Automated Unit Testing

Developing a prototype for .NET

Mikael Malm and Mads Bach-Sørensen

Master's Thesis - Spring 2008

SW10, project group d620a
Department of Computer Science

Aalborg University
Department of Computer Science
Selma Lagerlöfs Vej 300 ▪ 9220 Aalborg Ø
Phone: 9635 8080

Project Title:
Automated Unit Testing - Developing a Prototype for .NET

Project Period:
SW10, Spring 2008
Feb. 4th - Jun. 4th, 2008

Project Group:
d620a

Group Members:
Mads Bach-Sørensen,
mbs@cs.aau.dk

Mikael Malm,
mmal@cs.aau.dk

Supervisor:
Lone Leth Thomsen,
lone@cs.aau.dk

Abstract

A lot of research has been done in the area of automated unit testing on the Java platform, but almost none for the .NET framework. The goals of this project are to create a prototype tool and document the efforts needed in order to create a complete implementation for automated unit testing on the .NET platform.

An iterative approach has been used to explore the problem and focus has been put on having a running prototype as early as possible where both testing algorithms (back-end) and the integration with an IDE (front-end) has been priorities. In addition some focus has been put on the combination of random testing and symbolic execution as a test generation technique.

The goal creating a functional prototype was never achieved but some important aspects and technical problems associated with automatic test generation for the .NET-platform have been identified. This includes the merit of random testing as a fast and easy to implement test solution, difficulties of creating a plug-in for Visual Studio and most importantly the need for a proper analysis/instrumentation library for CIL code if symbolic execution should be used as a test generation technique.

Publications: 5
Number of pages: 78
With appendices: 80
Finished: June 4th 2008

Signatures

Mads Bach-Sørensen

Mikael Malm

Preface

This master thesis is written by group *d620a* and documents the effort done as part of a 10th semester Software Engineering project at the Department of Computer Science, Aalborg University. It was written during spring 2008 under the Programming Technology group and the theme for the project is Software Testing.

The report documents the development of a prototype tool for automatic generation of unit tests on the .NET platform, and the additional efforts needed in order to provide a complete implementation. It is assumed that the reader is familiar with basic concepts of unit testing, the .NET framework and computer science in general as some of the discussed concepts are not described in detail.

Reading Guide

As an aid to the reader the parts of this report along with a short description of the content is provided below.

Chapter 1 (Introduction) Introduces the problems related to unit testing and document why automating the process would be beneficial.

Chapter 2 (Previous Work) Summarizes the results achieved through our previous work that was a survey of tools and techniques used for automated unit testing.

Chapter 3 (Project Goals) Defines the overall goals of the project and methodology used during the project period.

Chapter 4 (Initial Analysis) Describes the initial analysis and gives an overview of the overall architecture and design of the prototype.

Chapter 5 (Stage 1) Describes the work done during stage 1. This includes analyzing and implementing the initial random testing technique and considerations regarding IDE integration.

Chapter 6 (Stage 2) Describes the work done during stage 2. This includes the implementation of an IDE plug-in and an analysis of symbolic execution. This chapter also revisits the implementation of random testing.

Chapter 7 (Stage 3) Describes the work done during stage 3. Describes the practical experiences gained during the work on implementing symbolic execution into the prototype.

Chapter 8 (Discussion) Discusses and reflects upon the experiences archived during the project period.

Chapter 9 (Conclusion) Concludes upon the experiences and knowledge achieved working with automated test generation.

When using references the type of the reference is capitalized followed by the rank, e.g., Chapter 2 or Figure 1.1. A list of figures, and appendices may be found in the back matter.

Citations refer to the bibliography found in the back matter and are written as a number encapsulated by brackets, e.g., [1]. The entries in the bibliography contain the author of the citation, the title, how it was published, the year of publishing, and other relevant information, respectively.

Acronyms and abbreviations are only used after the term has been written in its entirety. The acronym or abbreviation will be written in parentheses after the first time the term is used, and afterwards it is solely the acronym or abbreviation, which is used, e.g., Control Flow Graph (CFG) the first time and henceforth just CFG.

Content of CD

On the accompanying CD the following content can be found:

- A copy of this report in PDF format
- Source code for the various experiments

It should be noted that, due to the issues found during the course of this project, none of the projects found on the CD will produce a working executable. The source code is only attached as documentation for the work done.

Contents

1 Introduction **1**

2 Previous Work **5**

2.1 Test Case Generation 6

2.1.1 Random Testing 6

2.1.2 Symbolic Execution 7

2.1.3 Other Techniques 8

2.2 Test Oracles 9

2.2.1 Specification Languages 10

2.3 Other Approaches 11

2.4 Summary 12

3 Project Goals **15**

3.1 Prototype 16

3.2 Project Stages 17

4 Initial Analysis **19**

4.1 Architecture and Design 20

5 Stage1 **25**

5.1 Random Testing - Design and Implementation 25

5.1.1 Randoop Algorithm 26

5.1.2 Implementation 27

5.2 IDE Plug-in - Analysis and Design 29

5.2.1 Integrated Development Environment 29

5.2.2 Analysis of Existing Tools 31

5.2.3 Desired Functionality 35
5.3 Summary 40

6 Stage2 43
6.1 IDE Plug-in - Implementation 43
6.1.1 Basic GUI Elements 44
6.1.2 General Observations 47
6.2 Symbolic Execution - Analysis 48
6.2.1 Problems 49
6.2.2 Instrumentation 51
6.2.3 Constraint Solver 53
6.3 Random Testing - Refactoring 53
6.3.1 JCrasher Algorithm 54
6.3.2 Implementation 54
6.4 Summary 55

7 Stage3 57
7.1 Symbolic Execution - Implementation 57
7.1.1 Code Instrumentation 58
7.2 Summary 63

8 Discussion 65
8.1 Test Generation 65
8.2 IDE Integration 67
8.3 Iterative Development 68
8.4 Additional Considerations 69

9 Conclusion 71

List of Figures 73

List of Listings 74

Bibliography 75

A Project Time Schedule 79

CHAPTER 1

INTRODUCTION

Ensuring reliability and correctness of software by testing or verification is a vital part of the software development process, but it is also a costly and time-consuming activity. An activity which according to several studies can account for as much as 50% of the development time[2].

An important step in reducing this cost is minimizing the time before a defect is detected. The longer a defect is left to propagate through the various stages of the development process before it is found, the more expensive it is to repair. This can be seen from Figure 1.1 which also shows that the majority of errors are introduced during the programming phase.

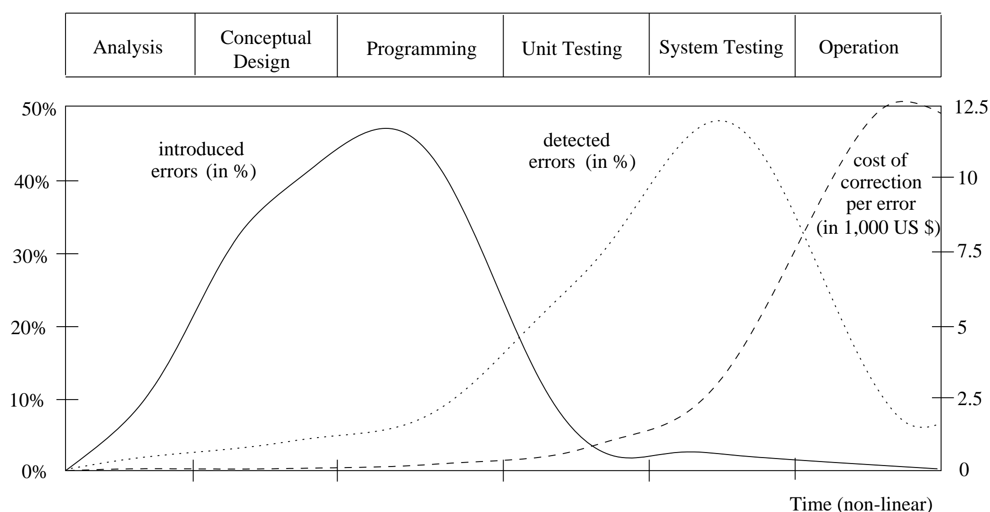

Figure 1.1: System life-cycle: Error introduction, detection and cost of repairing (Source: Liggesmeyer 1998[3]).

From this it should be clear, that whatever can be done to find defects early, and

means to minimize the time spent on testing, can have a substantial financial value to both the developers and the customers.

One of the most effective ways to minimize the number of defects that propagate beyond the programming stage is to do unit testing. Detecting and correcting unit-level defects during the programming phase not only reduces cost, but also lets the testers in Quality Assurance (QA) focus on their prime concern, namely integration and system testing.

Usually unit tests are written manually and only automated in special cases. The problem with this is that manual unit testing it is time-consuming, error-prone and usually not exhaustive. The time factor can to some extent be mitigated by using a unit testing framework such as JUnit[4], which has become the de facto standard when writing unit tests for Java code, or one of the numerous other frameworks available for various languages.

These frameworks usually provide much of the boiler-plate code associated with writing unit tests, as well as an execution framework to run the tests. In many cases the ability to output reports in various formats is also available. While this alleviates the programmer from manually running the test suites and provides for good integration into the build process, in the form of automated regression testing during nightly builds, etc., the programmer is still required to manually write the individual test cases.

This still leaves most of the mentioned problems unsolved:

Time-consuming: The quality of the test is largely dependent on the time used by the developer on a given test case. For any piece of complex code this can be a time-consuming activity.

Error-prone: The prime objective of unit tests is to break code. If the same programmer that writes the code is not intent on finding errors in the code, the unit tests will reflect this.

Exhaustive: Even if the programmer writes a lot of test cases he might not be able to achieve a high degree of coverage. This will diminish the confidence in the code-base.

Automatically generating test cases can solve a lot of these problems, especially if it can be done within a reasonable time. The idea of automatically generating the tests is not new, but many of the program analysis techniques required is computationally expensive and has not been tractable for real-world applications. In the last decade, however, the advances in hardware, constraint-solving techniques, and

the popularization of strongly typed languages such as Java and C# has spiked a renewed interest in this area and a number of research and commercial tools has been developed.

Most of these techniques and tools have been developed in the context of the Java language and to the authors knowledge only two existing tools target the .NET framework: *.Test* which is a commercial tool developed by Parasoft and *Pex* which is currently in development at Microsoft.

This report documents the efforts needed to create a tool to automate the generation of unit tests from .NET assemblies. The work is focused on creating a complete tool for automating the creation of unit tests. This includes trying to examine existing techniques in order to determine how to combine them in order to achieve better error detection, but also how the tool can be integrated into the development environment in order to provide better feedback to the developer.

CHAPTER 2

PREVIOUS WORK

In our previous work[5] we analyzed the current research directions in the field, and surveyed a selection of existing tools. Much of what is described here builds on that work, with the addition of a few new articles which were published more recently.

The chapter is divided into two sections, each dealing with the two major problem areas in automated unit testing:

Test Case Generation: This is the process of determining the input to use when testing a given program. Automatically generating input can roughly be classified into black-box techniques (heuristics, random) and white-box techniques (program analysis) known from the standard literature on software testing.

Generation of Test Oracles: This is the problem of how to automatically generate a heuristic/procedure that can determine whether or not a test case passed or failed. In manual testing this is the done by the programmer, usually in the form of assertions inserted into the test case to check certain program properties, but when automatically generating test cases this becomes much harder as the programmers knowledge of program behavior and specifications is usually not known.

In addition to this a survey of some of the existing tools was also made. These findings will not be described in this chapter, but will be elaborated on in the analysis in Section 5.2.

2.1 Test Case Generation

The generation of test case input is the most researched area of automated test generation and several advances have been made in the recent years. There are especially two techniques that have received a lot of attention: random testing and symbolic execution. Random testing because it is simple and fast, and symbolic execution because it is able to achieve a high degree of coverage and thereby potentially discover more errors.

2.1.1 Random Testing

Random testing has several advantages that make it attractive in the context of automatic test generation: It is relatively easy to implement, it is very efficient with respect to execution time, and because it is a black box technique, i.e. does not depend on the complexity of the code, it scales well to large code bases.

Several tools have implemented random testing, both as a standalone technique as used in JCrasher[6], AutoTest[7], and Randoop[8] or in conjunction with more sophisticated and systematic techniques as implemented in DSD-Crasher[9], Eclat[10], DART[1], CUTE[11] and the commercial tools Jtest[12] and AgitarOne[13]. The following gives a short overview of the tools that only rely on random testing. The tools that use more than one technique either use very similar approaches, in the case of the commercial tools, it is not known exactly how it is implemented.

The implementation in JCrasher[6] generates random input by constructing a mapping from types to methods returning objects of that type. This mapping is then used to generate a graph for each method under test where the nodes represent methods and their arguments, and edges goes from types (arguments) to methods and represent a way to create a value of that type. Creating input for a test case is then done by selecting a random path through the graph.

While the paper provides a thorough description of the method it lacks a good comparison with other techniques making it difficult to judge the effectiveness of the method in practice.

Ciupa et al.[7] implements a random testing strategy in the tool AutoTest. AutoTest maintains a pool of already constructed objects. Based on a probability it chooses whether to create a new object or use one of the existing ones from the pool when an object of a given type is needed. Before an object is added back to the pool there is also a probability that the object will be diversified, i.e. by calling random procedures in order to try to reach new program states. The motivation for this is

that some bugs in object oriented software are first found after an object has been used and mutated. Like JCrasher the algorithm used by AutoTest also starts out from the method under test and tries to construct input by recursively creating the needed objects.

In their results they both report that random testing can in fact be very effective at finding bugs, but also makes the important observation that random testing finds a large number of bugs within the first few minutes of testing.

Pacheco and Ernst[8] describes the work on the tool Randoop. Randoop also generates random test cases, but unlike JCrasher and AutoTest, the test cases are built by creating sequences of method calls starting with sequences only containing predefined values for primitive types. Each iteration of the algorithm picks a random method from the classes under test and tries to construct a new sequence by using values from previously constructed sequences as input. If it succeeds the sequence is added to the pool and can be used to construct new sequences. Randoop, like AutoTest, also tries to introduce mutation of objects. This is done by introducing a probability that a method call can be added to a sequence more than once when selected.

According to the paper this approach is, in some cases, able to achieve both better error detection and coverage than more systematic techniques. The authors also conclude, like Ciupa et al. in [7], that random testing, while effective, can be even more useful when combined with other more systematic techniques.

There are, however, one serious drawback to random testing namely that it fails to exercise program behavior for a large portion of the input domain[2, 14]. The implication of this is that random testing is unlikely to find errors at singularities in the code, i.e. randomly generating test cases is unlikely to find division by zero errors, etc. This does not mean that random testing should be discarded but rather, as several of the papers above conclude that it should be used in conjunction with other, more systematic, techniques in order to improve error detection. The same papers also show, by using simple heuristics in order to mutate program state, that random testing can be viable even when not combined with more computationally expensive techniques.

2.1.2 Symbolic Execution

The most widely recognized technique for systematically generating test input[15] is symbolic execution[16]. Instead of concrete input a symbolic execution engine is supplied with symbolic constants for inputs. As the program is executed assignments along a path updates the program state with symbolic expressions, and every

conditional generates a constraint in terms of the symbolic inputs. These constraints can then be solved, using a constraint solver, in order to generate concrete input values that are guaranteed to execute along the given path.

The advantage of systematic techniques like symbolic execution is that they solve the problem of not testing corner cases, which is the main problem of random testing. There are however several drawbacks to this technique, of which the largest is its inability to scale due to its computationally expensive nature. This is a problem that is also inherit in other systematic techniques such as model checking, i.e. the state explosion problem. Other problems include detection of infeasible paths and how to handle loops. These problems and others are discussed in greater detail in Section 6.2.

There are several tools that uses symbolic execution in order to generate test cases such as Symstra[17], which exhaustively explores the method sequences with symbolic arguments up to a given length. In order to speed up the exploration Symstra employs pruning of the state space based on state subsumption. Results show that Symstra is able to generate method sequences that reach a high degree of branch and intra-method path coverage for a number of complex data structures.

