will be referred to as being in the *second / later part of thepredicate*. Short-circuiting hinders test input search. Conjunction predicates have similarities with nested control flow statements (see 2.5), in that only one predicate is being satisfied at a time. The test inputs find a solution to the first predicate first, and only when this is satisfied, the search for a solution to the next may start. This must be found while still not violating the solution from the first. Baresel *et al.* [6] point out that "Whenever an individual is found that fits one more atomic condition, the probability of finding a solution which also fits the next one decreases considerably."

In the case of disjunctions, when the first predicate is true, the second is not evaluated. The branch distance for the **true** branch is trivially zero. The branch distance for the predicate being **false** is then calculated on the first predicate alone. Here as well, both predicates shouldideally be considered. A remedy to this situation would be to calculate the branch distances for both predicates at the same time, and let the total value be decided by the sum in the case of conjunctions, or the minimum value in the case of disjunctions. This is the formula preferred by among others Tracey *et al.* [25]. Baresel *et al.* [6] say that enabling all predicates to be evaluated in spite of short-circuiting operators is a preferred solution, but may not be possible because of side effects.

For the same reason it is necessary to calculate the branch distance once and only once in the case of singular predicates, it is necessary to take care when calculating the branch distance of a predicate that would normally not be evaluated. Any evaluation might cause unwanted state changes (see 2.6.5). To complicate matters further, predicates not normally evaluated might cause exceptions (see 2.6.6).

In the example " `P && x++ == 2`", the value of `x` is incremented if both predicates were evaluated, regardless of the truth value of the first predicate. This constitutes an unwanted side effect.

Arcuri and Yao[3] question the lack of treatment on this issue. They think that the "[o]nly answer could be that so little work on OO has been done, and on very few classes". Further, "[t]his shows a big problem with current (OO) testing: test clusters are too small".

McMinn *et al.* [19] did acknowledge the problem of possible exceptions arising from unduly evaluating predicates, and did find such predicates in an empirical study. However, they did not discuss any solution, as those predicates were found in control flow statements they did not consider for their methodology at the time.

To circumvent the side effects and exception-causing statements, one can substitute them with approximations, or *heuristics*. Finding rules to decide when side effects can be avoided and when the predicates need to be substituted and what to substitute them for, will help in gaining more information for the test input search.

The heuristics may well not be totally correct, it is enough to approximate the branch distance that would have been computed in the regular case. The information gained is still much better than nothing, and will help guide the search. Using all information possible will help make a better fitness function [6].

2.6.5 Side effects

Harman *et al.* [16] define side effects as "any state change caused by the evaluation of an expression. A side-effect free expression, when evaluated simply returns its value, causing no change in state". Examples of side effects can then be assignments, increments and decrements of numerical variables and counters, construction and destruction of objects and so on.

In calculating branch distance of predicates, it is necessary to avoid evaluating the predicate or parts of the predicate more than once, precisely for this reason. Any part of the predicate with side effects will cause this side effect to change the program state more than the intended number of times.

The same goes for side effects in the later parts of a compound predicate. If the second part of a predicate would not be evaluated, the side effects would not be executed. When defining rules for heuristics to replace regular branch distance calculations, the main goal is to avoid executing these side effects.

Purity analysis is a field of research regarding the safety of methods and the detection of externally visible side effects [24].

When method calls are present in the predicates, the presence of side effects must be expected. Methods can cause any number of side effects, and analysis to ascertain this may be impossible due to inaccessible code. Even when the code is available, checking for side effects is not trivial.

Further exploration of this field of research is not within the scope of the thesis.

2.6.6 Exception-causing factors

In evaluating later parts of compounds, care must also be taken to avoid throwing exceptions. An example is seen in compound predicates where the first part is a null-check of a dynamic memory reference, and the second part accesses the object in this reference. If trying to access the objects when the reference is null, an exception is thrown.

The first part of a compound can also be used to check if some value is within acceptable ranges, and then use this value in the second part. Failure to check if a value is zero could result in an exception because of division by zero [19]. Also, an exception could be thrown if trying to access an array index out of bounds.

3 Empirical analysis

3.1 Test setup

To get an overview of the frequency of the different branching statements, as well as the frequency of the expression types in the predicates, statistics were gathered from the classes of a selection of real-world programs.

3.1.1 Purpose of analysis

The relative frequency of the different branching statements is interesting, as well as the frequency of the different expression types. Whether the expressions occur before or after the compound binary operators "&&" and "||" is also of great interest. This will allow us to see which rules for branch distance heuristics will have the most impact.

3.1.2 Test requirements

For the findings to be relevant for and generalizable to Java code used in real-world programs, it is desirable to analyze a sample of such programs. It is also desirable for reliability purposes to analyze programs for which source code is openly available. Considering these requirements, open source software projects seem to be fitting candidates.

3.1.3 Choice of test population

For our open source programs, the Software-artifact Infrastructure Repository [5] has proven very useful. Here, the source code from a selection of Java programs ranging from the very small (13 lines of code, 1 class) to the very large (503,833 lines of code, 1967 classes) is available upon request [9]. Along with the source code, test cases and experimentation frameworks, as well as versions of the program code seeded with faults are available.

Ten of the programs were chosen, ranging from a stated 838 lines of code to a stated 503,833 lines of code. The intent was to capture a range of programming styles due to plausible variation of programmer preference and project guidelines between the programs.

3.2 Execution of the test

A Java program was given the location of the files to test. Each Java source file found in that location was parsed using *javacc* (see 4.2.7). Files containing interfaces or annotation definitions were ignored. Due to limitations in the version of *javacc* currently used, such as inability to parse some newer language features, analysis was aborted for a small number of files.

The expressions in the syntax tree produced by the parsing were counted using a visitor class (see 4.2.2) traversing the tree. Each occurrence of a branching statement (*if-*, *while-*, *do-while-*, *for-*, *conditional* and *switch* statements) were counted. The number of predicates containing different expression types and operators were also counted, and a distinction was made between expressions occurring before or after the compound operators "&&" and "||". The statements determining control flow in *switch* statements were not counted, as these are of a different format than the conditions of the other branching statements.

3.2.1 Overview

In Table 3, the number of lines of code and the stated number of classes refer to numbers that are stated on the short "bio" of the programs on the SIR website. As the software repository typically offer multiple versions of a program, these numbers presumably refer to one specific version of a program. Some programs include a number of test classes, which may constitute a large number of classes. This is the case in for example *jboss*, where the stated number of classes is about two thousand, and the number of analyzed classes is over four times that amount. Also, being a component library, *nanoxml* is offered as a bundle together with three different applications using it, which are all analyzed. The stated number of classes may relate to only the component. Finally, the *lines of code* refer to code and comments in all Java files. Since only regular classes that the version

Table 3: Size of test population

	Stated lines of code	Stated number of classes	Total files (*.java) analyzed	Analyzed files	Ignored files	Files with aborted search
ant	80 500	627	904	828	71	5
daisy	883	22	15	15	0	0
deos	838	24	24	24	0	0
Derby	503 833	1 967	2 235	1 880	307	48
jboss	116 638	1 126	8 490	6 137	2 333	20
jmeter	43 400	389	389	340	42	7
jtopas	5 400	50	50	30	20	0
nanoxml	7 646	24	51	37	8	6
siena	6 035	26	22	17	5	0
xml-security	16 800	143	145	128	17	0
TOTAL	781 973	4 398	12 325	9 436	2 803	86
				76.56%	22.74%	0.70%

of *javacc* is able to parse are analyzed, the actual lines of code analyzed is somewhat reduced. Regardless, the stated lines of code give an indication of size.

3.3 Statement, expression type and operator distribution

3.3.1 Comment on branching statement distribution

In Table 4, the vast majority of branching statements (80%) are *if-statements*. The branching statements not considered for instrumentation in this thesis, the *do-while* and the *switch* statements, have the smallest counts, with the *do-while* making up only 0.2% of the branching statements.

3.3.2 Compound predicates and null-checks

In Table 5, predicates with at least one compound operator number 7245 in total. Out of a total of 78,012 branching statements analyzed, this constitutes 9.3%

Table 4: Branching statement distribution

	If-then-else	Switch	Cond-itional	While	Do-While	For	TOTAL
ant	6 862	30	287	467	22	884	8 552
daisy	71	2	4	9	0	36	122
deos	69	1	2	1	0	6	79
Derby	28 483	994	1 248	1 056	100	3 593	35 474
jboss	23 778	234	828	1627	23	2 982	29 472
jmeter	1 552	6	20	235	1	195	2 009
jtopas	435	37	26	32	6	22	558
nanoxml	155	2	3	36	0	27	223
siena	157	35	5	35	5	14	251
xml-security	972	28	26	49	1	196	1 272
TOTAL	62 534	1 369	2 449	3 547	158	7 955	78 012

Table 5: Number of compound predicates, and number of null-checks and later use

	Two sub-predicates	Three sub-predicates	Four sub-predicates	Five+ sub-predicates	Null-checks and later access
ant	938	119	34	18	297
daisy	13	0	0	0	0
deos	1	0	0	0	0
Derby	2 585	397	127	76	709
jboss	2 104	209	60	43	999
jmeter	178	23	2	2	44
jtopas	57	11	8	3	20
nanoxml	11	2	0	0	3
siena	22	8	4	2	2
xml-security	150	27	6	5	65
TOTAL	6 059	796	241	149	2 139

Compound predicates where an object reference was null-checked and later accessed were found in a total of 2139 predicates. This means it is relevant for 29.5% of all compound predicates.

