Automatic Unit Test Generation

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Menno den Hollander
born in the Hague, the Netherlands

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

Logica
George Hintzenweg 89
3068 AX Rotterdam
www.logica.com

Automatic Unit Test Generation

Author: Menno den Hollander
Student id: 12174969
Email: `menno.den.hollander@gmail.com`

Abstract

While test generators have the potential to significantly reduce the costs of software testing and have the ability to increase the quality of the software tests (and thus, the software itself), they unfortunately have only limited support for testing object-oriented software and their underlying test generation techniques fail to scale up to software of industrial size and complexity. In this context, we developed JTestCraft, a state-of-the-art test generator for the Java programming that deals effectively with *all* object-oriented programming concepts, such as object array types, inheritance and polymorfism. Furthermore, JTestCraft can locate *all* relevant test cases due to the use of the novel Candidate Sequence Search algorithm. Other novel concepts introduced in this thesis include the Constraint Tree data-structure to improve scalability and the Heap Simulation Representation to simplify the implementation of the test generator. We evaluated JTestCraft by looking at its ability to generate tests that obtain high code coverage and compare the results to human crafted tests. In addition, the performance of JTestCraft is compared against similar tools. Finally, we give pointers for further research to improve the performance and usability of future test generators.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
	Dr. C.J. Boogerd, Faculty EEMCS, TU Delft
Company supervisor:	Drs. B. Vranken, Logica
	Ir. E.C. Essenius, Logica
Committee member:	Dr. E. Visser, Faculty EEMCS, TU Delft
	Dr. Ir. W.-P. Brinkman, Faculty EEMCS, TU Delft

Preface

This thesis discusses the work I have done for my master project at the Delft University of Technology. The idea for my master's project was born when I took the Software Quality and Testing course given by Arie van Deursen. During this course I learned that software testing is an essential, yet very time consuming activity. I considered writing unit tests especially as a very mundane task and, even though it is theoretically one of the hardest problems to solve, I figured that most of it could be automated. As part of this work, I have developed a unit test generator that incorporates novel techniques to generate unit tests for object-oriented software. I am really pleased with the results and hope that in the future this work will save software developers considerable time, effort and frustration to write unit tests.

Most of this work was conducted at Logica as part of the Working Tomorrow programme. I wish to thank the people and my fellow students at Logica for providing a warm and constructive working environment. My daily supervisors at Logica were Edwin Essenius and Bram Vranken. I want to thank them for providing valuable input, support and also for sharing their insights in the software industry. My university supervisors were Andy Zaidman and Cathal Boogerd, who I have to thank for significantly increasing the quality and readability of my writings. I am grateful to all my supervisors for their time and effort spent on listening to my ideas, carefully reading my work and providing useful feedback.

Finally, I would like to thank my parents, Gabriël and Wendy for encouraging me to do what I wanted to do and for their support throughout my university years.

Menno den Hollander
Leiden, the Netherlands
July 1, 2010

Contents

List of Figures

Chapter 1

Introduction

Software testing is an integral component of the software development process. It is the main software verification technique applied today to determine whether software adheres to its specification. Software testing constitutes a major part in software development expenses. It is estimated that the cost of software testing is typically half that of the total cost of software development and maintenance [3]. Without any kind of software quality verification, software is unlikely to function correctly. The National Institute of Standards and Technology estimates that inadequate software testing costs the United States economy $59.5 billion annually [26]. Moreover, empirical results show that when software testing is postponed to a later development phase, the cost of repairing software defects increases several times [3]. In this thesis we aim to reduce costs of software testing, which should translate into a significant decrease in software development expenses. This cost reduction is achieved by using a test generator that automatically generates tests. Another advantage of this approach is that a test generator constructs test cases systematically, which improves the quality of the constructed software test suite and reduces the room for human error in software testing.

This thesis investigates techniques that automatically generate software tests from program code and focuses on techniques that target object-oriented software. This approach and its alternative are discussed in section 1.1. Section 1.2 discusses the type of tests that these test generation techniques target. The two following sections discuss the most important problems that limit the applicability of automatic test generation to industrial software. These are the limited scalability of test generation techniques and the current level of support for object-oriented language features. In section 1.5 we formulate the goal of this thesis, which is to address both problems. Finally, the last section provides an overview of the contents of this thesis.

1.1 Program versus Specification-based Test Generation

Software testing serves multiple purposes, such as assessing the performance, reliability or security of a software system. In this thesis software testing is used to assert the correctness of software. In this case, some type of oracle is needed to differentiate between right and wrong software behavior. There exist two ways to provide this oracle to a test generator.[1] Both options are currently evaluated for usage at Logica, a global IT and management consultancy company.

Specification-based techniques provide a test generator with formal specifications to generate the software tests from. Unfortunately, this requires that the software specifications are available in a (sufficiently detailed) machine-readable form. In practice, this is almost never the case. Moreover, for developers to incorporate this in their software development process, they have to learn a new specification language.

Alternatively, program-based techniques only depend on the system under test to generate test cases. After the test set has been generated it is left to the developer to extend these test cases with an expected behavior specification. Since the generated test cases are written

[1]In the software testing literature program-based testing is often referred to as white-box testing, whereas specification-based testing is often referred to as black-box testing.

```
public static int weightCategory(double mass, double length ) {
    assert mass > 0.0;
    assert length > 0.0;
    double bmi = mass / ( length * length );
    if (bmi < 18.5)
        return BMICategory.UNDERWEIGHT;
    else if (bmi < 25.0)
        return BMICategory.NORMAL;
    else
        return BMICategory.OVERWEIGHT;
}
```

Listing 1.1: Body Mass Index example

in the same language as the system under test, a developer is only required to learn a new tool. Another advantage of program-based techniques is that they obtain structural coverage more easily than specification-based techniques, since concrete source code provides more information to guide the test generators than abstract specifications [30]. It is possible to extend program-based techniques, such that they take (partial) formal specifications into account, such as requiring the absence of segmentation faults or requiring no violations of design by contract rules [23].

Due to these advantages, we decided to research program-based test generation techniques. We are particularly interested in techniques that generate unit tests, since their construction requires considerable time and resources. The automatic generation of unit tests is discussed in the next section.

1.2 Automatic Unit Test Generation

A software unit is the smallest testable part of an application and a unit test is a piece of code that checks whether a software unit behaves as intended. In object-oriented software a unit is anything that ranges from a method to a small cluster of related classes. Unit tests allow developers to locate software defects early in the development cycle. This reduces the associated repair costs significantly [14]. Unit tests also reveal defects that other software quality insurance methods are unlikely to find [10]. Unfortunately, constructing unit tests requires considerable time and resources. The construction of a unit test can be split into two activities. First, the tester determines a specific case that is likely to reveal a software defect, e.g. a case that increases statement coverage. Then the tester checks whether in this specific case the software behaves according to its specifications. The first activity can be completely automated and we estimate, based on our experience writing unit tests, that this activity is responsible for up to 70% of the total costs of unit testing. The second activity cannot be automated completely, since the expected behavior specification to identify software defects must be provided by a tester.

For example, the program shown in listing 3.1 calculates the body mass index of an adult and uses this information to determine the person's weight category [15]. The listing in table

```
public void testWeightCategory0 () {
  int ret0 = BMI.weightCategory(34.0, 7.5);
}

public void testWeightCategory1 () {
  int ret0 = BMI.weightCategory(414.7, 0.7);
}

public void testWeightCategory2 () {
  int ret0 = BMI.weightCategory(498.6, 4.7);
}
```

(a)

```
public void testWeightCategory0 () {
  int ret0 = BMI.weightCategory(34.0, 7.5);
  assert ret0 == BMI.UNDERWEIGHT;
}

public void testWeightCategory1 () {
  int ret0 = BMI.weightCategory(414.7, 0.7);
  assert ret0 == BMI.OVERWEIGHT;
}

public void testWeightCategory2 () {
  int ret0 = BMI.weightCategory(498.6, 4.7);
  assert ret0 == BMI.NORMAL;
}
```

(b)

Table 1.1: Tests for the BMI example. Listing (a) shows the test generator's output and listing (b) shows the tests after a tester has extended the tests with their behavior specification.

1.1(a) shows a test set that could be generated such that full statement coverage is obtained. The listing in table 1.1(b) shows these tests after a tester has added an expected behavior specification for each test. These specifications are encoded as `assert` statements and when the test set is run the tester is immediately informed when a test has succeeded or failed.

The two most important problems that limit the applicability of automatic test generation to industrial software are the scalability of current test generation techniques and the current level of support for object-oriented language features [2]. In the following two sections both issues are addressed.

1.3 Problem Complexity

The problem of generating tests that satisfy a certain coverage criterion, e.g. that every program statement must be executed by the test set, is undecidable [13]. This means that is impossible to construct an algorithm that always achieves maximum coverage. In practice it is often possible to generate test sets that achieve high coverage for programs of limited size and complexity. However, test generation algorithms fail to scale to larger and more complex software, because the number of invocation sequences that these algorithms need to consider grows exponentially with the sequence length. Moreover, each invocation sequence can have many execution paths. This is caused by the path explosion problem, which states that the number of paths through a program grows exponentially with the size of this program. When there are loops present in the program, then there are often a near infinite number of execution paths. Finally, determining the inputs that cause the execution of a path is a NP-Complete problem.[2] Clearly, only a limited number of sequences and paths can be explored by a test generator. When developing new test generation techniques this should be taken into consideration, since this limits the scalability of test generators.

1.4 Testing Object-oriented Software

Even though most software developed today is constructed using object-oriented languages, little research has focused on test techniques that target object-oriented software [2]. From a tester's perspective, testing object-oriented software is more difficult than testing procedural software. This is due to object-oriented language features such as encapsulation, inheritance and polymorfism. For example, encapsulation in Java is enforced by private, default and protected access modifiers. These access modifiers prevent a tester to directly modify non-public member variables. Instead, the tester must invoke a sequence of methods that modify these member variables before the method under test is invoked. Access modifiers also prove to be difficult when the access to methods is restricted. In this case, a method can only be tested indirectly by invoking a method that in turn calls the method under test. Other language features lead to similar increases in complexity. This thesis introduces novel test generation techniques that deal effectively with object-oriented language features.

1.5 Research Objective

Recent advances in automatic test generation have made it possible to construct test suites for real software [7]. As previously discussed, there are two important problems that limit the applicability of automatic test generation to industrial software. First, program-based test generation techniques are still limited to program units of limited size and complexity due to scalability problems. The second problem is the current limited level of support for object-oriented language features [2]. The objective of this master thesis is to address both problems simultaneously. More specifically, we will introduce novel techniques that enable the generation of high quality tests for object-oriented software. These techniques add full support for object-oriented language features, such as encapsulation, inheritance and polymorfism. In order to address the scalability problem, these techniques are constructed such that they only need to explore the minimal number of invocation sequences and program paths. As part of this research, we develop a state-of-the-art test generator for the Java programming language to evaluate the effectiveness of these techniques.

Based on these objectives, we formulated the main research question as follows:

> "How do we add support for object-oriented software to program-based test generation techniques, such that their scalability to large and complex software is maximized?"

In the remainder of this thesis we introduce new algorithms and techniques to answer this question. The most promising algorithms and techniques are implemented in a proto-

[2]NP-complete problems are considered hard to solve, because no algorithm is known that can solve these problems in less than exponential time [28].

type test generator. In order to assess the performance of the prototype test generator and to answer the main research question we investigate the following sub-questions:

1. "How does the quality of the generated test cases compare to test cases written by software developers?"
2. "What are the main areas of improvement for the prototype test generator?"
3. "How does the performance of the prototype test generator compare to other test generators discussed in the literature?"

1.6 Contents

The second chapter provides a literature overview and a promising program-based testing approach is chosen to investigate further. In chapter 3 this program-based testing approach, Concolic Testing, is discussed in more detail. This chapter also introduces two new techniques that improve the scalability of automated test generation. Chapter 4 introduces new algorithms to generate invocation sequences. Invocation sequences put the system under test in the right state in order to test state depended program components. The most promising techniques and enhancements are implemented in a prototype test generator, which is discussed in the following chapter. In chapter 6 the performance of proposed techniques and enhancements is evaluated. Finally, the last chapter discusses the most important findings of this thesis and proposes several new directions for further research.

Chapter 2

Related Work

This chapter provides an overview of the test generation approaches that are subject of active research. These approaches are search-based testing, symbolic testing and concolic testing. Search-based testing techniques formulate the test data generation problem as a search problem and solve this problem using random or directed search techniques. Another approach, symbolic testing, derives a set of symbolic constraints that describes the conditions necessary for the execution of a certain path. This constraint set is then solved to generate the concrete test data input that executes the associated path. In this chapter we also consider a variant of symbolic testing, called concolic testing. This variant uses concrete execution to collect the symbolic constraints.

The first section briefly describes Search-based Testing techniques. This is followed by a discussion of Symbolic Testing. This includes the Concolic Testing variant on which the developed prototype test generator is based. In section 2.3 a comparison is given of the approaches discussed in this chapter. Finally, we identify the most important areas of improvement for existing test generation techniques.

2.1 Search-based Testing

Search-based Testing methods formulate the generation of test data as a search-problem and solve this problem using random or directed search techniques, such as local search (Hill Climbing, Simulated Annealing) and Evolutionary Algorithms (Genetic Algorithms, Evolution Strategies, Genetic Programming). The search algorithm repeatedly invokes the system under test with candidate test inputs and uses code instrumentation to observe the result and adjusts the input accordingly. This process repeats until the intended element is covered or the number of attempts exceeds a pre-defined threshold.

An overview of these techniques can be found in the survey by McMinn [22]. Originally, Search-based Testing techniques only supported procedural code. Recent work, which is not covered in McMinn‘s survey, has added support for heap data [19] and object-oriented software [31].

2.2 Symbolic Testing

Symbolic Testing uses symbolic execution to generate test data [7]. Originally proposed by King [16, 17], symbolic execution assigns symbolic expressions instead of concrete values to program variables as a path is followed through the code structure. The technique is used to derive a set of constraints that describes the conditions necessary for the execution of a certain path. The solution of these constraints is given in terms of the input variables and, in object-oriented software, the program state in which the program must reside before executing the input values. In Symbolic Testing, the path constraints are collected for the system under test, and solved using a constraint solver. The solutions represent the concrete test data that executes these paths.

For example, consider the following function that clamps a value to the specified minimum and maximum range:

```
int clamp(int x, int min, int max) {
        if (x < min)
            return min;
        else if (x > max)
            return max;
        else
            return x;
    }
```

This function has 3 possible paths. The first path clamps the value of x to the specified minimum. The constraint associated with this path is: $(x < min)$, and any input that satisfies this constraint will lead to execution of the first path, e.g. when $(x = 3, min = 5)$. The second path clamps the value of x to the specified maximum and its constraint is: $\neg(x < min) \wedge (x > max)$. Finally, the constraint associated with the last path that does not clamp this value is: $\neg(x < min) \wedge \neg(x > max)$. For example, the concrete input $(x = 3, min = 0, max = 5)$ causes the execution of this path.

Unfortunately, each path constraint passed to the constraint solver is (at least) a NP-Complete problem. This class of problems are known to be hard to solve [28]. Due to this complexity, the constraint solver is one of the most critical components that determine the performance of the test generator.

2.2.1 Concolic Testing

Concolic Testing is a variant of symbolic test generation that combines the concrete execution of a program with symbolic execution [12, 27]. Test generators based on this approach collect the path constraint during concrete (normal) execution of the system under test. After the execution finishes, the collected path constraint is modified. The solution of this modified constraint is then used to execute another path. This process repeats until a stopping criterion is met, for example when the number of iterations exceeds a threshold or when sufficient code coverage has been obtained.