DART[1] and similar projects such as CUTE and jCUTE[18] uses a combination of concrete and symbolic execution (concolic). Symbolic execution is used to generate input that directs the program through new paths. If the symbolic state for a path is too complex to be handled by the constraint solver, the symbolic values are substituted by concrete values allowing the tool to continue. More recently several improvements to the original algorithms have been proposed in order to make them more scalable. SMART[19] extends DART with interprocedural analysis combined with summaries of the pre- and post conditions on methods, and Majumdar[11] adds a random testing step to CUTE in order to drive the analysis until no new paths is found after which symbolic execution takes over.

Other tools using symbolic execution to generate test cases include the Java PathFinder project (JPF)[20], a model checker for Java programs, which combines symbolic execution with its model checking capabilities[21]. The commercial tools Jtest[12] and AgitarOne[13] also uses symbolic execution to some extent, although the specifics is not publicly available. The, not yet released, Pex[22] tool from Microsoft also uses a form of symbolic execution similar the work done in DART/SMART.

2.1.3 Other Techniques

There are a number of other techniques for creating unit tests. One of the approaches is to statically analyze the source code and look for specific patterns.

This is used in both FindBugs[23] and in the two commercial offerings Jtest and AgitarOne.

Trying to automatically infer invariants for the program under test is also a technique that is employed by a large number of tools. Usually this is done with the help of an external tool such as Daikon[24] or DySy[25], and requires a driver program that will exercise the code. The invariant detector then instruments and monitors the program during execution in order to infer likely invariants. The output, in the form of invariants, can then be used by other tools in order to improve test selection and oracles.

DSD-Crasher[9], the successor to JCrasher, uses Daikon to annotate the program using Java Modeling Language (JML)[26]. The annotated program is then processed by the static checker ESC/Java[27]. If ESC/Java finds an error it is reported in the form of a constraint system which DSD-Crasher then solves in order to produce a concrete test case. The generated test case is then executed in order to verify that the reported error is indeed an error. This step is necessary because ESC/Java is unsound and may produce false positives. A similar approach is also used by Jtest and AgitarOne.

2.2 Test Oracles

A test oracle is, as described in the beginning of the chapter, a procedure that given a test case can determine if the test passed or failed. This is closely related to the problem of determining if input for a given test case is valid or not, and is a question of whether or not a specification exists for the program under test.

Consider the example in Listing 2.1. The comments contain the informal specifications for the method which may not even be present, but can be an implicit understanding of how the code is supposed to be used. When manually generating test cases for this method a programmer might write a single test case as shown in Listing 2.2.

```
1 // Assumes that b != null and b.x > 0
2 // Returns 10 divided by b.x
3 int foo(Bar b) {
4   int z = 10 / b.x;
5   return z;
6 }
```

Listing 2.1: Example method with informal specifications.

```
1  void testFoo()
2  {
3    Bar b = new Bar();          // Create input
4    b.x = 2;
5
6    int result = foo(b);        // Call method under test
7
8    Assert.AreEqual(result, 5); // Oracle: make sure we get the
9                                // expected result
10 }
```

Listing 2.2: Manual unit test for the code in Listing 2.1

The test case uses the implicit knowledge that the programmer has in order to create legal input and construct a proper oracle. When automatically generating the test cases either an operational specification for the program has to be present, or the tool has to use heuristics or other means to infer as much as possible about the code.

2.2.1 Specification Languages

An operational specification could be in the form of an embedded specification language such as JML[26] for Java or, as is the case for Eiffel, a part of the core language. The specification can then be consumed by other analysis tools or compiled into the code in form of assertions which are checked on each method call. Listing 2.3 shows Listing 2.1 with JML annotations added.

```
1  // @require b != null and b.x > 0
2  // @ensure \result == 10 / b.x
3  int foo(Bar b) {...}
```

Listing 2.3: JML specifications added to Listing 2.1

In these cases a tool would be able to generate test cases very similar to the one shown in Listing 2.1 except that the test tool would not have to generate the assert statement in line 8. This assertion would already be a part of the normal code.

Generating input that does not produce illegal test cases, i.e. that does not violate the preconditions of the method, also becomes easier. A test tool could either try to create input that would not violate the constraint system expressed by the precondition or simply try to run the test and see if a pre-condition violation occurs.

In the case of AutoTest which is developed for Eiffel this is used with great success[7] and has allowed them to test several large real-world applications using random testing.

In the case of JML the situation is somewhat different. Research conducted by the developers of JML indicate that Java developers in general is very reluctant to add the specifications[?]. This could indicate that relying on the user to annotate the code with specifications is not a viable solution for languages which does not have built-in support. For tools that automatically try to generate the specifications, i.e. invariant detection, this is of course not a problem.

It should also be noted that the type of project is a factor. Certain projects such as mission critical software are more likely to use specifications, because of the high cost associated with program failure, while joe-programmer's notepad might not even have informal specifications apart from 'being able to enter text'.

2.3 Other Approaches

In the absence of specifications a test tool is left with the errors reported by the language. In this case the tool can either consider all exceptions as failed test cases or try to infer the invariants or use heuristics in order to determine if a test case passed/failed or indeed is a valid test. Listing 2.4 shows a test case for the method in Listing 2.1 which could be generated randomly by a tool such as JCrasher.

```
1  @test
2  void testFoo()
3  {
4    Bar b = new Bar();          // Illegal input
5                                // b.x == 0
6
7    int result = foo(b);        // Error: division by zero
8  }
```

Listing 2.4: Manual unit test for the code in Listing 2.1

This test generates a `ArithmeticException` in Java. At the language level it is perfectly valid to consider this a failed test case, in fact there are several tools such as DART and CUTE/jCUTE, that uses this approach.

The problem here is that, for any larger program, this can generate a large amount of test cases. Test cases which the programmer has to inspect in order to determine if it really is an error in the context of the program. The last point here is important because at the user level this is not a failed test, but an illegal one, i.e. the informal specification states that `b.x` is always greater than 0.

JCrasher handles this by filtering the exceptions based on a heuristic. Exceptions are classified and depending on their type and where they originate it is determined whether or not the error is propagated to the user. While this reduces the number

of produced test cases, and in many cases is consistent with what the user expects, there are still cases where the tool reports errors which would be a false positive at the user level. In the case of the example the `ArithmeticException` would be suppressed, but if `foo` is called with a `null` argument the error would be propagated to the user, and this is clearly not an error according to the (informal) specification.

The commercial tools Jtest and AgitarOne uses a large range of techniques such as invariant detection, static analysis, symbolic execution in order to produce a set of observations about the code. These observations can then be promoted to assertions by the programmer, the tools does therefore not try to solve the oracle problem, but delegates it to the programmer. In the case of the previous example Jtest/AgitarOne would report the observation that `b.x > 0` and `b != null`. It would then be up to the programmer to actually mark these as valid assertions.

Pex[22] uses an approach called Parameterized Unit Tests (PUT)[28] which is a generalized version of the standard parameterless test method. The PUT acts as a specification of the behavior of the code under test and describes a set of normal unit tests which can be obtained by instantiating the PUT with different arguments. The PUT for Listing 2.1 is shown in Listing 2.5.

```
1  [PexMethod]
2  void TestFoo(Bar b)
3  {
4    PexAssume.NotNull(b);             // Specification is part of test
5    PexAssume.AreNotEqual(0, b.x);
6
7    int result = foo(b);              // Call method under test
8
9    Assert.AreEqual(10, ((10 % b.x) + (b.x * result))) // Oracle
10 }
```

Listing 2.5: Parameterized unit test for the code in Listing 2.1

Pex then uses symbolic execution in order to generate normal unit tests which call the parameterized unit test with different input. This way the creation of the oracle and specifications is still done by the programmer, but it is done in the context of a unit test which may be more intuitive for people not used to writing operational specifications in languages like JML.

2.4 Summary

Automatically generating test cases has seen a number of advances in the recent years. Random testing and symbolic execution is two of the techniques which show the greatest potential, especially if they can be combined in order to complement

each other and make up for the individual weaknesses. This has already been done with good results in CUTE, and the commercial tools Jtest and AgitarOne takes this even further and combines almost all of the discussed techniques, i.e. random testing, symbolic execution, static analysis, pattern analysis, etc.

Test oracles are a harder to generate automatically. In order to do this properly, either some form of operational specification is needed or the programmer has to be consulted. The latter approach is taken by all the commercial tools; Pex uses parameterized unit tests which can act as a specification, AgitarOne and Jtest both uses the concept of observations which can be promoted to assertions, and in addition Jtest also has its own specification language JContract.

The tools that are a result of research projects either try to infer invariants automatically, which is the case for DSD-Crasher, Eclat, etc. Another approach is to use heuristics in order to filter based the type of errors like JCrasher does. The last approach is to simply disregard the issue and only concentrate on errors at the language level, i.e. like DART and CUTE.

Currently there is no conclusive evidence as to which approach is better (if any), and it also comes down to preference and project constraints. The only conclusion that can be made is therefore that any serious testing tool would do well to accommodate for several different ways to treat the oracle creation problem.

CHAPTER 3

PROJECT GOALS

During the research documented in the previous chapter several observations was made. The first was that there is only one tool that currently targets the .NET framework namely .TEST from Parasoft. Although this particular tool was not surveyed in[5], the information available at their website indicates that the product has the same features as Jtest tool. Pex from Microsoft is another tool that targets the .NET framework, but this tool is at the time of writing still in development and the feature set and release date is not known. Both of these are commercial products and to the authors knowledge no research or open-source implementations exist.

It is the authors' opinion that such a tool would be a valuable contribution. Both as an alternative to the commercial offerings but also as a research platform for future development of testing solutions for the .NET platform.

The second observation was that, except from the commercial offerings, many of the tools either only ran as a command line tool or produced error messages that were hard to decipher. This does not mean that the tools themselves were poor, but it does take the programmer a longer time to switch between tools in order to get the needed information. The only tool, apart from the commercial ones, that provided good information about errors and fixes was FindBugs.

The authors argue that in order to maximize the time and development effort that can be saved by using an automated tool, and in order to encourage its use, it is essential that a tool provides good integration into the development process. This includes proper error messages, solutions to fixes, reports, etc. This information should be provided to the programmer while he is coding so possible defects can be handled as quickly as possible. Integrating the testing tool into an Integrated Development Environment (IDE) could provide all of this, but would also allow the tool to provide visual clues and automatic refactoring in order to fix bugs.

The last observation was that only the commercial tools provided a wide range of techniques and options in order to maximize the number and types of bugs found. This is of course understandable as many of the tools are research projects and therefore focuses on one or two techniques, but as our previous research concluded it would be preferable if several different techniques could be combined.

If the development-time of any of the existing tools for Java is any indication, creating such a tool is a multi man-year effort, and clearly outside of the scope for a semester project. The goals of this project are therefore to develop a simple prototype in order to uncover any potential problems with creating such a tool for the .NET platform, and provide a basis for further development. The prototype should still incorporate all of the mentioned elements, but with reduced functionality.

3.1 Prototype

The prototype should both be able to function as a command line tool and as a plug-in for an IDE. As a command line tool the output should be in the form of NUnit test cases and when running under an IDE the output should be in the form of visual clues in the interface. In order to generate the test cases the prototype will combine random testing with symbolic execution.

The motivation for choosing a combination of random testing and symbolic execution has several aspects. The first is that the results from our previous works showed that this is a very promising combination, that could provide very high error detection rate as well as good coverage.

The second is that random testing will allow a fast initial implementation. This can be used when implementing the functionality in the IDE to identify problems with the interface between the backend tool and the IDE frontend as early as possible.

The next is that it is assessed that symbolic execution is the most difficult of the testing techniques to implement. Not only is there a lot of inherent problems in the technique, but it also requires a lot of auxiliary functionality in order to work, i.e. code instrumentation, constraint solving, etc. Identifying these as early as possible in order to find or build the needed functionality could provide a framework that future extension can build on, as it is not expected to be able to do a full working implementation.

The prototype will not cover the creation of oracles or any advanced functionality in the IDE. Furthermore the prototype will only be implemented to handle very basic programming constructs and types, but a description and analysis of the more

advanced problems will be provided where appropriate.

3.2 Project Stages

In order to structure the development of the prototype the project was divided into several stages. The motivation for this is that while the authors' previous survey and work on the subject provided a good overview, there are still a lot of areas that need further analysis and research in order to provide an implementation.

Using an iterative model and using prototyping would allow research for the next stage to be conducted in parallel with the implementation of the prototype for the current stage. The initial plan is summarized in the following:

Initial Analysis: Initial analysis in order to see how much of the needed functionality can be taken from existing libraries. This also contains the initial design of the architecture.

Stage 1: The design and development of a prototype implementation of a random testing tool. In parallel to this the analysis and design of the IDE plug-in is conducted.

Stage 2: This stage should see the implementation of the IDE plug-in and the analysis of symbolic execution module.

Stage 3: The implementation and integration of symbolic execution into the prototype.

As the rest of this report will show there were several implications and problems, which resulted in development not being able to follow this initial plan. These were mostly related to the implementation of the IDE plug-in and symbolic execution and ultimately resulted in not being able to produce a working prototype. This will be discussed and reflected upon in Chapter 8.

CHAPTER 4

INITIAL ANALYSIS

Before the development of the prototype started an initial analysis was conducted. The purpose of doing this analysis was to examine if some of the needed functionality could be provided by already existing libraries.

Many of the tools for Java depend on existing libraries for specific functionality such as reflection, instrumentation, and byte code analysis which can be done via libraries/projects such as Soot[29] and BCEL[30]. Even whole parts of the program such as symbolic execution could possibly be provided in the form of JPF[31].

The assessment was that in order to develop a tool for .NET it would also be necessary to depend on external libraries for at least some of the functionality. Partly because implementing the functionality would be too time-consuming, but also because it would remove focus from the goal of creating an automated testing tool.

Most of the functionality needed is related to symbolic execution, so the ideal situation would be if that entire part of the program could be provided as an off the shelf component. Unfortunately no such library was found. It was concluded that in order to perform symbolic execution at least code instrumentation had to be performed. A constraint solver also had to be at hand to help solving the path constraints generated by symbolic execution.

In order to do symbolic execution it is necessary to instrument the code, and therefore a proper library for instrumentation/analysis is needed. By investigating tools and libraries available for the .NET platform the following was found:

- Phoenix[32] (Microsoft Research)
- RAIL[33] (University of Coimbra)
- .NET profiling API[34] (Microsoft)

In addition to instrumentation a constraint solver is also needed. Many of the Java projects use the *Simplify*-solver[35], but several other solvers are available including a solver developed by Microsoft Research called *Z3*[36] that provides a managed API.

Due to these observations the assessment was made, that despite the lack of a tool able of performing actual symbolic execution, it would still be possible to integrate already existing libraries into a tool in order to gain the functionality needed for symbolic execution.

4.1 Architecture and Design

In this section the overall architecture of the tool is described. The architecture mainly consists in two parts. The first part consists of a command line interface and a plug-in for Visual Studio. These elements are the top layer of the architecture, the *frontend*. The other part describes the actual tool and all of the elements needed for supporting automatic test generation. This part is the lower layer of the architecture, the *backend*. This can be seen in Figure 4.1.

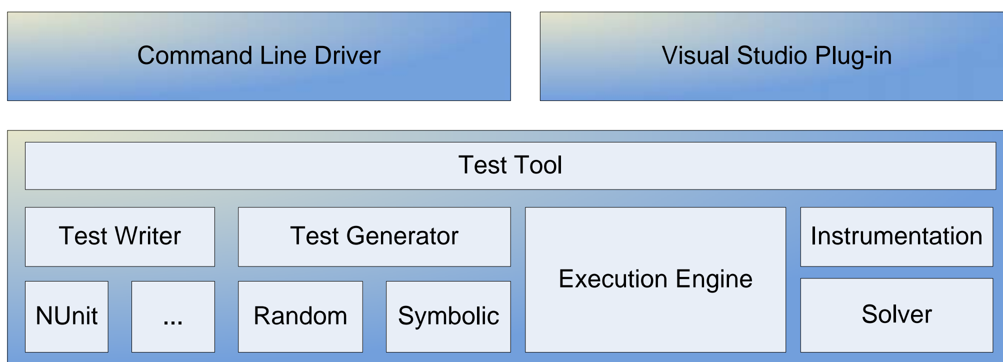

Figure 4.1: Overview of the test tool.

