Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master’s Thesis Nr. 60

Improving Automatic Software Repair with Probabilistic Symbolic Execution

Tobias Rudolph

Course of Study:	Softwaretechnik
Examiner:	Dr. rer. nat. Antonio Filieri
Supervisor:	Dr. rer. nat. Antonio Filieri
Commenced:	2016/10/28
Completed:	2015/04/28
CR-Classification:	D.2.5 D.2.8

Abstract

Automatic software repair techniques aim at repairing error-prone code automatically. In particular, test-based automatic software repair approaches use test cases to locate a bug and evaluate the automatically created repair code. However, this evaluation is only based on the successful or failing execution of the test cases but ignore the behavior of the software under the majority of of usage scenarios, which are not covered by the given test cases. Despite a variety of test-based program repair approaches have appeared in literature, the problem of a repair code altering the behavior of the program for uncovered scenarios has not addressed.
This thesis introduces a new metric to quantify the discrepancy in the set of possible execution paths introduced by different repair candidates, and computes it from the source code of the program using probabilistic symbolic execution. This allows for an exhaustive analysis that takes into account also the behaviors not covered by the available test cases. The approach has been implemented extending Nopol, an opensource tool for test-based automatic software repair. The new approach has been evaluate on a set of Java benchmarks, demonstrating an improved quality of the computed program repairs, which do not only pass the tests but also limit as much as possible the alteration of the uncovered program behaviors.

Zusammenfassung

Die automatische Softwarereparatur ist eine Technik, die entwickelt wurde, um das Finden und Beheben von Softwarefehlern zu automatisieren. Oft werden bei solchen Techniken Tests verwendet um Softwarefehler automatisch zu lokalisieren und eine generierte Reparatur zu evaluieren. Allerdings haben solche Verfahren nur Kenntnis darüber, wie sich eine Software unter genau diesen Testfällen verhält. Darauf, wie sich die Software unter den nicht durch Testfällen abgedeckten Anwendungsfällen verhält, wurde in keinem der bisher veröffentlichen Verfahren eingegangen.
In dieser Masterarbeit wird eine neue Metrik eingeführt, mit der die Diskrepanz hinsichtlich der gesamten möglichen Programmpfade von verschieden Softwarereparatur-Ergebnissen bestimmt werden kann. Diese Metrik ermöglicht eine Analyse, die über die Informationen hinausgeht, die aus den Testfällen generiert werden, und auch das Verhalten der Szenarien, die nicht Bestandteil der Testfällen sind, beachtet. Die Open Source Reparatursoftware Nopol wurde in dieser Thesis für die neu eingeführten Metrik angepasst. Während der Evaluation der Erweiterung wurde gezeigt, dass die Qualität der generierten Reparaturen verbessert wurde, indem die generierten Reparaturen nicht nur die Testfälle erfüllen, sondern auch die Beeinflussung der nicht in den Testfällen enthalten Szenarien möglichst gering gehalten wird.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

Developers spend about 90 % [Seacord et al., 2003] of the lifetime of a software fixing bugs, with significant costs for the whole project. In order to help developer to find and repair bugs, several automatic software repair tools have been introduced. The goal of automatic software repair is to find possible errors automatically and to propose a change in the code to fix these errors. Some promising approaches like Nopol [Xuan et al., 2016] or GenProg [Forrest et al., 2009] have been presented to automatically create repairs. But the study reported in [Qi et al., 2015] showed that these and the others tools produced so far do not provide effective solutions for fixing the detected bugs. In particular, most solutions alter the code to make the parts failing one or more tests or assertions unreachable; hence, ultimately, in most of the cases they fail to fix the program. Finally, the proposed repair are often very different from corresponding fixes produced by human developers, and their rational is hard to understand.

Most of automatic software repair approaches use a set of provided test cases to both find errors and evaluate candidate repairs. However, test cases do not provide, in general, a complete specification of the program, covering only a limited set of a program behaviors. Passing all the tests may indeed not be a sufficient condition to determine a repair actually fixed the bug. Furthermore, basing the synthesis of the repair only on the satisfaction of the available test case may ignore the alteration such repairs could introduced on behaviors of the software not covered by any the test case. This could not only fail to repair possible errors, but may also alter or disable other functionalities of the software.

In [Filieri et al., 2015] the use of probabilistic symbolic execution is proposed to evaluate and improve the quality of automatically created repair candidates. Probabilistic symbolic execution is a technique that allows the computation of quantitative information about the behaviors of a program. In particular, it can be used to quantify the fraction of the input domain that follows each execution path of the program. This information can be used to analyze and evaluate differente repair candidates generated by an automatic program repair tool against the original version of the program in order to quantify the discrepancy in the execution flow introduced by each of them. The candidate passing all the tests and introducing the lowest discrepancy with respect to the original behaviors of the program is then proposed to the developer.

In this thesis, a discrepancy metric computable via probabilistic symbolic execution is defined and used to extend the automatic software repair tool Nopol. The implemented tool is then evaluated on a set of benchmarks, demonstrating the effectiveness of the metric in quantifying the discrepancy introduced by a repair candidate, in turn driving the elicitation of the best solution among the candidates generated by Nopol.

1.2 Goals

In order to investigate the possibility of improving automatic software repair based on the quantification of the discrepancy between the original program and the repaired version, this thesis will pursue the following goals:

- Survey the state of the art in automatic software repair
- Define a metric of discrepancy between the original version of a program and a proposed repair candidate
- Implement a discrepancy quantification procedure using probabilistic symbolic execution on top of the opensource automatic software repair Nopol
- Evaluate on a set of benchmarks the effectiveness of the proposed metrics in reducing the behavioral discrepancy between the original and the repaired version of the program

1.3 Document Structure

The remaining thesis is structured as follows:

- Chapter 2 provides an overview of previous work that focuses on the analysis of software evolution and automatic software repair approaches.
- Chapter 3 introduces a new metric which can be used to compare the quality of repair candidates.
- Chapter 4 presents an implementation of ProRepair an enhancement of Nopol which benefits form the newly defined metric
- Chapter 5 evaluates the new metric and the enhancement of Nopol
- Chapter 6 presents a conclusion of the thesis and future work

Chapter 2

Foundations and Technologies

2.1 Automatic Software Repair

The first goal of this thesis is to provide an overview over the state of the art of automatic software repair approaches. In this chapter a definition of automatic software repair will be introduced, state of the art automatic software repair approaches are described and the approach which is used within this project is chosen. This chapter is based on test case based automatic software repair approaches because this is the technique which is used in nearly all state of the art approaches. Other techniques like the use of assertions [Elkarablieh et al., 2007] or pre- and postcondition [Wei et al., 2010] are not subject of this thesis.

2.1.1 Test Case Based Automatic Software Repair

The goal of automatic software repair is to transfer source code, which is error-prone, into an error free state. During the last years, especially since 2009, different automatic software repair approaches have been published to achieve this goal. Most of the state of the art automatic software repair approaches are based on test cases. Those approaches use test cases to determine if a project is bug free and to verify that a correct repair candidate is created. The benefit of this technique is that the code of the error-prone project does not have to be prepared. The only thing which is needed for test case based automatic software repair is the project itself and test cases. The use of test cases is common practice in state of the art software development. Most of the published automatic software repair approaches are based on test cases. The general process of repairing a bugged code using test cases based automatic software repairing approach is shown in figure 2.1. The process can be compared with a control loop in the area of control engineering. First of all the error prone code is analysed by the automatic software repair tool. After this step, the project is manipulated, respectively repair candidates are created. These repair candidates or the manipulated code is tested against the given test suite. If all test cases within the test suite passes, a correct patch has been created, the repair process is successful. Otherwise the project is manipulated and repair candidates created until the test cases pass or all possibilities of the approach are exploited and the approach is not able to find a repair.