The first advantage of this approach is that each constraint formula passed to the constraint solver results in the exploration of a whole path, whereas in normal Symbolic Testing the constraint solver needs to be invoked for the exploration of each branch in a path. This not only reduces the load on the constraint solver considerably, but also allows it to reason about complex code, such as cryptographic hash functions. Another advantage of this approach is that a test generator can observe the concrete values, which allows it to reason about native library calls. Unfortunately, Concolic Testing suffers from the same complexity problem as Symbolic Testing, since both approaches use a constraint solver to locate inputs to execute another path.

2.2.2 Object-oriented Software

Symbolic Testing and its variants have been extended in various ways to support object-oriented software. Most approaches require formal specifications to work, which requires that software developers have to put effort in creating them. For instance, JPF [30] and KUnit [9] generate heap configurations that need an invariant function to identify invalid heap configurations. These heap configurations are then used to generate test data for the methods in the system under test. This approach requires that the heap configuration can be initialized directly, i.e. without calling any methods. Unfortunately, this is often impossible due to encapsulation, which is enforced by (private) access modifiers.

Another approach is to generate method invocation sequences [6, 33]. This avoids both problems, since it does not require any formal specifications and such a method invocation sequence can be transformed directly to a test case. A drawback of this approach is that the test generator loses the ability to directly set the field values in objects and needs to determine invocation sequences that do this instead. This makes it significantly more complex to locate test data. Another drawback is that the generated test cases may use the system under test differently than it was intended. In this case, it is left to the user of the test generator to remove these redundant test cases.

2.3 Comparison of Test Generation Approaches

In this chapter we have discussed Search-based Testing, Symbolic Testing and Concolic Testing. In the previous section we already explained that Concolic Testing should outperform Symbolic Testing, because it needs fewer constraint solver invocations to obtain similar code coverage. Since the most recent performance comparison between Search-based Testing and Concolic Testing, many optimizations have been developed that improve the performance of Symbolic Testing and Concolic Testing approaches considerably [4, 7, 11, 20]. Moreover, the performance of state-of-the-art constraint solvers have also increased significantly. Due to the lack of similar performance improvements in Search-based Testing, we expect Symbolic and Concolic Testing to outperform Search-based Testing by a wide margin.

2.4 Conclusion

This chapter discussed three approaches that generate test input data. The Concolic Testing approach seems the most promising of these approaches. As all test data generation techniques, it suffers from the path explosion problem. Another problem is that the employed constraint solver has to solve computationally hard problems, which may take considerable time. In the next chapter we discuss Concolic Testing in more detail and propose new techniques to alleviate the path explosion problem and reduce the load on the constraint solver.

Chapter 3

Concolic Testing

This chapter discusses the theoretical foundations of test data input generation using Concolic Testing. In this chapter we introduce two new techniques to simplify and improve the scalability of automatic test generation for object-oriented software. The first technique, Simulated Heap Representation, simplifies the implementation of a test generator when the system under test contains array, field, reference and typing instructions. The second technique, the Constraint Tree data structure, improves the performance of automatic test generation when testing methods that have a small number of possible execution paths. Although these techniques are aimed at testing object-oriented software, they can also be used for procedural software.

The first section of this chapter introduces Concolic Testing. The following section discusses symbolic representations for the heap and focuses in particular on array types, which can significantly increase the problem complexity of test generation. In section 3.3 we propose the Constraint Tree data structure that avoids exploring the same execution paths twice. Finally, we summarize the main conclusions of this chapter.

3.1 Concolic Testing

In the previous chapter Concolic Testing was selected to generate test cases, because it performs well, and can handle complex code and native library calls. Concolic Testing generates test cases by first executing the system under test with random argument values. During execution both the concrete values and symbolic constraints are collected for the executed path [12, 27]. The next execution of the system under test is then forced to take a different path. Most concolic test generators achieve this by first negating one of the collected symbolic constraints that determine the execution path (i.e. those constraints that are associated with a branch predicate) and then by solving the resulting set of constraints. This process repeats until a stopping criterion is met, for example when the number of iterations exceeds a threshold or when sufficient code coverage has been obtained.

For example, consider again the body mass index example discussed in the first chapter. This example program calculates the body mass index of an adult and uses this information to determine the person's weight category. For the readers' convenience the BMI program is shown again in listing 3.1. Concolic Testing starts by executing the method with random arguments. Assume that in this case, the method argument `mass` is set to `22.0` and the method argument `length` is set to `-5.0`. During execution of the method with these input values both the concrete values and symbolic constraints are collected for the executed path. For an input to execute the same path, it is necessary that each expression associated with a branch evaluates to the same value. The first branch instruction is encountered when executing the first assert statement. In this case, the branch predicate evaluates to true, because the variable `mass` is set to `22.0`. For an input to take the same branch, it is necessary that the following constraint, called the branch constraint, holds:

$$(mass > 0.0)$$

The next branching node is encountered when executing the second assert statement. Now the branch predicate evaluates to false, because the variable `length` is set to a nega-

```
public static int weightCategory(double mass, double length) {
    assert mass > 0.0;                                    // Node 1
    assert length > 0.0;                                  // Node 2
    double bmi = mass / (length * length);                // Node 3
    if (bmi < 18.5)
        return BMICategory.UNDERWEIGHT;                   // Node 4
    else if (bmi < 25.0)                                  // Node 5
        return BMICategory.NORMAL;                        // Node 6
    else
        return BMICategory.OVERWEIGHT;                    // Node 7
}
```

Listing 3.1: Body Mass Index Calculation

tive value. The branch constraint associated with this branch evaluation is:

$$\neg(length > 0.0)$$

This branch constraint is combined with the previous branch constraint to form new path constraint:

$$(mass > 0.0) \wedge \neg(length > 0.0)$$

After executing the second assert statement, the method terminates with an `AssertionException`. Hence, the resulting path constraint describes the constraints on the input variables such that the method fails on the second assertion check. After the path constraint is obtained, it is altered by negating one of the branch constraints. When the last branch constraint is negated, the altered path constraint becomes:

$$(mass > 0.0) \wedge (length > 0.0)$$

This new path constraint is then passed to a constraint solver to determine if there exists an input that executes the new path. One of the many solutions the constraint solver might return is a test data input where `length = 1.0` and `mass = 50.0`. In the next iteration the method is executed with this input and the path constraint is again collected. This input causes the method to be executed without throwing an exception and returns the overweight category. This execution path has the following constraint:

$$(mass > 0.0) \wedge (length > 0.0) \wedge \neg(bmi < 18.5) \wedge \neg(bmi < 25.0)$$
$$\text{where} \left(bmi = \frac{mass}{length \times length}\right)$$

This process of modifying path constraints, solving them, executing the resulting test inputs, and collecting the path constraint repeats until a stopping criterion is met. This could be when the number of iterations exceeds a threshold or when sufficient code coverage has been obtained.

If the constraint solver is unable to compute a test input that satisfies the altered path constraint, then another altered path constraint is passed to the constraint solver to solve. This happens when either the path constraint is infeasible, i.e. when no input exists that satisfies the constraint, or when the constraint solver fails to compute a solution within the given amount of time.

3.2 Symbolic Heap Representation

The heap is the memory area where object and arrays are typically stored. In this section we discuss two ways to represent a heap in a test generator. The difference between these two representations is how they deal with arrays. Arrays increase the complexity of the path constraints, since an array load operation can result in loading different primitives or references. Which primitive or reference is loaded depends on the used array index. When a value loaded from an array is used in a branch constraint, it could influence the outcome of the branch evaluation. Hence, in the presence of array types, the control flow does not necessarily depend only on the traversed execution path, but also on the primitives and references that are loaded from arrays. The problem becomes even more complex when objects are stored in an array, since the result of any operation that depends on these objects, i.e. field load and store operations, reference checks and type checks, depends on the used array index. Due to these reasons many test generators have limited or even no support for array types [6, 27, 33].

For example, consider the following code fragment. Whether the target statement is reached depends on whether the index i is within the array bounds and which boolean value is loaded from the array.

```
boolean[] array = {false, true, true, false, true};

public int x(int i) {
    if(array[i]) {
        ... // target statement
    }
}
```

The first way to represent the heap symbolically is the Heap Simulation Representation. This novel technique that we propose reasons only about one element in an array at a time. This is achieved by considering each array load and store as a branch constraint on the array index, which sets the array index to a constant value. As a result, an array load can result in loading only one primitive or reference, and therefore the control flow only depends on the path constraint. In this case, the symbolic heap only has to simulate the workings of the concrete heap during concolic execution.

For example, a generator that implements this solution would first execute the method of the last example with a random input. Assume it executes the method with $i = 10$. When the test generator encounters the array load operation during execution, it adds boundary constraints to the path constraint. Since, the concrete value of i does not fall within the boundaries of the array, the path constraint becomes:

$$\neg((0 \le i) \wedge (i < 5))$$

Then the (concrete) array load operation is executed and an `ArrayIndexOutOfBoundsException` is thrown. To locate an input that executes another path, the collected boundary constraint is negated and solved. A possible solution is when $i = 4$. The test generator invokes the method with this input. When the array load operation is encountered, both the boundary constraints and the used array index are added to the path constraint, which becomes:

$$((0 \le i) \wedge (i < 5)) \wedge (i = 4)$$

Then the execution continues and a symbolic expression is loaded from the symbolic heap. In this case, the expression stored in the symbolic array represents the boolean `true` constant. The test generator then encounters the if instruction and adds the branch constraint, which is the loaded expression, to the path constraint:

$$((0 \le i) \wedge (i < 5)) \wedge (i = 4) \wedge \texttt{true}$$

Finally, the Concolic Testing process continues until a stopping criterion is met.

The array example illustrates that simulating the heap is a simple way to add support for arrays. Another advantage of his solution is that the constraints can be kept relatively simple to solve, since the constraint solver does not need to reason about heap operations (load and store operations for object fields and arrays), references (reference equality, null checks) and types (type casts, type checks). The reason why the constraint solver does not need to reason about these instructions is that the outcomes of these instructions are deterministic. If array store and load operations target fixed indices, then only one particular object or primitive is loaded. This means that objects will always have the same reference value and type in the same execution path. Therefore, it is impossible to change the evaluation of reference and type constraints. The only drawback of this solution is that it explores many execution paths more than once in the presence of arrays. For example, in the code fragment there are only two execution paths, but since there are five array indices, it explores these paths five times.

The other solution, called Heap Reasoning Representation, lets the constraint solver reason about the complete heap, including the contents of each array. For the array example, this would mean that the constraint solver can determine which index to select to negate the branch constraint and avoid exploring the same execution path twice. Unfortunately, this solution increases the load on the constraint solver considerably, since it must reason about references, types and all the heap operations that were executed. Another drawback of this solution is that when native code makes changes to the heap, the changes must be observable by the test generator, since otherwise the state of the symbolic heap becomes invalid. This solution requires also significantly more effort to implement.

Both solutions have their advantages and disadvantages. The Heap Simulation Representation can be implemented with relatively little effort and produces less complex path constraints than the Heap Reasoning Representation. However, the Heap Simulation Representation can only reason about one array element at a time. Hence, this solution is only

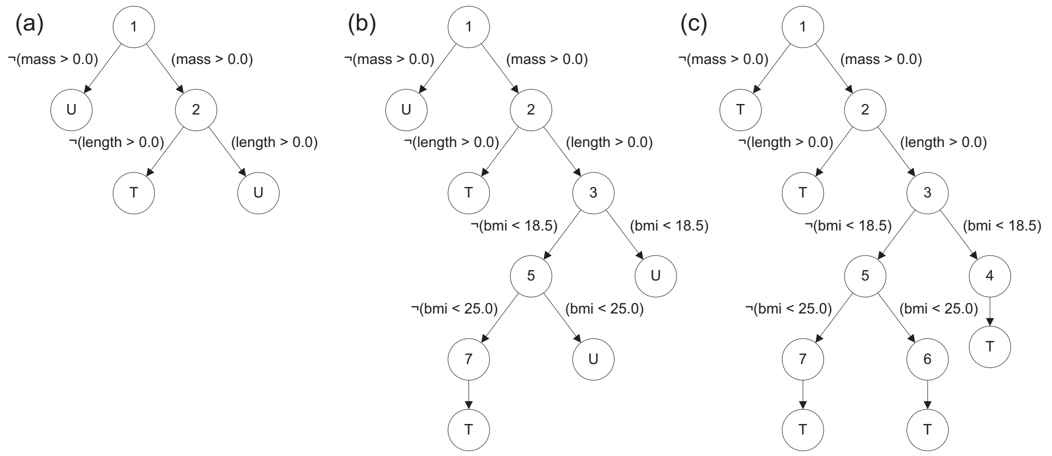

Figure 3.1: Path constraints represented in a tree structure. Figure 3.1(a) shows the tree representation after the first execution of the BMI example, figure 3.1(b) shows the tree after the second execution, and figure 3.1(c) shows the fully explored tree structure for the BMI method. All numbered nodes are computation nodes that correspond to part of the code of the BMI example. Nodes labeled T are terminal nodes, and nodes labeled U are unexplored nodes.

appropriate for testing software that makes little use of arrays. Otherwise, it is best to use the Heap Reasoning Representation. Note that it is also possible to combine both solutions, for instance, by only letting the constraint solver reason about the contents of primitive arrays. The techniques discussed in the remainder of this thesis are compatible with both solutions.

3.3 Constraint Tree

A problem with Concolic Testing as it is presented in the previous section is that the same execution paths are often explored multiple times, since the algorithm only remembers which path has been taken in the last execution. We propose to represent the collected path constraints of a method in a tree structure such that each execution path is explored only once. In addition, the generated test data can be stored in the Constraint Tree. After having generated a large volume of test data, the tree structure facilitates easy extraction of a small test set of high quality, e.g. a small test set that achieves high code coverage. Finally, the Constraint Tree provides a (partial) summary of a method.

The idea of this tree structure is that each path through the tree starting at the root node and ending at a leaf node represents an execution path. Figure 3.1 shows the path tree at different stages for the body mass index example. Each branch of the tree represents a branch constraint, and all the branch constraints along a path represent the path constraint. All nodes except for the leaf nodes are computation nodes. These nodes represent the computations that happen between branch constraints. In the case of the body mass index calculation program, node 3, where the BMI is calculated, is such a node. At the other nodes no computations happen, but these are still considered computation nodes. Any path starting at the root node leading to a computation node is per definition feasible. There exist multiple kinds of leaf nodes: unexplored nodes, terminal nodes, infeasible nodes and time-out nodes. Unexplored nodes are nodes for which the path constraint has not yet been tried to be solved and such a node could either be feasible or infeasible. Terminal nodes are feasible nodes, where the execution path ends, i.e. when a function returns or when a thrown exception fails to be caught. Paths leading to an infeasible node are infeasible and hence no input exists that executes that path. Time-out nodes are nodes where the path leading up to it, is unknown to be feasible or infeasible, because the constraint solver is unable to compute an answer in the given amount of time. It is important to note that it is often impossible to explore all execution paths of a method, because there can exist almost an infinite number of paths. Hence, it is not uncommon that a Constraint Tree is not fully explored.

3.3.1 Method Invocations

It often happens that a method calls another method. In a Constraint Tree this call can be represented by referencing the Constraint Tree path of the method called. Since many methods call the same method path, this approach reduces memory consumption considerably over an approach that represents the called method path in the original Constraint Tree. Similar to branch constraints, in the Constraint Tree a method invocation branches to different nodes depending on the method and path executed.[1]

For example, consider the following recursive method that implements Euclid's algorithm, which is an efficient algorithm for computing the greatest common divisor.

```
int gcd(int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}
```

The Constraint Tree associated with this method is shown in figure 3.2. Figure 3.2(a) shows the trivial approach. Figure 3.2(b) illustrates the effectiveness of referencing the called method path, since for each possible execution path only one additional tree node is required.