The design focuses on the elements in the backend of the tool. The analysis of the plug-in part will be described through the analysis of graphical elements seen in other tools and designed based on these observations. The description of this process can be found in Section 5.2.

Figure 4.2 shows the package diagram of the packages identified to be essential in the test tool. As implied from the figure the package called *Tool* is the central package of the tool. It is through this package all the functionalities are linked together and exposed to the world.

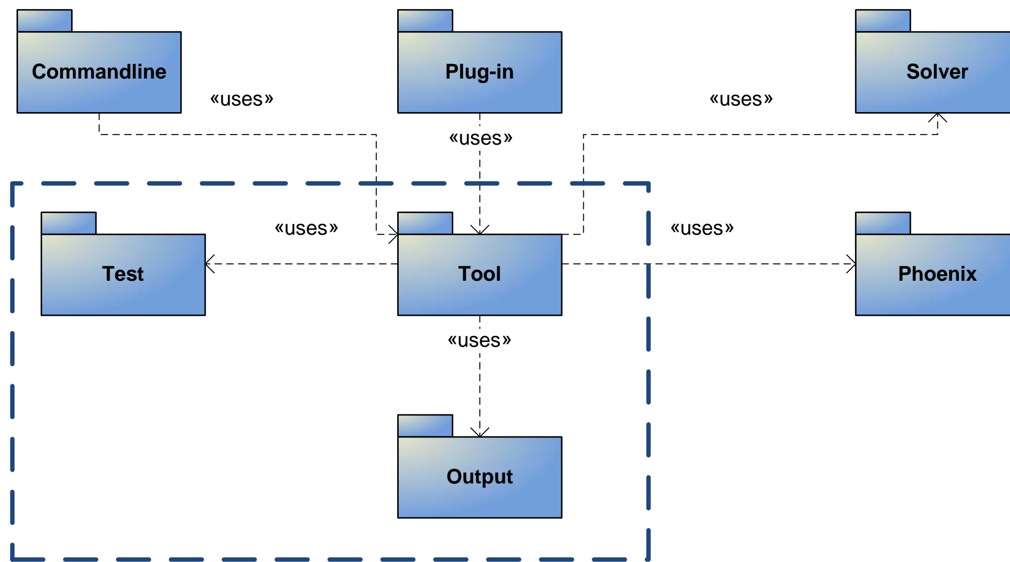

Figure 4.2: Package diagram of the tool.

The two packages associated with the user interfaces *Commandline* and *Plug-in* makes use of the functionalities exposed by the tool-package. The packages *Phoenix* and *Solver* contains functionalities associated to symbolic execution. Subjects related to these packages will be explained in Section 6.2 and Chapter 7.

The main focus of this section is the packages enclosed by the dashed line in Figure 4.2. The goal of the architecture is to create a solid foundation both facilitating a fast implementation of a prototype supporting basic functionality, and allowing future extensions.

A class diagram illustrating the design of the packages *Tool*, *Test* and *Output* can be seen in Figure 4.3. The plan is to base the implementation on this design, and then change the design if needed based on how the project evolves.

In the following the main purpose of the classes in Figure 4.3 is briefly described.

TestTool This is the main class of the tool. It exposes the functionality of the tool to the outside world and handles initialization tasks.

Testee This is mainly a container class holding information needed to handle a test project like output path etc. The class also contains the *Class*- and *TestCase*-objects associated to the assembly under test.

Class This is a wrapper for the reflected type of a class. Also holds a list of *Method*s.

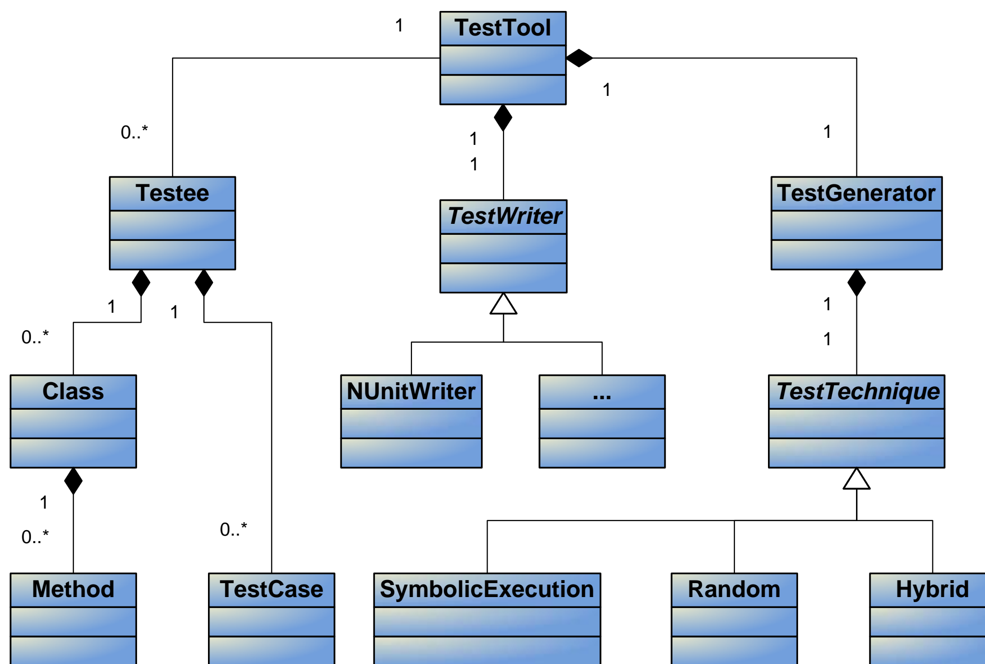

Figure 4.3: Preliminary class diagram of the test tool.

Method Wrapper class for the methods and constructors found in *Class* through reflection.

TestCase Represents a constructed test case. Can either be serialized to a file or executed directly.

TestWriter This abstract class that facilitates writing *TestCases* to files. Concrete test writers for various languages/frameworks inherit this class.

NUnitWriter Example of a concrete test writer. It writes NUnit test cases to a file. Other variants could be implemented.

TestGenerator This class is responsible of generating tests based on the technique of choice. The generated tests can then be executed by the tool or serialized to disk.

TestTechnique This abstract class holds general information needed to execute concrete a *TestTechnique*. A concrete *TestTechnique* inherits form this class.

Random The concrete implementation of a random test generation technique, that can be executed by the *TestGenerator*.

SymbolicExecution This class is not initially implemented but should contain an concrete implementation of a test technique based on symbolic execution.

Hybrid This class is not initially implemented but should contain an concrete implementation of a test technique based on a hybrid of a random- and symbolic technique.

CHAPTER 5

STAGE1

This chapter describes the research and work done during the first stage of development on the prototype. The goals of this stage is to create an initial implementation of the tool using one of the random testing algorithms and to analyze and design the interface portion of the IDE plug-in.

5.1 Random Testing - Design and Implementation

In the process of selecting a random testing algorithm for the tool the techniques employed by the JCrasher[6], AutoTest[7] and Randoop[8] were examined.

The advantage of the JCrasher approach is that it creates a graph of the input space up front. This allows the testing process to limit the number of generated test cases for a method either by limiting the depth of the graph or by counting the number of possible parameter combinations and only selecting a subset. This can be done before actually creating or running any of the test cases. The disadvantage is that the technique is very simple and does not try to mutate objects in order to reach new states that could produce errors.

AutoTest uses an approach very similar to JCrasher, but uses heuristics that for each iteration of the algorithm determines if an object is reuse or mutated. The drawback of the algorithm used by AutoTest is that it only allows limiting the running-time based on a time limit.

Randoop, like AutoTest, also reuses objects and provides mutation, but instead of working backwards from the method under test like JCrasher and AutoTest, it works its way forwards, and only appends a method to a sequence if it can be constructed from previously constructed objects.

The results from the Randoop project show that they get a high degree of coverage. Unfortunately Ciupa et. al. used error discovery rate as a metric in the AutoTest paper, making it difficult to compare the two techniques.

The reason for focusing on the mutation of program state is that it creates better coverage, i.e. different states explore different program paths. This is important when combining random testing with symbolic execution, because one of the objectives of doing this is to explore as much of the program as fast as possible, and only switching to symbolic execution when no new paths are explored.

For this reason it was decided to implement random testing as described in Randoop[8] in the prototype tool. During the analysis of symbolic execution in stage 2 of the project this decision proved to be less than optimal because of interface problems between the random and symbolic modules. This will be elaborated on in Section 6.3.

5.1.1 Randoop Algorithm

The Randoop algorithm builds the test cases by constructing sequences of method calls. It starts out with a number of sequences containing predefined values for primitive types and built-in types such as arrays. For each iteration of the algorithm a random method from the classes under test is chosen. If the method can be called using any of the values from the existing sequences, the sequences are concatenated and the method call is appended producing a new sequence.

The new sequence is then executed and if no error is found it is added to the pool of sequences that can be used on subsequent iterations. Listing 5.1 shows the pseudo code for the base algorithm. It should be noted that this is a reduced version of the algorithm, where anything related to oracles/contracts is left out. The full algorithm can be found in [8].

The `execute()` method in Listing 5.1 simply executes a sequence and returns the result, the other methods related to the selection and construction of a new sequence is described below.

`selectRandomMethod`: This method first selects a random class from the set of classes under test. From this class a random public method or constructor is selected.

`randomSequences`: This method takes a list of existing sequences and a list of types representing the input arguments if any. For each of the arguments it adds a sequence to `sequences` and an index specifying the value to use

```
1  GenerateSequences(classes, contracts, filters, timeLimit)
2  {
3    errorSequences  := {} // Contract violation or exception thrown
4    normalSequences := {} // No contract violations or exceptions
5
6    while(time() < timeLimit)
7    {
8      // Create new sequence
9      m(T1...Tk) := selectRandomMethod(classes)
10     <sequences, indexes> := randomSequences(normalSequences,
           T1...Tk)
11     newSequence := extend(m, sequences, indexes)
12
13     // Discard duplicates
14     if(isDuplicate(newSequence))
15       continue;
16
17     // Run sequence and check for errors
18     outcome := execute(newSequence)
19
20     if(isError(outcome))
21        errorSequences += newSequence
22     else
23        normalSequences += newSequence
24   }
25 }
```

Listing 5.1: Pseudo code for the algorithm used by Randoop[8]

for the argument is added to `indexes` . If the argument is a primitive type the sequence is selected from a fixed pool of sequences that only contain one primitive value. If the argument is a reference type either a sequence containing the `null` value is used or a value produced by one of the previous sequences.

extend: The extend method produces a new sequence by concatenating the sequences in the order they appear and appending the method `m` .

5.1.2 Implementation

The implementation of the randoop algorithm closely follows the description in the previous section. There is however two issues related to the representation of the sequences that warrant some consideration.

The first is how to represent sequences so that they are easily serialized by the `TestWriter` , i.e. written as NUnit test cases, but also in a way so they can be easily executed. The second is how to represent input as a the output from a previously executed statement in the sequence.

Sequence Representation

In the prototype a sequence is represented by the `TestCase` class found in the class diagram in Figure 4.3 which holds a list of `Statements`, i.e. a test case is a sequence of statements. The `Statement` class is then specialized based on the type of statement. This can be seen in the class diagram in Figure 5.1.

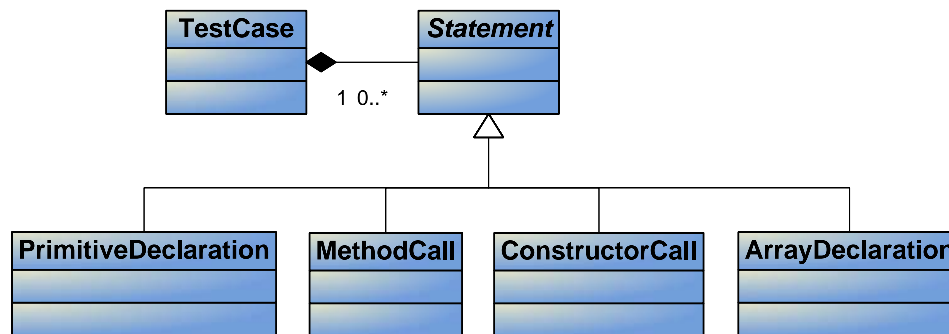

Figure 5.1: Class diagram for the `TestCase` class.

The abstract base class `Statement` holds the reflected type of the statement, a value representing the outcome when executing the statement, and exposes two abstract methods: `Execute()` and `ToCode()` wich is implemented by the sub-classes. The specialized classes also holds additional information, such as method parameters in the case of `MethodCall`.

When a `TestCase` is written to a file by the `TestWriter` this is simply a matter of traversing the list of statements calling `ToCode()` to obtain a string representation. In the case of `PrimitiveType` this would produce a string like `int arg0 = 0;` or `string arg1 = "Hello World"`. The `TestWriter` is responsible for writing the actual method bodies of the test cases.

Executing a `TestCase` is done in a similar manner, i.e. the list of `Statemets` is traversed and `Execute()` is called on each statement. For a statement taking input, such as a `MethodCall`, the value of the statements producing input for the method is obtained from the previous statements and reflection is used to execute the call. The *value* of a statement is the return value of the method call or in the case of constructors a new instance of the enclosing type.

It should be noted that `PrimitiveStatements` cannot be executed and always holds the value that was assigned at creation. Calling `Execute()` simply does nothing.

The `TestCase` monitors the execution outcome of the entire sequence and records if any exceptions was thrown or it was a normal run. This information is used by

the algorithm as shown in Listing 5.1 to determine if the `TestCase` can be used to produce further test cases (sequences).

Input Representation

There is one practical issue regarding the representation of input values as the result of a previous statement in a sequences. This is when the `extend` operation from Listing 5.1 is called it has to concatenate sequences. If the index is stored as a normal index from the beginning of the sequence it would have to be recalculated each time two sequences is concatenated.

The authors of Randoop notes this as a bottleneck and has proposed to store the index as a relative negative value from the current position. The result of this is that the concatenation operation is much cleaner as it does not need to modify any indexes during concatenation.

5.2 IDE Plug-in - Analysis and Design

To encourage as many developers as possible to use an automated testing tool it is of importance that the tool is fast, that it is easily accessible and that it is easy to use. For this reason an integration of the prototype into an IDE has been found beneficial.

This section will serve as the analysis and design of the IDE-plug-in. This includes an analysis of already existing tools and the graphical elements they use. The elements desired in the implementation of the plug-in will be pointed out and described in details.

5.2.1 Integrated Development Environment

Before the implementation of the plug-in can begin, the IDE to host the plug-in has to be chosen. The investigation of potential candidates was conducted simultaneous with the analysis and design of the plug-in. The requirement was that the IDE had to support C# since the test cases produced by the tool are serialized as C#-files.

Below three tools that all supports C# are listed. They all have in common that they are available in a free version. Several commercial IDEs are available but none of them are mentioned here.

Microsoft Visual Studio The most conspicuous product for developing .NET and C# is at the moment *Microsoft Visual Studio* which in addition to being available is several different versions under a commercial license also comes in a slimmer version targeting C# namely *Microsoft Visual C# Express Edition*[37]

SharpDevelop is an open source free to download IDE for C# written in C# by ic#code[38]

MonoDevelop is an IDE that addresses the Mono implementation of .NET, which allows .NET code to be executed on Linux[39]

It seems that Microsoft Visual Studio is the most widespread IDE targeting .NET and C#. Although this could not be documented, it has been decided to target Visual Studio. More specifically it was decided to target Visual Studio 2008 which is the newest version in the series.

A screenshot of Visual Studio 2008 showing the different areas in the editor can be seen in Figure 5.2.