Table 6: Number of predicates containing stated expressions after "&&" or "||"

	Array Access	Assign-ment	Field Access	Instance-of	Method Call
ant	33	3	76	11	720
daisy	3	0	4	0	3
deos	0	0	1	0	0
Derby	155	16	691	85	1 701
jboss	74	5	401	187	1 412
jmeter	12	5	20	16	120
jtopas	13	5	8	0	51
nanoxml	0	0	2	0	10
siena	13	0	14	0	13
xml-security	5	0	73	0	134
TOTAL	308	34	1 290	299	4 164

3.3.3 Prevalence of predicates with exception-inducing expression types

In Table 7, we see that the *division* operator, potentially causing an exception if the divisor is zero (see 2.6.6) has a relatively small presence in the later parts of a compound predicate. 15 such predicates were found. The *remainder* operator, potentially causing the same problem, was actually found in 13 predicates, although 9 of those were found in the same program.

Expressions prone to causing null pointer exceptions are far more plentiful. Field access expressions, array access expressions and the *instanceof* operator were found in a total of 1897 later sub-predicates in compound predicates, as seen in Table 6.

Table 7: Number of predicates containing stated operators after "&&" or "| |"

	pre-increment	pre-decrement	post-increment	post-decrement	division	remainder
ant	0	0	3	0	1	1
daisy	0	0	0	0	0	0
deos	0	0	0	0	0	0
Derby	5	0	0	0	5	9
jboss	1	1	3	3	5	0
jmeter	0	0	0	0	0	0
jtopas	0	0	0	0	2	0
nanoxml	0	0	0	0	0	0
siena	1	0	0	0	0	0
xml-security	0	0	0	0	2	3
TOTAL	7	1	6	3	15	13

3.3.4 Prevalence of predicates with side-effect-causing expression types

Increments and decrements in later parts of compound predicates are also rather rare. 17 cases were found (Table 7).

Assignments as parts of predicates are used a bit more often, for a total of 34 found in later predicates (Table 7). These assignments are of the form $((x = y) > 0)$. None of the other assignment operators, such as "+=" were used.

Method calls are found the most often of all the problematic expression types, in fact over twice as often as the others combined (Table 6).

Some object creation expressions, causing the same problems as method calls due to the implied constructor calls, were also found in Derby and jboss, contained in a total of 8 predicates late in compounds.

3.3.5 Nested conditionals

Conditional expressions inside other branching statements were found in a total of 69 predicates before compound operators, and in two predicates after. This nesting means that the outside predicates were decided based partly on the outcome of the inner conditional.

3.3.6 Non-typical predicates

Some rather marginal expression types were found in the larger programs. In Derby, anonymous classes were being defined and used as parameters in method calls. However, parameters are not part of branch distance calculations, and can be safely ignored.

4 Method

Part of the thesis consists of constructing a program to provide proof-of-concept for various instrumentation and heuristic rules. Of course, it also provides a means to experiment, explore and test theories and solutions.

In this part, I will first sketch out the requirements of the program. Then I will explain how these requirements were met, and give a brief overview of the execution of the program. Some detail is provided for main features of the program. Lastly, important problems and proposed solutions are explained in some detail.

4.1 Requirements of the program

4.1.1 Choice of files for instrumentation

The program should accept one or several files for instrumentation. Depending on test setup and dependencies to other classes, sometimes one file is all that needs to be instrumented. In other cases, several files might have to be instrumented and/or loaded in the Java Virtual Machine in order to access necessary information about fields and types referred to in the class being instrumented.

4.1.2 Accommodating search-based test data generation

The information gathered and generated by the probes and instrumentation while the instrumented class is under test needs to be accessed.

4.1.3 Statistics

For debugging and logging purposes, various statistics about the instrumentation must be available. Especially important in this project is the frequency of different expression types, and how many of them were handled and instrumented properly by the program. The files and predicates that could not be transformed properly should be listed.

4.1.4 Instrumenting control flow statements

Ideally, the program should instrument every type of control flow statement, but prioritizing the most common or interesting is necessary.

4.1.5 Semantics

It is vital that the control flow statements have equivalent semantics in the instrumented and original versions. This ensures the generated tests are useful for the program under instrumentation.

4.1.6 Heuristics

As mentioned in 2.6.4, whenever a short-circuiting operator prevents both parts of a compound predicate to be evaluated at the same time, we lose valuable information. The solution is to adopt a conservative heuristic for handling these situations. If the evaluation of the first predicate normally leads to the other not being evaluated, we will utilize rules that generate heuristics instead of regular evaluations. For instance a look-ahead in the code to see if the predicate is safe or without side-effects, or a transformation of parts of the code that will approach or simulate the value of the regular predicate without causing side effects.

4.2 Implementation of the program

4.2.1 Quick overview

The name of the program is "*VIns*", for "*Verde Instrumentor*". The user of VIns supplies a Java source file or collection of source files through a command-line interface. The source code is pre-processed so that names and types of fields, superclasses and references are saved for later processing. Using *javacc* grammar, (see 4.2.7) an abstract syntax tree of the source of the class to be instrumented is built.

VIns is implementing the "visitor" pattern [29]. Several visitors, each one with its own purpose, process and alter the source. This might in-

Listing 4: Steps of the instrumentation and transformation procedure

```
1. User starts VIns through the command-line, specifying file(s)
to instrument.
2. Files are pre-processed, and names and types of fields are
stored for later use.
3. For each Visitor, the file is parsed, manipulated and saved.
Predicates are manipulated through analyzing the constituent
expressions, and transformed with relation to rules for each
expression type.
4. Logs detailing the transformation are created.
```

volve for example instrumenting methods and control flow statement with probe calls (see 2.3). The final visitor will insert needed fields and method implementations from the “Instrumented” interface.

4.2.2 Visitors

When employing the visitor pattern [29], *visitor* objects are used when traversing, or visiting, the nodes of an abstract syntax tree (AST). They are usually altering or doing some form of computation on the values of the nodes in the AST. In this project, and in the helper programs used (see 4.2.7), the role of the visitors is to print source code associated with the nodes in the abstract syntax tree. In our case, relevant structures are altered or transformed in certain ways. The procedure chosen for this project is to use one visitor for each task we want to perform on the source code. One for changing the package name and import statements, one for changing the class name, one for instrumenting and transforming if-statements, and so on. The source code is run through a parse-and-visit cycle for each visitor we add.

A structure such as this helps separate the concerns of relatively unrelated tasks, and makes adding or removing visitors easy, making it customizable to whatever goals one has with instrumenting.

An important visitor which is run first of all is the MetaInfoVisitor. This does not or alter any code, but it gathers meta-information about the class under instrumentation, to be used by other visitors. The types and names of fields, super-classes and imported classes are some examples.

4.2.3 The anatomy of instrumented classes

Instrumented classes have certain features that determine behavior during test generation. An example of these is seen in Listing 5.

Interface methods and fields The "Instrumented" interface, which every instrumented source code implements, constitutes the link between the source and the testing framework, allowing extraction of statistics about the execution of test cases.

The interface stipulates a number of methods to be implemented. Some return simple statistics of the class, such as the number of branches or methods found in the class, while the bulk deals with the *execution trace*.

delta, *max_time* and *setTimeProperties()* are properties for use by testing frameworks using VIns.

ExecutionTrace The class ExecutionTrace is the main link between the instrumented class and the testing framework, through which branch coverage and branch distance information is conveyed at run-time, and which is queried for information about these distances and the trace after executing a test case.

DistanceCalculator The DistanceCalculator class is added as an import for all instrumented classes. It contains static methods for calculating branch distances, which in turn are fed as parameters to calls to the execution trace.

4.2.4 Functionality during run-time

Once the source code is instrumented, it can be used for test generation. As the code is semantically equivalent to the original, the tests generated can be used in both versions.