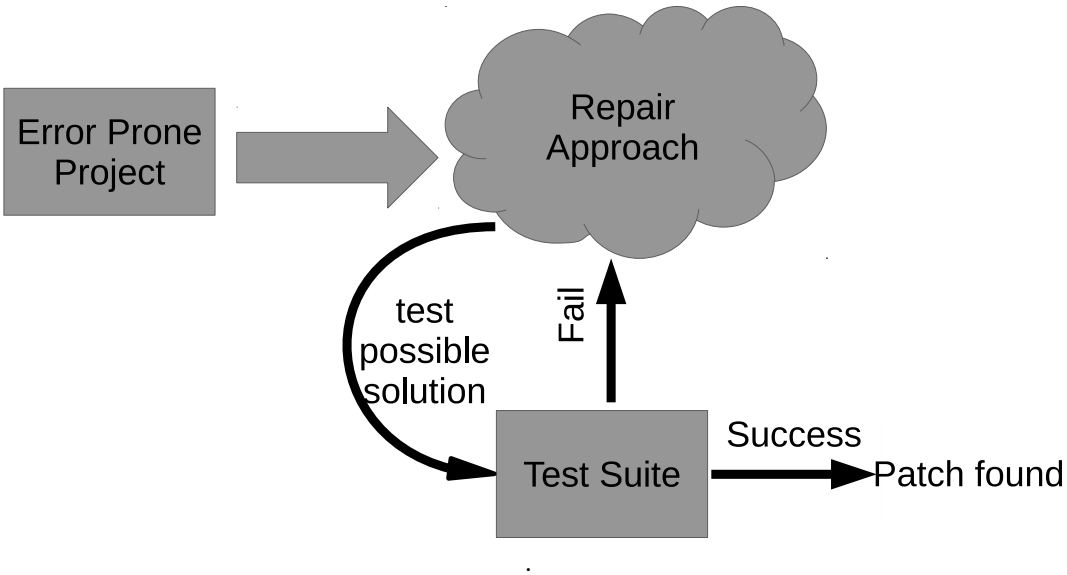

Figure 2.1. Test based automatic software repair

2.1.2 Survey approaches

To determine which automatic software repair approach can be improved with probabilistic symbolic a survey of the major test based automatic software repair approaches is given in this section. Through the big amount of automatic software repair approaches published, there will be some approaches which are not covered in the following survey. An overview of all can be found at [Monperrus, 2015].

JAFF

The approach of JAFF (Java Automatic Fault Fixer)[Arcuri, 2011] use evolutionary algorithms to create fixes in order to repair buggy source code. JAFF is one of the first approaches which was published with the possibility to repair code with only limited restriction. In order to fix an error prone code, JAFF combines different software engineering techniques.
Like all automatic software approaches discussed in this chapter JAFF is based on test cases which are used to evaluate the correctness of a created repair. As first step JAFF searches for the cause of the bug in the code. To find a possible cause of an error, all test cases are executed and every line gets ranked after its suspiciousness. The suspiciousness of a line of code is the ratio of a line passed by executing test cases which fail and test cases which pass. A statement with a high suspiciousness is often passed by failing test cases

and therefore more likely causing the error. The exact equation which is used within JAFF is shown in equation 2.1

$$susp(s) = \frac{failed(s)/totalfailed}{passed(s)/totalpassed + failed(s)/totalfailed} \tag{2.1}$$

After ranking the statements of the code, evolutionary programming is used to create a fix which lets all test cases pass. There are seven different operations which can be used by JAFF to create a repair. The statement where the repair is applied is randomly chosen out of the statements which are ranked the highest. The modifications done by these operations are ranked with a fitness function. With the result of this function the repair is chosen which passes the most test cases. On this solution the evolutionary algorithm is applied until a repair is found.
JAFF can only deal with a subset of the JAVA language. Therefore JAFF was only evaluated by the developer with bugged programs which are suitable to this limitations. An evaluation of real world project which would make the results of JAFF comparable to other automatic software repair program is not available.

GenProg

The award winning automatic software approach GenProg [Goues et al., 2012] is one of the most important contributions in the area of automatic software repairing. The foundations of GenProg have been published in 2009 [Forrest et al., 2009] and the approach is based on generic programming.
In order to start the repair process, GenProg generates an AST (abstract syntax tree). On this AST three different operations are used to manipulate the source code in order to create a bug free program code:

- deletion
- addition
- replacement

Trough the manipulations GenPorg generates different manipulation of the AST. All of these manipulated AST are ranked according to a fitness level. A fitness function ranks those candidates which are closest to the desired outcome, an error free software. To find such a result, GenProg uses test cases. Repair candidates with test cases which outcomes are closer to the desired outcomes are ranked with a higher fitness level. The patches with a high fitness are used for further manipulation in order to find a valid repair. GenProg repeats this process: manipulating the AST and choosing the best results according to the fitness value until a valid repair has been found which passes all test cases.

Statements which are visited during the execution of passing test cases are not manipulated by GenProg to ensure that these test cases do not fail after executing the manipulation.

GenProg is evaluated on real world projects. The results of GenProg are used as a benchmark for most of the automatic software repair approaches which have been developed since.

RsRepair

RsRepair [Qi et al., 2014] is an automatic software repair approach which is based on GenProg. The major innovation of RsRepair is the use of randomized search for automatic software repair. It has been shown that random search outperforms the GenProg mechanism on average. Analyses of GenProg suggest that most of the found patches are generated not during the generic programming part of the algorithm but on the initialization process of the generic programming. The initialization process of GenProg is based on randomized search. But RsRepir only uses random search as technique to find a valid patch in order to save the overhead generated through generic programming. Instead of the use of a fitness function to find a correct patch that let given test cases pass, test case prioritization is used by RsRepair. Test case prioritization ranks test cases according to their probability to fail during the evaluation of the project candidates. Test cases which seem to fail more often are applied before test cases which fail less. If a test case fails during the evaluation, the evaluation is aborted and a new repair candidate is searched. This leads to a more efficient execution compared to GenProg. Due to the absence of a fitness function RsRpair performs with a higher effectiveness level compared to GenProg caused by a lower amount of created repair candidates. The amount of successful crated repairs by GenProg is slightly better because RsRepair can only create a subset of the repairs created by GenProg.

AE

AE [Weimer et al., 2013] is focused on reducing the costs which are generated by test based automatic software repair approaches like GenProg. Similar GenProg or RsRepair, AE is based on a generate and validate mechanism. Multiple possible patches are generated trough generic programming and validated with test cases. When all test cases pass the patch is correct, otherwise more possible patches are generated. The major difference between AE and GenProg is, that semantically equal creation of repair candidates are not evaluated by the fitness function. The results of running test cases against the manipulated code are the same as the outcome of semantically equal code. This leads to significant cost savings. According to their calculation GenProg is three times more expensive compared to AE in terms of costs in dollars by executing on an amazon cloud environment in 2013. In terms of success, AE is almost as good as GenProg. Since the created patches of AE are a subset of patches created by GenProg, AE can not outperform GenProg in this area.

Kali

In order to improve the quality of test based automatic software repair tools, especially RSRepair and GenProg, the automatic software repair tool Kali [Qi et al., 2015] has been introduced.