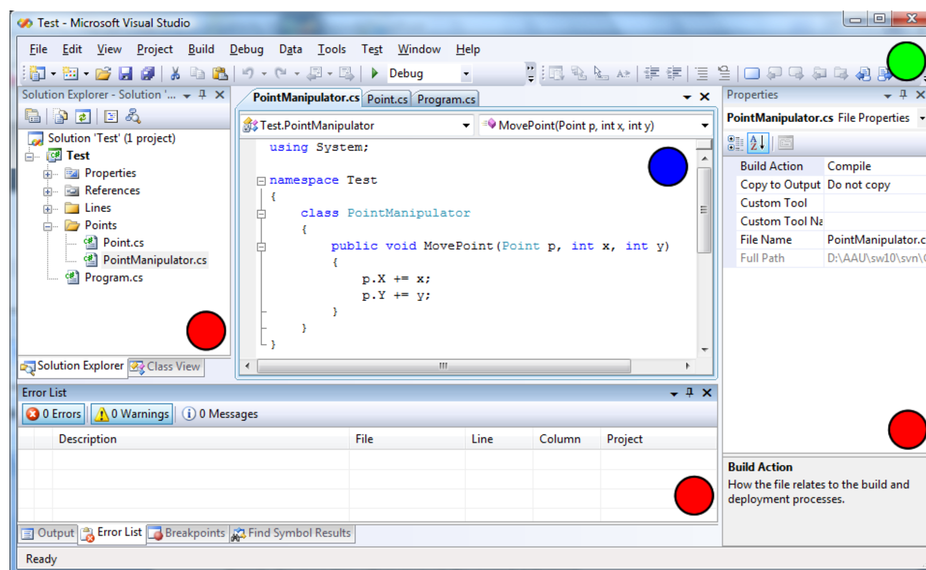

Figure 5.2: This figure illustrates the Visual Studio 2008 IDE. Areas marked with • are tool windows. Areas marked with • are editors and designers. Areas marked with • are commandbars.

Basically Visual Studio offers three types of graphical user interfaces. These are *Menus and Toolbars*, *Tool Windows* and *Document Windows*. Document windows can be used to host editors such as graphical designers and text/source editors.

When it is chosen to extend the functionality of Visual Studio, the programmer has three ways of doing this. These are described in the following.

Macros can be used to record actions in Visual Studio. These are recorded as Visual Basic code and can be saved and replayed. Macros can be bound to menus and toolbars. [40]

Add-ins are more powerful than macros. They makes it possible for the programmer to interact with many of the features and tools in Visual Studio. This includes code editors, the code model, output and debugging windows, editors, menus, and more.[40]

VSPackages are the most powerful way of extending the functionalities of Visual Studio. Visual Studio itself is composed of VSPackages and they provide deeper integration with the IDE. [40]

From the Visual Studio SDK webpage it appears that VSPackages are more complex to integrate than add-ins because the rely on different extension models. It would be preferable to use add-ins for the extension since these rely on a simpler model, but because of contradictory information on what the different approaches allows the programmer to gain access to, it was decided to go for the complex approach by using VSPackages. This approach however turned out to be somewhat overkill, because it appeared that using add-ins would properly have been sufficient. This will be discussed in Chapter 8.

5.2.2 Analysis of Existing Tools

The following section will serve as an analysis for the integration of the plug-in in an IDE. The analysis should shed light on the elements needed for user interaction with the plug-in, it will be conducted through the investigation on the visual elements provided by existing tools.

In our previous work[5] three test-tools targeting Java were examined. The two commercial tools AgitarOne[13] by Agitar and Jtest[12] by Parasoft both came as plug-ins for Eclipse. The academic tool jCUTE[41] by Sen et al. has a standalone Graphical User Unterfaces (GUI) allowing the user to run tests and view the resulting test cases and errors found. AgitarOne and Jtest are very similar and only Jtest will be considered.

Beside these tools, a tool called FindBugs[23] described in Section 2.1 is also reviewed. FindBugs is available both as a standalone application and as a plug-in for Eclipse.

The main elements found during the analysis will next be described briefly and listed. The analysis will not deal with testing algorithms or implementation details it will only point out graphical elements.

Jtest

Figure 5.3 shows a screenshot of the Jtest perspective in Eclipse. The list below explains how errors are highlighted and communicated to the user. The numbers in the figure maps to the list.

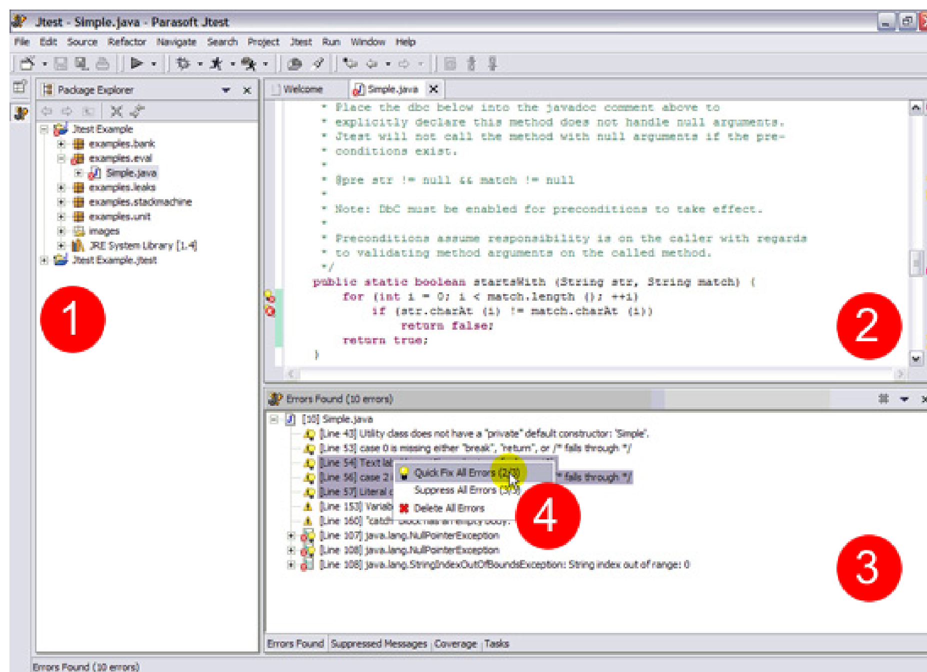

Figure 5.3: Screenshot showing the Jtest perspective in Eclipse (Source: Parasoft 2007[12]).

1. In the package explorer pane packages and classes containing problems are marked with a red icon.
2. In the code editor pane the lines containing errors are marked with an red icon in the gutter. Icons illustrated as a *light ball* indicates that Jtest has a proposition for a *hot fix*.
3. In the information pane in the bottom of the screen the programmer can view a comprehensive list of all errors and problems found in the program.

4. By selecting on one or more of the problems listed in 3, a tooltip appears allowing the programmer to apply *hot fixes* suggested by Jtest. It is also possible to suppress or delete problems.

Beside the functionality described her both Jtest and AgitarOne has coverage analysis capabilities. This is described in our previous work[5]. This functionality consists in that every statement executed under test is marked with green color in the source code editor to illustrate that the statements has been tested. After a test run test reports are generated showing which parts of the program, that has been tested, helping the programmers to gain the appropriate confidence in the program.

jCUTE

In Figure 5.4 a screenshot showing the jCUTE GUI can be seen. As mentioned jCUTE hasn't been integrated in an IDE. The numbers in the figure maps to the list.

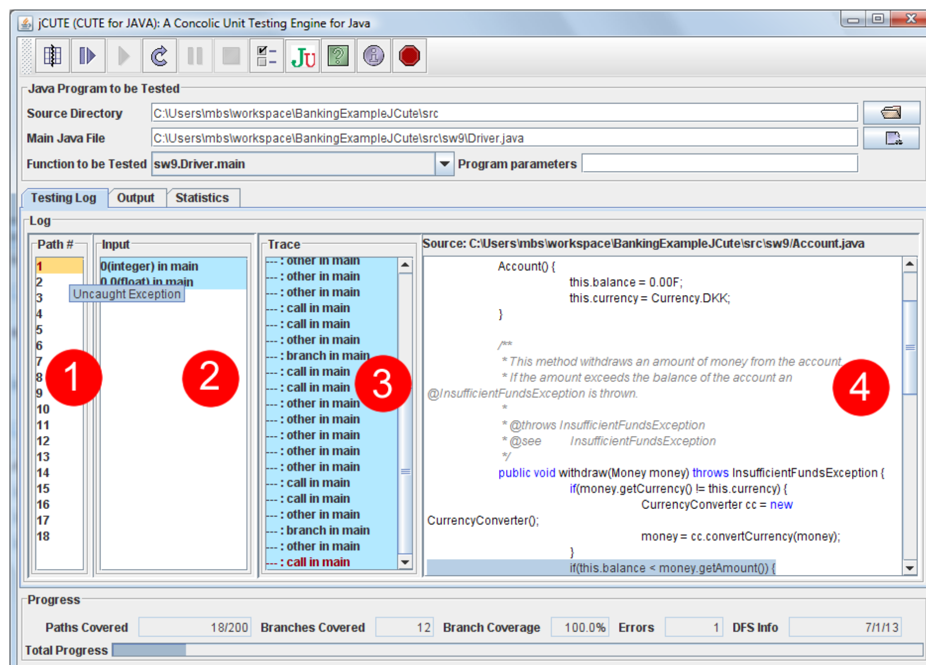

Figure 5.4: Screenshot showing the jCUTE user interface.

1. The path pane shows a comprehensive list of the different paths tested by the tool. By selecting a specific path the content in 2, 3 and 4 will change. If an error has been found in a path, the path will be highlighted with yellow.

2. The input pane show the initial input parameters used to drive the test case through the specific path.

3. In the trace pane the programmer is able to explore the full call trace. If a call results in an error, the specific call will be marked red. A red call will always be the last call in the trace.

4. In the source code pane, the programmer can view the source code associated with the different calls in the call trace.

   By selecting a call in 3, the source code pane will change. The failing statement will be highlighted with blue in the source code pane.

FindBugs

In Figure 5.5 a screenshot showing the FindBugs perspective in Eclipse can be seen. The numbers in the figure maps to the list.

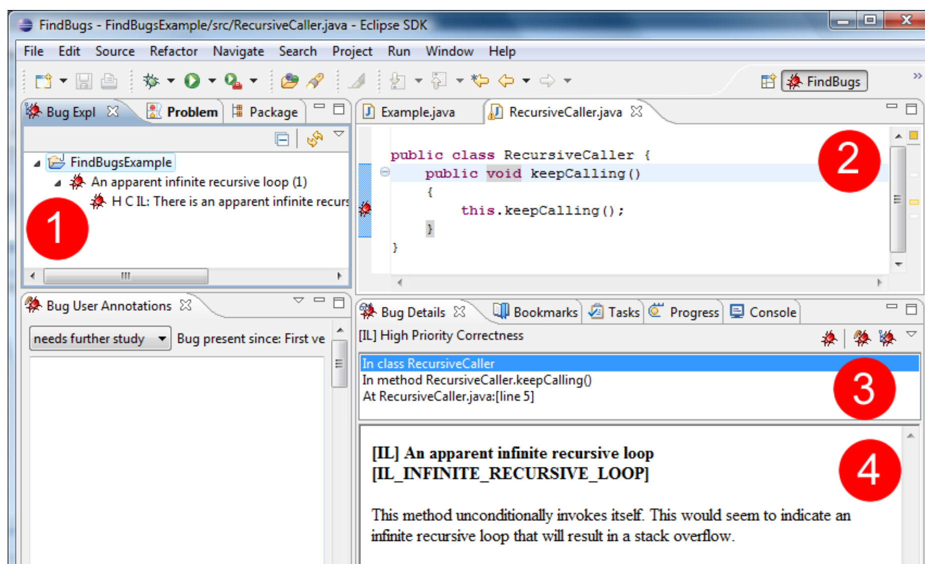

Figure 5.5: Screenshot showing the FindBugs perspective in Eclipse.

1. In the bug explorer pane all problems encountered are organized in a project structure. For each project the problems are organized in different categories like e.g. *infinite recursive loops* or *possible null pointer references*.

   By selecting a specific problem the content of panes 2, 3 and 4 will change.

2. In the source editor pane the source code is shown and is editable like in the normal Java perspective. The problems found by running the FindBugs-tool are enumerated with a small red *bug* in the left gutter.

   By clicking this icon the content of panes 1, 3 and 4 will change.

3. In the bug details pane the programmer can see in which class, method and line the selected bug is located.

4. In the bottom part of the bug details pane, the programmer can view a detailed description of the selected problem.

   In this case the problem is a method, that recursively keeps calling itself.

5.2.3 Desired Functionality

In this section some selected features and functionalities pointed out in the previous section will be explained in more detail. These are considered as candidates for implantation in the plug-in. Examples on how the IDE should communicate with the programmer and vice versa will be illustrated through some mockups and examples from existing tools.

Package Explorer

To help the programmer gaining an overview of which parts of the program that contains errors, the folders and classes containing errors should be visually marked in the package explorer window. This could be done as it was seen in FindBugs and Jtest. An example on how this could look like is illustrated in Figure 5.6 which shows a mockup of a package explorer.

Source Code Editor

Figure 5.7 shows a mockup of a source code editor. In this case a possible *null-reference error* has been encountered. This is due to the fact that the method *Move-Point* could be called with `null` as an argument.

The error is in this case marked with a red and black squiggly under the statement as well as an error icon in the left gutter.

In the case illustrated in the figure only one error is marked. The reader would notice that the statement following the first error contains an equivalent problem. The

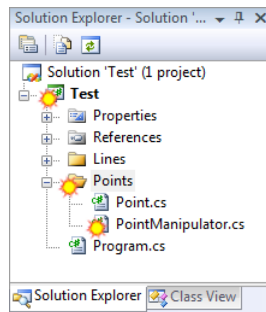

Figure 5.6: This figure is a mockup showing how errors should be visualized in the package explorer. In this case an error has been encountered in the calss *PointManipulator*, which is located in the folder *Points* and is part of the project *Test*.

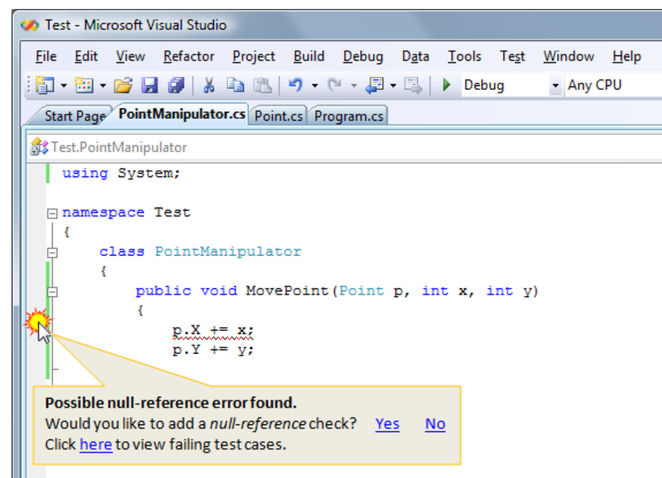

Figure 5.7: This figure is a mockup showing a source code editor.

reason for this is due to the fact that when checking the code a test case traversing the specific path of the program and uses `null` as an argument to the *MovePoint* method would throw an exception when the first statement is reached and thereby the execution will never reach the second statement.

If the programmer is hovering the mouse over the icon or the underlined statement a tooltip should appear applying some useful information on the problem. If the tool has knowledge about an applicable hot fix to the problem, the user should be able to apply this fix by answering *yes* to the question *"Would you like to add a null-reference check?"*. The tooltip functionality could also provide an easy way to access the source code of the failing test case that triggered the error.

Problem Details

It is important to provide the programmer with useful and understandable information on the problems encountered during testing. It is not sufficient to print a line number and inform that the line contains an error. Many programmers have experienced some of the less informative syntax-error messages provided by some SQL engines that could say something like this: *"Query failed: ERROR: syntax error at or near "MyTable" LINE 42: SELECT. . . "*.

This problem is of course somewhat different because the mentioned message deals with a syntax error and the errors found by the tool would be runtime errors. The point is however that it would be much easier for the programmer to correct an error, if it is known in details what kind of an error that has been found and where and why that error caused an exception to be thrown.