Listing 5: example of code implemented in an instrumented class

```

import vins.Instrumented;
import vins.ExecutionTrace;
import vins.distance.*;

public class Ins_BranchExample implements Instrumented {

 static public ExecutionTrace ET_;
 private long ET_delta, ET_max_time;
 public int getNumberOfBranches(){ return 56; }
 public int getNumberOfMethods(){ return 11; }
 public void resetExecutionTrace() {
        getExecutionTrace().reset(
                        getNumberOfMethods(),
                        getNumberOfBranches(),
                        ET_max_time,
                        ET_delta );
        ET_ = getExecutionTrace();
        }

 public ExecutionTrace getExecutionTrace() {
        return ExecutionTrace.getTrace();
                }

 public void setTimeProperties(long max_time, long delta) {
        ET_max_time = max_time; ET_delta = delta;
        getExecutionTrace().setUsingTime(true);
        }
.
.
.
}
```

When running the code in a test, whenever a method or branch is executed, calls from the probes go to the execution trace, which keeps track of the branches taken. Also, when evaluating a predicate, a call to the execution trace sets the calculated branch distance in both branches, regardless of the outcome of the predicate. The original predicate is replaced by a check to see if the branch distance equals zero, which signifies that the condition occurred.

By querying the ExecutionTrace, testing frameworks get access to the branch distances and which branches were executed for a given test input, informing the search algorithm's objective function.

4.2.5 Branch distance calculations

Following the strategy used by Tracey *et al.* [25], whenever a branching construct is encountered in the code under instrumentation, VIns will generate a code snippet consisting of a call to the distance calculator to be called at run-time. The snippet will replace the original predicate. This call to the distance calculator will simultaneously calculate and store the branch distance of the predicate, as well as determine the truth value of the predicate, thereby determining the control flow.

An array of two numbers are generated in the distance calculation call, one distance measure for each outcome of the predicate. Depending on where the target branch lies, each of these numbers can be needed. This means that both the branch distance as well as the branch distance of the negation of the predicate are calculated in the same distance calculation call. In doing this, we prevent calling statements more than once, preventing possible side effects. As one of the two branches will be taken, one of them will be 0.0.

Making the calculations through a call, rather than adding them directly into the source, accomplishes a few things. Firstly, cluttering the code is avoided. Although this should not pose much of a problem, since only the computer will normally read the code, it will help debugging and maintaining the program. Secondly, it allows the actual implementation

Listing 6: Transformation of if-statements

```
//DC = DistanceCalculator

if( firstInt < 10 ) { /* statements */ }

        ⇓

double[] dist_00 = (DC.distLess(firstInt , 10));
Ins_ExampleClass.ET_.setDistance(00, 01, dist_00);
if (dist_00[0] == 0) {
 Ins_Example.ET_.setExecuted(00);
 {
        /* statements */
 }
} else {
 Ins_Example.ET_.setExecuted(01);
}
```

of the distance calculator to be changed, which allows for improved calculations later.

4.2.6 Instrumentation of branching constructs and methods

The Java programming language has a number of different constructs that, when evaluated, determine the flow of a program. In VIns, they differ in how or whether they are instrumented and treated in regards to branch distance calculation calls.

The basic approach consists of the introduction of a line of code situated above the actual predicate. This line of code is the call to the distance calculation. The actual predicate is then replaced with a simple check to see if this distance is equal to the value "0.0". This value means there is zero distance, which means that the predicate value is "true".

If-then-else The basic *if-then-else* construct follows the template depicted above.

While loops The *while loop* consists of the body of statements and the condition, determining whether the loop will run an(other) iteration. A boolean variable called "is_first" is introduced, which will determine

Listing 7: Transformation of while loops

```
//DC = DistanceCalculator

while( firstInt < 10 ) { /* statements */ }

        ⇓

double[] dist_00 = (DC.distLess(firstInt , 10));
Ins_ExampleClass.ET_.setDistance(00, 01, dist_00);
boolean is_first_00 = true;
while(is_first_00 ? dist_00[0] == 0 : firstInt < 10) {
 is_first_00 = false;
 Ins_Example.ET_.setExecuted(00);
  {
        /* statements */
  }
}
Ins_Example.ET_.setExecuted(01);
```

which of the original and instrumented predicates will be evaluated in the head of the loop. The branch distance should only be calculated once, hence the instrumented predicate will only be evaluated the first time.

For loops The *for loop* is instrumented much in the same way as the while loop, with an "is_first" variable introduced. Additionally, the initialization variable is moved outside of the loop head and initialized here. To preserve the correct scope for this variable, and to avoid name space problems, additional curly braces surround the initialization variable and the loop itself.

Conditional expressions Logically, conditional expressions are if-then-else-statements in one line. However, being "in-line" means that they cannot be instrumented over several lines and still guarantee semantic equivalence with the original predicate. The several lines of instrumentation for the if-then-else construct must be condensed into one line, while preserving the property of expressions that they can be nested into other expressions and statements. The instru-

Listing 8: Transformation of for loops

```
//DC = DistanceCalculator

for( int i = 0; firstInt < 10 ; i++) { /* statements */ }

        ⇓

{
 int i = 0;
 double[] dist_00 = DC.distLess(firstInt , 10);
 Ins_ForExample.ET_.setDistance(00, 01, dist_00);
 boolean is_first_00 = true;
 for (; is_first_00 ? dist_00[0] == 0 : firstInt < 10; i++) {
  is_first_00 = false;
  Ins_ExampleClass.ET_.setExecuted(00);
  {
        /* statements */
  }
 }
  Ins_ForExample.ET_.setExecuted(01);
}
```

mented conditional must thus remain an expression while executing its normal functions and reporting and calculating the branch distance.

The one line of the instrumented actually condenses the instrumentation seen in the if-statement. The "setDistance()"-method both reports the distance, sets what branch was executed, and returns "**true**" if the first branch distance is 0.0.

Not considered The do-while loop is very similar to the regular while-loop, the only difference is that it is always executed at least once. This loop is fairly uncommon in use (see Table 4), and for this thesis it is ignored.

The switch-statement is also ignored in this thesis due to little use compared to the other branching expressions. Logically, if need be it could be transformed into a series of if-then-else statements.

Instrumentation with unrecognized predicates Whenever VIns

Listing 9: Transformation of conditional expression

```
//DC = DistanceCalculator

(firstInt < 10 ? then : else)

        ⇓

Ins_ExampleClass.ET_.setDistance(
        00, 01, DC.distLess(firstInt, 10))
        ? then
        : else;
```

Listing 10: Instrumentation of methods

```

Public String aMethod() { /* statements */ }

        ⇓

public String aMethod() {
 Ins_ExampleClass.ET_.setCalledMethod(0);
 /* statements */
}
```

does not instrument the predicate, for example in cases where proper transformation for the predicate's expressions is not defined, the original predicate is used as the control flow statement condition. This ensures correct performance even if the branch distance calculation does not function. The execution trace is still maintained by inserting a probe call in the statement body, which will set the branch distance to 1.0 or 0.0 with respectively a false or a true condition. This probe call is inserted by default in most instrumentations. This could be seen as redundant in the cases where proper transformation is defined, given that reporting the branch distances will also signify which path was taken.

Methods Methods are instrumented as part of keeping track of the flow of control during execution of the instrumented classes. A probe is inserted as the first statement in the code block of methods.

4.2.7 Tools used

The abstract syntax tree is generated using *javacc*, and the visitors are part of *JavaParser* [11]. It currently parses code up to Java version Java 1.5.

Janino [26] is used for testing of instrumented source and semantic equivalence. It is a convenient compiler that allows simple compilation-on-the-fly of source code from any place, including memory. It is fully Java 1.4 compatible, but is unfortunately missing key features from 1.5 and later. It still fulfills its role, and is not used for the main functionality of VIns.

4.3 Transformation

4.3.1 Call generation

While the different branch constructs differ in how they are instrumented, the predicates themselves are of the same format for all of the constructs, and the branch distance is thus calculated in the same manner. The predicates are transformed and called in the same way for any control flow statement.

The expressions encountered in each predicate determine what form the branch distance call will have, as there are rules of transformation for each type of expression. Some of the expressions are deemed safe, and / or are representing "non-logic" pieces of the predicate. Variable references, numbers and boolean literal values are examples of this. These expressions are not changed, they are simply included "as is". Others, for example those part of the logical structure, are transformed.

The aim of the transformation is to calculate the branch distance at run-time, while at the same time managing to choose the right path in the branching. The predicate needs to be transformed into a form which accomplishes those two things at the same time. The solution used here follows Tracey *et al.* (see 2.2.4, and involves using the original predicate to calculate a branch distance, and then substitute the predicate in the control flow statement with a comparison between the resulting branch distance and zero. The predicate is true if the branch distance is zero. Which functions to be called are determined by the operators of the expressions, if any. Parameters to the function calls are the expressions from the predicates, or rather what these expressions are evaluated to at run-time.

Expressions under the heading "rules" are expressions that are transformed in a given way. "Safe" denotes expressions that are never instrumented or transformed (due to being primitive building blocks). "Not considered" means expressions that are not found to be part of a predicate in any of the classes investigated.

An example of the transformation of a control flow statement is shown in Listing 11. Line 1 represents the original control flow statement, and the arrow in line 3 signifies the transformation.

The array called "dist_N" in line 5 contains the branch distance values for both branches of the predicate. The "N" represents the index of the first branch in the predicate, numbered from 00.