3.4 Conclusion

This chapter introduced the concepts of Concolic Testing. We introduced a new technique to represent the heap symbolically. This technique, called Simulated Heap Representation,

[1]Recall, that when a polymorfic method invocation is made on an object, the method that is actually executed is determined by the object's type. Since this approach considers the method that was actually executed, it also supports polymorfic method calls.

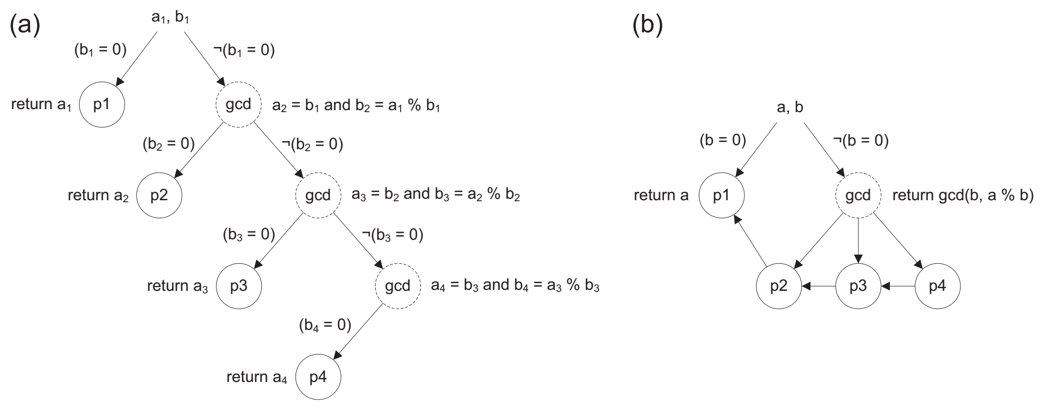

Figure 3.2: Constraint Trees for the recursive method that implements Euclid's algorithm. These trees represent the same four possible execution paths, labeled p1 to p4, through the recursive method. The dashed gcd node is a method call node. In this case, gcd calls itself recursively. Path p2, p3 and p4 represent the path where gcd calls itself one time, two times and three times respectively. The trivial approach shown in 3.2(a) also stores the path taken through the called methods, whereas the approach of figure 3.2(b) represents the method path called by referencing its leaf node.

can be implemented with relatively little effort and produces less complex path constraints than the Heap Reasoning Representation. However, the Heap Simulation Representation can only reason about one array element at a time. Hence, this representation is best used for testing software that makes little use of arrays. We also proposed the Constraint Tree data structure that can be used to avoid exploring the same paths twice, which improves the scalability of Concolic Testing. The Constraint Tree data structure provides two additional benefits. First, the Constraint Tree represents a (partial) summary of a method, which can be used to significantly improve performance when exploring method invocation sequences. Second, it provides a place to store the generated test cases in.

In the following chapters we will discuss the design, implementation and evaluation of a Concolic Testing tool that uses all of the techniques introduced in this chapter. However, the next chapter addresses invocation sequence generation algorithms, since the success of testing object-oriented software depends primarily on the ability to set the unit under test in the right state. As such, we expect that an efficient invocation sequence generation algorithm will provide the largest scalability improvement, especially when it is combined with the Constraint Tree data structure.

Chapter 4

Candidate Sequence Search

From a software testing perspective, object-oriented software is very different from procedural software. Testing object-oriented software often requires considerably more effort than testing procedural software. This is due to language features such as encapsulation, inheritance and polymorphism, which make testing object-oriented software significantly harder. One of the most important differences between testing units of procedural software and object-oriented software is how functions (procedural) and methods (object-oriented) depend on program state. In procedural software, most functions do not depend on state (i.e. static and global variables in C), or when they do, this state is easily manipulated by a test generator. Whereas, in object-oriented software almost every method depends on its object's state, which is much harder to manipulate due to encapsulation. Hence, the success of testing object-oriented software depends primarily on the ability to set the unit under test in the right state. This is achieved by generating invocation sequences that consist of constructor invocations, method invocation and field assignments. These elements of invocation sequences are called invokables and each invokable represents a statement in a test case. The complete test sequence represents a test case, where the object, static fields and the method arguments are put in the right state, and finally, the method under test is executed. In this chapter we introduce the new Candidate Sequence Search algorithm to generate invocation sequences. This algorithm wastes very little time on redundant sequences and deals effectively with language features, such as arrays, inheritance and polymorfism.

The first section discusses the limitations of current approaches that generate invocation sequences. Then an overview is given of the Candidate Sequence Search algorithm. The following two sections discuss the two steps of this algorithm in more detail. The last section summarizes the main conclusions of this chapter.

4.1 Current Approaches and Their Limitations

In this section two approaches are discussed to generate invocation sequences. As far as we are, nearly all algorithms discussed in the literature use parameter types of constructors and methods to generate invocation sequences. The only exception is the algorithm of Buy et al. that uses method summaries that are computed using symbolic execution to generate invocation sequences [6]. Both approaches and their limitations are discussed in the following two subsections.

4.1.1 Type-based Approaches

An often used approach is to use the parameter types of the invokables of the system under test to generate invocation sequences [8, 31]. However, this approach suffers from two problems. First of all, this approach generates many redundant sequences, because it is only necessary to explore sequences where every invocation influences the execution of the method under test. All other sequences are redundant, since the same effect can be obtained by executing a shorter sequence where the methods that do not influence the method under test are removed.

For example, getter methods, methods that provide read access to access restricted member variables, do not alter the program state. If the return value of these methods is not used later on in the sequence, then it does not influence the system under test. Hence, a sequence that is exactly the same except that it does not contain this getter method call will execute exactly the same paths though the method under test and produce exactly the same states as the original sequence. Therefore, the original sequence does not need to be explored.

The second problem is that this approach does not take (static) class variables into account, which means that this approach is likely to fail generating useful test cases for methods that depend on static state.

4.1.2 Combining Execution Paths

The approach by Buy et al. uses symbolic execution and define-use pairs to address the two problems associated with the type-based approaches [6]. A define-use pair describes two locations in the program code: where a variable is defined and where a variable is used. A variable is defined when it is assigned a value and a variable is used when it is used in a comparison or computation. For example, the following line of code shows an example of a define and a use.

$$\underbrace{a}_{\text{define}} = \underbrace{b}_{\text{use}} + 1;$$

Their algorithm works as follows. First, the methods of the system under test are summarized using symbolic execution. Each summary describes an execution path with a path constraint and the state changes it causes. After having computed these summaries, these method paths are combined to generate invocation sequences that exercise all define-use pairs of the fields in the class under test.

To illustrate this method we use the coin box example of Buy et al. [6]. The CoinBox class is shown in listing 4.1. The code represents control code for a vending machine. The vending machine requires at least two coins before serving a drink. The CoinBox class contains a known defect [18], since the method *returnQtrs()* does not reset variable *allowVend*. Hence, it is possible to get your money back after you have inserted two coins and get a drink for free.

The computed summary of the method of this class is shown in table 4.1. The algorithm will then try to combine pre- and post-conditions of the method paths to generate invocation sequences. For example, the sequence of the following method paths is infeasible: *Coinbox*$_1$*(); vend*$_2$*()*. The number in subscript corresponds to the path number in the summary table. This sequence is infeasible, because the postcondition '*allowVend' = 0*' of the *Coinbox*$_1$*()* path cannot be combined with the precondition '*allowVend* $\neq$ *0*' of the *vend*$_1$*()* path. Eventually the algorithm will find the following feasible sequence that uncovers the known defect: *Coinbox*$_1$*(); addQtr*$_2$*(); addQtr*$_1$*(); returnQtrs*$_1$*(); vend*$_1$*();*.

Martena et al. extended this approach to generate tests for clusters of classes that compose a program or subsystem [21]. There are still some limitations to their work. First, the computed method summaries can be infeasible because they depend on a state that does not exists. As a result, resources are wasted on exploring infeasible invocation sequences. Second, their prototype has limited support for pointers and does not support arrays, exceptions, inheritance and polymorfism. While support for these features can be added, we expect it to

```
class Coinbox {
    private int totalQtrs = 0;
    private int curQtrs = 0;
    private boolean allowVend = false;

    void addQtr() {
        curQtrs = curQtrs + 1;
        if (curQtrs > 1) {
            allowVend = true;
        }
    }

    void returnQtrs() {
        curQtrs = 0;
        //missing: allowVend = false;
    }

    void vend() {
        if (allowVend) {
            totalQtrs = totalQtrs + curQtrs;
            curQtrs = 0;
            allowVend = false;
        }
    }
}
```

Listing 4.1: Coinbox example of Buy et al. rewritten in Java [6]

Method	**Path**	**Precondition**	**Postconditions**
Coinbox	1		*totalQtrs'* = 0 *curQtrs'* = 0 *allowVend'* = 0
addQtr	1	*curQtrs* $>$ 0	*curQtrs'* = *curQtrs* + 1 *allowVend'* = 1
	2	*curQtrs* == 0	*curQtrs'* = 1
vend	1	*allowVend* $\neq$ 0	*totalQtrs'* = *totalQtrs'* + *curQtrs* *curQtrs'* = 0 *allowVend'* = 0
	2	*allowVend* == 0	
returnQtrs	1		*curQtrs'* = 0

Table 4.1: Execution summary computed using symbolic execution for the Coinbox class.

be very complex to do so for arrays and pointers, since their prototype needs to be extended with symbolic heap support. Moreover, when a symbolic heap is incorporated in their solution, we expect that only a small fraction of the method paths can be combined to construct feasible sequences and the resulting algorithm therefore becomes very inefficient.

4.2 Algorithm Overview

In the previous sections we discussed the problems that limit the applicability and scalability of current invocation sequence generation algorithms. These are that current approaches can overlook some sequences, consider too many sequences or miss (efficient) support for language features such as arrays, inheritance and polymorfism. We propose the new Candidate Sequence Search (CSS) algorithm to address these problems. This algorithm is based on the idea that the test generator only has to explore sequences where every invokable in sequence influences the execution of the methods under test. Unfortunately, it is very hard to determine whether an invokable influences another invokable without executing them. This is because we do not know beforehand which execution paths and array loads are possible. CSS addresses this problem by first generating candidate sequences where every invokable could potentially influence the method under test. A candidate sequence is a sequence of invokables where the invokable arguments have not yet been specified. Although the generated set of candidate sequences likely contains many redundant sequences, it is also guaranteed to include all sequences where every invokable influences the method under test. After a candidate sequence is generated, the algorithm explores execution paths through these sequences to locate test cases. During the exploration of a candidate sequence, partially explored execution paths that cannot influence the method under test are pruned from the search to improve performance.

It is important to note that the CSS algorithm only determines what possible sequences need to be explored and under which conditions their exploration can be pruned. Hence, it can be combined with any test generation technique, including search-based testing. In this thesis we present an implementation of this algorithm that uses concolic execution, in combination with the Heap Simulation Representation and the Constraint Tree optimization, which are discussed in the previous chapter. In the following two sections the generation and exploration steps of the CSS algorithm are discussed in more detail.

4.3 Sequence Generation

The first step of the CSS algorithm is to generate candidate sequences such that every invokable in sequence could potentially influence the execution of the methods under test. In this section we first discuss the criteria that determine whether an invokable in a sequence could influence the method under test. Then the candidate sequence generation algorithm is discussed that uses these criteria to generate a set of candidate sequences.The last two subsections discuss two optimizations that further reduce the number of generated sequences that are redundant.

Invokable	**Argument Types**	**Return Type**
Array Constructor	Array Element Type*	Array Type
Constructor Method Invocation	Constructor Argument Types*	Constructed Object Type
Instance Field Load	Instance Type	Field Type*
Instance Field Store	Instance Type, Field Type*	-
Instance Method Invocation	Instance Type, Method Argument Types*	Method Return Type*
Static Field Load	-	Field Type*
Static Field Store	Field Type*	-
Static Method Invocation	Method Argument Types*	Method Return Type*

Table 4.2: Invokable Types. Argument and return types marked with a * could also have a primitive type or no type at all.

4.3.1 Criteria

There exist many techniques to determine whether an invokable could influence the execution of another invokable [29, 32]. These techniques vary in accuracy and complexity. For the CSS algorithm we developed a simple and fast static analysis algorithm. This algorithm checks whether at least one data-flow pair could exist between the two invokables. A data-flow pair consists of two operations of which the first operation could affect the value loaded by the second operation. There exist three types of data-flow pairs:

Return-Argument Pair The invokable returns a reference type that can be used as an argument by the other invokable. The analysis assumes that the returned value is not null and that the value is actually used by the other invokable. Note that an invokable argument is not strictly an argument that is used in a method call. For example, an argument for an array constructor is an element that can be stored within the array. Table 4.2 shows the argument and return types for different types of invokables.

Define-Use Pair The program code of the invokable contains a field define and the other invokable contains a field use of the same field. The analysis does not determine whether the define and use access the same object.

Store-Load Pair The program code of the invokable contains an array store operation and the other invokable contains an array load operation of the same array type. The analysis assumes that both array references and indices could be the same.

The execution of an invokable also depends on the constructors and methods it calls. Hence, the algorithm also checks the above criteria for all the constructors and methods that can be called by the invokables. In order to support polymorfism, all methods that overload the called methods are considered as well.

An invokable in a sequence can influence the method under test in two ways. First, an invokable can influence the method under test directly with the criteria discussed above. Second, an invokable can influence the execution of another invokable in the sequence that in turn influences the method under test. The method under test is always the last invokable

```
void candidateSequenceSearch(Method mut) {
    List<Sequence> candidates = new List();
    // the initial candidate sequence is the method or constructor under test
    Sequence mutSequence = new Sequence(mut);
    candidates.add(mutSequence);
    if (mutSequence.isExecutable())
        exploreSequence(mutSequence, new Path());
    for (Sequence sequence : candidates)
        for (Invokable invokable : sut) // i.e. method, field assignment, array or object constructor
            if (invokable.influences(sequence)) {
                Sequence successorSequence = new Sequence(invokable, sequence);
                candidates.add(successorSequence);
                if (successorSequence.isExecutable())
                    exploreSequence(successorSequence, new Path());
            }
}
```

Listing 4.2: Candidate Sequence Search Algorithm - Generation

of a sequence. Hence, when each invokable could influence one of the following invokables in the sequence, all invokables in the sequence could always influence the method under test directly or indirectly.

4.3.2 Algorithm

The basic candidate sequence generation algorithm is shown in listing 4.2. Initially, one candidate sequence is created that consists only of the method under test. New candidate sequences are created by inserting new invokables in front of previously generated candidate sequences[1,2]. In order to avoid generating redundant sequences, only invokables are added that can influence the execution of the rest of the sequence. When a new candidate sequence is found, it is added to the list of candidate sequences. If the new candidate sequence can be executed, then the execution paths through this sequence are also explored. It is only possible to execute a candidate sequence when all non-static method calls in the sequence have an object on which the method can be invoked.

An example that demonstrates the candidate sequence generation algorithm is shown in listing 4.3. We are interested in testing the *warning()* method. In order for this method to return *true*, an TemperatureMonitor object must be constructed, enabled and set to a temperature higher than 80. First, the test generator performs static analysis. The results of this

[1]The tester controls which invokables are allowed to appear in the generated sequences by specifying the invokables that belong to the system under test.