In order to provide the programmer with useful information on the problems encountered, the IDE should offer a tool window that shows what kind of an error that has been found e.g. like it is done in FindBugs.

Beside the detailed description of the problem FindBugs provide information on which class, method and line the error is located in.

Quick Fix Functionality

Earlier the quick fix functionality of Jtest was mentioned. Such a feature would help the programmer to easily modify some faulty source code.

The quick fix functionality could be based both on pattern matching and symbolic execution. By using pattern matching, the system would be aware of bad/dangerous patters and recognize these. Then the tool could offer to change the code to a known solution to the problem. A simple example could be to check the nominator in a division for zero and add a *throw IllegalArgumentException* statement to the source code.

If the symbolic execution approach is used, the tool could use the different constraints and try to find a counterexample, which could either be promoted to a check in the source code or a new test case. Something similar to this is used by AgitarOne and Jtest.

An example of how the user interface of the quick fix feature could be implemented can be seen in figure Figure 5.7.

Coverage Functionality

The commercial tools Jtest and AgitarOne discussed in our previous work[5] both had some build in coverage functionality. Such a feature can be good because it can help influence the programmer's confidence in the program under test. Standalone coverage tools are also available. Two examples are Clover[42] by Atlassian and NCover[43] by Gnoso which respectively are used to generate coverage reports for Java and .NET programs.

These tools typically generate coverage reports based on how many lines in a program that have been executed. If a program has 100 lines of code and 80 of them has been executed under test, the coverage report will show that 80% of the program has been tested.

In our previous work different approaches for generating coverage reports were discussed. It was found that predicate coverage provides a better impression on how much of the program that has actually been tested. Predicate coverage is sometimes referred to as path coverage. By using predicate coverage it is the number of paths through the program that is basis of the coverage calculation.

Of course there are some problems when choosing predicate coverage. Like in symbolic execution unbounded loops and recursive calls have to be handled in some way. One way of doing this is to decide that one iteration of a loop or a recursive function is sufficient to gain full coverage. That is if all statements in the body have been executed at least once.

To visually support this functionality the code in the source editor could be highlighted like it is seen in e.g. Clover and base this highlighting on line coverage. See Figure 5.8. Additionally the simple coverage highlighting could be supported by the more specific predicate coverage. This could be done by listing the possible paths through the program like described next, and then highlight the paths that have been traversed by the tests.

The user could be presented to the results by listing the coverage results by program, package, class and method and do this for both statement coverage and path coverage. This could be done something like shown in Table 5.1 on page 39

Program Path List

To support the coverage functionality just described a comprehensive list of possible paths through the program would be of some use. The list would only be comprehensive in the manner that paths through recursive functions and unbounded loops

Figure 5.8: Screenshot of showing some of the coverage functionalities in Clover. (Source: Atlassian 2007[42])

Statement coverage	70%	(70/100)
Path coverage	50%	(20/40(∞))

Table 5.1: This table shows a possible outcome of a coverage report generated for a method. The method has 100 lines of code and 40 different paths, when recursive calls and unbounded loops are only counted once. The infinity symbol indicates that the method in this case due to a loop has an unknown number of paths.

would only be counted once.

A list like the one implemented in jCUTE and illustrated on Figure 5.4 could be used. Like just described this list could be highlighted and illustrate which paths of the program that have been tested. By clicking on one of the paths the associated test case(s) could be shown.

Other Elements

Beside the features and functionalities mentioned in the preceding sections some additional elements have to be implemented in the IDE. These are mostly associated with menu items. An example would be a settings window in which the programmer would be able to change the configuration of the tool. As an example the programmer should be able to change the folder structure of the output files and change the settings for the testing algorithms.

The settings for the testing algorithms should be highly configurable to allow the programmer to be in control of the memory and time needed for testing. This could be done by allowing the user to specify the maximum depth of the graphs used by the symbolic execution and the length of sequences used by the random testing

library. The user should also be allowed to change other settings like turning on and off an eventual *test while typing* feature.

In the following list desirable features for an IDE integration are listed. The comment in parentheses indicates which kind of Microsoft Visual Studio 2008 component that is needed for the implementation.

- Package explorer (Tool Window)
- Source code editor (Document Window)
- Problem details (Tool Window)
- Quick fix functionality (Document Window, Tool Window)
- Coverage functionality (Document Window)
- Program path list (Tool Window)

All the elements in the list of functionalities will not be immediate candidates for implementation. The initially implementation should only treat the most basic needs. Figure 5.7 illustrates this need. To proof the concept the Visual C#-source code editor should be extended to visualize where an error has been encountered. This should either be done by making a tag in the gutter or by underlining a statement like shown in the figure.

The experiences gained during integrating with the Visual Studio IDE are described in Section 6.1.

5.3 Summary

The GUI of the existing tools *Jtest*, *jCUTE* and *FindBugs* have been explored in order to determine how encountered problems are reported. The graphical elements used by these tools were identified and it was found that they all use individual methods related to the way their algorithms works, but that they all maps the problems found to the source code editor to notify the programmer. The different visual elements and how they could be used by the tool was also discussed.

Although an initial analysis indicated the integration of a tool in Visual Studio 2008 would be challenging it was decided to target the IDE integration to this environment. The different ways of doing integration with Visual Studio were briefly described, and it was concluded that integrating the tool in a *VSPackage* would the best way to ensure that the all needed functionality in Visual Studio was accessible.

It was decided initially only to implement a very basic part of IDE related part of the tool. The elements implemented were to be error-highlighting in the Visual C#-editor. All other elements were left to be dealt with in future stages.

CHAPTER 6

STAGE2

This chapter describes the research and work done during the second stage of development on the prototype. The goals of this stage is to implement the Visual Studio plug-in and do an analysis of symbolic execution. The analysis of symbolic execution also uncovered some problems regarding the implemenation of the random testing technique which required some refactoring. This is described at the end of the chapter.

6.1 IDE Plug-in - Implementation

In Section 5.2 the design of the desired features for the plug-in was described and a set of elements to be integrated was selected. The intension is to develop a base set of functionality in the IDE and then incrementally expand the functionality as the tool becomes more sophisticated.

Broadly the creation of the plug-in for Visual Studio can be seen as consisting of three parts. First the package that holds all the different visual elements such as buttons and menus should be defined along with other elements such as icons also used in the package. This part of creating a package for Visual Studio is somewhat straight forward. Many manuals, tutorials and videos are available on the Visual Studio SDK webpage and on the web in general.

The second part consists of hooking into the different services that Visual Studio provides. This includes the C#-editor in order to draw squiggles and icons in the editor and access to the user's active solution and projects. This proved to be much more laborious than first anticipated and eventually caused the project time schedule to slip.

The third is to define the interface that is needed in order to provide the interaction between the backend of the prototype and the GUI. This also includes developing extra functionality such as providing line numbers and type of error. This was never implemented because of the problems encountered while implementing the second part.

The next section describes the steps required in order to create the basic GUI elements such as buttons. This is followed by some general observations done while trying to implement the actual functionality in the IDE.

6.1.1 Basic GUI Elements

Starting form Visual Studio 2008 Microsoft has changed the way how graphical elements are integrated in Visual Studio. Now all graphical elements are defined in a *.vsct* file. The extension is short for Visual Studio Command Table. The *.vsct*-file has an XML-based structure and is used to define graphical elements and associate these with commands.

In Listing 6.1 the basic outline of the *.vsct*-file is illustrated. The top-element in the file is `CommandTable` . In the following description some of the child-elements of `CommandTable` are shortly described.

Extern This element is used to reference elements from the Visual Studio IDE. If a menu item is to be added to the Visual Studio main menu bar, the Globally Unique Identifier (GUID) of the menu bar must be referenced. The `Extern` -element is used for this purpose.

Commands This element is used to assign a commands to the owning VSPackage. The GUID of the package is used for this.

VisibilityConstraints This element can be used to control in which context the different elements should be visualized. As an example a `VisibilityItem` which is a child of `VisibilityConstraint` can be set to the context-type *UICONTEXT_SolutionHasSingleProjects* which means that the item will only be visible if the solution only has one project.

KeyBindings This element is used to bind keyboard shortcuts to the defined commands.

Symbols These elements are used to link a GUID with a logical name so the programmer can reference a readable name instead of a GUID.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <CommandTable xmlns="http://schemas.microsoft.com/VisualStudio/
      2005-10-18/CommandTable"
      xmlns:xs="http://www.w3.org/2001/XMLSchema">
3   <Extern href="stdidcmd.h"/>
4   <Extern href="vsshlids.h"/>
5   <Extern href="msobtnid.h"/>
6   <Commands>
7    <!-- ... -->
8   </Commands>
9   <VisibilityConstraints>
10   <!-- ... -->
11  </VisibilityConstraints>
12  <KeyBindings>
13    <!-- ... -->
14  </KeyBindings>
15  <Symbols>
16    <!-- ... -->
17  </Symbols>
18 </CommandTable>
```

Listing 6.1: Outline of the Visual Studio Command Table-file

In Listing 6.2 the definition of a top menu bar item can be seen. The XML in the listing will be described hereunder. The resulting menu in Visual Studio can be seen in the screenshot in Figure 6.1. The menu items in the screenshot are defined as buttons.

In line 2 in Listing 6.2 a `Menu` -element is defined. This element has some different attributes.

The `guid` - and `id` -attributes are used as element identifiers and are used to associate the UI-element with the appropriate VSPackage.

The `priority` -attribute indicates the priority of elements grouped together and can be used to control in which order the elements should be displayed. `type` describes the stereotype of the element and helps the IDE to determine the behavior of the element.

In line 2 the `Parent` -element is defined. This helps the IDE to place the menu element in the right logical container. In this case the value *guidSHLMainMenu* indicates that the menu-element should be placed in the main menu and the value *IDG_VS_MM_BUILDDEBUGRUN* indicates that the menu element should be placed in the *Build* and *Debug* menu-group.

In line 5 and 6 the `CommandName` and `ButtonText` are set to *Icarus*. In the top-level menu like in this case the `CommandName` -attribute isn't used but the `ButtonText` is used to define the visual name of the menu.

The actual placement of the menu can be seen in Figure 6.1. The optional attribute `priority` could be specified and be used to place the element very specific.

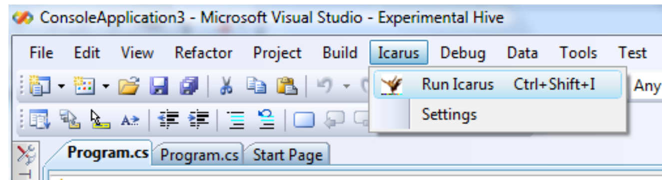

Figure 6.1: Screenshot showing visual elements integrated in Visual Studio.

```
1 <Menus>
2  <Menu guid="guidIcarusPluginCmdSet" id="TopLevelMenu"
       priority="0x100" type="Menu">
3   <Parent guid="guidSHLMainMenu" id="IDG_VS_MM_BUILDDEBUGRUN" />
4   <Strings>
5    <ButtonText>Icarus</ButtonText>
6    <CommandName>Icarus</CommandName>
7   </Strings>
8  </Menu>
9 </Menus>
```

Listing 6.2: The definition of a top menu bar item with the label *Icarus*

In Listing 6.3 the XML-definition for a button (menu-item) can be seen. Basically the XML defining the button is the same as in Listing 6.2. However there are a few differences.

In line 2 the parent GUID is associated to the package experimented with in this project instead of a GUID associated with Visual Studio functionality.

The `CommandFlags` in line 3 and 4 sets the menu-item to be invisible if the package hasn't been loaded. In line 7 the text of the button is set to the value *Settings* and finally in line 6 the button is associated to the user-defined command *cmdidIcarus-Tool*. When the menu-item is pressed this command will trigger an event that will be handled by our package.

The actual definition of the VSPackage is done in a C#-file that contains global information on the package and some load-keys verified by Microsoft needed by Visual Studio when loading a package.

The package definition will not be described here but the main functionality implemented in the file consists in handling callback functions from the IDE, that are associated with the e.g. the GUI-elements just described. The callback functions can be used to trigger some user defined external functionality such as the tool described in this paper.

```
1 <Button guid="guidIcarusPluginCmdSet" id="cmdidIcarusTool"
      priority="0x0310" type="Button">
2  <Parent guid="guidIcarusPluginCmdSet" id="TopLevelMenuGroup" />
3  <CommandFlag>DefaultInvisible</CommandFlag>
4  <CommandFlag>DynamicVisibility</CommandFlag>
5  <Strings>
6   <CommandName>cmdidIcarusTool</CommandName>
7   <ButtonText>Settings</ButtonText>
8  </Strings>
9 </Button>
```

Listing 6.3: The definition of a button (menu-item) placed in a toplevel-menu

6.1.2 General Observations

In Section 5.2 it was mentioned that the VSPackage approach was the most complex approach, and while this is true it was not the actual choice of plug-in type that required the most effort. As mentioned in the beginning it was relatively easy to implement the basic package setup. The only thing that caused some minor problems was the need to register the plug-in with Visual Studio before use. This could be done via a command line script, but the authors chose to create a small installer that registered the plug-in as part of the install process.

The task that proved to be much more complicated was finding the right services and events that would allow us to interact with the C#-editor and solution explorer. Examples that show how to create tools and interact with these are available through Visual Studio SDK webpage. The problem with the examples is that I many cases they show only trivial and very basic functionalities. These are helpful when used to gain the larger overview, but lack in details when wanting to implement more specific functionalities.

Furthermore we found that the documentation on the Visual Studio SDK libraries in many cases needed to be described in greater details. The problem consists in that many of the more important classes are very large and may contain more than hundred different methods and properties. The detail level of the documentation on these methods and properties are very low and leaves much unsaid. For this reason it has been hard to understand what information and functionality that is available through the classes and how exactly they are used and accessed.

The basic need of interacting with the C#-editor and the solutions and projects hosted by the IDE is fundamental in order to achieve the desired functionality. Unfortunately the authors did not succeed in getting hold of the necessary documentation and, after having spent the better part of two weeks without any significant progress, it was decided that the integration had to be abandoned. This is further discussed in Chapter 8.

6.2 Symbolic Execution - Analysis

Symbolic execution is a method to symbolically execute a program opposite to executing the program concretely as seen in normal execution of a program. Symbolic execution according to testing were first purposed by James C. king in 1976 [16].

When a program is executed symbolically a Control Flow Graph (CFG) is obtained, where the nodes represents the different program states, and the directed edges represents transitions between the states. Symbolic execution is a simulated run of a program, where symbols, instead of values, are used to represent variables in the program.

An example could be a function `add` , that takes two integers as input and returns an integer that is the sum of the inputs. This could be represented with X and Y as symbolic input-arguments and the symbolic output $X+Y$. This output could then be used as input to a an other function.

The different possible states of the program are described by the symbolic variables, a Path Condition (PC) and a program counter where the program counter describes the next statement to be executed.

The path condition is described by a conjunction of boolean expressions on the form: $< e_1 \wedge e_2 \wedge \ldots >$, where the different conditions describing a given path are accumulated. Initially the PC is true. If the expression describing the PC becomes false it means that the path in question is infeasible. A tree representing the symbolic execution of the program in Listing 6.4 can be seen in Figure 6.2. The PC for each state is shown in the figure. The path condition marked with red text is infeasible. The comments in Listing 6.4 illustrates a possible concrete exectution of the program.