Kali can apply the following three types of modifications:

- Setting an if statement to true or false in order just to execute always one branch of an if clause
- Insert a return statement before the statement where the error occurs
- Remove one or more statements

On the one hand, these manipulations are easy to evaluate since they are simple to read and understand by a developer. But on the other hand the search space of finding a repair is limited with only these three manipulation types are available. In an evolution with a benchmark set of error prone software created by GenProg developers, Kali achieves a similar success rate compared to AE but creates patches faster and easier to understand.

PAR

PAR [Kim et al., 2013] is an automatic software repairing tool based on the use of predefined patterns. The aim of PAR is to improve the readability of patches. The developers of PAR analysed repairs created by human developers and transferred the most often and easy to use techniques into eight different patterns. These patterns are used during the repair process of PAR after the statements are ranked according to their suspiciousness. These patterns are used to manipulate the code until the error is repaired. With the use of these patterns the created patches are similar to fixes created by human developer. Therefore the created patches are meaningful and easy to read.

MutRepair

[Debroy and Wong, 2010] describes a technique which uses mutation in order to repair automatically faulty programs. The technique of mutation testing is adapted. First of all the statements get ranked after their suspiciousness. The statement with the highest suspiciousness is mutated. If a mutation passes all test cases, a repair is successful generated.

ASTOR

Astor [Martinez and Monperrus, 2014] is a framework which contains different automatic software repair approaches. This framework was published to solve the problem that the software of some automatic software approaches is not published by their developers. The framework contains the approach GenProg, MutRepair and a subset of PAR.

SemFix

An other approach for repairing a bug automatically was introduced with SemFix [Nguyen et al., 2013]. Like the approaches described in this section, SemFix is based on find and evaluate a patch with test cases.

The novelty of SemFix is to use symbolic execution in order to create a repair. First the statements of the project are ranked by their suspiciousness. Again the most suspicious statement is going to be replaced by an newly created statement. Symbolic execution is used to create a repair. Other than normal symbolic execution, SemFix uses this technique not from the beginning of the project but starts the execution of symbolic analyses at the suspiciousness line of code. Assuming within this line a variable is an assignment, this variable value is transferred into a symbolic value and it is analysed which values the variable has to be assigned in order to let all test cases pass. Out of these information the SMT (Satisfiability Modulo Theories) solver calculates a solution which leads to a repair.

SemFix was improved by the enhancement DirectFix [Mechtaev et al., 2015]. The main improvement compared to SemFix is that DirectFix searches for the most simplistic possible repair. A simple repair enhances the readability of the repair and is in most of the cases closer to the repair a human developer would have created.

Nopol

Nopol [DeMarco et al., 2014] [Xuan et al., 2016] is an automatic software repair approach focused on fixing broken branch conditions and missing preconditions. Like many automatic software repair approaches Nopol is based on test cases. The approach is divided into two steps: finding a suspicious line and generate a fix.
In order to find the cause of a bug, the lines of code are bases on their suspiciousness. The suspiciousness of a line is calculated considering the ratio of visited passed and missed test cases like described in Equation 2.1. After the possible cause of a bug is found, Nopol tries to fix the possible broken line of code. Therefore a novel technique called *angelic* has been introduced. For a broken branch condition this approach sets the condition to true respectively to false.

If all test cases pass with either of both settings Nopol tries to generate a repair. The parameters which are available at that point where the repair is applied are combined due to a combination which leads the branch condition to true respectively for all given test cases. If such a combination can be created, the created equation is the repair for the broken branch condition.
Recently Nopol was extended with symbolic execution with Java Pathfinder to integrate the approach similar to the approach introduced SemFix.

SPR

An other technique which is focused in repairing branch condition is SPR [Long and Rinard, 2015]. SPR adds disjunction or conjunction to a branch condition to let all test cases pass. Unlike Nopol which generates repair with solving an equation which contains knowledge, SPR uses the generate and validate mechanism in order to find a correct repair. Additional SPR inserts if statements or inject control flow statements like return or break to let all test cases pass.

2.1.3 Limitations of automatic software repair approaches

In the previous section an overview of the major state of the art test based automatic software repair approaches has been provided. In this section the limitations of those approaches will be discussed.
To be able to describe the limitations, criteria have to be defined to measure the quality of automatic software repair approaches. In []three attributes are defined to rate repair tools:

1. scalability: ability to repair large real world programs
2. repair ability: handling a variety of different bugs
3. quality of repair: repair makes minor changes to the code, repairs are comparable to repairs done by a human developer etc.

The most important aspect of automatic software repair approaches is the ability to create valid repairs. Independently of which approach is used to automatically repair an error prone software, the success rate is very low. This was shown within two studies which present benchmarking results of automatic software repair approaches applied on real world projects.
In [Qi et al., 2015] four test based automatic software repair approaches written with the ability to repair C programs: GenProg, RsRepair, Kali and AE are compared. This benchmark consists of 8 eight program with 105 bugs. These bugs are part of the original evaluation of GenProg. GenProg creates 18, RsRepair 20 and both AE and Kali 27 RsRepair repairs. Out of these plausible repairs, 2 repairs by both GenProg as well as by RsRepair are correct. A correct patch is a plausible patch which can be verified to be correct. AE and Kali created 3 correct repair each. Only a maximum of 27 % have been repaired. But 2% respectively 3% of all bugs are correctly repaired.

The other study [Durieux et al., 2015] used the framework Defect4J to compare the effectiveness of Nopol, Kali and GenProg. Defect4J [Just et al., 2014] is a framework of error prone Java projects. The framework contains four projects with 224 bugs and 12182 test cases. Kali and GenProg are originally written in C and only able repair C code. While Nopol is written in Java and is only able to repair Java code. To make these projects comparable, Kali and GenProg have been transformed to Java.

Out of 224 bugs, Nopol generates 34 patches, Kali 9 patches and GenProg 16 patches. The amount of repairs which could been verified to be correct are even smaller. Only four repairs created by GenProg, three created by Nopol and one repair created by Kale are verified as correct This study was done by the developers of Nopol, Defect4J has been provided by an other team.

The results of these two studies show how good state of the art automatic software repair approaches are considering the three attributes: scalability, repair-ability and quality of repair. All approaches which have been analysed during the two benchmarks are able to deal with large projects. Therefore the scalability is satisfying.
The repair ability of state of the art automatic state repair approaches is not satisfying. The success rate of creating repairs is between 25% to 4% dependent on approach and benchmark.

In terms of quality the benchmarks showed that most of the created repairs are not created with a high quality. The created repairs are often not or only hard readable. The automatic software repair approaches covered by the benchmarks were only able to create correct repairs with a success rate of 10% to 0%, dependent on approach and benchmark.

2.1.4 Choose of approach to work on

In the previous chapters an overview has been provided in Chapter 2.1.2 and the limitations have been shown in Chapter 2.1.3. In order to choose an automatic software repair approach which can be improved with probabilistic symbolic execution these findings have to be considered.

One precondition for the choice of an automatic software repair approach is that it should be able to repair Java code because the approach has to be compatible to symbolic analysis software which are described in Chapter 2.2 .

There are some projects available at this time which meet this requirement. The following list is taken from the homepage of Martin Monperrus, a leading researcher in the area of automatic software repair, who provides an overview of available automatic software repair tools (http://www.monperrus.net/martin/automatic-software-repair-tools). This list contains the available Java based automatic software repair tool.