The branch distance is calculated and assigned in line 6, with calls to static functions in the "DistanceCalculator" ("DC").

In line 8, the branch distances are reported to the ExecutionTrace. The name of the original class is "ExampleClass", and the instrumented class gets "Ins_" as a default prefix. The ExecutionTrace is referenced in the first part of the instrumented class (see Listing 5).

If the predicate would evaluate to **true**, the branch distance for the first branch of the predicate would be 0.0, and this is checked in line 9.

Line 14 to 18 is a abbreviated version of the same transformation. This format will be used for most of the following examples.

4.3.2 Rules

Binary Expression Binary expressions are any two expressions bound together by an operator such as "==" or "<" or "&&". Of special interest are the "AND" ("&&") and the "OR" ("||") operators, which

Listing 11: Transformation of expressions

```
if (x > y && m < n ) { /* statements */ }

        ⇓

double[] dist_N =
 DC.distAnd(DC.distGreater(x, y), DC.distLess(m, n));

Ins_ExampleClass.ET_.setDistance(N, N+1, dist_N);
if (dist_N[0] == 0) {
        /* statements */
}


(x > y && m < n )

        ⇓

DC.distAnd(DC.distGreater(x, y), DC.distLess(m, n))
```

signify short-circuiting boolean expressions. Here, the left side is evaluated normally. The expressions of the right side are checked, and if deemed safe they are also instrumented normally. The entire binary 'is then surrounded with a "distAnd()" or "distOR()" construction. If not safe, the right side and its possible inner binaries must be transformed into heuristics.

```
(x > y)

        ⇓

DC.distGreater(x, y)
```

```
(x > y && someMethod())

        ⇓

DC.distAnd(DC.distGreater(x, y), DC.distTrue(someMethod()))
```

Unary Expression No special instrumentation is done except for unary expressions with the "!" ("not") operator. If the unary expression is the only expression in a predicate, then only a boolean value is possible, hence the operator will be "not". The "!" is replaced with "distNot" call to get a distance call. If the unary is not the sole expression in a predicate, the unary's inner expression is instrumented normally, and the "!" is attached.

```
(!booleanVariable)

         ⇓

DC.distNot(booleanVariable)
```

Method Call Expression Normally not instrumented

Exceptions: When used as lone predicate, surround with a distTrue(). Only a boolean return value is possible in these situations.

```
(booleanMethod())

         ⇓

DC.distTrue(booleanMethod())
```

```
(intMethod < intVariable)

         ⇓

DC.distLess(intMethod(), intVariable)
```

Enclosed Expression Generate the instrumentation for the inner expression, surround with "( )"

Qualified Name Expression Normally not instrumented

Exceptions: When used as lone predicate, surround with a distTrue(). Only a boolean return value is possible in these situations.

Name Expression Normally not instrumented

Exceptions: When used as lone predicate, surround with a distTrue(). Only a boolean return value is possible in these situations.

Boolean Literal Expression Normally not instrumented

Exceptions: When used as lone predicate, surround with a distTrue(). Only a boolean return value is possible in these situations.

Field Access Expression Normally not instrumented

Exceptions: When used as lone predicate, surround with a distTrue(). Only a boolean return value is possible in these situations.

Array Access Expression Normally not instrumented

Exceptions: When used as lone predicate, surround with a distTrue(). Only a boolean return value is possible in these situations.

Super Member Access Expression Normally not instrumented

Exceptions: When used as lone predicate, surround with a distTrue(). Only a boolean return value is possible in these situations.

Conditional Expression Instrumented in own visitor, no other instrumentation later.

Exceptions: When used as lone predicate, surround with a distTrue(). Only a boolean return value is possible in these situations. Rare cases, if used at all, but syntactically possible.

Instanceof Expression Instrumentation of this expression does not produce a call to a dedicated distance function, but instead produces a distance array "in-place" with an added ternary. The only values possible are 0.0 and 1.0 for true and false, respectively.

Ferrer *et al.* [10] propose a distance function for the *instanceof* operator based on the class hierarchy, but due to time constraints this is not treated in the thesis.

```
if(object1 instanceof Object) { /* statements */ }

         ⇓

double[] dist_00 = object1 instanceof Object ?
  new double[] { 0.0, 1.0 } : new double[] { 1.0, 0.0 };
Ins_ExampleClass.ET_.setDistance(00, 01, dist_00);
```

```
if (dist_00[0] == 0) {
        /* statements */
}
```

4.3.3 Safe

- Class Expression
- “This” Expression
- “Super” Expression
- Assign Expression
- Null Literal Expression
- Char Literal Expression
- Double Literal Expression
- Integer Literal Minimum-Value Expression
- Integer Literal Expression
- Long Literal Minimum-Value Expression
- Long Literal Expression
- Cast Expression
- Array Creation Expression
- Array Initializer Expression

4.3.4 Not considered

- Annotation Expression
- Variable Declaration Expression

4.4 Heuristics

4.4.1 Heuristic generation

In the second part of a compound predicate, some of the expressions are transformed so as to provide a measure (heuristic) of the branch distance even though they would not be executed in the program. When generating heuristics for predicates that "should not" be evaluated, certain expressions are again deemed as safe, and these are generally the same as in regular instrumentation (see 4.3.3). Other expressions require analyzing further, to determine whether they can introduce side effects or if their evaluation should otherwise be prohibited or modified.

Of special note, predicates which are containing method calls are automatically ignored, and a default value is assigned as the branch distance. Execution of a method as part of a predicate can lead to an arbitrary number of state changes / side effects. A "safe" method, such as a function without side effects, e.g. a "getter"-method, would pose no problem or harm to a heuristic evaluation. The trouble is figuring out which methods are safe. Although ways exist to assess the purity and safety of methods (see 2.6.5), this lies outside the scope of the thesis, and will remain unimplemented.

In general the procedure in compound predicates with the operator "&&" is to first calculate the branch distance of the first predicate and store the value. If the value is 0.0, i.e. the predicate evaluates to "true", the total distance can be calculated without fear of side effects, as the second predicate would be evaluated in a normal, uninstrumented version of the predicate. However, if the value is bigger than 0.0, the predicate evaluates to false, and due to the short-circuiting nature of the "&&", the second part and its possible side effects would not be evaluated. In this case, we must calculate a heuristic value without causing side effects. The distance calculation call would generally have the form of Listing 12.

The methods *pushDist()* and *popDist()* represent the storage and retrieval of distance values in the framework, so we do not have to calculate any distances more than one time.

Listing 12: Branch distance calculation for &&

```
//DC = DistanceCalculator

DC.pushDist([distance of first predicate]) == 0.0
// if the first predicate evaluates to true
 ? DC.distAnd(DC.popDist(), [distance of the second predicate])
//then calculate the distance for the entire compound
 : DC.distAnd(DC.popDist(), [heuristic of the second predicate]);
// else calculate the distance for the first predicate and
// the heuristic of the second.
```

Listing 13: Branch distance calculation for ||

```
//DC = DistanceCalculator

DC.pushDist([distance of first predicate]) == 0.0
// if the first predicate evaluates to true
 ? DC.distOr(DC.popDist(), [heuristic of the second predicate])
// then calculate the distance for the first predicate and
// the heuristic of the second.
 : DC.distOr(DC.popDist(), [distance of the second predicate]);
//else calculate the distance for the entire compound
```

If no heuristic can be found, for example if none is defined for a given expression, a default value is used. The value "`new double[] { 1.0, 0.0 }`" is then inserted, to give the second part a default distance of 1.0.

The above explanation is for the "AND" (&&) operator. Transformation of the short-circuiting "OR" (||) operator has a similar solution. The first predicate is evaluated and the branch distance is stored. In this case, if the value is 0.0, the first predicate is true, and the short-circuiting would prevent the second predicate from being evaluated. As we are calculating the branch distance for the condition both being true and false, we need to get the heuristic value for the second predicate. So in this case, if the branch distance of the first predicate is 0.0, we extract the heuristic value, and if not, we can evaluate the whole condition normally. The distance calculation call would generally have the form of Listing 13.

As with regular instrumentation, expressions under the heading "rules" are expressions that are transformed in a given way. "Safe" denotes ex-

Listing 14: Example of transformation with heuristic

```
1  if(x > y && secondInt == firstInt++) { /* statements */ }
2
3          ⇓
4
5  double[] dist_N =
6   DC.pushDist(DC.distGreater(x, y)) == 0.0
7    ? DC.distAnd(DC.popDist(), DC.distEquals(secondInt, firstInt++))
8    : DC.distAnd(DC.popDist(), DC.distEquals(secondInt, firstInt));
9
10 Ins_ExampleClass.ET_.setDistance(N, N+1, dist_N);
11 if (dist_N[0] == 0) {
12         /* statements */
13 }
14
15 /* Heuristic part: */
16 (x > y && secondInt == firstInt++ )
17
18         ⇓
19
20 DC.distEquals(secondInt, firstInt)
```

pressions that are never instrumented or transformed (due to being primitive building blocks). “Not considered” means expressions that are not found to be part of a predicate in any of the classes investigated. “Not implemented” are expressions that were not implemented due to time constraints. They are still discussed.