```
1  int x, y;                     //x = 1, y = 0
2  if(x>y){                      //is (1 > 0)? : YES
3      x = x + y;                //x = 1 + 0 = 1
4      y = x - y;                //y = 1 - 0 = 1
5      x = x - y;                //x = 1 - 1 = 0
6      if(x - y > 0){            //is (0 - 1 > 0)? : NO
7          assert(false);
8      }
9  }
```

Listing 6.4: Concrete execution. Example from [21]

Symbolic execution is widely accepted as the most systematic way for automatically generating an exhaustive set of test cases [15].

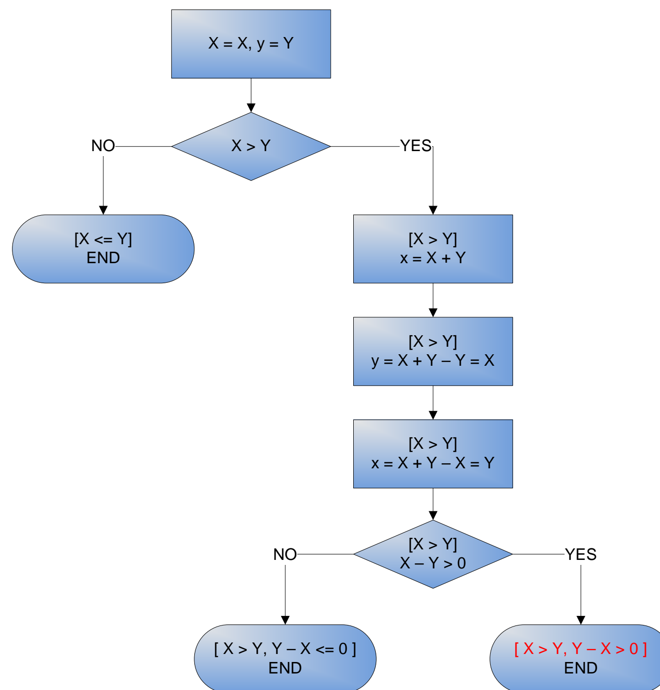

Figure 6.2: Symbolic execution. Example from [21]

6.2.1 Problems

Through our previous work it was found out that several problems arises when dealing with symbolic execution. Some of these are:

Computationally expensive Symbolic execution is an expensive technique because of e.g. constraint solving and maintaining CFGs representing complex programs. This fact is one of the reasons that it is not until recently that symbolic execution has been integrated in advanced test generation tools despite the technique were suggested already in 1976.

Unbounded loops The presence of unbounded loops and recursive calls in a program results in an infinite number of paths through a program[44]

Infeasible paths This problem consists in that some paths in a program may be infeasible. The problem isn't that some code is unreachable or dead, but the fact that it can be impossible to find out if a path is in fact infeasible. If the predicate in a conditional branch is linear, the problem is easy to solve. If on the other hand the predicate is nonlinear like $X^3 + Y^2 = 5$ there is no guarantee that the problem can be solved[44]

Aliasing Lee et al. in their works[15] specifies two related kinds of aliasing that should be handled when performing symbolic execution. These are array element aliasing and reference aliasing.

The array element aliasing arises when analyzing indexed arrays. The problem consists in, that in some situations the functionality of a method can be changed significantly because of this problem. This is illustrated in Listing 6.5. The method reads two elements from the array, and writes their sum in one of the elements and the difference in the other. The problem arises when $i = j$. Then the sum eventually be set to 0.

```
1 int butterfly(int[] a, int i, int j) {
2     int sum = a[i] + a[j];
3     int diff = a[i] - a[j];
4     a[i] = sum;
5     a[j] = diff;
6     return a[i];
7 }
```

Listing 6.5: This method illustrates the array element aliasing problem. The method replaces a pair of elements with their sum and difference. Source: [15]

Different solutions have through time been proposed to these problems. Anand et al.[45] suggests an approach targeting the computationally problem called Demand-Driven Compositional Symbolic Execution which basically means the symbolic execution will only be performed when needed and only on fractions of the program.

The unbounded loop problem is addressed by Anand et al with a method they call *Abstract Subsumption Checking*. The method uses some state matching techniques to minimize the state space search by checking if a symbolic state is subsumed by another symbolic state[46].

The ability of constraint solving has to some extent evolved along with computers getting more powerful. But in the end it all comes to that some problems can be solved and some problems may not be solvable. In some systems like DART this problem is approached by using an a technique called concolic testing, which means that when the solver resigns the system replaces the symbolic value with a concrete one and resumes the traversal of the path.

If one wants to generate concrete test cases to be used in an e.g. NUnit test suite on background of symbolic execution it is necessary to transform the symbolic path constraints and values generated through the symbolic execution to concrete values, thereby ensuring that all feasible paths of the program under test are executed by the test suite. For this purpose a constraint solver is necessary.

The array element aliasing problem is addressed by Lee et al where an approach that determines all possible values of each array index, which makes it possible to generate equivalence classes for all possible combination of array element aliases. [15]

6.2.2 Instrumentation

When using symbolic execution to generate tests it is necessary to instrument the program under test. This is because it is necessary to enumerate the different paths traversed in the CFG representing the program to monitor which paths in the program that has been tested.

It is also necessary to gain control of the inputs to the program along with different values used internally in the program. Therefore pieces of code have to be injected in the program allowing monitoring and manipulating the variables in the program.

The approach taken in this project is meant to combine random testing with symbolic execution, in the way that when the random part of the tool fails to improve the test coverage the symbolic execution part gains control over the process of generating test cases. For this reason it is necessary to monitor which paths that have been visited and to inject code to control and monitor the inputs, assignments and branches in the program under test. The consequence of this is that a instrumentation/analysis library that is able to provide manipulation at the instruction level is needed.

During the initial analysis in Chapter 4 three different libraries for .NET code instrumentation was found. In the following each of these libraries will be analysed further in order to determine which one to use for the implementation of the symbolic execution module.

RAIL

The Rail library[33] was one of the first libraries for .NET code manipulation. This library supports analysis/instrumentation at the instruction level, and also seems to have a simple interface. Unfortunately this library is no longer maintainted and has not been updated since 2005, it is therefore not considered further.

.NET Profiling API

The .NET Profiling API is an unmanaged API that hooks into the CLR and provides callbacks for various events. One of these events is when the CLR is about to JIT compile the CIL code. This allows an application to inspect and rewrite the code before execution.

The advantage of this is that it allows for runtime instrumentation and therefore does not need to modify the existing executable or create an instrumented copy. Runtime instrumentation could possible also be an advantage in the presence of dynamic class loading, where the possible code paths cannot be determined statically, although this is not something that has been studied.

The disadvantages of the Profiling API is that it has to be implemented as an in-process COM server and cannot be implemented in a managed language. This would mean that in addition to the actual COM server a managed library providing callbacks via IPC mechanisms would have to be implemented. In addition to this the Profiling API does not support instrumentation at the instruction level directly, so this would also have to be implemented.

Phoenix

Phoenix is a software analysis and optimization framework in development at Microsoft. The goal of the project is to provide a basis for all of their future compiler technologies. The framework is general and is not directly targeted at .NET, but allows assemblies to be loaded and analyzed. The project has already been used in several research projects such as the `Phx.Morph` library of Wicca[47].

The advantages of Phoenix is that it allows for full analysis and rewriting at the instruction level. The framework also contains a lot of functionality that could help other types of analyses, i.e. such as the code pattern-matching used by FindBugs. Another advantage is that it is available now and does not require coding a full component.

Disavantages include the complexity of the library and that it is still a research project. This means that interfaces might change, and that all functionality might not be available. Another disadvantage is that it introduces a dependency on a huge library, with a restrictive license (non-commercial research).

For this project the Phoenix framework was chosen. This was mainly due to the fact that implementing an instrumentation library on top of the Profiling API would be too time consuming and remove focus from the real goal of doing symbolic

execution. Since this is also a research project the disadvantages of Phoenix is less of a problem, but if creating an open-source tool or a tool that was also intended to run on the Mono implementation of .NET, these would not be acceptable.

6.2.3 Constraint Solver

Should concrete test cases needed to be generated on background of symbolic execution e.g. for use in a NUnit test suite, it is necessary to transform the symbolic path constraints and values generated through the symbolic execution to concrete values, thereby ensuring that all feasible paths of the program under test are executed by the test suite. For this purpose a constraint solver is necessary.

In the initial analysis the Simplify solver was mentioned as well as Z3[36]. For this project the Z3 solver was chosen for the simple reason that it provides a well documented .NET API which would make integration easy. The solver is still under development but available through Microsoft research under a non-commercial research license.

Regarding speed and capabilities the solver is used internally by Microsoft in several applications including their own testing tool Pex. An earlier version was also entered into the SMT-COMP'07 competition[48], which is a yearly competition for SMT-solvers where it won several of the categories.

6.3 Random Testing - Refactoring

During the analysis of symbolic execution it became apparent that the first algorithm chosen for random testing was not ideal. The reason for this is that the implementation builds test cases in a bottom-up fashion, starting from primitive types and adding method calls. This was not considered until examining how to implement symbolic execution.

When doing symbolic execution this is done in a top-down manner starting from a method and exploring the paths. If random testing should drive the symbolic execution a better approach would be the one proposed in JCrasher[6]. This would allow the implementation to perform random testing on a specific method. The algorithm is described in detail in the following section.

6.3.1 JCrasher Algorithm

As briefly described in Section 2.1, JCrasher works by building a graph of input generation methods and traversing this graph in order to create test-cases which is then executed in a runtime environment.

When JCrasher fist loads a class, reflection is used to iterate over each of the public methods. A mapping is then created which maps types to predefined values and methods returning the type.

```
1  class T
2  {
3    T()
4    {
5      ...
6    }
7
8    C f(A a, int b)
9    {
10     ...
11   }
12 }
```

Listing 6.6: Example class for JCrasher testing

When examining the class `T` in Listing 6.6 JCrasher would map the type `T` to the constructor `T()`, i.e. in order to create a type `T` it knows that it can call the default constructor for `T`. For the method `T.f` in JCrasher would map the type `C` to the method `T.f`. This is done recursively for each of the arguments until all types have been mapped. Generating a test case can now be done by selecting a path through the graph. Figure 6.3 shows an example of a generated graph taken from the JCrasher paper[6].

6.3.2 Implementation

Changing the existing prototype to use this approach would require that the graph described in the previous section is constructed when the `TestGenerator` is loading the classes. Other than this the change only affects the class acutally implementing the random algorithm.

The design of the `TestCase` class done in the last section on random testing can still be used. This only requires that the statements are constructed as a path in the graph is traversed.

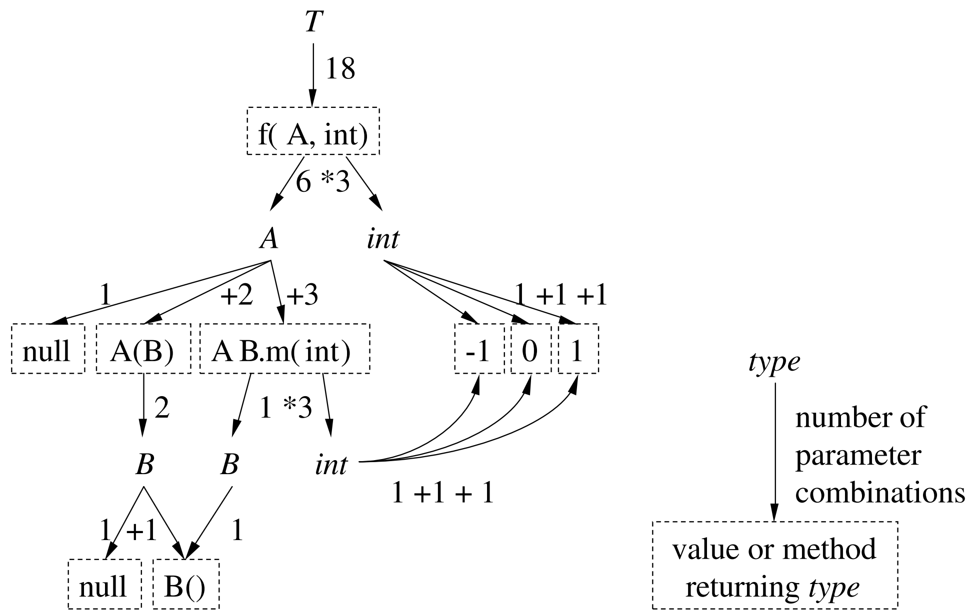

Figure 6.3: Parameter-graph generated by the JCrasher algorithm.

6.4 Summary

Through stage 2 the random algorithm being implemented in stage 1 was changed from the *Randoop*-algorithm to the *JCrasher*-algorithm. This was done because it turned out that the nature of the *JCrasher*-algorithm seems better fit for integration with symbolic execution.

An integration of the tool with the Visual Studio IDE was tried out and some experiences were gained. The objective was to be able to illustrate errors found in the IDE and more specific in the Visual C#-editor. This was to be done by drawing a squiggly under the failing statement or by drawing an icon in the gutter of the editor.

Unfortunately the interoperation done with Visual Studio through VSPackages turned out to be more complex than first expected. This hurdle entailed that the integration with the IDE was abandoned.

An analysis of symbolic execution and the elements needed to accomplish this was conducted. It turned out that no package solution providing the functionality needed for symbolic execution is available on the .NET-platform. However different libraries are supporting subsets are available. It was chosen to focus on instrumentation since this part is essential to the rest of the process.

For instrumentation the libraries *Phoenix*, *RAIL* and the *.NET Profiling API* were identified. It was decided to use *Phoenix* since *RAIL* turned out to be a dead project. *Phoenix* was chosen rather than the *.NET Profiling API* due to the reason that Phoenix supports a rather comprehensive set of functionalities, and also because it was assessed that the *.NET Profiling API* would involve a large amount additional implementation.

CHAPTER 7

STAGE3

This stage contains the initial implementation of the symbolic execution module. The goal was to provide a simple implementation of symbolic execution that did not handle any of the problems such as array aliasing described in Section 6.2 and concentrate on the integration with random testing. However, due to problems with the Phoenix framework that was chosen for code instrumentation this was never achieved. The results of trying to implement the instrumentation is detailed in the following section.

7.1 Symbolic Execution - Implementation

The purpose of combining random testing with symbolic execution is to use random testing to quickly generate as much coverage as possible. When random testing does not increase coverage, symbolic execution takes over and tries to generate input for the paths not covered yet. Listing 7.1 illustrates the concept in pseudo-code.