- Astor: Implementation of GenProg, PAR and MutRepair
- Kali: Fixes code by applying three simple modifications
- Nopol: Specialized in fixing branch conditions and preconditions
- JAFF: Evaluationally approach to fix bugs
- Java-RSRepair: Java version of RsRepair

The comparison of Kali, Astor and Nopol with the Defects4J framework shows that Nopol is able to fix the highest amount of bugs, additionally Nopol produced the highest number of fixes which are correct. According to this study Astor, Kali and RsRepair have a lower success rate. JAFF is not evaluated with real world programs. Based on this analyses of available automatic software repair tools, Nopol is chosen as the automatic software repair tool which will be extended within this thesis with probabilistic symbolic execution. An other reason why Nopol is chosen is the way how Nopol creates repairs. As described in chapter 2.1.2 Nopol changes branch condition in order to create repairs. This leads to changes of the path flow which could be analysed via symbolic execution.

Nopol

In Chapter 2.1.2 a short introduction into Nopol was given. After choosing Nopol as approach which is improved with probabilistic symbolic execution a more detailed analyses of Nopol is needed.
Nopol is an automatic software repair approach which has been developed by the SPIRAL Group. Nopol includes different modi which can be chosen in order to determine what kind of errors have to be be repaired.

1. fixing missing precondition
2. fixing branch condition via angelic
3. fixing branch condition via symbolic analyses
4. fixing infinite loops

During this thesis the modus fixing branch condition via angelic is used. Angelic is a new method of repairing bugs introduced in context with Nopol. Angelic can be used to create repairs by fixing branch conditions. To be able to repair a new condition, Nopol sets the original branch condition to false respectively true. If based on this adjustment all test cases pass, a new condition is created with parameters which are used in this project setting the condition for all test cases to false respectively true.

The creation of a repair by Nopol is shown in Figure 2.2 and can be split into four different parts:

1. As first step Nopol ranks the suspicious. of every line of the analysed project. A statement with a high suspicious. is more likely to be the cause of the bug. In order to rank the statement the fault localization library GZoltar [Riboira and Abreu, 2010] is used.
2. In order be able to manipulate the project and in order to insert the fix and be able to get information out of the project which are needed to create a repair, the project gets

transformed with SPOON [Long and Rinard, 2015]. SPOON is a library which makes it possible to transform and analyse Java code.

3. As next step the repair is created. Based on the results of GZoltar the branch condition with the highest suspiciousness is chosen. On this statement a repair is created with angelic fixing. To create a repair based on angelic fixing a SMT equation is created. This equation contains all the information from the test cases and the angelic analysis. If the SMT solver can create a solution, the result is transformed into Java code. Otherwise the next branch condition with a lower suspiciousness is analysed

4. Finally the created fix is inserted by using SPOON into the project and transformed back into Java code. At the end Nopol runs all test cases again to evaluate the created repair.

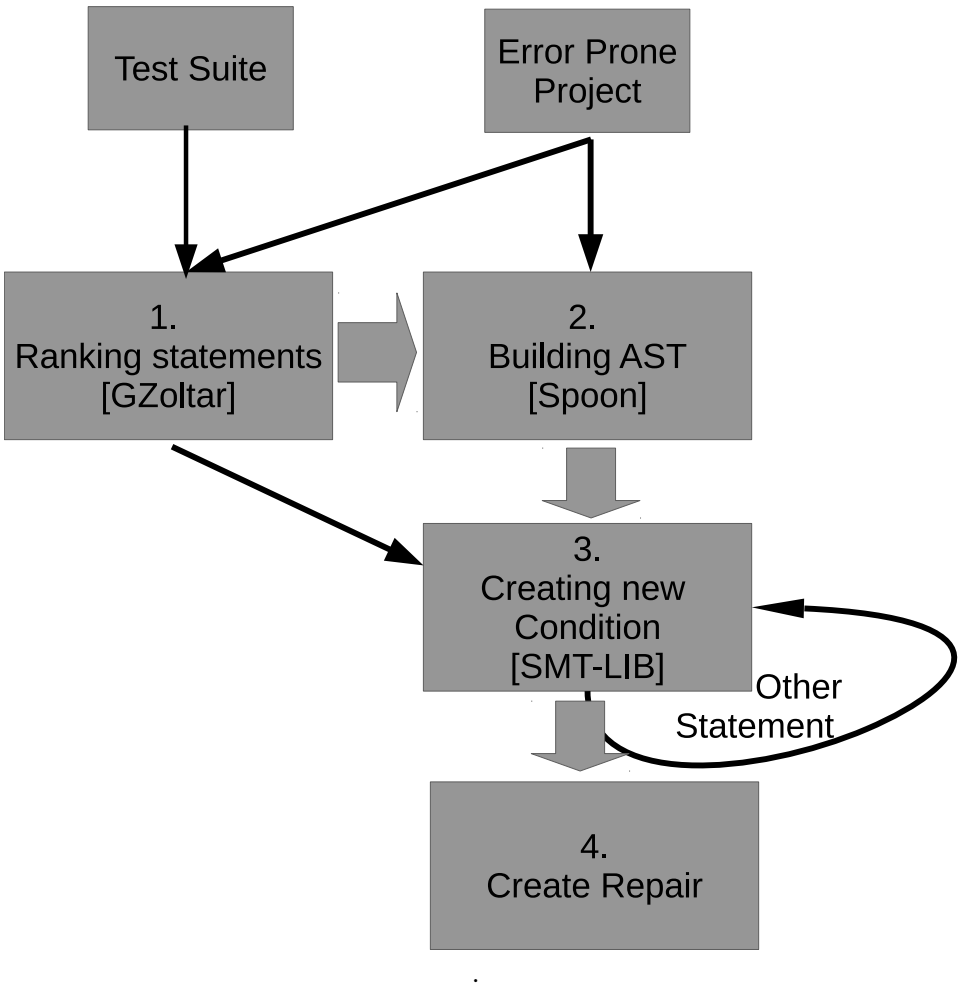

Figure 2.2. Nopol overview

2.2 Symbolic Execution

Symbolic execution [King, 1976] is a technique which is used to get knowledge about the overall behaviour of a program. While normal testing uses concrete values and is therefore only able to cover a small amount of possible input parameters, symbolic execution uses symbolic values instead of concrete values to be able to cover the whole input variance.

During symbolic execution all branches of a program are executed and the different branch condition are saved as path condition.(PC). Assuming the code has a branch condition c with two branches B1 and B2 like shown in listing 2.1. The PC gets split into the branch condition $PC_{B1} = PC \wedge c$ and $PC_{B2} = PC \wedge \neg c$. After this branch both branches are inspected with symbolic execution. During the execution PC are checked with constraint checker if every PC is satisfiable.If a PC does not become satisfiable the PC is dropped.

```
1 ...
2 if (c==true)
3          B1
4 else
5         B2
6 ...
```

Listing 2.1. Path condition

The benefit of symbolic execution compared with test cases is that symbolic execution covers the hole range of possible input instead of only a subset of them. Especially errors in branches which are reached with a low possibility could be discovered.

2.2.1 Probabilistic Symbolic Execution

An enhancement of symbolic execution is probabilistic symbolic execution [Geldenhuys et al., 2012]. Goal of an probabilistic symbolic execution is to get the probability of each possible path of a program. The probability of reaching a certain path can be calculate via model counting as shown in formula 2.3 Where #(c) is the amount of element which passes a certain PC and #(D) is the size of the domain.

$$Pr(c) = \frac{\#(c)}{\#(D)}$$

Figure 2.3. Probabilistic Symbolic Execution

Assuming the snippet in listing 2.1 is the first branch condition in the program, then the branch condition would be $B1 = B2 = 50\%$.