Listing 14 shows an example of transformation involving a heuristic. As only the heuristic (line 8) is different from regular transformation, only this will be shown in most of the subsequent examples, as illustrated with line 16-20.

4.4.2 Division by zero

Whenever a division is carried out in a program, and the divisor has a possible range that includes zero, programmers are expected to take this into consideration and control for zero in the divisor. This is accomplished for example through an outright check of the divisor value, or by some other measure that guarantees a non-zero value.

In "`if (variable > 0 && (othervariable / variable) > 10)`", the first part of the compound does the checking. If care was not taken, a heuristic would happily check both the first and the last predicate at the same time, even though the variable was in fact zero. An exception would be thrown.

Not only explicit tests are used to guarantee non-zero values of the divisor. In "`(methodWithSideEffects() && othervariable / variable > 10)`", the methodWithSideEffects() with a boolean return value could be checking the divisor. Another scenario is that a boolean flag could signify if the division was safe or not. Or the division could be guaranteed safe by design.

It is necessary to be certain that no exception is thrown due to a zero in the divisor. In VIns, whenever a division is encountered, the expression that constitutes the divisor is extracted and a zero-check based on this expression is inserted before the original expression. This will allow or disallow the computation of the second value in the predicate. The check is applied recursively on the divisor expression, so that any nested divisions will be checked as necessary.

4.4.3 Null checks

In cases where an object is referred to in later parts of a compound predicate, the same reference may have been checked for null earlier, or another design or programmatic feature may have checked the reference. In cases where the reference is null, it clearly cannot be accessed, and would not be accessed in the original version of the predicate, unless a programmer error had been made. An example of this would be "`if (object == null || object.field == something)`". To take into account these scenarios, all references to fields of objects must have the objects checked for null.

A number of the heuristic rules depicted in 4.4.5 require that objects be checked for null, for many of the same reasons as division is checked for zero. That is, the earlier, non-heuristic predicates in a compound predicate may have checked or altered the reference to the objects, either explicitly, implicitly or by a side effect. Accessing an object that is null will cause a null pointer exception in Java.

The same process is applied in these cases, as were applied when checking for zero division. A look-ahead is made to see if any expressions require a null-check. Any such null-checks are combined in a single, composite ternary check, in order to allow or disallow the full computation of the heuristic.

4.4.4 Determining which expressions to check

Not any variable can or should be null-checked. If a primitive variable is checked, an exception is thrown. Likewise, when a static field such as "Aclass.afield" is referenced, the check "Aclass != **null**" would cause an exception to be thrown.

In the context of instrumenting and transforming a source file, big challenge is to distinguish between references to objects and references to primitives, and also between instance and static field. The parsing tool used does not distinguish this, so information gained through other means is needed.

When only considering one class, the only real information to be gained is concerning the variables declared within the class itself. We are then missing possible inherited fields. If the class is extending regular Java API classes, we could gain information about inherited fields from these through reflection, as they are available to the JVM.

An additional problem lies in the possible necessity of null-checking the field itself. This should only be done in the event the field is an object, and not a primitive. Through reflection it should be easy to see what type a variable is, but this requires the class to be loaded, together with other classes that the class under instrumentation references. Thus, if using reflection to gather information about references and fields, ideally the entire program under instrumentation should be loaded in memory.

Alternatively, a pre-processing step could help to keep track of the types of fields in the classes of the program. This is perhaps a more primitive method, but a simpler one, and it is the preferred method for VIns. Before instrumentation and transformation is carried out on the classes specified in the command-line, each file is analyzed, and information about fields is stored. When later instrumenting the files, this stored information is accessed, and the correct null-checks are constructed.

If a file was not supplied during the pre-processing stage, and the information is unavailable, an attempt is made to find the class and field with reflection. This would work if the class in question was, for example, part of the core Java API, and thus available to and loaded in the JVM. If the class is not found through reflection, the information is unavailable, and thus no safe null-check can be constructed. The default behavior for VIns is then to not transform the predicate, and instead supply a default value for the branch distance.

4.4.5 Rules

Unary Expression When the unary operator is the "!" (not) operator, the same rule as for instrumentation is used. For pre/post -decrement or -increment, the resulting value is checked, but the assignment is not carried out.

Exceptions: There exist some theoretical situations where the evaluations of unary expressions are dependent on the evaluation of previous expressions (see 6.2.3).

```
(x > y && secondInt == ++firstInt)

          ⇓

DC.distEquals(secondInt, (firstInt + 1))
```

Boolean Literal Expression Same as instrumentation

Method Call Expression Always substitute with a default distance value. If the method is guaranteed no side effects, return normal instrumentation of this construction. However, the analysis of side-effect-free methods is not within the scope of this thesis, and therefore not implemented.

```
if(x > y && intMethod() > z) { /* statements */ }

            ⇓

double[] dist_00 = DC.pushDist(DC.distGreater(x, y)) == 0.0
 ? DC.distAnd(DC.popDist(), DC.distGreater(intMethod(), z))
 : DC.distAnd(DC.popDist(), new double[] {1.0, 0.0});

Ins_ExampleClass.ET_.setDistance(00, 01, dist_00);
if (dist_00[0] == 0) {
            /* statements */
}
```

Field Access Expression If the field is an object field, the entire scope must be null-checked (see 4.4.3). The field itself should be null-checked as well. If the field is a static field, then it is clearly a mistake to null-check the scope, as this is just a class name / class reference. *Exceptions:* If the scope is a method call, or has a method call as part of it, the search for a heuristic is aborted. The method call may have unforeseen side effects.

```
(x > y && Math.PI > 3.0)

            ⇓

DC.distGreater(Math.PI, 3.0)
```

```
(x > y && this.anInt == 1)

            ⇓

DC.distEquals(this.anInt, 1)
```

```
(x > y && this.anInteger < 1)

            ⇓
```

```
firstInteger != null
 ? DC.distLess(this.anInteger, 1)
 : new double[] { 1.0, 0.0 }
```

InstanceOf Expression The InstanceOfExpression is handled mostly the same way as in regular transformation. Arrays and field access expressions must be null-checked (See 4.4.3). A null check is technically not required on the object itself, since "`null instanceof Object`" just returns false, no exception is thrown.

Exceptions: Objects can be returned from potentially unsafe methods (See 2.6.5). InstanceOf-expressions with method calls are thus deemed unsafe.

```
(x > y && getInteger() instanceof Object)

         ⇓

DC.pushDist(DC.distGreater(x, y))
 ? DC.distAnd(DC.popDist(), getInteger() instanceof Object
  ? new double[] { 0.0, 1.0 }
  : new double[] { 1.0, 0.0 })
 : DC.distAnd(DC.popDist(), new double[] { 1.0, 0.0 })
```

```
(x > y && anObject.anotherObject instanceof Object)

         ⇓

DC.pushDist(DC.distGreater(x, y))
 ? DC.distAnd(DC.popDist(),
         anObject.anotherObject instanceof Object
  ? new double[] { 0.0, 1.0 }
  : new double[] { 1.0, 0.0 })
 : DC.distAnd(DC.popDist(), anObject != null
  ? anObject.anotherObject instanceof Object
   ? new double[] { 0.0, 1.0 }
   : new double[] { 1.0, 0.0 }
  : new double[] { 1.0, 0.0 })
```

Enclosed Expression Same as instrumentation. Only the inner expressions are checked, as the enclosement does not confer any change.

4.4.6 Not implemented

Binary Expression Nested binary expression within the later predicate of a compound is potentially complex and not implemented yet, except for expressions with the *division* operator (see 4.4.2).
Exceptions: Possibly problems with side effects supposed to be present in the previous predicate in the binary, similar to the problem in the unary expressions.

```
(x > y && anInt / (anotherInt -1) == 2)

        ⇓

DC.pushDist(DC.distGreater(x, y))
 ? DC.distAnd(DC.popDist(),
        DC.distEquals(anInt / (anotherInt - 1), 2))
 : DC.distAnd(DC.popDist(), (anotherInt - 1) != 0
  ? DC.distEquals(anInt / (anotherInt - 1), 2)
  : new double[] { 1.0, 0.0 });
```

```
(x > y && someMethod())

        ⇓

DC.distAnd(DC.distGreater(x, y), new double[] { 1.0, 0.0 })
```

Conditional Expression Observed in only two cases in the later part of a compound predicate in the empirical analysis, so this expression is not a priority.
Exceptions: Plenty of side effects scenarios possible. Maybe checking each part individually (condition, then-expression, else-expression) will prove manageable.

Array Access Expression Here the issues with field access are multiplied with the issues with side effects in the index expression, for instance method calls or increments.