```
1  foreach(Method f in Testee.Methods)
2  {
3    Instrument(f);
4
5    while(coverageIncreased)
6    {
7      Input i = RandomInput();
8      ExecuteConcrete(f, i);
9    }
10
11   InterpretSymbolic(f);
12 }
```

Listing 7.1: Pseudo code for comined random/symbolic testing.

In order to do this it is necessary to both be able to monitor the program while the concrete execution takes place in order to update the symbolic state, but also in order to perform the acutal symbolic execution. This requires that the method under test is instrumented in order update the symbolic environment.

7.1.1 Code Instrumentation

During the analysis of symbolic execution it was decided to use the Phoenix framework for instrumentation. This is a rather large and complex framework, so it was expected that there would be an initial learning curve associated with it. Unfortunately many of the same problems that was experienced during the plug-in implementation (described in Section 6.1), i.e. missing or wrong documentation, made this even harder.

Phoenix is also meant as a general compiler optimization framework, so it provides its own internal intermediate representation for code and does not use use the Common Intermediate Language (CIL) that is used in .NET. This coupled with the fact that the documentation was lacking made it mostly a process of trial and error.

During the work done in this stage a new version of the Phoenix framework became available. This version updated the documentation and fixed a number of unimplemented features related to Phoenix' internal processing of CIL code. It was not until this update that the small example described in the following could be made to work.

The following sections documents the work on a small test example that was made to part of the initial effort to get familiar with Phoenix. The basic idea of the example is to insert a call to the static `TestClass.Test()` method from the `TestTool.dll` at the beginning of all mehtods in the instrumented assembly. The functionality in the prototype would ofcourse require more advanced functionality at the granularity of individual instructions, the example only serves as a proof of concept.

Initialization

In order to use the phoenix framework it has to be initialized to use the right subcomponents depending on the target achitecture and the runtime system. This is shown in Listing 7.2.

There are several things to note in this example. The first is that Phoenix implements a custom memory management scheme, so all objects are created by calling the

```
1  Phx.Targets.Architectures.Architecture msilArch =
       Phx.Targets.Architectures.Msil.Architecture.New();
2  Phx.GlobalData.RegisterTargetArchitecture(msilArch);
3
4  Phx.Targets.Architectures.Architecture x86Arch =
       Phx.Targets.Architectures.X86.Architecture.New();
5  Phx.GlobalData.RegisterTargetArchitecture(x86Arch);
6
7  Phx.Targets.Runtimes.Runtime msilRuntime =
       Phx.Targets.Runtimes.Vccrt.Win.Msil.Runtime.New(msilArch);
8  Phx.GlobalData.RegisterTargetRuntime(msilRuntime);
9
10 Phx.Targets.Runtimes.Runtime x86Runtime =
       Phx.Targets.Runtimes.Vccrt.Win32.X86.Runtime.New(x86Arch);
11 Phx.GlobalData.RegisterTargetRuntime(x86Runtime);
```

Listing 7.2: Phoenix initialization code.

static `New()` method. The second is that target architecture and runtime for x86 also have to be initialized even if we only handle CIL code.

This is not intuitive and was not properly explained in the documentation or examples, and manifests itself in a `FatalError` exception at runtime with no additional information. When inspecting the stacktrace it can be seen that the failure occurs when trying to load the global symbols. The authors guess is therefore that, since Phoenix is a general framework for both managed and unmanaged code, the problem is related to the unmanaged startup code that was used on previous versions of windows. In order to read and write the symbols and code for this part of the of the file it is necessary to register the architecture and runtime for native x86.

Main Instrumentation

After Phoenix has been initialized and an assembly loaded it is now possible to loop through the methods in order to do the instrumentation. The small test example only requires that the first instruction is located and a method call is inserted at the beginning. The code for this is shown in Listing 7.3, but has been simplified in order to make it more readable.

There is one interesting thing to note here. Before the call instruction at line 30 can be created, the necessary symbol for the `Test()` method has to be created. This involves adding the `TestTool.dll` to the import table of the instrumented assembly and adding entries in the symbol and type tables.

The examples and documentation was very vague on this part and the only usable example was an example of inserting a `nop` instruction into the beginning of a function, and does not involve the symbol/type tables at all. A substantial amount

```
1  private static void Instrument(Phx.PEModuleUnit module)
2  {
3    // Import the symbol for the Test() method in TestTool.dll
4    Phx.Symbols.FunctionSymbol testFuncSymbol =
         ImportTestFunc(module);
5
6    // Loop through the FunctionUnits in the module
7    foreach (Phx.FunctionUnit functionUnit in module.Functions)
8    {
9      // Raise code to Low Level IR.
10     functionUnit.DisassembleToBeforeLayout();
11
12     //  Don't care about unmanaged code
13     if (functionUnit.Name.Equals("UnmanagedEntryPoint"))
14       continue;
15
16     // Find the first instruction
17     Phx.IR.Instruction firstInstruction = null;
18     foreach(Phx.IR.Instruction  instruction in
           functionUnit.Instructions) {
19       if(instruction.IsReal) {
20         firstInstruction = instruction;
21           break;
22         }
23     }
24
25     // Create new call instruction and insert it before
           firstInstruction
26     Phx.IR.CallInstruction callInstruction =
           Phx.IR.CallInstruction.New(functionUnit,
           Phx.Targets.Architectures.Msil.Opcode.call,
           testFuncSymbol);
27     firstInstruction.InsertBefore(callInstruction);
28
29     // Make sure that instruction sequence is legal
30     functionUnit.Legalize.Instruction(callInstruction);
31   }
32 }
```

Listing 7.3: Code for the main instrumentation loop.

of time was therefore used to figure out how to do this.

Adding Assembly Imports

The code for adding the assembly import for the `TestTool.dll` is shown in Listing 7.4. First the import table is inspected to see if it already contains the import, if this is the case the found symbol is returned. If the symbol is not found it needs to be added to the table.

This is the point where it gets a little complicated. In order to add the symbol the manifest of the assembly is needed, and the only way to obtain the manifest of

an assembly is to load it. Since Phoenix already contains functionality to load an assembly and get the manifest, this was used (line 9-11).

```
1  private static Phx.Symbols.AssemblySymbol
       AddAssemblyImport(Phx.PEModuleUnit module, string
       assemblyNameString)
2  {
3    // Check to see if the import already exists
4    Phx.Symbols.AssemblySymbol assemblySymbol =
         FindAssemblySymbol(assemblyNameString);
5    if(assemblySymbol != null)
6      return assemblySymbol;
7
8    // Load the TestTool.dll to obtain the manifest
9    Phx.PEModuleUnit library =
         Phx.PEModuleUnit.Open("TestTool.dll");
10   library.LoadGlobalSymbols();
11   Phx.Manifest manifest = library.Manifest;
12
13   //Possible solution: Can only have one active PEModuleUnit
14   Phx.Threading.Context context =
         Phx.Threading.Context.GetCurrent();
15   context.PopUnit();
16
17   // Create a assembly reference to library
18   Phx.Name libraryName = Phx.Name.New(module.Lifetime,
         "TestDll");
19   assemblySymbol = Phx.Symbols.AssemblySymbol.New(null, manifest,
          libraryName, module.SymbolTable);
20
21   return assemblySymbol;
22 }
```

Listing 7.4: Code for the assembly import method.

The problem here is that there can only be one active `PEModuleUnit` at a time, so the newly loaded assembly has to be removed from the current thread's context as done in line 14-15. This did not manifest itself in any runtime errors, but had the result that no instrumentation was added to the assembly.

This was completely undocumented and a lot of time was spent on debugging to find the cause, before a fix was found as a side note in a forum thread discussing a topic not related to this. After installing the new version of the framework line 14-15 is no longer required so either this was a bug or the functionality was changed.

Creating a Function Symbol

After creating the code to add the assembly reference all that is needed is to implement the actual `ImportTestFunc()` that will be called from the main instrumentation loop in Listing 7.5, line 4.