Probabilistic symbolic execution can be used to optimize testing effort. Depending on the strategies, branches with a higher or with a lower probability can be focused. An other use of probabilistic symbolic execution has been shown by [Filieri et al., 2013]. They performed a reliability studies based probabilistic symbolic execution. Other possible areas which could benefit from probabilistic symbolic execution are presented in [Filieri et al., 2015]. One example mentioned in this paper is automatic software repair, which is investigated within this thesis.

2.2.2 CATG

CATG [Tanno et al., 2015] is a concolic testing tool for Java programs. Concolic testing is a combination of symbolic and concrete testing. As a result of such a testing techniques, a test suite can be created which covers all branches of the program which is tested. Besides the concrete testing with concrete parameters, a symbolic testing is executed. With this technique limitation of symbolic and concrete technique can be compromised by the advantage of each other. Concolic testing starts with concrete testing. Random values are generated as input parameters for the fist execution of the program. During this execution knowledge about branch condition is collected. Based on this knowledge the symbolic part of concolic testing is executed. During this part all reachable branch condition are discovered and collected.

The goal of the second part, symbolic execution is, to generate information which test cases are needed to cover all possible branches of the project.

1. The class of the analysed function needs a main function.
2. Within the main function the input parameter of the to analysed function have to prepared for the conclolic testing as shown in listing 2.2

```
1 public static int main(String[ ] args){
2 int x = CATG.readInt(1);
3 boolean b = CATG.readBollean(true);
4 function(x,y);}
```

Listing 2.2. Initialization of CATG

2.2.3 PathConditionsProbability

A probabilistic symbolic execution tool which is under development by the University of Stuttgart and the Imperial College London is PathConditionalProbability. PathConditionalProbability generates out of the the outcome of the concolic analyses of CATG the probability of every possible path in a program.

Chapter 3

Comparing Repair Candidates

In state of the art automatic software repair approaches the correctness of a patch is the only metric which is used to evaluate a patch created by automatic software repair tools. In this subsection a new metric is introduced to improve the quality of automatically created patches.

3.1 Comparing Results of Different Automatic Software Repair Approaches

As investigated in Chapter 2.1.2 there are various test based automatic software repair tools published which are based on different approaches. In this section we want to compare results of two different approaches repairing the same error prone code.
One example of an error prone code which has been repaired by automatic software repairing tools is a snippet out of Tcas [Do et al., 2005]. Tcas is a traffic collision avoidance system, which is taken from the software-artefact infrastructure repository. This snippet is a part of the evaluation of SemFix as well as Nopol and shown in Listing 3.1. Both approaches are based on test cases, but use different approaches to generate a repair. The same test cases and the same Tcas snippet were used for the evaluation of SemFix as well as for the evaluation of Nopol.

```
1  int is_upward_preferred(boolean inhibit,int up_sep,int down_sep) {
2
3          int bias;
4          if(inhibit)
5                  bias = down_sep; //SemFix fix: bias=up_sep+100
6          else
7                  bias = up_sep;
8          if (bias > down_sep) //NOPOL fix : if (up_sep != 0)
9                  return 1;
10         else
11                 return 0;}
```

Listing 3.1. Tcas snippet

Five test cases have been used by both approaches to find the cause of the error and generate a valid repair. As shown in Table 3.1 two of the given five test cases fail. This implies that the Tcas snippet is not bug free. The test cases could be incorrect, which would lead to invalid results. But it is assumed that all test cases are correct and expect the correct output. The code snippet of Tcas is shown in Listing 3.1. The function in the snippet has three input parameter and the possible output of 0 or 1. Besides the original code the snippet, the Listing 3.1 shows also the fixes applied by SemFix and Nopol.

Table 3.1. Test Cases for Tcas snippet

Test Case	Input			Expected output	Observed output	Status
	inhibit	up_sep	down_sep			
1	true	0	100	0	0	pass
2	true	11	110	1	0	false
3	false	100	50	1	1	pass
4	true	20	60	1	0	false
5	false	0	10	0	0	pass

Obviously Semfix and Nopol create different repairs because they are using different techniques to repair a bug. The results of fixing Tcas are taken from the published papers about these tools [DeMarco et al., 2014] [Nguyen et al., 2013]. We could evaluate the results which have been produced by Nopol. SemFix's approach is not available, therefore we could not evaluate SemFix's results.
Through different approaches which line is to change in order to let all test cases pass, SemFix finds line 5 as the line which is most likely the line which repair makes all test cases pass, Nopol finds line 4 and line 8 both equally likely to be the cause of failing the two test cases.
SemFix's repair approach is to replace line 5 statement change $bias = down\_sep$ to $bias = upsep + 100$. The repair approach of Nopol changes the statement at line 8 $if(bias > downsep)$ to $if(upsep! = 0)$. Both approaches are able to change the code snippet to let all test cases pass. According to the definition given that a code is repaired if all test cases passed, both were able to repair the program, too. Never the less both fixes are obviously quite different. SemFix changes a statement, Nopol changes an if condition. This leads to the question whether the quality of these both fixes is the same and which fix should be preferred.

Even if the result of these two automatic program repair tools seems to be correct, the repairs are only based on five test cases. With this limited knowledge these fixes are created. This means that most of the input variants which are possible are not covered with these test cases. Even if the code would have a high test case coverage, it is almost impossible to cover all possible inputs with test cases. But test based automatic software repair tools can only consider defined test cases, all other possible scenarios are not considered. The

repair which is implemented by an automatic software repair tool, or implemented by a developer on his own could change the behaviour of these uncovered scenarios and turn a correct behaviour into a wrong one. Whenever this happens, the goal of fixing an error in the code would not be achieved and maybe a even create a worse error has been created.

Table 3.2. Test suite with an additional test case

Test Case	Input			Expected output	Observed output	Nopol Status	SemFix Status
	inhibit	up_sep	down_sep				
1	true	0	100	0	0	pass	pass
2	true	11	110	1	1	pass	pass
3	false	100	50	1	1	pass	pass
4	true	20	60	1	1	pass	pass
5	false	0	10	0	0	pass	pass
6	**false**	**400**	**300**	**0**	**1**	**fail**	**pass**

In order to show that such a creation of a false positive repair is possible, a new scenario is added to the existing five test case. This new scenario is shown in Table 3.2 as test case number 6. The table shows the test cases and the results of the test cases after the bug is fixed by the approaches Nopol and SemFix. The new test case fails with the program repaired by Nopol. But the test case number six is a scenario which should pass. With the program being repaired by SemFix the new test case passes. In this specific case, the fix of SemFix seems to be better, compared to the fix that Nopol created. But as mentioned before this is just based on a low amount of scenarios. Therefore the question which repair is better is still not answered. To be able to answer this section a new metric is introduced in this thesis, which considers the program flow of all possible inputs in order to be able to rank different repair candidates.
As discussed before a major disadvantage of using test cases to find a repair is the lack of a complete coverage over all possible inputs.

A combination of test based automatic software repair approaches and probabilistic symbolic execution might help to overcome this problem. Test cases are still used to evaluate a repair. Additionally the behaviour of the original program and the repaired program are compared. The probability of reaching every branch is measured via probabilistic symbolic execution. Repairs which influences a lower amount of scenarios should have similar results of a probabilistic symbolic analysis compared to the results of a probabilistic symbolic analysis of the original program. A repaired program which changes the behaviour of many scenarios should have an unlike result compare to the original program. Definition 1 introduces the LoC probability which is similar to branch probabilities in probabilistic symbolic execution.