[2]The access to invokables is often restricted by `private` and `protected` access modifiers. Normally, a software tester would want a test generator to only generate sequences that do not violate these access restrictions. In this case, the algorithm should not generate sequence that contain these restricted invokables. However, a software tester sometimes overrules these access restrictions in cases where it is otherwise too hard to test a specific case. Hence, it might be a good idea to give a test generator the option to use a selection of access restricted invokables.

```
class TemperatureMonitor {
    private boolean enabled = false;
    private int temperature;

    void setEnabled(boolean enabled) {
        this.enabled = enabled;
    }

    void updateTemperature(int temperature) {
        if(enabled)
            this.temperature = temperature;
    }

    int getTemperature() {
        return temperature;
    }

    boolean warning() {
        return getTemperature() > 80;
    }
}
```

Listing 4.3: TemperatureMonitor example

analysis are shown in table 4.3. Note that the *warning()* method includes the field defines and uses of the *getTemperature()* method, since it calls this method. Then the algorithm constructs the initial candidate sequence that consists of only the method under test, which is the *warning()* method. The first invokable that could influence the initial sequence is the *TemperatureMonitor* constructor, since it returns a TemperatureMonitor object that can be used by the initial sequence and it modifies the *temperature* field, which is used by the initial sequence. The *updateTemperature()* method could define the *temperature* field as well. These invokables are then used to construct the following two new candidate sequences: *TemperatureMonitor(); warning();* and *updateTemperature(); warning();*. These candidate sequences are used in turn to generate new candidate sequences. Figure 4.1 shows some of the other candidate sequence this algorithm generates. This figure reveals that the algorithm generated the sequence that we were looking for: *TemperatureMonitor(); setEnabled(); updateTemperature(); warning();* .

The candidate sequence generation algorithm also works in cases where multidimensional arrays must be constructed. For example, when a two-dimensional integer array is required, the algorithm adds a two-dimensional integer array constructor in front of the candidate sequence. This array is then filled by adding array constructors of one lower dimension in the front of the candidate sequence.

Special care needs to be taken when generating tests for methods that have argument types with many subtypes. For example, the *equals(Object obj)* method returns whether the object that invokes the method is equivalent to the passed argument object. When a

Invokable	Field		Reference Type	
	Defines	Uses	Arguments	Return
TemperatureMonitor()	enabled, temperature	-	-	TemperatureMonitor
setEnabled()	enabled	-	TemperatureMonitor	-
updateTemperature()	temperature	enabled	TemperatureMonitor	-
getTemperature()	-	temperature	TemperatureMonitor	-
warning()	-	temperature	TemperatureMonitor	-

Table 4.3: Static analysis results for TemperatureMonitor example

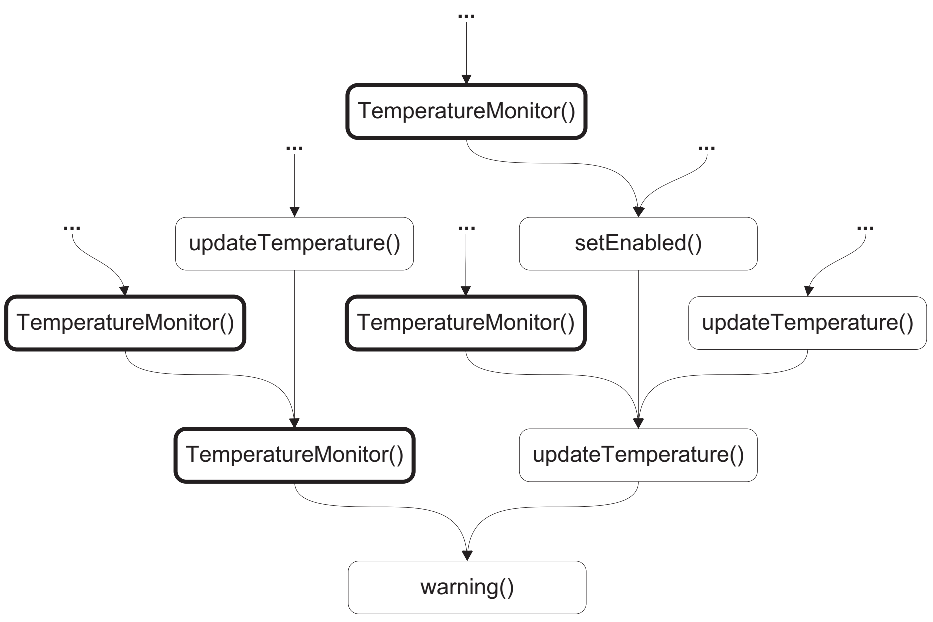

Figure 4.1: Candidate Sequence Generation for the TemperatureMonitor example. This figure shows the generated candidate sequences for the *warning()* method. These sequences are grouped in a tree. Each node represents an invokable and each line represents an influence relationship between the starting node and the candidate sequence it points to. Executable sequences start with an invokable with a bold outline. The dots represent other candidate sequences that are not shown in the tree.

class overrides this method, it could take any reference type as its argument. Instead of considering all possible reference types, it is best to only consider types that are used in the code of the method. These are the types used in type cast and type check instructions. In addition, classes that may show different behavior in the context of the method should be considered as well. These are the classes that overload one of the methods called in the method code.

It is important to note that the presented candidate sequence generation algorithm can be implemented more efficiently by making two adjustments. First, the algorithm should cache the results of the influence check that determines whether an invokable influences a candidate sequence, since otherwise the algorithm would be bottlenecked by these checks. Second, the algorithm should be implemented such that only one candidate sequence is stored in memory. The stored sequence is used to computed the next candidate sequence, which then replaces the old candidate sequence. This adjustment reduces the exponential number of sequences stored in memory to a single sequence.

4.3.3 Data-flow Check

An invokable in a sequence can only influence the method under test when there is data-flow possible between them. Data-flow is only possible when every invokable in the sequence can be linked to each other by return-argument pairs or by define-use pairs on static fields. Hence, a test generator could perform a check if data-flow is even possible before exploring the candidate sequence to reduce the number of generated redundant sequences.

The following example demonstrates a sequence where no data-flow is possible between all the invokables and the method under test. In this example the following candidate sequence is generated to test the *hashCode()* method of the *Integer* class:

```
Integer ret0 = new Integer( primitiveVariable0 );
Integer ret1 = new Integer( primitiveVariable1 );
referenceVariable0 .hashCode();
```

The first two invokables of this sequence create an *Integer* object by invoking the constructor that takes a symbolic variable as its argument. This symbolic variable is stored in a primitive integer field called *value*. The *hashCode()* method uses this field to compute the hash code of an Integer object. Since the *hashCode()* method takes only one reference argument, there is only data-flow possible between either the first or the second invokable of the sequence and the method under test. Hence, this candidate sequence is redundant and can be pruned from the search.

4.3.4 Partial Order Reduction

Another optimization is to explore only one candidate sequence of a set of similar candidate sequences where the invokables are executed in different orders, but perform the same computations. For example, consider the *calculate()* method in the following example:

```
class PartialOrderReductionExample {
    private boolean a, b, c;

    void setA(boolean a) { this.a = a; }
    void setB(boolean b) { this.b = b; }
    void setC(boolean c) { this.c = c; }

    void calculate () {
        if (a && b && c) {
            // target
        }
        return;
    }
}
```

In this example, the target statement can only be reached when the private fields *a*, *b* and *c* are set to *true*. In order to do this, the setter methods associated with these fields must be called first. These three setter methods can be called in six different orders. However, the order in which they are executed before the *calculate()* method is called does not matter. Hence, it is only necessary to explore one of these six sequences.

To implement this optimization each invokable is associated with a number. When an invokable is inserted in front of a candidate sequence, the invokable must either directly influence the first invokable of the sequence or have a number lower than or equal to the first invokable of the sequence. This ensures that for parts in candidate sequences where the invokables do not influence each other, only one subsequence is generated.

This optimization can also be used during the exploration of these candidate sequences. When the first of two following invokables in a sequence has a higher number than the second invokable, then execution paths where the first invokable does not influence the second invokable directly can be pruned from the search.

4.4 Sequence Exploration

The second step of the CSS algorithm is to explore the execution paths through a candidate sequence to locate test cases. First, the criteria that determine whether an invokable in a sequence has influenced the method under test are listed. Then the candidate sequence generation algorithm is discussed. The last section addresses an optimization that groups similar sequences together before exploring them to avoid unnecessary work.

4.4.1 Criteria

It is easy to check whether an invokable influenced another invokable directly after an invocation sequence has been executed. This is done by analysing the execution trace to see if one of the following criteria holds:

Return-Argument Pair The invokable returned a non-null reference type that is used as an argument by the other invokable.

Method 1	Method 2	Method 3	Redundant Sequence
def(a)	use(a)	use(a)	Yes, the execution of method 2 does not change the program state.
def(a)	def(a)	use(a)	Yes, the define of method 2 overwrite the define of method 1.
def(a)	def(b)	use(a), use(b)	No, both defines directly influence the uses at method 3.
def(a)	use(a), def(a)	use(a)	No, the define of method 2 is first influenced by the define of method 1, thus both methods influence the execution of method 3.

Table 4.4: Redundant Sequence Examples. This figure shows four method invocation sequences, each one is represented by a row in the table. The first three columns of the row show which fields are defined and used by the executed path. Whereas, the last column explains whether the sequence is redundant.

Define-Use Pair The invokable defined a field that the other invokable used. Note that if the field is a (non-static) object field, then the define and use must access the field of the same object.

Store-Load Pair The invokable stored an element in array at a specific index that the other invokable loaded from.

This analysis considers the complete execution trace. Therefore, all the constructors and methods that are called by the invokables are considered as well. Moreover, these data-flow pairs describe the three cases where an invokable has influenced the execution of another invokable by passing a value between the two invokables. This requires that the value is not redefined before it is used, since otherwise another value is used instead of the passed value. To check whether all invokables have influenced the method under test, the execution trace must show that all invokables have influenced one of their following invokables in the sequence. Table 4.4 shows some typical example sequences that illustrate in what cases invocation sequences are redundant. It is interesting to note that, in general, the order in which the fields and array elements are defined and used in an invokable does not matter. We illustrate this using the following example, where a and $flag$ are both object fields:

```
public void m() {
    flag = false;
    if(a > 0)
        flag = true;
}
```

This example shows that the value of the flag variable depends of the value of a, even though the $flag$ variable is defined before variable a is used.

```
void exploreSequence(Sequence candidateSequence, Path path) {
    for (ReferenceAssignment referenceAssignment : candidateSequence.getReferenceAssignments(path)) {
        ConstraintTree tree = new ConstraintTree();
        Path exploredPath = invokeSequence(candidateSequence, referenceAssignment,
                candidateSequence.getRandomPrimitiveAssignment(path));
        tree.addSatisfiablePath(exploredPath);
        if (exploredPath.passesExplorationCriteria(candidateSequence))
            exploreSequence(candidateSequence, exploredPath);
        while (tree.hasUnexploredPath()) {
            Path unexploredPath = tree.nextUnexploredPath();
            ConstraintSolverResult result = solver.solve(unexploredPath);
            if (result.isSatisfiable()) {
                Path exploredPath = invokeSequence(candidateSequence, referenceAssignment,
                        result.getPrimitiveAssignment());
                tree.addSatisfiablePath(exploredPath);
                if (exploredPath.passesExplorationCriteria(candidateSequence))
                    exploreSequence(candidateSequence, exploredPath);
            } else {
                tree.addInfeasiblePath(path);
            }
        }
    }
}
```

Listing 4.4: Candidate Sequence Search Algorithm - Exploration

4.4.2 Algorithm

Listing 4.4 shows the second step of the CSS algorithm that explores a candidate sequence using concolic execution. Initially, the explored path is empty and the first invokable is executed with random values using concolic execution. If the invokable has reference argument types, then these are all null, since no references exists yet. The collected path constraint, is then added to the Constraint Tree. The sequence is explored further when the executed path can influence the rest of the sequence. If executed path must be influenced by a previous invocation, then this condition must also hold before the sequence is explored further. When a path does not meet the exploration criteria, then the algorithm continues to search for another path that does meet these criteria. Otherwise, the sequence is explored further by calling the sequence exploration method recursively with the explored path and the same candidate sequence. Since it is often not possible to explore all execution paths, a configurable limit is placed on the number of explored paths per invokable in a sequence. From the passed path the algorithm concludes what the current invokable is that needs to be explored. The current invokable is explored with each possible assignment of previously obtained references[3]. Then the paths that start with the passed path and continue through

[3]If the symbolic heap is implemented with the Heap Reasoning Representation, the constraint solver is able to reason about the heap and also about the previously returned references. Hence, it is not necessary to explore all possible reference argument assignments for each invokable in a sequence. Instead, the constraint

this invokable are explored. This algorithm continues until the whole sequence is explored.

4.4.3 Sequence Grouping

The CSS algorithm discussed so far explores each candidate sequence directly after it has been generated. The drawback of this approach is that it involves a lot of rework, since many sequences start with the same (subsequence of) invokables. Another approach is to group invokables with the same starting invokables together before these sequences are explored. This avoids exploring the same set of paths many times. For example, consider the following two sequences: *a(); b(); c();* and *a(); b(); d();*. Both sequence start by calling the subsequence *a(); b();*. By grouping the two sequences together the subsequence *a(); b();* needs to be explored only one time. As a result, this optimization significantly reduces the time needed to explore shared subsequences.

4.5 Conclusion

In this chapter we introduced the new Candidate Sequence Search (CSS) algorithm. This algorithm generates invocation sequences that are essential for testing object-oriented software. An invocation sequence consists of constructor invocations, method invocation and field assignments and is used to set the unit under test in the right state. Current approaches overlook some sequences, consider too many sequences or miss (efficient) support for language features such as arrays, inheritance and polymorfism. CSS is a two-step algorithm that addresses these issues. First, candidate sequences are generated where every invokable could potentially influence the method under test. In these candidate sequences the arguments of these invokables have not yet been specified. After a candidate sequence has been generated, the algorithm explores the execution paths through these sequences to determine arguments of the invokables and as a result, locate test cases. We also provided various enhancements that improve the efficiency of the CSS algorithm. The next chapter discusses how the CSS algorithm in combination with the Heap Simulation Representation and the Constraint Tree optimization, which were discussed in the previous chapter, are implemented in a prototype test generator.

solver could compute both the values of the primitive arguments and which of the returned references have to be assigned to the reference arguments.

Chapter 5

JTestCraft

In this chapter the design of JTestCraft is discussed. JTestCraft is a test generator prototype that implements the new algorithms and techniques proposed in the previous chapters. This chapter starts with an overview of the architecture of the test generator. After this section we discuss the components that form this test generator in more detail. The component that is first discussed is the Sequence Generator. This is followed by a description of the Instrumentation Class Loader, which alters the program code of the system under test such that it informs the Symbolic Virtual Machine of its execution. The Symbolic Virtual Machine that collects symbolic constraints for the executed paths is explored in the next section. Section 5.5 explains how JTestCraft simplifies the collected constraints. Then in section 5.6 we briefly discuss the constraint solver that determines which concrete method arguments lead to the execution of an unexplored path. The following section describes how JTestCraft deals with floating point arithmetic. Section 5.9 discusses the measures taken to ensure the correctness of the test generator prototype. Finally, we describe how the collected execution traces are transformed in a test suite and conclude this chapter.

5.1 Architecture Design Overview

Figure 5.1 shows the architecture of the test generator prototype. The test generator starts by generating a candidate sequence. Then the paths through each candidate sequence are explored and their associated constraints collected.

In our prototype the component responsible for collecting the symbolic constraints is divided in two parts: the instrumentation class loader and the Symbolic Virtual Machine (SVM). The job of the instrumentation class loader is to instrument the system under test, such that the instrumented code informs the SVM of the concrete execution. The SVM uses this information to construct the path constraint of the executed path. After having executed a concrete invocation sequence, the collected path constraint is stored in a Constraint Tree data structure, which has been discussed in chapter 3.