Exceptions: Access to an array returned from a method, as in "getArray()[0]" is unsafe.

CastExpression Observed in 145 cases late in compound predicates. Some possible complicated scenarios where type checking is done in the first predicate, which must then be taken into account in the latter predicate.

```
(anObject instanceof T &&
        (anotherObject = (T)object).fieldInT == x)
```

Super Member Access Expression Super members must be null-checked if they are objects. They can also be internal classes, which would be a mistake to null-check. Field access problematics thus apply here as well.

```
(super.member != null && super.member == object)
```

Assignment Expression Assignments seem to be dangerous territory because assignments are what we inherently wish to avoid when defining heuristics. However, there are cases with similarities to unary crements, i.e. it is possible to check the resulting value without actually going through with the assignment. "i+=10" can be written as "(i+10)". "(o = variable) != **null**" can be written as (variable) != **null**. Method calls can be part of the assignment, and are as always unsafe.

Class Expression Observed in 114 cases in the empirical analysis. They likely pose no problems.

```
(java.lang.Double.TYPE == double.class)
```

Array Creation and Initializer Expressions Observed in a very few cases in the empirical analysis. They likely pose no problems.

4.4.7 Safe

- “This" Expression
- “Super" Expression
- Null Literal Expression
- Char Literal Expression
- Double Literal Expression
- Integer Literal Minimum-Value Expression
- Integer Literal Expression
- Long Literal Minimum-Value Expression
- Long Literal Expression
- Name Expression
- Qualified Name Expression

4.4.8 Not considered

- Annotation Expression
- Variable Declaration Expression

5 Validity-testing VIns

5.1 Functional testing

5.1.1 Correct transformations

Correct transformation of every possible combination of predicate and predicate expression may be impossible to guarantee, due to the sheer number of such combinations. Through testing during development however, a number of expression types have been thoroughly evaluated. Also, samples of instrumented code from real-world programs have been scrutinized for any inconsistencies.

5.1.2 Compiler pass

Due to the architecture of the program, the source code is parsed several times during the instrumentation and transformation. This includes parsing just before completion of this process, after all predicate transformations have been completed. Any parse errors caused by illegal transformations would be caught.

5.1.3 Semantic equivalence

To test for semantic equivalence(see 2.4.1) between the original and the transformed source code, a Java source file is written, composed of a selection of predicates of different expressions. The source code is instrumented, and the two versions are both compiled using *janino*. Unit tests then assert whether a method executed with the same inputs in each of the versions yields the same output.

5.2 Branch distance validation

The correct automatic calculation of branch distances has been tested alongside developing the application. In a number of tests, instrumented source code has been compiled, and the set of branch distances computed when invoking a methodhas been captured. A set of correct branch distances have been computed manually, and the two sets have been compared.

6 Discussion

6.1 Discussion of empirical analysis

6.1.1 Compound operators and null-checks

With compound operators present in 9.3% of all control flow statements (Table 5), the empirical analysis shows a marked benefit of handling such predicates.

The number of predicates with null-checks is also of importance, since it applies to 29.5% of all compound predicates. Failure to consider these would cause exceptions in many tests.

6.1.2 Program bias

An effect of the diversity of programming styles can be seen in some of the statistics gathered in the empirical analysis. The two largest programs have a pronounced effect on the distribution of expression types (Table 6). Some expression types are even only found in the largest. Even so, ignoring the contribution from both of the two largest programs together actually does *not* show a huge effect on the distribution of control flow statements. The largest changes are that the frequency of if-statements is decreased a couple of percentage points, and the frequency of while-loops is increased a couple of percentage points. That the changes are so small seems to come from the curious fact that the diversities in the programs cancel each other out to a degree.

There is some additional variability between the largest programs. *Derby* uses the *switch* and the *do-while* four times more than *jboss*, despite having only 20% more control flow statements. *jboss* on the other hand, uses the while-statement a lot more than what *Derby* does (see Table 4).

When looking at the expression types present in later part of compound predicates, we can again see rather large relative differences between the programs. *jboss* for example, have twice as many predicates with *instanceof* operators than *Derby*. The reverse situation is true for array access expressions, and Derby has over 70% more field accesses than *jboss* (Table 3).

It is clear that the larger programs influence the distribution of these expressions and control flow statements, and perhaps a bigger sample would be needed to draw more valid conclusions about such distributions. However, the absolute ordering of the control flow statements and predicate expression types relevant for this thesis remains very consistent regardless of including these larger programs or not. The empirical analysis is thus useful, since it can serve as a help in prioritizing which expressions to define heuristics for.

6.1.3 Effect on distribution by including test cases

As some of the programs included tests, the question arises whether these files induced a bias into the distribution of control flow statements and expressions. After all, tests are not production code, and such distributions could be completely different.

The analysis of Derby included 568 analyzed test files out of the total 1880 files. This constituted 7812 control flow statements. The analysis of jboss included 2308 analyzed test files out of the total 6137, with a total of 4607 control flow statements. Together, the test files make up 16% of the control flow statements, so a large bias could skew the distribution.

When running an analysis on the tests alone, the results were very similar to the main analysis. The distribution of expressions stayed more or less within a couple of percentage points. The most striking numbers relevant to the thesis were the if-statements, which were down to 69% from 80%, the for-loops which were up to 19% from 10%, and the while-loops which were up to 7% from 4.5%. Also, the field access expressions were down from 17.8% of predicates to 7.8%. However, even comprising 16% of total control flow statements, the distribution of expressions in the tests is not different enough from the distribution in the production code to have any large impact on the whole, given our purposes. The order of expression types still hold to a very large degree.

6.2 Problems addressed in the thesis

6.2.1 Conditional expressions

Conditionals nested within other control flow statements, such as "`if( x == ( y<z ? 1 : 1000 ) )`", should not pose any problems. There is no special function in the distance calculator (see 4.2.5) for them, as there is for numerical values or objects. The value they output is simply treated as a parameter to the distance call function of the enclosing statement. Nested conditionals were found in 69 control flow statements in the empirical analysis, or about one in thousand.

When evaluating the transformed conditional expression, the branch distance calculation and the boolean value resulting from the evaluation are carried out internally in the conditional, at the time this conditional is evaluated as part of the enclosing control flow statement.

One might consider letting the calculation of this inner branch distance value influence the outer. After all, the evaluation of the inner does have an influence on the evaluation of the outer, and the state of the program when evaluating the enclosing predicate is the same as for evaluating the inner. There should thus be no problem letting them be evaluated together.

Consider the control flow statement given in the example above, "`if( x == ( y<z ? 1 : 1000 ) )`". The branch distance in the inner predicate is calculated on the basis of the distance between y and z. The branch distance of the outer predicate is calculated on the basis of the distance between x and either 1 or 1000. So, the calculation and evaluation of the outer is dependent on the magnitudes of y and z. In this case, the inner predicate may well provide a beneficial fitness landscape in terms of guidance for the search algorithm, given that y and z are values of a suitable kind (see 2.2.5). However, the calculation of the outer will not give good guidance, since what x is compared to has only one of two values,

This scenario is at the moment not well enough understood, and the cases in which it would work or not are not clearly defined. More investigation is needed. What further complicates things, is that side effects and exception-causing factors have to be taken into consideration. At the moment, the two values are kept separate in the implementation of VIns.

Heuristics in "stand-alone" conditional statements function just as in the other control flow statements. This comes as a function of having the same format on the predicate as the others, and the predicate is what the branch distance is calculated from. (As before, the switch-statement is not considered in this context).

However, some problems arise when considering conditionals nested inside control flow statements. As above (see 6.2.1), one solution is to merely treat the conditional as another source of input for the branch distance calculation of the outer predicate. The problem is that this can cause exceptions and side effects when the same variables are changed and / or accessed in both the outer and inner predicates.

In the example of "`if( O != null && x == (O.someValue < y ? 1 : 1000) )`", calculating the two separately would not work, as the *O* in the conditional is dependent on the *O* in the first predicate. The procedure in VIns is, at the moment, to instrument and transform the conditionals first. This is due to the later transformations of some predicates containing conditionals, and transforming conditional expressions after this would cause complications. However, when encountering conditionals after a compound operator (&& or ||) inside a control flow statement, this procedure may have to be reworked.

The conditional could of course reside in the first part of the compound as well. In "`if( x == ( y++ < z ? 1 : 1000 ) && y == t )`", the y in the second subpredicate is dependent on the evaluation of the first.

It is worth mentioning that these examples may be purely theoretical, in that they might never occur in real-world code in the same way they are depicted here. However, the principle that side effects and exceptions can occur goes beyond these examples, and other, more common constructions may have the same underlying problems.

The procedure when encountering nested conditionals in a compound is currently to treat it as an unknown, that is return a default value when the heuristic would be calculated.