Definition 1. ***LoC probability*** *is the probability of using a certain line of code over all possible inputs*

$$LoCPr(P) = \frac{\sum_{i=0}^{i=\#passingP}}{\sum_{i=0}^{i=\#possibleInputs}}$$

.

In order to be able to compare the program flow of two programs the path flow of every program has to be determined. With the defined LoC probability, the probability of a line can be calculated. To be able to compare the difference in terms of the probability considering a whole program a new metric is introduced.

Definition 2. ***Path Probability Delta(PPD)*** *is the difference of the LoC probability of every line of a code between two similar programs. The smaller this value is, the similar are the program flows of these two programs.*

As proposed in this thesis the new defined path probability delta metric can be used as a new metric in the area of test case based software repair. Valid repair candidate which satisfies the most important metric, passing all available test cases, can be ranked according to their PPD. The PPD should be calculated between the original program and every repaired version based on created repair candidates. The candidate with the lowest PPD is the repair candidate which affects the least of all possible scenarios which are not covered by the given test cases.

By calculating the PPD metric of the original snippet and the snippet modified by SemFix as well as the original snippet and Nopol, the question which repair approach generates the better repair can be answered by comparing the PPD of both repairs. The automatic software repair approach with the lower PPD creates the better fix for the Tcas snippet.

3.2 Measurement of the Path Probability Delta

After having the PPD defined as a new metric in comparing the program flow of a original code and a modification of the original code, this subsection describes how this new metric can be measured before the question about the best repair candidate can be answered.
In order to calculate the PPD of a repair, the probability of reaching every line of the modified and the original code has to be compared. To calculate the metric, the LoC probability of every line has to be measured. In this thesis we assume that every possible input has the same probability, all possible inputs have the same weight.

$$PPD = \sum_{i=0}^{i=\#lines} LoCPr_{original}(i) - LoCPr_{modified(i)}$$

Figure 3.1. First draft of calculating path probability delta

The calculation of the PPD is shown in Figure 3.1. For every line of the code within the original project the LoC probability is measured and compared with the LoC probability of the same line measured within the modified program. Every difference between these LoC probabilities are summed up to the PPD. The smaller this sum is, the smaller is the difference of the overall behaviour of the compared programs.

3.2.1 Reducing the Amount of Measurement Points

An analysis of the measurement of the PPD accorfing to Figure 3.1 shows several problems which are discussed and resolved within this section.

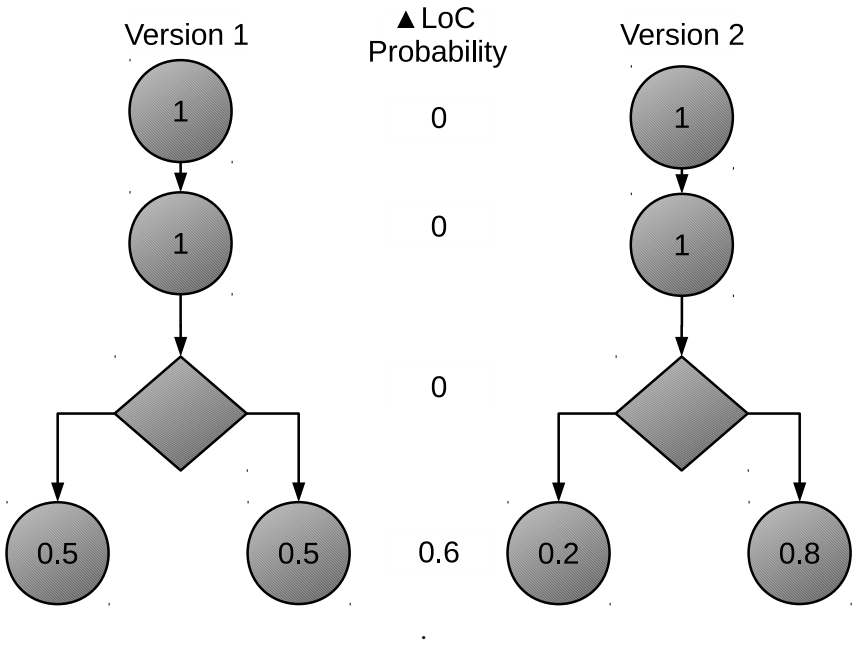

Figure 3.2. Measurement of LoC Probability

By looking into the LoC probabilities of every line of a program it seems that most of the LoC probabilities are the same. Only after an element which changes the program flow, like

a branch condition, the probability is changed. This is shown in Figure 3.2, where program flow diagrams of a slightly different programs are shown. The LoC probability only changes in that example after the branch condition. Only the branch condition changes the LoC probability. An improvement of measuring the PPD would be to set only a measuring point after each statements which change the LoC probability. A formula which describes an improved calculation of the PPD is shown in Figure 3.3 This improvement speeds up the measurement and calculation of the PPD.

$$PPD = \sum_{i=0}^{i=\#Branches} |LoCPr_{original}(i) - LoCPr_{repair}(i)|$$

Figure 3.3. Improved calculation of the PPD

An other problem which is addressed by the improved calculation of the PPD is that it prevents different weight of branches through a different amount of measurement points. The formula as in Figure 3.1 uses the LoC probability of every line of a program to calculate the PPD. Different amount of measurement points can occur, if for example the *then* and *else* branch of an if statement contains a different amount of statements. Figure 3.4 shows a similar program as shown in Figure 3.2. The only different is that the left branch contains a additional measurement point. In both variations, the probability for reaching the left branch is reduced by 0.3, and the probability of reach the other branch's probability enhanced by 0.3. But even if the probability of reaching the branches is the same for both examples the resulting PPD is different. The measurement with the old formula would result into in PPD of 0.6 for the first example and 0.9 for the second example. These unequal PPDs are caused by the different amount of measurement points within the different branches. The branches in Figure 3.2 consist out of one statement and therefore the LoC probability is measured once within every branch. The left branch of the second example consists out of two statements. Therefore the LoC probability gets measured twice within this branch. This results into an unequal weight of the different branches. By measuring the LoC probability only once within every branch, every branch has the same amount of LoC measurements and therefore has the same weight.