The test generator uses the Constraint Tree to select a random unexplored path and its associated constraints are passed to the constraint solver. The constraint solver determines whether a concrete invocation sequence exists that satisfies that path constraint. If it exists, the concrete invocation sequence is executed by the instrumented system and stored in the Constraint Tree. Otherwise, the path constraint is marked as infeasible to avoid exploring this path again and another unexplored path is tried.

This process repeats until enough code coverage has been obtained or when the number of iterations exceeds a threshold. After each iteration the state (both concrete and symbolic) of the system under test is reset by reloading all its classes. This avoids situations where the generated test sequences depend on a global state (static fields) that is the result of executing other sequences.

Finally, a selection of the concrete invocation sequences stored in the execution tree is sent to the test code generator to generate JUnit test cases that execute these sequences.

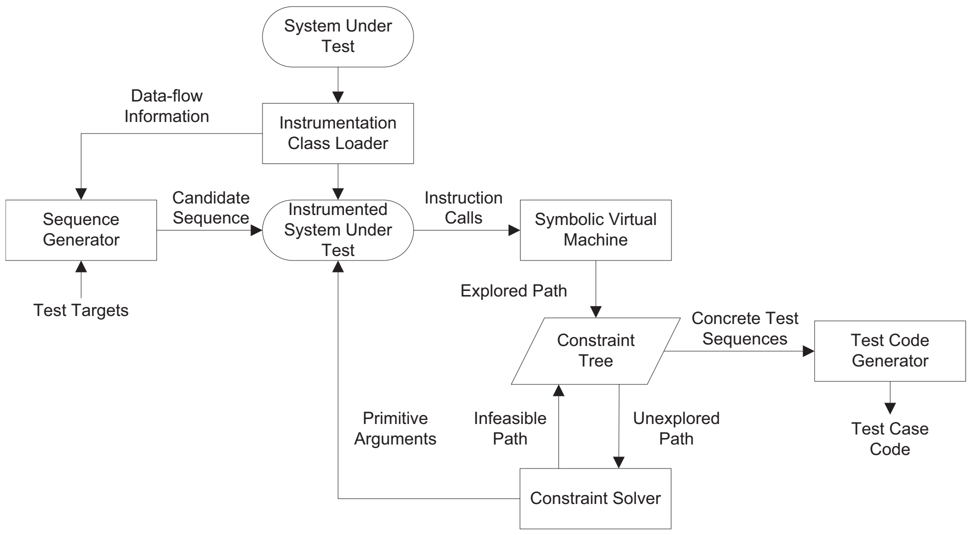

Figure 5.1: JTestCraft Architecture

5.2 Sequence Generator

The sequence generator is responsible for generating candidate sequences that need to be explored by the test generator. The previous chapter introduced the Candidate Sequence Search (CSS) algorithm that generates only candidate sequences that could influence the methods under test. The CSS algorithm needs data-flow information of the system under test in order to generate these candidate sequences. This data-flow information is obtained from the instrumentation class loader that also analyses the classes for data-flow information when it instruments the system under test. Initially, we implemented the CSS algorithm in JTestCraft without any optimizations and with only support for invokables (methods, constructors, array constructors) that have a public access modifier. In early experiments this version of JTestCraft spent most of its time on exploring redundant sequences where there was no data-flow possible between some of the invokables in the sequence and the method under test. Hence, we decided to also implement the data-flow check optimization that avoids exploring these kinds of redundant sequences. This optimization significantly increased the maximum length of the sequences that can be explored by the test generator.[1] Unfortunately, the Partial Order Reduction and Sequence Grouping optimizations were not implemented due to time constraints.

[1]In our experiments the data-flow checks increased the maximum sequence length from 3 or 4 statements to 5 or 6 statements.

5.3 Instrumentation Class Loader

This section discusses the component of the prototype test generator that executes a path through an invocation sequence and collects its symbolic constraints. The first subsection explains why byte-code instrumentation of the system under test was chosen to monitor the concrete execution. The second subsection explains how this technique is implemented in the prototype test generator. Finally, we discuss the Symbolic Virtual Machine, which is responsible for collecting the symbolic execution traces.

5.3.1 Execution Monitoring Techniques

We considered collecting the symbolic constraints by either modifying an existing Java Virtual Machine or using instrumentation of the Java source code or byte-code executed by the JVM.[2] All three solutions are able to monitor individual instruction executions.[3]

We have opted for Java byte-code instrumentation, since this has the following advantages over adapting an existing JVM:

1. Full compatibility with the Java platform, because it allows to use a certified JVM.
2. It offers higher performance, because the constraint collector runs in the same virtual machine as the code that is being tested. It also allows us to take advantage of highly performance-optimized JVM's.
3. It requires significantly less debugging effort, since both the concrete and symbolic state of a program are visible within the same debugger.

Java byte-code instrumentation has the following advantages over Java source code instrumentation:

1. Does not require the availability of source code, which allows us to test closed-source software.
2. It requires less development effort to instrument correctly, because byte-code requires significantly less semantic analysis than Java source code.

The following sections describe how the byte-code of the executed code is instrumented to trace the execution of the system under test.

5.3.2 Byte-code Instrumentation

Byte-code instrumentation is achieved by using a custom class loader for the classes that need to be tested and all the classes on which the system under test depends.[4] The classes

[2]An instrumented program has additional instructions added to its code that allows another program to monitor the behavior of the instrumented program.

[3]We also considered hooking our test generator into the Sun JVM using the Java Platform Debugger Architecture (`http://java.sun.com/javase/technologies/core/toolsapis/jpda/`). However, this infrastructure is not suited to be used at the instruction execution level, which is the only level at which the constraints can be collected.

[4]This custom class loader is written using the Apache Byte Code Engineering Library, see `http://jakarta.apache.org/bcel/` for more details.

are instrumented such that they invoke a method of the SVM for each instruction executed. This method invocation describes the instruction and allows the SVM classes to monitor the concrete execution of the system under test and construct a symbolic representation of the executed instructions. However, not all instructions are instrumented by the test generator. Unconditional control transfer instructions, such as `goto` instructions, do not influence the path constraint due to their deterministic nature and are therefore ignored. Thread synchronization instructions are also ignored, because we are mainly interested in investigating techniques for testing object-oriented software.

The SVM operates on a simplified JVM instruction set. By abstracting the instruction set, the number of JVM instructions is reduced from 210 to approximately 50 instruction types that need to be handled by the SVM. This reduction is achieved by mapping multiple JVM instructions to a single symbolic representation type. The first kind of instructions that allow this reduction are arithmetic and conversion instructions. For most of these instructions there exists 4 variants, each for the primitives of the JVM, which are `int`, `long`, `float`, `double`.[5] For instance, `iadd` (integer), `ladd` (long), `fadd` (float) and `dadd` (double) are all instructions that pop two values from the operand stack, perform an addition, and push the resulting value on the stack. In the SVM primitive types are converted to Number objects and then, wherever possible, treated the same. The other kind of instructions that can be simplified are instructions that push constants on the stack or instructions that store and load variables. For these instructions there exist optimized variants of these instructions that have an operand or memory address hardcoded in their instruction, for example, `iconst_0` is an dedicated instruction for pushing a zero value on the operand stack. These optimized instructions are mapped to the same symbolic representation type as their unoptimized counter-parts.

For example, listing 5.1 shows a method that increments an object field. The Java compiler transforms this method in the JVM byte-code shown in listing 5.2. When this method is loaded by the test generator, it is instrumented by a custom class loader. The resulting byte-code of this process is shown in 5.3. The additional instructions added by the instrumentation class loader were written in Java instead of byte-code for the reader's convenience. Each time the instrumented method is executed by the JVM, it obtains a symbolic thread object from the SVM, which it uses to keep SVM informed of the instructions that are executed by the JVM. The next section discusses how these instructions are interpreted by the SVM. However, first we discuss how the Java Core Library Classes can be instrumented, since they cannot be instrumented using a custom class loader.

Instrumentation of Core Library Classes

The instrumentation of the core library classes is complicated by the fact that the SVM is build on top of these classes. Java, by design, requires that classes belonging to the java package or any of its subpackages are loaded by the default class loader.[6] This means

[5]The other Java primitives, boolean, byte, char and short primitives, are all represented by the int type in the JVM.

[6]Note that this requirement is enforced in the native code of the JVM, hence cannot be circumvented by altering the java runtime library.

```
public class Example {
    private int field;

    public void incrementField(int value) {
        field = field + value;
    }
}
```

Listing 5.1: Field Increment Example

```
aload_0                   # load InstrumentationExample instance
aload_0                   # load InstrumentationExample instance
 getfield Example.field   # load field from InstrumentationExample instance
 iload_1                  # load method argument
iadd                      # add loaded field and method argument value
 putfield Example.field   # store computed result
return                    # exit method
```

Listing 5.2: Original byte-code for Field Increment Example

```
SymbolicThread symbolicThread = SymbolicVirtualMachine.getSymbolicThread();
aload_0                   # load InstrumentationExample instance
symbolicThread.load (0);
aload_0                   # load InstrumentationExample instance
symbolicThread.load (0);
 getfield Example.field   # load field from InstrumentationExample instance
symbolicThread.getField (Example.class, "field", new Integer (loadedFieldValue));
 iload_1                  # load method argument
symbolicThread.load (1);
iadd                      # add loaded field and method argument value
symbolicThread.add ();
 putfield Example.field   # store computed result
symbolicThread.putField (Example.class, "field");
symbolicThread.ret ();
return                    # exit method
```

Listing 5.3: Instrumented byte-code for Field Increment Example

that only one version of these classes can exist in the JVM. Unfortunately, we need two versions. First, a normal (uninstrumented) version is needed that is used by the symbolic JVM. Second, for the system under test an instrumented version is needed that collects the symbolic constraints of such a library class. In Java 6, it is only possible to instrument the core library classes such that the structure of the classes is maintained, e.g. it is not allowed to add new fields or to introduce new methods.[7] Moreover, since it is impossible to move native method declarations to other packages, the solution must keep the classes in the same package as they were originally declared. The solution is to implement both versions in the same method. This is achieved by combining the normal and instrumented versions of the method as subroutines in the same method. The first instructions of this method then check whether the normal or instrumented subroutine should be run. The instrumented subroutine should only be run when the SVM thread has called that method, which is checked using method signatures. The resulting instrumented byte-code then looks as follows:

```
SymbolicThread symbolicThread = SymbolicVirtualMachine.getSymbolicThread();
if(symbolicThread.isTracing(methodSignature)) {
    instrumented bytecode...
} else {
    original bytecode...
}
```

5.4 Symbolic Virtual Machine

The previous section discussed techniques to observe which instruction are executed. In this section we discuss the Symbolic Virtual Machine that uses these observations to construct a path constraint that describes the executed path. This is achieved by symbolically interpreting the executed JVM byte-code instructions. The internal workings of the SVM are quite similar to those of a JVM. The only difference is that the SVM operates on symbolic expressions instead of primitive variables. The following subsections illustrate how JVM instructions are symbolically interpreted by the SVM and discusses the components of the JVM that are duplicated in the SVM.

5.4.1 Stack Instructions

The JVM creates for each method invocation a frame, that contains space for local variables (method variables) and an operand stack to store partial results. More specifically, instructions load constants or values from local variables or fields onto the operand stack, while other instructions take operands from the operand stack, operate on them, and push the result back onto the operand stack. In the SVM this operand stack is duplicated and holds symbolic expressions instead of primitive variables. Similar to a JVM operand stack, a SVM operand stack can also contain array and object references.

[7] See `http://java.sun.com/javase/6/docs/api/java/lang/instrument/package-summary.html` for more details on the Instrumentation API.

The JVM instruction set has dedicated instructions to manipulate this operand stack. These instructions pop, duplicate or swap operands located on the operand stack. For example, the *swap* instruction, which swaps the two operands at the top of the operand stack, is implemented as follows:

```
public void swap() {
    //take two operands from the stack
    Operand firstOperand = getCurrentFrame().popOperand();
    Operand secondOperand = getCurrentFrame().popOperand();
    //place the two operands in reverse order on the stack
    getCurrentFrame().pushOperand(firstOperand);
    getCurrentFrame().pushOperand(secondOperand);
}
```

Note that the actual implementation of the *swap()* method also implements numerous consistency checks (assertions) to quickly detect defects in the test generator prototype.

5.4.2 Constant Instructions

Instructions that push constants on the stack are represented by placing symbolic constant expressions on the symbolic stack. This includes instructions that load constant values from constant object fields. These instructions push either primitives or references on the operand stack, and the system under test is instrumented such that the corresponding method of the SVM is called. These methods are *pushPrimitive()* and *pushReference()*, which are implemented as follows:

```
public void pushPrimitive(Number constantValue) {
    getCurrentFrame().pushOperand(new Constant(constantValue));
}

public void pushReference(Object reference) {
        if(reference == null) {
        getCurrentFrame().pushOperand(new NullReference());
    } else {
        getCurrentFrame().pushOperand(new ObjectReference(reference));
    }
}
```

5.4.3 Arithmetic Instructions

Arithmetic and primitive type conversion instructions are represented by forming new expressions by composing them of the symbolic expressions located on the symbolic stack. For example, the following *add()* method belongs to this type of instructions, which is used to symbolically represents addition instructions for all primitive types.

```
public void add() {
    PrimitiveExpression right = (PrimitiveExpression) getCurrentFrame().popOperand();
    PrimitiveExpression left = (PrimitiveExpression) getCurrentFrame().popOperand();
    getCurrentFrame().pushOperand(new Addition(left, right));
}
```

5.4.4 Branch Constraints

Conditional control transfer instructions, that correspond to if-then-else and switch-case language constructs, determine the execution path taken through the program. Each time a branch is taken, a branch constraint is added to the path constraint. The branch constraint consists of a branch condition expression that captures the symbolic condition that determines which branch is taken and a branch evaluation that records which branch actually has been executed. Instructions that represent an if-statement that compares two values are implemented symbolically as follows:

```
public void if_cmp_xxx(int condition, boolean result) {
        Expression right = getCurrentFrame().popOperand();
        Expression left = getCurrentFrame().popOperand();
        pathConstraint.add(new IfEvaluation(new IfExpression(condition, left, right), result));
    }
```

The *if_cmp_xxx()* method takes two arguments. The *condition* argument describes the kind of comparison used in the branch constraint, i.e. <, <=, ==, != , >= or >. The *result* argument indicates whether the first branch was taken. These method arguments and the two symbolic expressions located on the symbolic operand stack are used to construct the branch constraint, which is then added to the path constraint. Even though the constraint solver does not reason about reference and typing constraints, these kinds of constraints are also added to the path constraint, since they are needed to discriminate between paths in the execution tree.

5.4.5 Load and Store Instructions

Primitives and references can be loaded from and stored in (local) method variables, class and object fields, and arrays. This subsection explains how the SVM deals with these different types of memory locations. This explanation uses the symbolic instruction implementation methods shown in listing 5.4.

Method variables are stored in the frame associated with the method that is being executed. For example, the *load()* method loads a symbolic expression from a local variable index.

Field variables and array elements are stored in the symbolic heap. The symbolic heap is implemented as a hash map data structure that maps heap locations to symbolic expressions. It can happen that a field or an array element is loaded, before a value is stored at that location. In this case, the JVM loads the default value associated with the type of that location. The *getstatic()* method loads a symbolic expression stored in a (static) class field.