6.2.2 Branch distance calculations

I have mentioned a few ways of calculating the branch distance (see 2.2.3), which are to some extent quite similar, and to some extent different. Perhaps the most marked difference lies in the way conjunctions and disjunctions, i.e. compound predicates, are calculated. Also, the normalization formula has quite diverse implementations. Although some variants must necessarily be chosen to serve in examples in this thesis, does not mean it is the only one suited.

The implementation of the DistanceCalculator class (4.2), which contains the formulas for calculating the numerical values of branch distances, is not discussed in any detail. It is not a focus of the thesis. The class can be switched out or improved upon, as long as the static method definitions remain. The requirement is that the branch distance is greater the larger the gap between the values compared, and the absolute values do not matter.

6.2.3 Compound predicates

The boolean logical operators "&" and "|" are not short-circuiting, and can theoretically be instrumented normally, combined with regular "distAnd()" and "distOr()" (see 4.3.2), provided the type of both the expressions is boolean. However, due to possible confusion when the expression types are non-boolean, this feature is at this moment not implemented. The empirical analysis showed that 401 out of 78,012 predicates contained these operators (not reported in tables).

In this thesis, discussion of compound predicates have not ventured far out of the context of one and only one compound operator — meaning a maximum of two sub-predicates. This is due to the time limits imposed for writing the thesis, and the possible complexities and testing requirements in considering more than one compound operator.

As we see from the empirical analysis, by far the largest majority of the cases of compound predicates are covered even when only considering two sub-predicates (See Table 5).

In Java, binary expressions with more than two sub-predicates are treated as nested, i.e. only two expressions of which either can consist of binary expressions. A possible solution is to simply tackle one sub-predicate at a time, in the same order they are normally evaluated. An important issue is finding out which (sub-)predicate is first evaluated, and then treating all other (sub-)predicates as needing heuristics to avoid side effects and exception-causing factors (See 2.6.5 and 2.6.6).

Some complexities arise with the possibility of side effects "spreading" over multiple sub-predicates. Take the example of "`if(x < y && ++x < z && ++x < t)`". In this case, when x is compared to t in the last predicate, its value has been increased two times. The normal procedure for calculating the heuristic in this case would be to replace *++x* with *x+1* in the second predicate, in order to avoid the side effect of incrementing. The same would be done in the last predicate, so that t would be compared to *x+1* instead of the correct *x+2*. Again, if based on the distribution of the incrementing operators seen in the empirical analysis, this might seem to be a very marginal example. However, other predicates prone to side effects may occur in predicates with three or more sub-predicates.

Assignments may be something to watch out for too. In "`if( P && ( y = x ) < t && z == y)`", the value of z is dependent on y which is dependent on x. When calculating the heuristic in the second predicate, the assignment cannot be carried out, and this would cause an erroneous heuristic in the third predicate. Assignments are more prevalent in later predicates than increments or decrements, at least according to the empirical analysis.

6.2.4 Side effects

Some solutions that are proposed here define heuristics for expression types that are rather marginal. They do provide fairly simple rules, and are interesting in their own right, and have thus been addressed. Others prove more of a challenge.

Method calls occur in 4164 out of a total of 7245, or 57.5% of the predicates after compound operators (Table 6). This means that even if all the other challenges are solved, involving defining heuristics for all other expressions, over half of the predicates cannot be heuristically analyzed unless the method calls are dealt with. This number may be substantially reduced if purity analysis (see 2.6.5) is applied. No solution will be offered in this thesis, but a possible extension of the project may see external software being used in this regard.

In a related, but much more marginal situation, the same problem applies to object creation through constructors as well.

In a few cases, *array creation* expressions were seen in the empirical analysis. This is of the form "`new int[ ] {1, 2}`". Although creating a new object, the array object, it is not something that influences the state of the program as it existed before the predicate (see 2.6.5), and is therefore considered harmless in itself. This would not be the case if the new array was assigned to a reference, for example.

Increments and decrements (Table 7) are shown in the empirical analysis to be rather rare in predicates later in compounds. However, the heuristic is so simple that defining it is effortless. It is simply a matter of using the expected value at the time of the evaluation, without carrying out the assignment itself. "`( P && ++x == y )`" is treated as "`(P && (x+1) == y )`".

A similar procedure can be used on *assignments* as well, where the proposed solution is to simply use the value of the variable at the time of the evaluation, without assigning anything. For example "`( P && ( x = y) == z )`" is treated as "`( P && (y) == z )`".

6.2.5 Exceptions

The *remainder operator and division by zero* are fairly uncommon (Table 7), but also pretty straightforward to define a heuristic for. Exceptions in predicates with division are avoided by extracting the divisor out of the predicate, and only evaluate the predicate if the divisor is not zero. The same procedure can be used for the remainder operator.

With *field access expressions*, the assumption is that no side effects can arise, given avoidance of other expressions prone to cause them such as method calls. After all, if fields in objects are accessed, they are already initialized, and any side effects caused by such initialization would be dealt with already, and thus irrelevant to the predicate.

However, when accessing static fields, those fields may or may not have been initialized, depending on whether the Java Virtual Machine uses lazy loading or not. Usually, the JVM loads, links and resolves all classes at initialization of the JVM, and a static field such as `Static Object o = new Object()` would be resolved at the same time. When using lazy loading, this initialization would not take place until the field was accessed.

The question then becomes whether such initialization should be considered a side effect, and whether the generation of heuristic values should take this into consideration (see 2.6.5).

The choice of procedure here is to ignore this possibility. The development of the software *may* be dependent upon the use of a specific JVM, but this seems unlikely.

Array access expressions are special in that they can cause exceptions both through accessing them when the references are *null*, and through an index that is out of bounds. Both must be taken into consideration.

The null-check is solved on the same basis as null-checking in field access and in *instanceof*-expressions. To find out whether the index is valid, a check must be made before the predicate containing the array access, in

the same manner as made in regards to division by zero.
"(array != **null** && index>0 && index<= array.length)" must be inserted as a prerequisite to evaluate the array access expression. This is currently not implemented in VIns, and has thus not been thoroughly tested.

A solution for combinations of null-checks, zero division checks and array access checks is to simply combine these checks, with the possibility of redundant checks. The possible performance hit is most likely negligible.

The order of these checks is important. Any null-checks should come first, as zero- and array checks might access objects.

6.3 Features and limitations of VIns

6.3.1 Functionality

VIns fulfills the requirements set up in the methods section (see 4.1) in the following ways:

- It accepts one or several files for instrumentation. Due to checking and pre-processing dependencies between the files, potentially more predicates can be transformed when more of a program is instrumented at a time. However, even one single file from a program will have some transformation carried out.
- Information about the trace of an instrumented program from its execution, as well as any computed branch distances, is available through calls in the Instrumented interface, which is implemented by each instrumented file.
- Metrics gathered when instrumenting are saved in various log files. One log gives an overview of the number and names of files that were either instrumented successfully, ignored due to being non-class, or aborted by reason of not being parsed. Another log gives a list of which predicates were not instrumented, helping in debugging efforts. The last log is the summary of the instrumentation, de-

tailing the number of different expressions, and whether they were instrumented, transformed to a heuristic or not treated. The instrumentation deals with the control flow statements specified in the requirements, i.e. the *if-then-else*, the *while-* and *for-loops* and the conditional expressions. The *switch* and the *do-while* are ignored.

- As far as testing and inspection shows, the transformations conserve the semantics (see 2.4.1) of the original predicates, meaning that any test cases generated with the help of the instrumented files will cause the same control flow in the original version.

- Heuristics are generated for a selection of eligible expressions. Side effects and exception-causing factors in these statements are avoided when evaluating these in the cases where they would normally not be evaluated.

6.3.2 Limitations of VIns

Some requirements were not completely fulfilled.

Transformation and instrumentation of the expressions are complete as regards those in the first part of compound predicates.

When considering heuristics, not all expressions are implemented in VIns, but most are discussed. Time constraints limit the possibility of fleshing out the solutions and implementing them. The expressions in question are named in the *Methods* section(see 4.4).

javacc, and the visitors in the Java Parser project (see 4.2.7) were not the newest possible, and some features of Java could not be parsed. However, as seen in the empirical analysis, this was only an issue for a few files of the total.

Using janino as a tool for compiling when testing semantics and branch distance proved useful, but could be more useful still, if janino had supported newer features. It only fully supported Java 1.4, as it lacked some important features from 1.5. This also prevented it from being used in pre-processing the files to capture the fields and other meta-information.

VIns as it stands at the time of writing is capable of fulfilling its use, namely instrumenting and transforming source code. Since it is outside of the scope of the thesis, it is not adapted to being used directly in a testing framework.

As it is used mostly as a proof-of-concept for the transformations, meta-information about the classes beyond the predicates themselves were not considered. Abstract classes for example, are instrumented just the same as normal classes. Branches are given an id unique to their own class, and its possible superclass is not taken into consideration.

Currently, *constructors* are instrumented in the same way as other method bodies. However, due to the requirement that calls to other constructors, either in the same class or in the superclass, are executed first, the probe cannot be placed first. This causes the probes in the other constructors, and in any methods they subsequently call, to be registered before the original probe.