```
1  private static Phx.Symbols.FunctionSymbol
       ImportTestFunc(Phx.PEModuleUnit moduleUnit)
2  {
3    // Holds the fuction symbol to be created
4    Phx.Symbols.FunctionSymbol testFuncSymbol = null;
5
6      // Get a reference to the TestTool.dll assembly.
7      Phx.Symbols.AssemblySymbol testToolSymbol =
           AddAssemblyImport(moduleUnit, "TestTool");
8
9      // Create a class type for TestTool.TestClass
10     Phx.Name testClassTypeName =
           Phx.Name.New(moduleUnit.Lifetime, "TestTool.TestClass");
11     Phx.Symbols.MsilTypeSymbol testClassTypeSymbol =
           Phx.Symbols.MsilTypeSymbol.New(moduleUnit.SymbolTable,
           testClassTypeName, 0);
12     Phx.Types.AggregateType testClassType =
           Phx.Types.AggregateType.NewDynamicSize(
           moduleUnit.TypeTable, testClassTypeSymbol);
13     testClassType.IsDefinition = false;
14
15     // Now, attach the class type to the TestTool assembly
           reference.
16     testToolSymbol.InsertInLexicalScope(testClassTypeSymbol,
           testClassTypeName);
17
18     // Build up a function symbol for the Test() method.
19     Phx.Types.FunctionType functionType =
           moduleUnit.TypeTable.GetFunctionType(
           Phx.Types.CallingConventionKind.ClrCall, 0,
           moduleUnit.TypeTable.VoidType, null, null, null, null);
20     Phx.Name functionName = Phx.Name.New(moduleUnit.Lifetime,
           "Test");
21     testFuncSymbol =
           Phx.Symbols.FunctionSymbol.New(moduleUnit.SymbolTable, 0,
            functionName, functionType,
           Phx.Symbols.Visibility.GlobalReference);
22
23     // Add it as a method of the TestTool.
24     testClassType.AppendMethodSymbol(testFuncSymbol);
25     testClassTypeSymbol.InsertInLexicalScope(testFuncSymbol,
           functionName);
26
27     return testFuncSymbol;
28 }
```

Listing 7.5: Main instrumentation loop.

The code to do this is shown in Listing 7.5 and is mostly here for completeness as it is rather straight forward. It did however take some time to implement as the documentation in the previous version lacked a proper description of some of these concepts.

In particular that it was necessary to use the `Phx.Symbols.MsilTypeSymbol` instead of the more general `Phx.Symbols.TypeSymbol`. This was not intuitive because the code had already been raised from CIL into the more general IR of the

Phoenix framework. Fortunately some examples from the Phoenix forums provided the missing information.

It should also be noted that the documentation on these classes in the newest version of Phoenix has been updated which also provide several examples that use the concepts.

7.2 Summary

The problems related to using the Phoenix framework meant that the time was spent on figuring out how to perform the instrumentation of the test code. Because of this work on the internal parts of symbolic execution such as how to represent the symbolic state or how to monitor the concrete execution of a program in order to update the symolic state was not started.

Using the phoenix framework for instrumentation of .NET code proved to be much more complicated than initially assesed, and using Phoenix in its current state cannot be recommended. Instrumentation of the code, however, is still needed, so other approaches should be looked at. Implementing the necessary functionality on top of the .NET profiling API would be the recommended solution, since instrumentation can be done at runtime, but this is not something that has been analysed.

CHAPTER 8

DISCUSSION

In this chapter the implementation of the different elements of prototype will be discussed and reflected upon. The first sections contain a discussion of the testing techniques and the IDE integration. The next discusses the iterative development model used in the process, and the last section gives a general discussion of the project and future directions.

8.1 Test Generation

The authors' previous work[5] and initial research showed that combining techniques is essential in order to create a tool that detects the largest possible range of errors. The two techniques, random testing and symbolic execution, was selected because research indicates that individually these are currently the most promising techniques.

The implementation of random testing confirms this as a simple and easy to implement solution. Although the current implementation does not try to handle any of the more complicated topics related to test oracles or filtering of test cases it still gives a relative indication of this being a simple technique to implement.

Symbolic execution is a comprehensive and systematic technique that has a lot of problems requiring special attention. In addition it also requires some external functionality. Due to the factors described in Chapter 7 only the external dependencies of symbolic execution was examined and the authors is therefore unable to provide any insight into the implementation of the actual algorithm, but instead point to work that must be completed before an actual implementation can take place.

In general the choice techniques proved to be a good example of both the ease and

simplicity with which a tool could be made, but also the complexity introduced if a more comprehensive solution is required. A discussion of the individual techniques is provided in the following sections.

Random Testing

The initial implementation of random testing into the developed tool was as expected relatively easy and did not depend on any external functionality. After a .NET assembly was loaded the built-in reflection capabilities of the .NET platform could be used to traverse and introspect the types and methods in order build the needed data structures. The biggest problem was to design a good representation of the test cases that allowed for both easy serialization and execution.

During stage 2 of the development, however, the analysis of symbolic execution uncovered one problem that required a refactoring of the random testing implementation. This was that the implementation of random testing (Randoop algorithm) used a bottom-up approach to generating the test cases. This complicated the integration with symbolic execution which works top-down and it was therefore decided to re-implement the random testing module, using the approach used in JCrasher, in order to accommodate this.

It could be argued that the choice to change implementation would degrade the effectiveness of the system, because the motivation for choosing the Randoop approach was that it achieved a high degree of coverage. While this is true, it is a situation that can be mitigated to some extent by incorporating some of the mutation heuristics from AutoTest.

Unfortunately the problems related to the IDE integration and symbolic execution forced the authors to stop the development of this module before it was complete.

Symbolic Execution

The basic idea of symbolic execution is rather simple, but there are a lot of special cases that must be handled in order to provide a full implementation, i.e. array aliasing, etc. There are also a number of components that is required in order to implement a symbolic execution engine. These include the following:

- Instrumentation
- Constraint solver

- Execution engine and main algorithm

During the analysis of the symbolic execution module it was decided to first implement the instrumentation of the assemblies. This was partly because all of the other components of symbolic execution depended on being able to instrument the program, but also to become familiar with Phoenix, the framework chosen for the task.

Phoenix is a large framework and provides a lot of functionality related to program analysis. This was one of the reasons that it was chosen over simpler approaches such as the .NET Profiling API or the Rail library. Unfortunately it was also the main reason for not being able to finish, or even provide a partial implementation of symbolic execution.

The fact that Phoenix is still in development, meant that documentation was missing or created for previous versions of the API. Since the framework uses its own multilevel intermediate representation for the code a large amount of time was spent trying to figure out how this worked. This coupled with the missing or out of date documentation made it a very time consuming task to achieve even some very simple instrumentation. Even this was only possible very late in the development, when a new version of Phoenix was released.

In retrospect it would have been better to use one of the other libraries and implement the missing functionality as explained in Section 6.2. This would also be a time consuming task, but it would be on the basis of well documented interfaces, and could be made much more light-weight.

A deeper investigation of the two additional modules never started although some possible constraint solvers were taken into account.

8.2 IDE Integration

As mentioned earlier the focus in this report has been put on identifying functionalities that is needed to implement all aspects of an automated test tool. In order to develop a tool that is meant to be used by developers it has been emphasized that the human computer interaction is an important aspect of the tool. This is important because previous experiences indicate that easy access and use of a tool or technique to some point determines the adoption.

It was decided to target Microsoft Visual Studio 2008 as the platform for the plug-in, as this IDE especially for .NET development is very widespread. Although the

integration was halted due to implementation problems, the authors still believe that the Visual Studio IDE is the best platform to use. It can be argued that integration on another platform could have been chosen for a proof of concept but it was unexpected that the integration would be as time consuming as it turned out.

One reason that the estimation of the needed efforts for developing a plug-in for Visual Studio was off is due to the fact that the estimate was based on previous experiences with a different IDE. On an earlier semester the plug-in structure for Eclipse was investigated and these experiences were used as measurement in this project. It was of course risky to base the estimation on experiences made with a different IDE, but it still came as a surprise that the differences in complexity were that significant.

Mainly the issue consisted of problems with obtaining the proper amount of documentation for the Visual Studio SDK, so if a tool like the one described in this report is to be integrated in an IDE in the future, it is recommended that the IDE integration begins as early in the process as possible, and that an ample amount of time and effort is reserved for this purpose. It should be noted that the authors did not have any previous experience with writing plug-ins for Visual Studio so this would also have to be factored in.

8.3 Iterative Development

The goal of the project was to develop a fully functional prototype of an automated testing tool which could be used as a basis for further development, and provide insight into potential problems, pitfalls or areas that need further development.

Because of this it was equally important to ensure that there was always a working version of the tool and that all the aspects of the tool were touched during development. For this reason the development of the prototype was divided into an initial analysis phase and three development phases, where each of the development phases should result in an increasingly more functional version of the tool.

This worked fine until late in stage 2, where the IDE integration proved to be more difficult than first anticipated. The consequence was that the time schedule slipped and shortened stage 3 by one week. This again meant that there was only enough time to explore a single aspect of the implementation of symbolic execution. The initial time schedule can be found in Chapter A.

It could be argued that doing more design from the beginning would have resulted in a clearer overview of the entire application and could have prevented some of the

encountered issues such as the reimplementation of the random module at the start of stage 2. However, this would have taken focus away from the practical details of developing the tool which was one of the project goals, and the authors therefore argue that the iterative and more agile approach used was better in order to achieve this goal.

The decision of postponing the final implementation of the random module resulted in not having a working prototype at the end. In retrospect this can be seen as a suboptimal decision, but it can also be argued that the time saved allowed for more work on the IDE integration and symbolic execution. It is therefore a question of whether having a functional prototype is deemed more important than fully exploring all the initial problem areas. The authors weighted the latter more important because of the explorative nature of the project.

8.4 Additional Considerations

The previous sections has discussed many of the aspects related to the development of an automated unit testing tool, and has mainly focused on the practical aspects. There are however also a number of more general considerations.

The choice of using Phoenix as an instrumentation and analysis library and Z3 as a solver restrict the use of the project to non-commercial projects only because of the license. Even if these eventually become a part of Visual Studio relying on them will essentially lock the tool to the windows platform. This may, or may not, be an issue depending on how important cross-platform support is, but should at least warrant some consideration. This is only an issue for the base tool as the GUI portion will necessarily be locked to a specific IDE.

During the late stages of the project a preview of the Pex tool from Microsoft was released. The authors have not had the time to do a thorough analysis of the tool, but it is clear that it tries to achieve many of same goals as this project, i.e. integration of the testing process into the IDE and it employs an advanced form of symbolic execution for test generation. More importantly it has an extensive API providing many of the functionalities related to symbolic execution that has proven to be time-consuming to implement. It would therefore be interesting to see if some of the functionality could be used or if this could be used as a complete component for symbolic execution.

It should also be noted that while this report documents many of the issues concerning an implementation of the chosen techniques it also leaves out the problem of generating oracles for the tests. This is an area that most likely will involve a

large amount of interaction with the users through the IDE, but also introduce several new modules in the back-end. It could therefore prove to be a significant task, but would be necessary in order to produce a complete tool.

CHAPTER 9

CONCLUSION

This project report has documented an effort to create prototype tool for automated unit testing on the .NET platform. Similar tools already exist both in academia and on the commercial market. However, except from .TEST from Parasoft none of the tools currently available targets the .NET platform. This limits testers either to write their own test cases or use the commercial tool.

The main purpose of this project report has been twofold:

- To develop and implement a working prototype of an automated testing tool targeting .NET. The tool should focus on the full spectrum of the implementation including advanced test generation algorithms, a command line interface as well as integration with an IDE.
- To identify the different components needed for implementing the fully functional automated testing tool targeting the .NET platform. This includes identifying which components are readily available for .NET and which ones that potentially needs to be implemented.

Implementing a working prototype was not accomplished, but several aspects and technical problems with creating such a tool for the .NET platform have been identified.

Several elements related to the interaction with the programmer through the GUI were analyzed and designed. The effort needed to implement them however, was greater than anticipated which means that nothing conclusive can be said.

The prototype also tried to combine random testing and symbolic execution as a test generation technique. Due to problems with instrumentation of .NET assemblies symbolic execution was never implemented and it was established that a proper

analysis/instrumentation library is needed. Such libraries are readily available in different versions for Java, but are not for .NET to the same extent. It was also established that the tools available were not mature enough and very complex to use.

Even though an implementation of a running prototype was not accomplished the main elements needed for an implementation on the .NET platform were identified and we are still confident in that the development and implementation of an automated testing tool for .NET would be beneficial for software developers wanting to deliver programs that have been well tested.

List of Figures

1.1 Error introduction, detection and cost of repairing. 1

4.1 Overview of the test tool. 20
4.2 Package diagram of the tool. 21
4.3 Preliminary class diagram of the test tool. 22

5.1 Class diagram for the `TestCase` class. 28
5.2 Screenshot of the Visual Studio 2008 IDE. 30
5.3 Screenshot of Jtest in Eclipse. 32
5.4 Screenshot of jCUTE 33
5.5 Screenshot of FindBugs in Eclipse 34
5.6 Mockup of the package explorer. 36
5.7 Mockup of a source code editor. 36
5.8 Screenshot of Clover. 39

6.1 Screenshot of visual elements in Visual Studio. 46
6.2 Symbolic execution. Example from [21] 49
6.3 Parameter-graph generated by the JCrasher algorithm. 55

A.1 Time schedule - February 79
A.2 Time schedule - March 79
A.3 Time schedule - April 80
A.4 Time schedule - May 80
A.5 Time schedule - June 80

List of Listings

2.1 Example method with informal specifications. 9
2.2 Manual unit test for the code in Listing 2.1 10
2.3 JML specifications added to Listing 2.1 10
2.4 Manual unit test for the code in Listing 2.1 11
2.5 Parameterized unit test for the code in Listing 2.1 12
5.1 Pseudo code for the algorithm used by Randoop[8] 27
6.1 Outline of the Visual Studio Command Table-file 45
6.2 The definition of a top menu bar item with the label *Icarus* 46
6.3 The definition of a button (menu-item) placed in a toplevel-menu . . 47
6.4 Concrete execution. Example from [21] 48
6.5 This method illustrates the array element aliasing problem. The method replaces a pair of elements with their sum and difference. Source: [15] 50
6.6 Example class for JCrasher testing 54
7.1 Pseudo code for comined random/symbolic testing. 57
7.2 Phoenix initialization code. 59
7.3 Code for the main instrumentation loop. 60
7.4 Code for the assembly import method. 61
7.5 Main instrumentation loop. 62

Bibliography

[1] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," *SIGPLAN Not.*, vol. 40, no. 6, pp. 213–223, 2005. [cited at p.vi, 6, 8]

[2] G. J. Myers and C. Sandler, *The Art of Software Testing*. John Wiley & Sons, 2004. [cited at p.1, 7]

[3] P. Liggesmeyer, M. Rothfelder, M. Rettelbach, and T. Ackermann, "Qualitätssicherung software-basierter technischer systeme - problembereiche und lösungsansätze," *Informatik Spektrum*, vol. 21, no. 5, pp. 249–258, 1998. [cited at p.1]

[4] "Junit.org - resources for test driven development." Web, 2007. `http://www.junit.org`. [cited at p.2]

[5] M. Bach-Sørensen and M. Malm, "Automated unit testing - a survey of tools and techniques." web, 2008. `https://services.cs.aau.dk/public/tools/library/details.php?id=1199950650`. [cited at p.5, 15, 31, 33, 38, 65]

[6] C. Csallner and Y. Smaragdakis, "JCrasher: An automatic robustness tester for Java," *Software–Practice & Experience*, vol. 34, pp. 1025–1050, Sept. 2004. [cited at p.6, 25, 53, 54]

[7] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Experimental assessment of random testing for object-oriented software," in *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, (New York, NY, USA), pp. 84–94, ACM, 2007. [cited at p.6, 7, 10, 25]

[8] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, (Minneapolis, MN, USA), May 23–25, 2007. [cited at p.6, 7, 25, 26, 27, 74]

[9] C. Csallner and Y. Smaragdakis, "Dsd-crasher: a hybrid analysis tool for bug finding," in *ISSTA '06: Proceedings of the 2006 international symposium on*

Software testing and analysis, (New York, NY, USA), pp. 245–254, ACM, 2006. [cited at p.6, 9]

[10] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, (Glasgow, Scotland), pp. 504–527, July 27–29, 2005. [cited at p.6]

[11] R. Majumdar and K. Sen, "Hybrid concolic testing," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, (Washington, DC, USA), pp. 416–426, IEEE Computer Society, 2007. [cited at p.6, 8]

[12] Parasoft, "Jtest: Java unit testing & code compliance - parasoft." web, 2007. `http://www.parasoft.com/jsp/products/home.jsp?product=Jtest`. [cited at p.6, 8, 31, 32]

[13] Agitar, "Agitar software: Automated unit testing for java applications." Web, 2007. `http://www.agitar.com/solutions/products/agitarone.html`. [cited at p.6, 8, 31]

[14] M. Pezze and MichalYoung, "Software testing and analysis: Process, principles, and techniques," 2007. [cited at p.7]

[15] G. Lee, J. Morris, K. Parker, G. A. Bundell, and P. Lam, "Using symbolic execution to guide test generation: Research articles," *Softw. Test. Verif. Reliab.*, vol. 15, no. 1, pp. 41–61, 2005. [cited at p.7, 48, 50, 51, 74]

[16] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976. [cited at p.7, 48]

[17] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," in *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LECTURE NOTES IN COMPUTER SCIENCE, pp. 365–381, Springer Verlag, April 2005. [cited at p.8]

[18] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *CAV*, pp. 419–423, 2006. [cited at p.8]

[19] P. Godefroid, "Compositional dynamic test generation," in *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, (New York, NY, USA), pp. 47–54, ACM, 2007. [cited at p.8]

[20] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engg.*, vol. 10, no. 2, pp. 203–232, 2003. [cited at p.8]

[21] S. Khurshid, C. S. Păsăreanu, and W. Visser, “Generalized symbolic execution for model checking and testing,” *IEEE*, 2003. [cited at p.8, 48, 49, 73, 74]

[22] Microsoft, “Pex: Dynamic analysis and test generation for .net,” 2007. `http://research.microsoft.com/Pex/default.aspx`. [cited at p.8, 12]

[23] T. U. of Maryland, “Findbugs - find bugs in java programs.” Web, 2008. `http://findbugs.sourceforge.net/`. [cited at p.9, 31]

[24] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, “The daikon system for dynamic detection of likely invariants,” *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 35–45, 2007. [cited at p.9]

[25] C. Csallner, N. Tillmann, and Y. Smaragdakis, “Dysy: dynamic symbolic execution for invariant inference,” in *ICSE ’08: Proceedings of the 30th international conference on Software engineering*, (New York, NY, USA), pp. 281–290, ACM, 2008. [cited at p.9]

[26] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Muller, , and J. Kiniry, “The jml reference manual.” web, 2007. `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmlrefman.pdf`. [cited at p.9, 10]

[27] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, “Extended static checking for java,” in *PLDI ’02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, (New York, NY, USA), pp. 234–245, ACM, 2002. [cited at p.9]

[28] N. Tillmann and W. Schulte, “Unit tests reloaded: Parameterized unit testing with symbolic execution,” *IEEE Softw.*, vol. 23, no. 4, pp. 38–47, 2006. [cited at p.12]

[29] McGill, “Soot: a java optimization framework.” web, 2008. `http://www.sable.mcgill.ca/soot/`. [cited at p.19]

[30] BCEL, “Bcel: The byte code engineering library.” web, 2006. `http://jakarta.apache.org/bcel/index.html`. [cited at p.19]

[31] N. A. R. Center, “Java pathfinder.” web, 2008. `http://javapathfinder.sourceforge.net/`. [cited at p.19]

[32] Microsoft, “Phoenix: a software optimization and analysis framework.” web, 2008. `http://research.microsoft.com/Phoenix/`. [cited at p.19]

[33] D. S. Group, “Rail: Runtime assembly instrumentation library project.” web, 2005. `http://rail.dei.uc.pt/`. [cited at p.19, 51]

[34] Microsoft, “Profiling api.” web, 2008. `http://msdn.microsoft.com/en-us/library/bb384547.aspx`. [cited at p.19]

[35] Hewlett-Packard, "Simplify: A theorem prover for proof checking." web, 2005. http://www.hpl.hp.com/downloads/crl/jtk/. [cited at p.20]

[36] L. de Moura and N. Bjørner, "Z3: An efficient smt solver." web, 2008. http://research.microsoft.com/projects/Z3/. [cited at p.20, 53]

[37] Microsoft, "Microsoft visual c# 2008 express edition." Web, 2008. http://www.microsoft.com/express/vcsharp/. [cited at p.30]

[38] ic#code, "The open source development environment for .net." Web, 2008. http://www.icsharpcode.net/OpenSource/SD/. [cited at p.30]

[39] MonoDevelop, "Monodevelop." Web, 2008. http://www.monodevelop.com/Main_Page. [cited at p.30]

[40] Microsoft, "Visual studio sdk." Web, 2008. http://msdn.microsoft.com/en-us/library/bb166441.aspx. [cited at p.31]

[41] K. Sen, "Cute :: A concolic unit testing engine for c and java." Web, 2007. http://osl.cs.uiuc.edu/~ksen/cute/. [cited at p.31]

[42] Atlassian, "Atlassian clover & code coverage analysis." Web, 2007. http://www.atlassian.com/software/clover/. [cited at p.38, 39]

[43] G. Inc., "Ncover." Web, 2008. http://www.ncover.com/. [cited at p.38]

[44] P. D. Coward, "Symbolic execution and testing," *Inf. Softw. Technol.*, vol. 33, no. 1, pp. 53–64, 1991. [cited at p.49]

[45] P. G. Saswat Anand and N. Tillmann, "Demand-driven compositional symbolic execution," pp. 367–381, 2008. [cited at p.50]

[46] C. S. P. Saswat Anand and W. Visser, "Symbolic execution with abstract subsumption checking," pp. 163–181, 2006. [cited at p.50]

[47] M. Eaddy, "Wicca v2." web, 2008. http://www1.cs.columbia.edu/~eaddy/wicca/. [cited at p.52]

[48] McGill, "The satisfiability modulo theories competition." web, 2008. http://smtcomp.org/. [cited at p.53]

Appendix A

Project Time Schedule

February 08						
M	T	W	T	F	S	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29		

Initial analysis Stage 1 Stage 2 Stage 3 Report

Figure A.1: Time schedule - February

March 08						
M	T	W	T	F	S	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

Initial analysis Stage 1 Stage 2 Stage 3 Report

Figure A.2: Time schedule - March

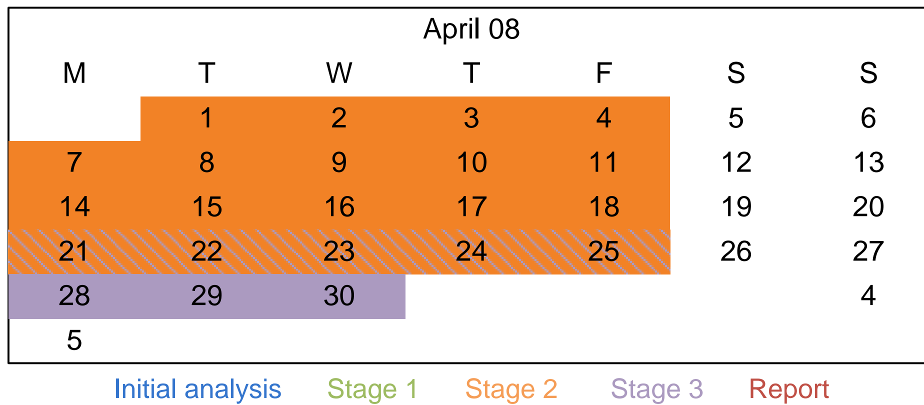

Figure A.3: Time schedule - April. The hachured area are overrun form the subsequent stage

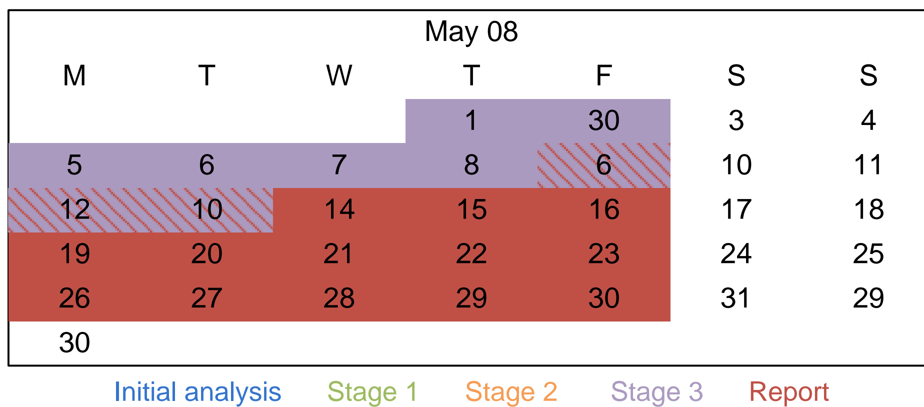

Figure A.4: Time schedule - May. The hachured area are overrun form the subsequent stage

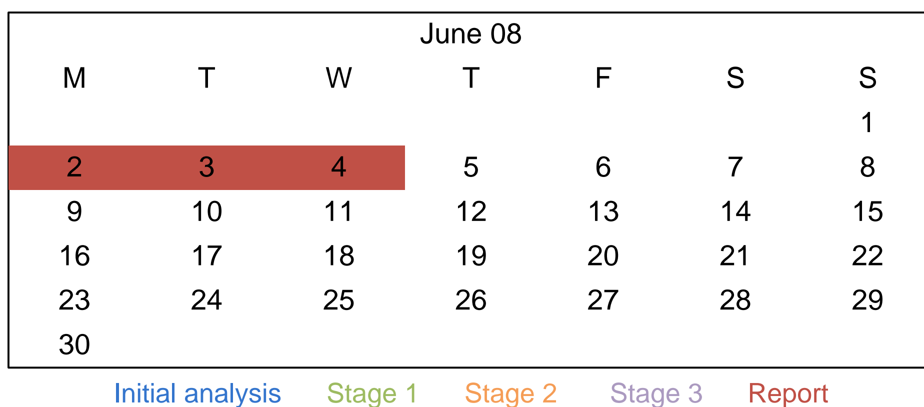

Figure A.5: Time schedule - June