Recall that the prototype test generator is implemented using the Heap Simulation Representation, therefore each time an element is loaded from or stored into an array, the concrete index used in this array operation is added to the path constraint. This is illustrated by the *astore()* method, which loads a symbolic expression stored in a (static) class field.

```
public void load(int index) {
    Expression expr = getCurrentFrame().getLocalVariable(index);
    getCurrentFrame().pushOperand(expr);
}

public void getstatic(Class clazz, String fieldName, Object result) {
    StaticField staticField = new StaticField(clazz, fieldName);
    Expression storedExpression = SymbolicHeap.get(staticField);
    if(storedExpression != null) {
        getCurrentFrame().pushOperand(storedExpression);
    } else {
        //if the symbolic heap does not contain an expression
        //then create a new expression, from the loaded result
        Expression expression = createExpression(result, staticField.isPrimitive());
        getCurrentFrame().pushOperand(unsoundExpression);
    }
}

public void astore() {
    Expression valueExpression = getCurrentFrame().popOperand();
    PrimitiveExpression indexExpression = (PrimitiveExpression) getCurrentFrame().popOperand();
    ArrayReference arrayReference = (ArrayReference) getCurrentFrame().popOperand();

    //update symbolic heap
    ArrayElement arrayElement = new ArrayElement(arrayReference, indexExpression);
    SymbolicHeap.put(arrayElement, valueExpression);

    //add index constraint
    Constant indexConstant = new Constant(indexExpression.getConcreteValue());
    IfEvaluation indexConstraint = new IfEvaluation(
        new IfExpression( IfExpression.EQ, indexExpression, indexConstant),
        true);
    pathConstraint.add(indexConstraint);
}
```

Listing 5.4: Symbolic Implementation of Load and Store Instructions

5.4.6 Method Invocations

When a method call is made, the JVM copies the argument methods argument from the stack frame from the calling method to the local variable space of the method that is called. The SVM duplicates this functionality. Recall, that when a polymorfic method invocation is made on an object, the method that is actually executed is determined by the object's type. Hence, the SVM also adds the executed method to the path constraint in order to keep track of polymorfic method calls. For example, the following SVM code handles (non-static) method calls:

```
public void invoke_method(String methodName, Class[] parameterTypes) {
    //push new stack frame that stores the method call arguments as local variables
    List<Expression> arguments = collectInvocationArguments(parameterTypes.length + 1);
    frameStack.push(new StackFrame(arguments));

    //add actually invoked method to path constraint
    ObjectReference objectReference = (ObjectReference) arguments.get(0);
    Class referenceClass = objectReference.getReference().getClass();
    Method method = CodeRepository.lookupMethod(referenceClass, methodName, parameterTypes);
    pathConstraint.add(new Invocation(method));
}
```

One of the advantages of Concolic Testing is that it can deal with native library calls. After a method is called, this method has to register itself with the SVM to indicate that it is instrumented. If it fails to register itself, the SVM knows that the method was a native method. Return values of native methods are copied to the SVM and treated as either symbolic references or symbolic constants depending on their type. This allows the test generator to reason about these native return values.[8]

5.4.7 Exception Handling

Exceptions are typically thrown by a program or the JVM to signal that something went wrong. When an exception is thrown or caught, this event is recorded in the execution trace. This allows the test case generator to filter test sequences that (do not) throw exceptions, or to add code to the generated test code to catch uncaught exceptions. The current implementation of JTestCraft does not support handling cases where exceptions are caught, but this feature can easily be added.

Many frequently occurring exceptions, such as a `IndexOutOfBoundsException` or a `ArithmeticException` (e.g. divide by zero), are thrown when an implicit check done by the JVM fails. In the Symbolic JVM these checks are made explicit by adding them to the path constraint. The result is that the prototype test generator actively tries to locate test data that triggers these exceptions.

[8]In a future version of JTestCraft we also want to record which object fields and array elements are used and defined by a native method. This would allow the test generator to build a model of a native method, which can then be used to reason more accurately about this native method.

5.5 Constraint Simplification

Recall that Concolic Testing collects both the concrete values and symbolic constraints for the executed path. Many of these symbolic constraints can be simplified to reduce the load on the constraint solver. The constant substitution simplification technique simplifies parts of the path constraint and is based on the observation that many of the subformulas passed to the constraint solver only contain constant values and no variables. The constraint formula can then be simplified by substituting the subformula with its associated concrete value. For example, the constraint $i = (2 * 3) + 1$ can be rewritten as $i = 7$.

The constant substitution simplification technique can be extended to deterministic branch constraints. When a branch constraint is completely independent from the arguments passed to the function, it is impossible to negate this branch constraint. We can use this observation in two ways to optimize the search. First, the search knows beforehand that the alternative path is infeasible, hence unnecessary constraint solver queries can be avoided. Second, because the branch constraint always evaluates to the same branch, it can be removed safely from the path constraint without influencing the set of feasible solutions.[9] For example, consider the following method that initializes the values of a buffer:

```
private void initialize_buffer() {
    for(int i = 0; i < BUFFER_SIZE; i++) { // BUFFER_SIZE == 2
        this.buffer[i] = −1;
    }
}
```

The completely explored Constraint Tree for this example is shown in 5.2. In this example the `BUFFER_SIZE` is set to only 2 to demonstrate the principle. The Constraint Tree shows that that even in this simple example this optimization avoids exploring 3 infeasible paths.

Both simplification techniques can be implemented with virtually no overhead. Depending on the type of code, it is expected that both optimizations significantly reduce the size and number of the constraint formulas passed to the constraint solver. It also reduces memory consumption, since smaller path constraints are stored in the Constraint Tree.

The optimizations discussed in this section are less advanced than the technique proposed by Cadar et al. [7]. For instance, their technique also simplifies the constraints that were already collected, which should result in simpler constraints. However, their approach takes more effort to implement, since it uses rewrite techniques to determine the concrete value, whereas our proposed approach only needs to perform a substitution. Moreover, the constraint solver that JTestCraft employs, uses advanced constraint simplification techniques, which significantly reduces the need for more effective constraint simplification techniques in the test generator prototype.

[9]Note that this optimization is in effect a more powerful variant of loop unrolling, which is a program transformation technique that replaces a loop by a list of repeating statements that achieve the same result as the original loop. Our technique is more powerful, because also works on complex expressions, even when these expressions are passed from another method. Another advantage of our technique is that is relatively easy to implement.

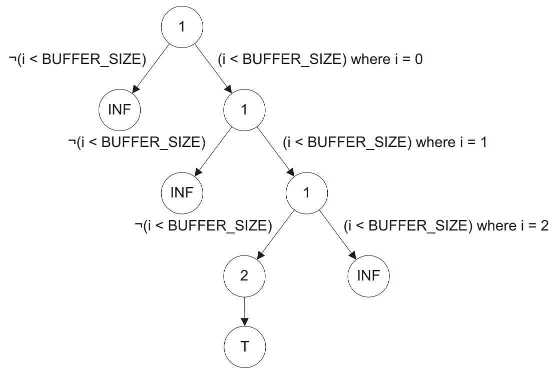
1
¬(i < BUFFER_SIZE)
(i < BUFFER_SIZE) where i = 0
INF
1
¬(i < BUFFER_SIZE)
(i < BUFFER_SIZE) where i = 1
INF
1
¬(i < BUFFER_SIZE)
(i < BUFFER_SIZE) where i = 2
2
INF
T

Figure 5.2: Tree constraints for the buffer initialization example. `BUFFER_SIZE` is a constant that is set to 2. All numbered nodes are computation nodes that correspond to part of the code of the power function example. Nodes labeled T are terminal nodes, and nodes labeled INF are infeasible nodes.

5.6 Constraint Solver

The constraint solver is responsible for determining the values of primitive arguments that satisfy the constraints of an unexplored path in an invocation sequence. All features of a modern programming language can be expressed in first-order logic. However, general purpose solvers for first-order logic lack the necessary performance to reason about software and most cannot deal efficiently with all programming language concepts, e.g. some only support linear constraints. This severely limits the applicability of symbolic execution to test data generation. Instead of solving the constraints using one general-purpose algorithm, Satisfiability Modulo Theories (SMT) solvers combine various optimized decision procedures to reason about software efficiently. These decision procedures target among others equality, linear arithmetic, fixed-size bit-vectors, arrays, lists and pointer logic problems. Due to the advantages, most recent symbolic and concolic test generation tools are built on top of SMT solvers, as is the prototype test generator.[10]

The `int` and `long` primitive types can be represented using linear arithmetic or fixed-size bit-vector theories. JTestCraft uses fixed-size bit-vectors to represent these primitive types, since they allow the constraint solver to reason about non-linear arithmetic constraints, such as division and bit-shift operations.[11] It also allows the constraint solver to reason about cases where `int` or `long` primitives overflow. Unfortunately, this exact reasoning scales worse than the approximate linear arithmetic theory, because of its added complexity. Due to the lack of support of floating point arithmetic in constraint solvers today, the prototype test generator is limited in its ability to reasoning about these kinds of

[10]The prototype employs the Microsoft Z3 SMT Solver [24], which supports all the theories and techniques needed to develop a test generator. Moreover, it ranks as one of the fastest solvers in the SMT competition. `http://www.smtcomp.org/`

[11]The prototype test generator only uses fixed-size bit-vector theories. Hence, the constraints passed to the constraint solver are NP-complete.

constraints. In the following section we discuss how JTestCraft deals with floating point arithmetic.

5.7 Floating Point Arithmetic

Many other test generators lack support for floating point arithmetic [7]. The test generators that do deal with floating point constraints use linear real arithmetic to solve these floating point constraints [27]. However, this approach requires considerable effort to implement and cannot deal with non-linear arithmetic operations, casts to integers, and Not-a-Number and 'infinite' floating point numbers. JTestCraft uses another approach that combines random testing with concolic testing to support floating point arithmetic. JTestCraft simply explores a sequence that requires floating-point variables, multiple times and each time with other random values. JTestCraft uses then concrete execution to deal with the floating point arithmetic and simply ignores the collected floating point constraints. This new approach is even less effective than the 'linear real arithmetic' approach, because it does not involve any reasoning about floating point constraints at all. However, we chose this approach because it requires very little effort to implement and is sufficient for evaluating the new algorithms and techniques introduced in this thesis.

5.8 Test Code Generator

The test code generator turns concrete invocation sequences in JUnit test cases. These concrete invocations are stored in the execution tree at the terminal nodes, i.e. the nodes where the method returns or where an exception is thrown and not caught within the same method. Unfortunately, due to time constraints we have chosen not to prioritize the concrete invocation sequences, i.e. selecting a small subset that provides sufficient test criteria coverage [25]. Hence, all the concrete invocation sequences stored in the execution tree are transformed into JUnit test cases. In practice, this means that JTestCraft generates test cases that aim to achieve full path coverage for the methods under test.

5.9 Correctness

It is important that a test generator functions correctly to avoid overlooking important test cases. A large number of assertions were added to the codebase of the test generator prototype to ensure its correctness. The most important set of assertions check whether the input that satisfies the constraints of an unexplored path does indeed cause the execution of that path. Similarly, if the constraint solver finds a path infeasible, then assertions check that this path is never executed. These checks ensure that the collected constraints are correct for the executed path and that the constraint solver interprets these constraints correctly.[12]

[12]The correctness checks have been very effective for locating (subtle) software bugs in the prototype test generator and these checks even found some defects in the Z3 SMT solver.

5.10 Conclusion

This chapter presented JTestCraft, a test generator for the Java programming language, which is able to reason accurately about almost all language features, including recursion, polymorfism, encapsulation and integer under- and overflows. Moreover, it can be used to test software that contains floating-point arithmetic and native library calls. The only unsupported features are exception catching and generating sequences that contain non-public constructor and method calls. However, both features can easily be added to the test generator. Since JTestCraft has full support for almost all language features, it can be used on real-world software. In the next chapter we will evaluate the performance of prototype test generator.

Chapter 6

Empirical Evaluation

In this chapter the performance of JTestCraft is evaluated using three experiments. Each experiment answers one the research questions posed in the first chapter. In the first experiment the prototype is run on a small and well-designed object-oriented program to evaluate the method invocation sequence algorithm and compare the quality (code coverage) of the generated test suite to a test suite written by a software developer. The second experiment identifies the bottlenecks of the prototype test generator using code profiling. The last experiment measures the performance of the prototype test generator on several algorithm implementations. The results of this experiment are used to compare the performance of JTestCraft to other test generators described in the literature. Finally, a summary is given of the three experiments discussed in this chapter.

6.1 Test Generation Performance

The goal of the first experiment is to answer the research question: "How does the quality of the generated test cases compare to test cases written by software developers?" This is achieved by comparing the quality of the generated test suite to a test suite written by a software developer and by measuring the maximum explored length of the generated invocation sequences. Note that the outcome of this experiment is determined primarily by the effectiveness of the Candidate Sequence Search algorithm. First, we discuss the JPacman application, which is used the evaluate the performance of JTestCraft, and then we discuss the experiment.

6.1.1 JPacman

JPacman is an object-oriented program based on the well known Pacman game. The program is used in a lab course to teach students about software testing, and the teacher's version includes a JUnit test set of high quality.

JPacman consists of approximately 3800 lines of code including whitespace and comments. The codebase is split in two packages. The `jpacman.controller` package is responsible for rendering the game on screen, processing input and updating the game state accordingly. Whereas the `jpacman.model` package contains the game logic and the classes that are used to represent the game internally. Even though JTestCraft can reason about the user interface classes perfectly, the current implementation needs future enhancements before it can unload the user interface. Therefore we decided to evaluate the test generator only on the `jpacman.model` package. This package consists of 15 classes and 2500 lines of code of which 612 lines are executable method body statements.

JPacman makes use of array types and object-oriented features, such as encapsulation and polymorphism. Another noteworthy fact of this program is that it uses assert statements throughout its codebase to represent contracts [23], i.e. class invariants and method pre and post conditions. This puts more constraints on the method input and makes it harder for JTestCraft to generate useful method sequences. Fortunately, it also increases the quality of the generated tests, since the generated code is more likely to represent a use case that occurs in practice.

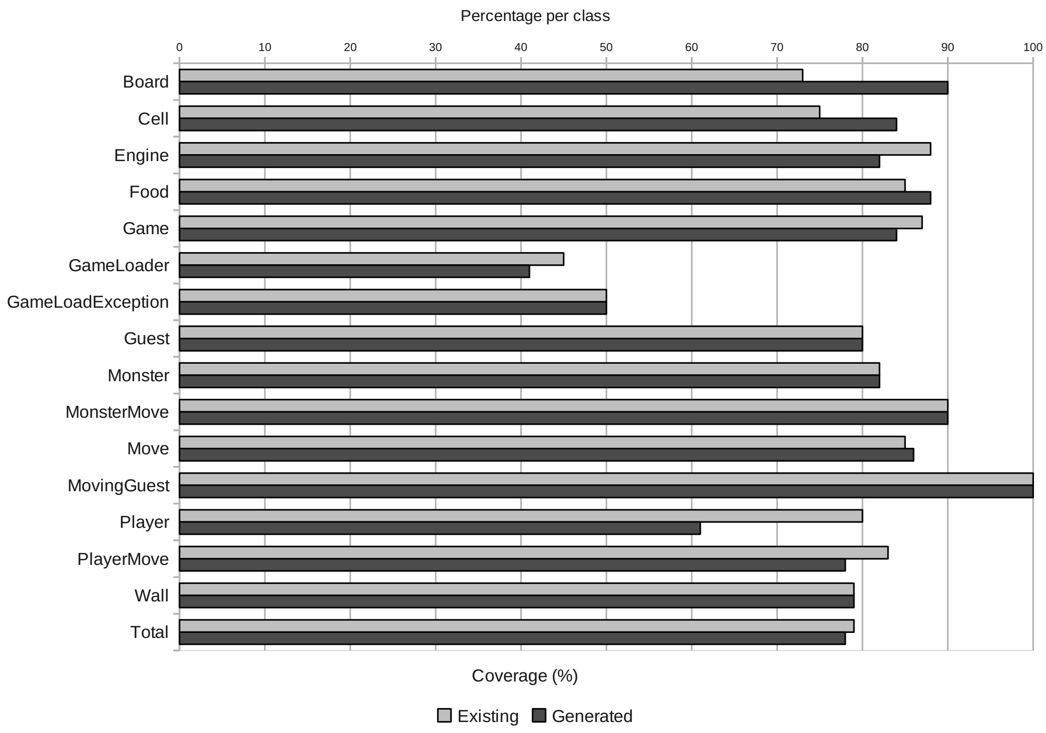

Figure 6.1: JPacman line coverage results for the existing and generated test sets.