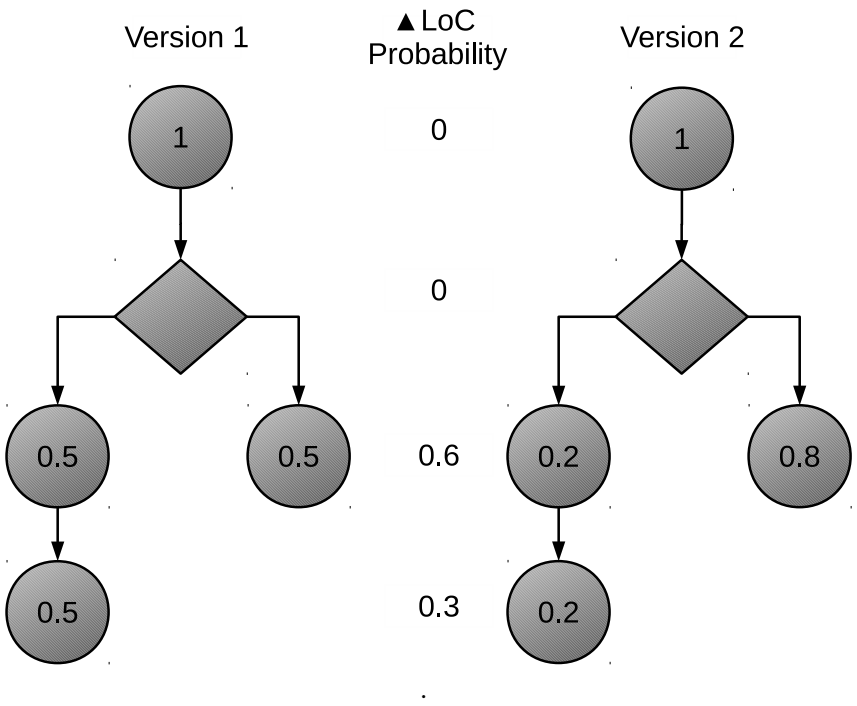

Figure 3.4. Different amount of measurement points in different branches

3.2.2 Dealing with nested if statements

With this improved equation it is possible to calculate the PPD in most of the cases. One remaining problem is similar to the problem of different weighted paths as discussed before. Nested if statements lead to the problem, that some branches could include more measurements of the LoC probability compared to other branches. This leads to an unequal weight of branches. On the left side of Figure 3.5 a program with an if statement which includes an *then*, an *else if* and an *else* branch is shown. The *else if* branch consists out of a nested if statement. On the left side of Figure 3.6 the same program is shown. The difference of the two figures is that the probability of executing the branches in version 2. In Figure 3.5 the probability of reaching the branch with the nested if is not changed, but the the probability of reaching the other two branches is changed. The changed probability of version 2 of Figure 3.6 is similar. But instead of changing the probability of the first branch, the branch of reaching the nested if branch is changed. In both variations, the probability of one branch is reduced by $\frac{1}{6}$ and an other branch's probability is enhanced by $\frac{1}{6}$. The third branch's probability stays the same. The PPD of the two programs in Figure 3.5 is $\frac{2}{6}$. The second mutation changed the probability of using the branch which includes the

nested if statement. As result the PPD is $\frac{3}{6}$, but should be the same as the previous PPD. This is caused by a three time measurement of the LoC probability within the branch with the nested if statement. Since the LoC probability is additional measurement at the two branches of the nested if statement the branch consists out of two additional measurement points. Each of the other branches contains only one measurement point each. This leads to an unequal weight of the different branches.The solution of this problem is, only to measure the LoC probability on the innermost if, $ifelse$ or $else$ branches if an if statement is nested. This solves the problem of different weighted paths and again saves costs trough measure and calculate less LoC probabilities.

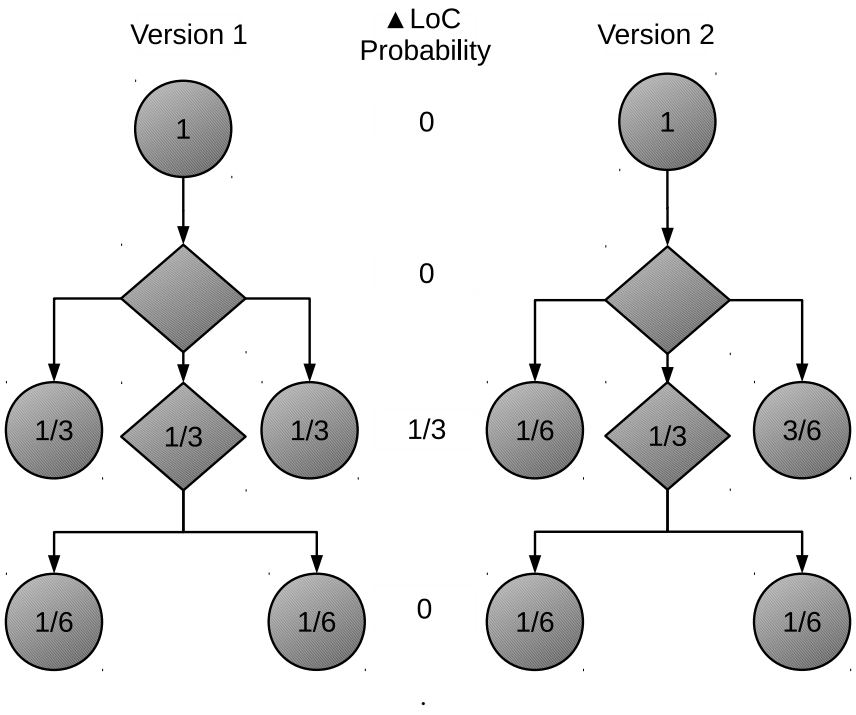

Figure 3.5. First version of the nested if statements

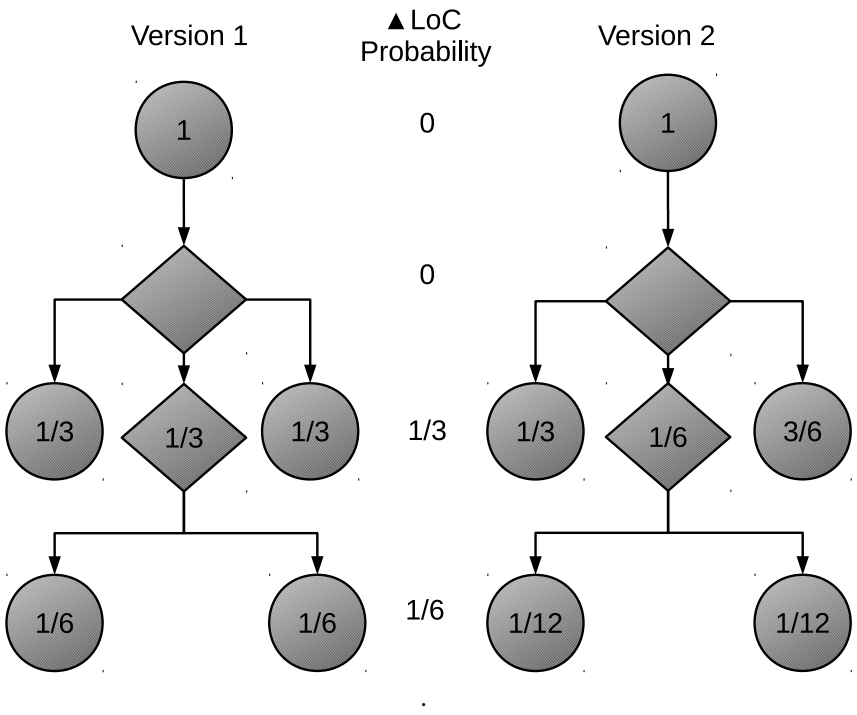

Figure 3.6. Second version of the nested if statements

$$PPD = \sum_{i=0}^{i=\#InnerBranches} |LoCPr_{original}(i) - LoC_{modificated}(i)|$$

Figure 3.7. Final calculation of the PPD

After applying these adjustments on the formula defined before in Figure 3.1 changes to the formula shown in Figure 3.7. Only the LoC probability of the branches are measured and compared between the original program and the slightly different program. If there are nested branches in the program, only the branches of the innermost branches are measured. With these improvements the PPD can calculated correctly and the calculation effort which is needed is reduced compared to the first draft of this formula.

$$normPPD = \frac{\sum_{i=0}^{i=\#InnerBranches} |LoCPr_{original}(i) - LoCPr_{modificated}(i)|}{\#InnerBranches}$$

Figure 3.8. Normalized PPD

In order to be able to compare the PPD between different projects the metric has to be normalized. Right now the metric depends on the number of measurement points. The more measurement points are within the analysed project, the higher is the PPD. As shown in Figure 3.8 the PPD is normalized by dividing trough the amount of measurement points.