The *efficency* of the program may be questioned when using the architecture of several parse-and-visit cycles (see 4.2). Would it be more efficient time-wise and system-resources-wise to only parse one time, and use the resulting syntax tree for all instrumentations?

In practice, the instrumentation of a single file takes very little time. However, VIns does need to be overhauled in regards to memory use. When instrumenting the largest currently considered program, *jboss*, with around 6000 files, the system went out of memory. Likely sources of this inefficiency have been identified, and given more time, fixing this would have been a priority.

7 Summary and conclusions

7.1 Summary

I have started by giving an account and overview of the field of Search-based Software Test Data Generation (SBSTDG) and its relation to the enclosing field Search Based Software Engineering (SBSE).

Some of the important problems of the field were presented, among them issues stemming from the object-oriented approach to software engineering.

More in-depth consideration was given on the topics of branch distance as it pertains to the objective function for meta-heuristic search, on instrumenting source code for gaining information about the execution of programs, and on transforming code to make test generation easier and/or possible at all. These three topics are of special importance to the thesis. At the end of Section 1, I gave an account of the most interesting challenges as it pertains to this thesis. A detailed account of instrumenting various control flow statements have not been seen in the literature, and the thesis attempted to show how to do this. The *conditional expression* got special mention, as instrumentation of its unique in-line structure requires different handling than the other statements.

How to transform predicates in the source code in order to calculate branch distances has not, as far as I know, been discussed in any detail in the literature. The thesis attempted to do so, for a selection of control flow statements.

Compound predicates, where sub-predicates are joined by either the "||" or "&&" operator, ideally have their branch distances calculated using all sub-predicates. However, the operators are short-circuiting. This means that the calculation of branch distances and thus the evaluation of the fitness function cannot be carried out in many cases, due to the possible side effects and exception-causing factors. The premise of keeping the transformed version of the program semantically equal to the original when generating test input, would be broken if such side effects were allowed to execute. In addition, exceptions could cause the premature abortion of search. In the thesis, rules were defined governing how and when such evaluation could take place, and how heuristics could be used in other cases.

I presented an empirical analysis containing 10 small and large open-source programs, and pointed out the prevalence of different control flow statements, prevalence of compound predicates and the distribution of expression types found in the sub-predicates situated after compound operators. This analysis, though limited in population, could serve as a way to prioritize the development of heuristics.

In the methods section, the program Verde Instrumentor, or VIns, was introduced, as well as an account of the expected functionality and requirements. Rules for transforming and generating branch distance calculation calls for the expressions making up predicates were fleshed out. This was also the case for heuristics. For both transformation and heuristics, I remarked what was implemented in the program, what was still only a hypothesis, and what was not handled. An account of the testing carried out on VIns was given.

Finally, I discussed the impacts of the empirical analysis, how it affected the direction of the thesis. How the thesis handled the challenges given in the theory chapter was discussed. Limitations to the functionality of the program as it is implemented by the time of thesis submission was reported.

7.2 Missing features / further work

Some features of VIns, had they been finished in time, would have made the program more powerful. For some of the heuristics, the theoretical solutions were defined, but there was no time to implement and / or test it thoroughly.

Also, VIns was created alongside learning the theoretical background of the field, and alongside developing solutions for transformations. As a consequence, the architecture of VIns is not optimal. Refactoring the code and making VIns more maintainable would be a desired goal.

For further exploration, similarities between compound and nested predicates would be interesting to explore, hopefully finding out whether principles from one could be used in improving the other. Could for example heuristics be of use in handling nested predicates?

For empirical investigation, the distribution of expression types between the different control flow statements would be interesting to find out. Are some expression types more common in loop conditions than in if-else-statements?

References

[1] Andrea Arcuri. It does matter how you normalise the branch distance in search based software testing. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2010.

[2] Andrea Arcuri, Per Kristian Lehre, and Xin Yao. Theoretical runtime analyses of search algorithms on the test data generation for the triangle classification problem. In *Proceedings of 1st International Workshop on Search-Based Software Testing (SBST) in conjunction with ICST 2008*, pages 161–169 (Best Paper Award). IEEE Computer Society, 2008. Best Paper Award.

[3] Andrea Arcuri and Xin Yao. A memetic algorithm for test data generation of object-oriented software. In *Proceedings of the 2007 IEEE Congress on Evolutionary Computation (CEC)*, pages 2048–2055. IEEE, 2007.

[4] Andrea Arcuri and Xin Yao. On test data generation of object-oriented software. In *Testing: Academic and Industrial Conference, Practice and Research Techniques (TAIC PART)*, pages 72–76. IEEE Computer Society, 2007.

[5] Software artifact Infrastructure Repository. Software-artifact infrastructure repository: Home, 2010. [`http://sir.unl.edu/content/sir.html`; accessed 16-July-2010].

[6] André Baresel, Harmen Sthamer, and Michael Schmidt. Fitness function design to improve evolutionary structural testing. In *GECCO '02: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1329–1336, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[7] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[8] Leonardo Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithms. In *Proceedings of the 2002 Conference on Genetic and Evolutionary Computation (GECCO '02)*, pages 1337–1342. Morgan Kaufmann Publishers, 2002.

[9] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[10] Javier Ferrer, Francisco Chicano, and Enrique Alba. Dealing with inheritance in oo evolutionary testing. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1665–1672, New York, NY, USA, 2009. ACM.

[11] Julio Vilmar Gesser. Javaparser java 1.5 parser and ast, 2010. [`http://code.google.com/p/javaparser/`; accessed 8-April-2010].

[12] James Gosling, Bill Joy, Guy L Steele Jr., and Gilad Bracha. The java language specification, third edition, 2005. [`http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html`; accessed 19-April-2010].

[13] Hamilton Gross, Peter M. Kruse, Joachim Wegener, and Tanja Vos. Evolutionary white-box software test with the evotest framework: A progress report. In *ICSTW '09: Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, pages 111–120, Washington, DC, USA, 2009. IEEE Computer Society.

[14] Mark Harman. The current state and future of search based software engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 342–357, Washington, DC, USA, 2007. IEEE Computer Society.

[15] Mark Harman, Lin Hu, Robert M. Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Transaction on Software Engineering*, 30(1):3–16, 2004.

[16] Mark Harman, Malcolm Munro, Lin Hu, and Xingyuan Zhang. Side-effect removal transformation. In *In 9 th IEEE International Workshop on Program Comprehension (IWPC'01*, pages 310–319. IEEE Computer Society Press, 2001.

[17] Xiyang Liu, Hehui Liu, Bin Wang, Ping Chen, and Xiyao Cai. A unified fitness function calculation rule for flag conditions to improve evolutionary testing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 337–341, Long Beach, CA, USA, 2005. ACM.

[18] Phil Mcminn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.

[19] Phil McMinn, David Binkley, and Mark Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Trans. Softw. Eng. Methodol.*, 18(3):1–27, 2009.

[20] Phil McMinn and Mike Holcombe. Evolutionary testing of state-based programs. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1013–1020, New York, NY, USA, 2005. ACM.

[21] Norbert Oster and Francesca Saglietti. Automatic test data generation by multi-objective optimisation. In Janusz Górski, editor, *SAFECOMP*, volume 4166 of *Lecture Notes in Computer Science*, pages 426–438. Springer, 2006.

[22] José Carlos Bregieiro Ribeiro. Search-based test case generation for object-oriented java software using strongly-typed genetic programming. In *GECCO '08: Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation*, pages 1819–1822, New York, NY, USA, 2008. ACM.

[23] Marc Roper, Iain Maclean, Andrew Brooks, James Miller, and Murray Wood. Genetic algorithms and the automatic generation of test data. Technical report, Semin. Arthr. Rheum, 1995.

[24] Ru D. Salcianu and Martin C. Rinard. Purity and side effect analysis for java programs. In *In VMCAI*. Springer-Verlag, 2005.

[25] Nigel Tracey, John Clark, Keith Mander, and John McDermid. An automated framework for structural test-data generation. In *Proceedings of the International Conference on Automated Software Engineering*. IEEE, October 1998.

[26] Arno Unkrig. Janino, 2010. [`http://docs.codehaus.org/display/JANINO/Home`; accessed 8-April-2010].

[27] Stefan Wappler and Ina Schieferdecker. Improving evolutionary class testing in the presence of non-public methods. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 381–384, New York, NY, USA, 2007. ACM.

[28] Wikipedia. Code coverage — wikipedia, the free encyclopedia, 2010. [`http://en.wikipedia.org/w/index.php?title=Code_coverage&oldid=358675600`; accessed 3-May-2010].

[29] Wikipedia. Visitor pattern — wikipedia, the free encyclopedia, 2010. [`http://en.wikipedia.org/w/index.php?title=Visitor_pattern&oldid=358524886`; accessed 4-May-2010].