6.1.2 Experimental Setup

The JTestCraft experiments were conducted on a Windows computer with a Core 2 Duo 2.4 GHz processor using Sun's Java 2 SDK 1.6.0 JVM with 1024 MB allocated memory. In these tests JTestCraft was given 5 minutes for each method under test, and the constraint solver time limit was set to 10 seconds per problem. For the JPacman experiments an upper limit of exploring ten paths per method invocation was set, since many of the methods under test have an almost infinite number of paths that can be explored. Note that the number of paths explored per sequence can still grow exponentially when the length of a sequence increases. The maximum allowed array size was set to 64 elements (per dimension) to avoid the creation of arrays of several hundreds of megabytes. The only modification made to JPacman was to rewrite `Engine.initialize()`. This method throws an exception even when used correctly, which makes JTestCraft discard potentially useful sequences that contain this method. Hence, this method was rewritten such that it does not throw an exception. The code coverage statistics for both the generated and the human-made test set were measured using EMMA[1].

[1]`http://emma.sourceforge.net/`

6.1.3 Results

We compared the generated test set against the existing (human-made) test set. The existing test set consists of 80 test cases while the generated test set consists of 315 test cases. We estimate that about 70% of the generated test cases are redundant (add no additional coverage). JTestCraft explored all sequences with lengths equal to four and less. In some cases, it reached exploring sequences with six method invocations. The obtained line coverage for both test sets is shown in figure 6.1. Both test sets achieve similar code coverage results. There are some noticeable differences between both test sets if we look at specific parts of the source code. First of all, the generated test set achieves significantly higher coverage in the Board and Cell classes, because the existing test set does not test their `toString()` methods. The other reason, which holds also for the Food and Move classes, is that the generated test suite also includes many test cases where the assertions are violated. The existing test set achieves higher coverage for the classes Engine, Game, Player and PlayerMove. These classes require longer sequences to obtain full coverage than JTestCraft can explore. JTestCraft currently only supports generating sequences that contain public methods invocations and methods with other access modifiers can only be tested indirectly. Hence, the test generator could achieve higher coverage with less effort when support is added for generating sequences that also include method invocations with default and protected access modifiers.

6.1.4 Threats to Validity

In this experiment only one object-oriented program was tested. Further experiments on other test subjects need to be conducted to determine the extent to which the obtained results are representative for different types of programs, such as web applications, industrial control systems and physics simulations.

6.1.5 Discussion

JTestCraft was designed to generate test cases for object-oriented program units. The current implementation was successful in testing JPacman. We are confident that performance improvements will allow JTestCraft to surpass the existing test set and perhaps even obtain 100% feasible code coverage. These optimizations consist of algorithmic optimizations, which have been discussed in chapters 2 and 4, and implementation improvements, which are identified in the following section.

6.2 Performance Measurements Using Profiling

This second experiment is used to answer the research question: “What are the main areas of improvement for the prototype test generator?” In this experiment the test generator was profiled while it generated tests for JPacman. The profiler measured the processor time and memory needed to perform different subtasks in the test generation process, which allowed us to determine the bottleneck in the current JTestCraft implementation.

6.2.1 Experimental Setup

The setup for the performance profiling experiment is the same as for the previous JPacman experiment. The only difference is that the JVM profiler was enabled.

6.2.2 Results

The performance profiling results show that most of the processor time (75%) and memory usage (90%) is spent on generating method sequences. Fortunately, memory usage can be brought down significantly (to a few megabytes), since method sequences are enumerable. In contrast, generating sequences will remain computationally expensive, because of the underlying exponential time algorithm. This problem could be reduced by implementing the algorithm more efficiently.[2] The remaining processor time (25%) is used to explore these sequences. Almost all this time is used to determine alternative paths. With a little bookkeeping we can change this algorithm such that it completes in constant time. Only 1% or less of the total time was spent on solving alternative path constraints.

6.2.3 Threats to Validity

The JVM profiler imposes a significant overhead on the computational performance of JTestCraft. As a result, the number of explored sequences was two to three times lower and the maximum sequence length was also shorter. We observed that as the sequences got longer, relatively more time was spent on generating invocation sequences. It is therefore likely that even more than 75% of the processor time was spent on generating invocation sequences.

In addition, similar to the previous experiment only one object-oriented program was tested. This makes it hard to draw general conclusions about the bottlenecks in JTestCraft. However, a similar performance profile was obtained for the algorithm experiments discussed in the next section.[3] Therefore, we are confident that the bottlenecks of JTestCraft were identified.

6.2.4 Discussion

While designing JTestCraft we tried to keep the constraints as simple as possible, since we know that the constraint solver is often the bottleneck in symbolic test generators. However, after noticing that at most 1% of the total processor time was spent in the constraint solver, we expect that shifting some of the complexity from the test generator to the constraint solver should result in significant performance improvements. For example, letting the constraint solver reason about reference types and arrays reduces the number of concolic invocations and calls made to the constraint solver.

[2]For instance, the current implementation relies heavy on String comparisons to check whether two methods, fields or types are equal. By assigning each method, field and type with an unique identifier, this String comparison can be replaced by an integer comparison, which speeds these comparisons up by roughly a hundred times.

[3]The only difference was that up to 2% of the total time was spent in the constraint solver.

6.3 Performance Comparisons Using a Standard Set of Algorithms

Up to now the research of test generators focused almost exclusively on applying the generator to algorithms. Hence, we tested the prototype also on algorithms in order to answer the following research question: “How does the performance of the prototype test generator compare to other test generators discussed in the literature?” We evaluated the performance of JTestCraft on the seven Java classes listed in table 6.1. These classes were also used to evaluate the performance of other test generators, such as Rostra [34], Symstra [33] and Korat [5] (only the last five classes). In this experiment the performance of JTestCraft is compared against Rostra and Symstra. It is important to note that the performance of JTestCraft is only compared against the other test generators on test subjects that require only simple invocation sequences to test, e.g. only sequences where the method arguments consist of primitive types. For test subject where more complicated sequences are required, e.g. where an object is passed as a method argument, Rostra and Symstra would fail to generate any test cases.

Rostra and Symstra are discussed in the following subsection, while the other subsections describe the experiment.

6.3.1 Rostra & Symstra

Rostra and Symstra are both test generators that also generate invocation sequences.

Rostra is a random test generator that draws its primitive arguments from a limited set of random values. This allows it to prune many redundant sequences based on the observation that similar program states result in the same behavior. Hence, when the resulting program state of a sequence has occurred before in another sequence, then the former sequence can be pruned from the search.

This technique is illustrated using the sorted list data structure that is implemented using a sorted array. When a new element is added to the array, the size of the array increases and the element is inserted at a location such that the array remains ordered. For example, the sequences *add(1), add(2), add(5)* and *add(5), add(1), add(2)* result in the same array configuration: *[1, 2, 5]*. Since both sequences result in the same program state, it is only necessary to explore one of these sequences further. Hence, only one of the following two

Class	Methods Under Test	Relevant Private Methods	Lines	Branches
IntStack	push, pop	-	30	9
UBStack	push, pop	-	59	13
BinSearchTree	insert,remove	removeNode	91	34
BinomialHeap	insert, extractMin, delete	findMin, merge, unionNodes, decrease	309	70
LinkedList	add, remove, removeLast	addBefore	253	12
TreeMap	put, remove	fixAfterIns, fixAfterDel, delEntry	370	170
HeapArray	insert, extraMax	heapifyUp, heapifyDown	71	29

Table 6.1: Algorithm classes used in the experiments

sequences needs to be explored: *add(1), add(2), add(5), add(4)* or *add(5), add(1), add(2), add(4)*.

Symstra is a symbolic test generator that uses a similar technique as Rostra. The difference is that Symstra uses symbolic states instead of concrete states to prune redundant sequences, which makes this technique even more effective. For example, this symbolic technique needs only one sequence for each sorted list of a different length. The sequences *add(1), add(2), add(5)*; *add(5), add(1), add(2)* and *add(4), add(5), add(6)* all result in the same symbolic array configuration: *[a_1, a_2, a_3] where $a_1 \leq a_2$ and $a_2 \leq a_3$*. As a result only one program path needs to be explored for each sequence length.

These techniques make both test generators well suited for the algorithms implemented in Java [33].[4] Recall that JTestCraft does not implement any of these optimizations, and thus explores all generated sequences and all paths through these sequences. Even though we did not expect JTestCraft to outperform the other test generators, we still wanted to know how much performance could be gained if a similar optimization was implemented in JTestCraft.

6.3.2 Experimental Setup

The experiments for Rostra and Symstra were performed on a Linux machine with a Pentium IV 2.8 GHz processor using Sun's Java 2 SDK 1.4.2 JVM with 512 MB allocated memory. The JTestCraft tests were conducted on a Windows computer with a Core 2 Duo 2.4 GHz processor using Sun's Java 2 SDK 1.6.0 JVM with 1024 MB allocated memory. In these tests JTestCraft was given 5 minutes for each method under test, and the constraint solver time limit was set to 10 seconds per problem. The only modifications made to the algorithm classes was that the inner classes were rewritten to normal Java classes, since JTestCraft does not support the instrumentation of inner classes. The code coverage statistics for JTestCraft were measured using Cobertura[5], whereas Rostra and Symstra measured the obtained coverage statistics themselves.

Redundant sequences In order to avoid the testing of redundant method invocation sequences, only sequences that were fully connected and that contained no null references were tested. The test generator was also instructed to only use the Integer.valueOf(int) method to obtain instances of Comparable, Object and Integer objects. Otherwise, JTestCraft would spent most of its time on exploring irrelevant sequences.[6] The following two sequences are examples of the many redundant sequences that would otherwise have been explored.

[4]As far as we are aware Rostra and Symstra are the fastest test generators for testing algorithm implementations.

[5]We use Cobertura instead of EMMA for this experiment, since only Cobertura can determine branch coverage statistics. `http://cobertura.sourceforge.net/`

[6]Without this adjustment we measured a decrease in the maximum explored sequence length by approximately 1 to 3 invocations.

```
HeapArray heapArray = new HeapArray();
String string = heapArray.toString();
Integer integer = Integer.valueOf(string);
boolean b = heapArray.insert(integer);
```

```
TreeMap treeMap1 = new TreeMap();
TreeMap treeMap2 = new TreeMap();
Integer integer = Integer.valueOf(699252145);
Object object = treeMap1.put(integer, treeMap2);
```

The sequence on the left illustrates a case where an Integer object is constructed from a String object. However, assuming that the Integer construction call does not throw an exception, then the resulting Integer object is exactly the same as if it was constructed using an integer primitive. Hence, it is unnecessary to explore these sequences. The other sequence creates an object (treeMap2) to store it in a Treemap object (treeMap1). However, the behavior of a TreeMap is not influenced by the type of the stored objects, since all of the TreeMap methods only use the reference of the stored object. Therefore, it suffices to only explore sequences where objects of only one type (e.g. Integer) are stored in a TreeMap.

Integer object method arguments In all of the algorithm classes Integer objects are used as method arguments. JTestCraft treats these arguments as objects and adds Integer object constructors to the generated sequences. By treating them as objects, it is possible to pass the same Integer object multiple times as an argument in a method sequence, for example:

```
HeapArray heapArray = new HeapArray();
Integer integer = Integer.valueOf(−195898993);
boolean b1 = heapArray.insert(integer);
boolean b2 = heapArray.insert(integer);
```

In contrast, Rostra and Symstra treat these arguments as primitives, and insert one Integer object constructor call for each method argument that has an Integer object type. The result is that JCraftTest generates more redundant sequences than the other test generators in this case. However, Rostra and Symstra will fail to generate test cases in other situations where, for example, the references of Integer objects are checked for equality.[7]

6.3.3 Results

The results of this experiment are listed in table 6.2 for each algorithm and shows the time needed to explore all sequences of a given length N. The table also shows for JTestCraft the number of explored paths through the sequences and the number of tests selected by the test generator that makes its selection based on the method path coverage criterion in the parentheses. The results for Rostra and Symstra were obtained from [33], where they recorded the number of generated tests and in the parentheses the cumulative number of tests that increase branch coverage. The table also shows the branch coverage that was obtained for generated test suites.[8] As expected, the results reveal that both Rostra and Symstra outperform JTestCraft with a significant difference in both the coverage obtained

[7]Note that this difference accounts for the lack of sequences of even length in the result table.

[8]A test suite consists of test cases of all lengths.

Class	**JTestCraft**				**Rostra**				**Symstra**			
	N	Time	Tests	Cov.	N	Time	Tests	Cov.	N	Time	Tests	Cov.
UBStack	5	7.41	1591(6)	90	9	4.98	1950(6)	100	9	0.85	43(5)	100
	6	*89.28	14830(8)		11	31.83	13734(7)		11	1.24	67(6)	
	7	*103.33	27119(8)		13	*269.68	*54176(7)		13	1.57	94(6)	
	8	-	-		15	-	-		15	2.33	141(6)	
IntStack	5	5.68	1022(4)	33	9	12.76	5766(4)	67	9	0.26	18(3)	100
	6	39.42	6062(4)		11	*207.59	*47208(5)		11	0.47	24(4)	
	7	- 338.9	47659(4)		13	*689.02	*52480(5)		13	0.54	32(5)	
	8	- 600.02	83098(4)		15	-	-		15	0.67	40(6)	
BinSearchTree	5	19.87	3865(9)	83	9	4.80	1460(16)	100	9	4.07	350(15)	100
	6	174.03	27543(14)		11	23.05	7188(17)		11	15.22	1274(16)	
	7	- 600.01	65718(22)		13	-	-		13	70.94	4706(16)	
	8	-	-		15	-	-		15	*251.30	*12626(16)	
BinomialHeap	5	27.44	5757(16)	79	9	4.97	1320(12)	84	9	1.41	40(13)	91
	6	493.68	88970(22)		11	50.92	12168(12)		11	3.59	66(13)	
	7	*	-		13	-	-		13	5.67	86(15)	
	8	-	-		15	-	-		15	17.53	157(16)	
LinkedList	5	21.08	4398(7)	100	9	32.61	8591(6)	100	9	0.56	25(6)	100
	6	248.01	41790(9)		11	*412.00	*20215(6)		11	0.66	33(6)	
	7	- *736.34	112206(9)		13	-	-		13	0.80	42(6)	
	8	*	-		15	-	-		15	0.94	52(6)	
TreeMap	5	69	12169(9)	15	9	3.52	560(31)	84	9	3.79	114(28)	84
	6	- 600.02	82578(9)		11	12.42	2076(37)		11	17.32	386(34)	
	7	-	-		13	41.89	6580(39)		13	38.15	698(36)	
	8	-	-		15	-	-		15	173.71	2074(36)	
HeapArray	5	7.1	1054(7)	70	9	3.75	1296(10)	76	9	2.79	51(9)	100
	6	53.04	6230(8)		11	-	-		11	5.77	96(11)	
	7	- 375.32	35614(13)		13	-	-		13	14.52	175(13)	
	8	- 600.02	59629(17)		15	-	-		15	28.50	389(13)	

Table 6.2: Results for the algorithm experiments. The symbol '*' represents a case where the test generation process timed out and the symbol '-' represents a case where a test generator exceeded the memory limit. It is important to note that JTestCraft generates tests for one method at a time and in this table the aggregated results are shown for each class under test.

and the explored sequence length reached, since their optimizations allow them to prune most sequences.[9]

6.3.4 Threats to Validity

The results for this experiment were obtained by running the tests on two machines that have different performance characteristics. The results for the test generators are so far apart that the performance difference could only have a little impact on the measured coverage and explored sequence length differences.

[9]As stated before, the sequence generation algorithm of JTestCraft treats Integer objects differently than Rostra and Symstra treat them. We conducted another experiment where we changed the Integer object arguments to integer primitives. In this case, JTestCraft explores exactly the same sequences as the Rostra and Symstra. In this experiment the performance of JTestCraft approaches the performance of Rostra and reaches sequence lengths of 9 to 11. We estimate that most of the redundant sequences that are avoided this way, could also have been pruned by the Partial Order Reduction optimization discussed in chapter 4. This indicates that the Partial Order Reduction optimization could be very effective at improving the scalability of the Candidate Sequence Search algorithm.

6.3.5 Discussion

The results show that JtestCraft in its current state is not particularly suited for testing algorithms. However, we are confident that adding the optimization that Symstra uses to prune redundant sequences would make the performance of JTestCraft comparable to Symstra.

6.4 Summary

In this chapter JTestCraft was tested on an object-oriented program and a set of algorithms. In the first experiment JTestCraft was evaluated by comparing its generated test set to an existing test set for an object-oriented Pacman implementation. This test revealed that the coverage of the existing (human-made) and generated test set achieve similar code coverage results. Moreover, we found that all test cases that consist of up to four invocation statements were located, and in some cases the test generator explored sequences of up to six invocation statements.

In the second experiment JTestCraft was profiled while it generated tests for the Pacman program. The results of this experiment show that there is room to significantly improve the performance of the implementation of the test generator. We also noticed that only a small amount of the total processor time was spent in the constraint solver. Therefore we expect that shifting some of the complexity from the test generator to the constraint solver could also result in significant performance improvements.

Finally, the algorithm experiment results indicate that there are also opportunities to considerably improve the performance of JTestCraft by implementing additional optimizations discussed in the literature.

Chapter 7

Conclusions and Future Work

This thesis introduced novel techniques to automatically generate unit tests for object-oriented software. These techniques promise to reduce the costs of unit testing up to 70%, while at the same time increase the quality of the software tests. This results in significantly lower development costs, less defects in both the software system as its test set, and shorter software development life-cycles. In this project we investigated techniques that generate test cases from program code and then leave it to the software developer to extend these test cases with an expected behavior specification. The first problem of these techniques is the path explosion problem that limits test generation techniques to scale up to software of industrial size and complexity. The other problem is that these techniques have only limited support for testing object-oriented software. In this thesis both problems were addressed and the main research question was formulated as follows:

> "How do we add support for object-oriented software to program-based test generation techniques, such that their scalability to large and complex software is maximized?"

In this thesis we introduced novel algorithms and techniques to answer this question. The most promising algorithms and techniques were implemented in a prototype test generator to evaluate their effectiveness. These contributions are discussed in the following section.

7.1 Contributions

The main contributions of this work are as follows:

Candidate Sequence Search We introduced the novel Candidate Sequence Search (CSS) algorithm to generate invocation sequences. This algorithm is able to find all invocation sequences that can influence the unit under test and deals effectively with object-oriented language features, such as arrays, inheritance and polymorfism. In addition, we proposed three optimizations that ensure that very little time is wasted on redundant sequences.

Heap Simulation Representation We introduced a new technique to represent the heap symbolically. This technique, called Simulated Heap Representation, simplifies the implementation of a test generator that supports software containing array, field, reference and typing instructions. This techniques also reduces the load on the constraint solver.

Constraint Tree We proposed the Constraint Tree data-structure that is used to avoid exploring the same program paths twice in order to improve the scalability of the test generation techniques.

JTestCraft We developed a test generator for the Java programming language, called JTestCraft, that uses the Candidate Sequence Search algorithm, Heap Simulation Representation and Constraint Tree data-structure. JTestCraft is able to reason accurately about almost all language features, including recursion, polymorfism, encapsulation and integer under- and overflows. Moreover, it can be used to test software that contains floating-point arithmetic and native library calls. Due to its support for almost all language features, JTestCraft can be applied to real-world software.

Evaluation We evaluated the performance of JTestCraft in three experiments to determine the bottlenecks in its implementation and to measure its performance against human testers and other test generators described in the literature.

7.2 Conclusions

In the introduction of this thesis we formulated the main research question and three specific research questions. These research questions are answered in this section.

Specific Research Questions

The most promising algorithms and techniques were implemented in a prototype test generator and to answer the main research question we investigated the following three sub-questions.

> "How does the quality of the generated test cases compare to test cases written by software developers?"

JTestCraft was evaluated by comparing its generated test set to an existing test set for JPacman, which is an object-oriented Pacman implementation. This experiment revealed that the coverage of the existing (human-made) and generated test set achieve similar code coverage results.

> "What are the main areas of improvement for the prototype test generator?"

The only language features that JTestCraft currently does not support are exception catching, instrumentation of inner classes and generating sequences that contain non-public constructor and method calls. We recommend to add support to JTestCraft for these features, since it is also relatively easy to do so.

The performance profiling results show that most of the processor time and memory space is used for generating method sequences. We expect that these numbers can be brought down significantly when the candidate sequence generator component is implemented more efficiently. At a later stage, the performance of JTestCraft can be improved further by implementing additional scalability optimizations discussed in this thesis (chapter 4), in other works (chapter 2).

> "How does the performance of the prototype test generator compare to other test generators discussed in the literature?"

As opposed to other test generators discussed in the literature, JTestCraft deals effectively with object-oriented language features, such as arrays, inheritance and polymorfism. Another unique feature of JTestCraft is that it does not overlook invocation sequences that influence the unit under test. Both features enable JTestCraft to generate test cases for program code where other test generators would fail. In order to improve its scalability JTestCraft explores only sequences that could influence the unit under test. The scalability of JTestCraft can be improved further by implementing other optimizations (chapter 2). The algorithm experiments seem to confirm this, as their results reveal that two other test generators outperform JTestCraft with a significant difference in both the coverage obtained and the explored sequence length reached, since their optimizations allow them to prune most sequences. However, we are confident that adding similar optimizations to JTestCraft should cancel out the performance difference.

Main Research Question

Now that we have answered the three sub-questions, we are ready to answer the main research question:

> "How do we add support for object-oriented software to program-based test generation techniques, such that their scalability to large and complex software is maximized?"

The success of testing object-oriented software depends primarily on the ability to set the unit under test in the right state. This is achieved by generating invocation sequences that consist of constructor invocations, method invocation and field assignments. In this work we proposed the Candidate Sequence Search algorithm that is guaranteed to find all invocation sequences that can influence the unit under test and deals effectively with all object-oriented language features including inheritance and polymorfism. The JPacman experiment confirms these claims and shows that JTestCraft generated a test set that obtains similar coverage as the existing (human-made) test set.

Invocation sequence generation algorithms fail to scale to large and complex software, because the number of invocation sequences that need to be considered grows exponentially with the sequence length. The CSS algorithm addresses this problem by only considering sequences where every invokable could influence the system under test. As a result, considerably less invocation sequences need to be explored. In order to further improve the scalability of the CSS algorithm, we proposed three optimizations that avoid the exploration of sequences that cannot reveal new test cases. Due to time constraints we were only able to implement the Data-flow Check optimization, which almost doubled the maximum length of the explored sequences. We are confident that the Partial Order Reduction and Sequence Grouping optimizations will also significantly improve the scalability of the Candidate Sequence Search algorithm. With these optimizations the CSS algorithm becomes a very effective and efficient solution to generate test cases for object-oriented software.

Although the scalability problem remains the subject of active research, we expect that the CSS algorithm combined with other optimizations, such as those discussed in the literature (chapter 2) and the Constraint Tree data-structure, can already be successfully applied to generate test cases for object-oriented program units of moderate size (at least 10 kloc). We are also confident that the ideas behind the CSS algorithm will form the foundation for future test generation techniques that target object-oriented software.

7.3 Discussion

Based on the results we are confident that the prototype can be used successfully to significantly reduce the effort needed to construct software test sets. In this case, a software developer is only required to write tests for cases that require long sequences, while the rest (most) of the test cases are generated by the test generator. It should be noted that the performance of JTestCraft depends on the testability of the system under test. Hence, to increase the effectiveness of a test generator, the system under test should be designed for testability.

7.4 Future Work

In this section we provide some directions for further research to improve the applicability and effectiveness of future test generators.

Usability

The most important thing that stands in the way of adopting JTestCraft for use in actual software development is that it lacks usability. This can be solved by turning this prototype test generator into a stand-alone GUI application or, even better, into a highly integrated IDE plug-in.

Test Case Prioritization

The current version of JTestCraft outputs a test for every executed path through a method under test. However, in practice software developer are often only interested in generating test cases such that every line of code or every branch in the system under test is executed. Therefore, a future version of JTestCraft should have the option to filter test cases based on a test criteria, such as statement coverage or branch coverage.

Regression Testing

The current version of JTestCraft is not particularly suited for regression testing, since it generates test cases without any assert statements. This can easily be resolved in a future version where the test generator records the values returned by the method under test and the state changes (field and array stores) this method causes. When the executed invocation sequence is transformed into a test case, the recorded values are turned into assert statements.

Test Criteria

JTestCraft attempts to locate test cases that cause exceptions to be thrown, such as assertion exceptions, divide by zero exceptions, array out of bounds exceptions and null pointer exceptions, since these test cases are likely to reveal software defects. Other common sources for software bugs include arithmetic overflows and wrong boundaries in branch conditions, e.g. $i \leq 5$ instead of $i < 5$ [35]. An improved version of JTestCraft should also try to find test cases where overflows occur and where the boundary values are tested, e.g. for the branch condition $i \leq 5$ two test cases should generated where $i = 5$ and $i = 6$. In addition, the software testing literature has an extensive array of other test criteria that should be considered as well [1, 35].

Symbolic Heap Representation Trade-Offs

This thesis discussed the heap simulation and heap reasoning solutions to represent the heap symbolically. JTestCraft implemented the heap simulation solution, since it can be implemented with relatively little effort and produces less complex path constraints than the heap reasoning solution. However, the heap simulation solution can only reason about one array element at a time, whereas the heap reasoning solution reasons about the whole heap configuration. Further research is needed to determine which solution (or combination of both solutions) is best used for automatic unit test generation.

Floating Point Arithmetic

Current constraint solvers can only approximate floating point arithmetic using linear real arithmetic. As a result, test generators are unable to deal effectively with floating point multiplication and division operations, infinite and NaN (Not a Number) values, and rounding errors. Especially these special floating point values and rounding errors are common sources for software defects. Therefore, full support for floating point arithmetic would significantly increase the chances of a test generator locating software defects in code that uses floating point arithmetic extensively. This requires the research of new decision procedures for floating point arithmetic.

Sequence Search Control

The test generation techniques presented in this thesis can be used to automatically generate unit tests for software that require little effort to set up the test environment, such as algorithms and industrial control systems. However, these techniques fail when program units require a specific test environment. For example, program units in a database application could require that a database connection exists. Unfortunately, the test generator tries random network addresses and ports to establish an connection and, as a result, will probably fail to connect. Another problem is that these techniques are not yet suited for integration tests due to scalability limitations.

Both problems can be alleviated when a software developer could provide the test generator with more guidance. For example, in order to test a complete program, a software developer could instruct the test generator to only influence the program state through simulated keyboard and mouse events. While we do not expect this approach to be very scalable, it scales much better than when all methods that could influence the program state are explored. In the case of the database example, the software developer could provide the test generator with an initial sequence that sets up the database connection. This potential solution raises several important questions: ‘In which ways can the software developer provide guidance to the test generator?’ and ‘How effective is this guidance?’

Another possible solution to the test environment and scalability problems are mock objects. Mock objects are objects that simulate the behavior of real objects. In this case, the test generator specifies the behavior of the simulated objects. For the database example it would mean that the test generator simulates the database connection and has complete control over what values are ‘read’ from the database. The advantage of this solution is that the test generator has more freedom to control the execution of the method under test. However, this freedom can also result in test cases that do not represent realistic situations, since the test generator has no idea how the real objects behave. Further research is needed to determine the effectiveness of this solution. We also need to discover if the quality of the generated test cases presents an actual problem and, if so, what solutions there are to this problem.

Bibliography

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008.

[2] Andrea Arcuri and Xin Yao. On test data generation of object-oriented software. In *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 72–76, Washington, DC, USA, 2007. IEEE Computer Society.

[3] Boris Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[4] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. Rwset: Attacking path explosion in constraint-based test generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366. Springer, 2008.

[5] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133, New York, NY, USA, 2002. ACM.

[6] Ugo Buy, Alessandro Orso, and Mauro Pezze. Automated testing of classes. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 39–48, New York, NY, USA, 2000. ACM.

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, San Diego, CA, December 2008.

[8] Christoph Csallner and Yannis Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software Practice & Experience*, 34(11):1025–1050, 2004.

[9] Xianghua Deng, Robby, and John Hatcliff. Kiasan/kunit: Automatic test case generation and analysis feedback for open object-oriented systems. In *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference*

Practice and Research Techniques - MUTATION, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

[10] Michael Ellims, James Bridges, and Darrel C. Ince. The economics of unit testing. *Empirical Software Engineering*, 11(1):5–31, 2006.

[11] Patrice Godefroid. Compositional dynamic test generation. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–54, New York, NY, USA, 2007. ACM.

[12] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[13] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208–215, 1976.

[14] Capers Jones. *Applied software measurement (2nd ed.): assuring productivity and quality*. McGraw-Hill, Inc., Hightstown, NJ, USA, 1996.

[15] Ancel Keys, Flaminio Fidanza, Martti J. Karvonen, Noboru Kimura, and Henry L. Taylor. Indices of relative weight and obesity. *Journal of Chronic Diseases*, 25(6-7):329–343, 1972.

[16] J. King. A new approach to program testing. In *Proceedings of the International Conference on Reliable Software*, pages 228–233. ACM Press, 1975.

[17] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[18] David Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, Chris Chen, Young-Si Kim, and Young-Kee Song. Developing an object-oriented software testing and maintenance environment. *Communications of the ACM*, 38(10):75–87, 1995.

[19] Kiran Lakhotia, Mark Harman, and Phil McMinn. Handling dynamic data structures in search based testing. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1759–1766, New York, NY, USA, 2008. ACM.

[20] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[21] V. Martena, A. Orso, and M. Pezze. Interclass testing of object oriented software. *Engineering of Complex Computer Systems, 2002. Proceedings. Eighth IEEE International Conference on*, pages 135–144, 2002.

[22] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[23] Bertrand Meyer. Applying “design by contract”. *Computer*, 25(10):40–51, 1992.

[24] Leonardo Moura, de and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer Berlin / Heidelberg, 2008.

[25] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[26] RTI. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, May 2002.

[27] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, New York, NY, USA, 2005. ACM.

[28] Michael Sipser. *Introduction to the Theory of Computation, Second Edition*. Course Technology, February 2005.

[29] Frank Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[30] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *ISSTA ’04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107, New York, NY, USA, 2004. ACM.

[31] Stefan Wappler and Joachim Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *GECCO ’06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1925–1932, New York, NY, USA, 2006. ACM.

[32] Mark Weiser. Program slicing. In *ICSE ’81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[33] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381. Springer, 2005.

[34] Tao Xie, Darko Marinov, and David Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE ’04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 196–205, Washington, DC, USA, 2004. IEEE Computer Society.

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