3.3 Applying the Path Probability Delta on the Tcas Snippet

In this section the above defined new metric, the PPD, is applied on the introducing Tcas snipped example. Finally the repairs suggested by Nopol and SemFix are ranked after the PPD and it is discussed which repair is the better one. Listing 3.2 shows the original Tcas snippet. The if condition has been changed to a path condition, which shows in detail at which input the condition is fulfilled. For every line which is required to calculate the PPD, the LoC probability is shown in Listing 3.2. The range for integer is between $-2147483648 \leqslant x \leqslant 2147483647$. Boolean can set to either to *true* or $false$.

```
1  int is_upward_preferred(boolean inhibit,int up_sep,int down_sep) {
2          int bias;
3          if(inhibit)
4                  bias = down_sep; 50%
5          else
6                  bias = up_sep; 50%
7          if(inhibit ∧ down_sep > down_sep ∨ ¬bias ∧ up_sep > down_sep)
8                  return 1; 24.999999997089616954326629638671875%
9          else
10                 return 0;} 75.000000029103830456733703613281 25%
```

Listing 3.2. Tcas snippet with LoC probability

An analysis of the snippet with focus on the LoC probability shows that the first branch condition separates the LoC probability for line 5 and line 7 to 50% and 50%, since the branch condition can be set to either true or false. The path condition of the second if clauses is more complicated. The first part of the condition $a \wedge c > c$ is not satisfiable, because c can not be smaller than c. The second part $\neg inhibit \wedge up\_sep > down\_sep$ is satisfiable. The fist part of this formula $\neg inhibit$ is to 50% satisfiable because *inhibit* is

a boolean. The second part of the formula $up\_sep > down\_sep$ is with probability of 49.99999999417923390865325927734374% satisfiable. This leads to a probability of $0.5 * 0.49999999994179233908653259277343 74 = 24.999999997089616954326629638671 87\%$.The else branch is executed with a probability of 75.000000002910383045673370361328125%. These LoC probabilities are used to calculate the PPD of the Tcas snippet repaired by SemFix and Nopol.

```
1  int is_upward_preferred(boolean inhibit,int up_sep,int down_sep) {
2
3          int bias;
4          if(inhibit)
5                  bias=up_sep+100; [50%]
6          else
7                  bias = up_sep; [50%]
8          if(inhibit ∧ up_sep + 100 > down_sep ∨ ¬inhibit ∧ up_sep > down_sep)
9                  return 1; [ 49.99999999417923390865325927734374 55%]
10         else
11                 return 0;} [ 50.00000000582076609134674072265625 45%]
```

Listing 3.3. Tcas snippet repaired by SemFix

In Listing 3.3 the Tcas snippet with the repair done by SemFix is shown. Like in the previous listing, the probabilities of using the if and *else* branches as well as the path condition is shown in detailed. Since the change made by SemFix is done in line 5, the first if clauses and the LoC probability of line 5 and line 7 are not affected by the manipulation. The condition of the second if clause changes after applying the repair. To enter the *then* branch the condition $a \wedge b + 100 > c \vee \neg a \wedge b > c$ has to be fulfilled. Because b is an integer with the range of$-2147483648 <= x <= 2147483647$, b+100 can be seen as b, since 100 can be negligible. This results to a condition of $a \wedge b > c \vee \neg a \wedge b > c => b > c$. This leads to 50% probability of entering the then branch what result to a LoC probability of 49.99999999417923390865325% for line 9 and 50.000000005820766091% for line 11.

$$PPD = |0.5 - 0.5| + |0.5 - 0.5| + |0.24999 - 0.4999| + |0.75000 - 0.50000| = 0.49999$$

Figure 3.9. Calculating the PPD for SemFix's repair

The PPD between the original program and the program repaired by SemFix is 0.49999. As shown in Figure 3.9 the results are rounded to guarantee a better readability . This result has to be compared to the PPD of the repair created by Nopol. This allows to compare the two repair candidates of both approaches and answer the question which repair effects the less of all possible scenario not covered by the test cases and is the better repair.

```
1  int is_upward_preferred(boolean a,int b,int c) {
2
3          int bias;
4          if(a)
5                  bias = c;  //50%
6          else
7                  bias = b; //50%
8          if(b! = 0)
9                  return 1; //99.99999997671693563461303710937 5%
10         else
11                 return 0;} //2.32830643653869628906251 0^-8%
```

Listing 3.4. Tcas snippet repaired by Nopol

In Listing 3.4 the Tcas snippet repaired by Nopol is shown. Again with the results of the LoC probability for every line which is needed to calculate the PPD. Since the change which is made by Nopol is done in line 8, the LoC properties of line 5 and line 7 are not affected by the manipulation and stays with 50% each. The second if condition is changed by Nopol in order to repair the Tcas snippet. The path condition which has to be satisfied to enter the if clause is $b! = 0$. This results in massive changes of the LoC probability of line 9 and line 11. The LoC probability of line 9 is 99.99999976716935634613037109375%. This leaves the LoC probability of Line 11 to $2.3283064365386962890625 \times 10^{-}8\%$.

$$PPD = |0,5 - 0,5| + |0,5 - 0,5| + |0.24999 - 0.99999| + |0.75000 - 0.00001| = 1.49999$$

Figure 3.10. Calculating the PPD of the repair done by Nopol

The PPD of the original snippet and the snippet repaired by Nopol is 1.49999. The calcualtion in Figure 3.10 uses rounded numbers to provide a better readability.

By calculating PPD of the repair done by SemFix and the repair done by Nopol the question which repair is better can be answered. Comparing the PPD of the SemFix repair shows that Nopol has a PPD of 0.5, and the snippet manipulated by Nopol has a PPD of 1.49999. SemFix repair results in a smaller PPD. Therefore the manipulation done by SemFix has a smaller impact in changing the behaviour of all possible scenarios compared to Nopol. This leads to the answer that the repair done by SemFix is the better repair in terms of fixing the Tcas snippet.

Chapter 4

Implementing Improved Automatic Software Repair Approach

One scenario where the PPD metric could be used is improving the quality of patches created by automatic software repair approaches. In this chapter Nopol, an automatic software repair approach is extended with the PPD metric. Goal of this improvement is to improve the quality of created repairs. The repairs created with the new approach should affect the behaviour of scenarios which are not covered by the test suite the less as possible. The extension of Nopol which is developed in this thesis is called ProRepair.

4.1 ProRepair

ProRepair is an extension for Nopol and uses probabilistic symbolic execution in order to find a patch which changes the behaviour of the code as less as possible. The new extension consists of several components. An overview is given in Figure 4.1. In order to be able to create repairs for an error prone project the automatic software repair tool Nopol is used. The reasons why Nopol is used are discussed in Chapter 2.1.4. Nopol has several different modi for fixing bugs implemented. The mode which is used and adapted within ProRepair, is the technique of manipulating branch conditions to pass all test cases and create a repair. In order to be able to create repairs, Nopol needs a the project which is to be repaired and on test cases. Some of these test cases should not pass, otherwise no error can be found and no repair can be created.
The adapted Nopol creates several repair candidates. Everyone of these repair candidates leads to a repaired program which let all test cases of the test suite pass. In order to choose the candidate which affects scenarios which are not covered by the test suite the less as possible the PPD of all repair candidate is calculated. For such a calculation the LoC probability of certain points in the project is needed. To measure the LoC probabilities, probabilistic symbolic execution used. With the LoC probabilities of certain points of the original code and of the same points in the code repaired by the repair candidates the PPD for every repair candidate is calculated. According to the results of their PPD the repair candidates are ranked. The candidate with the lowest PPD is considered the best repair and chosen as fix for the error prone project.

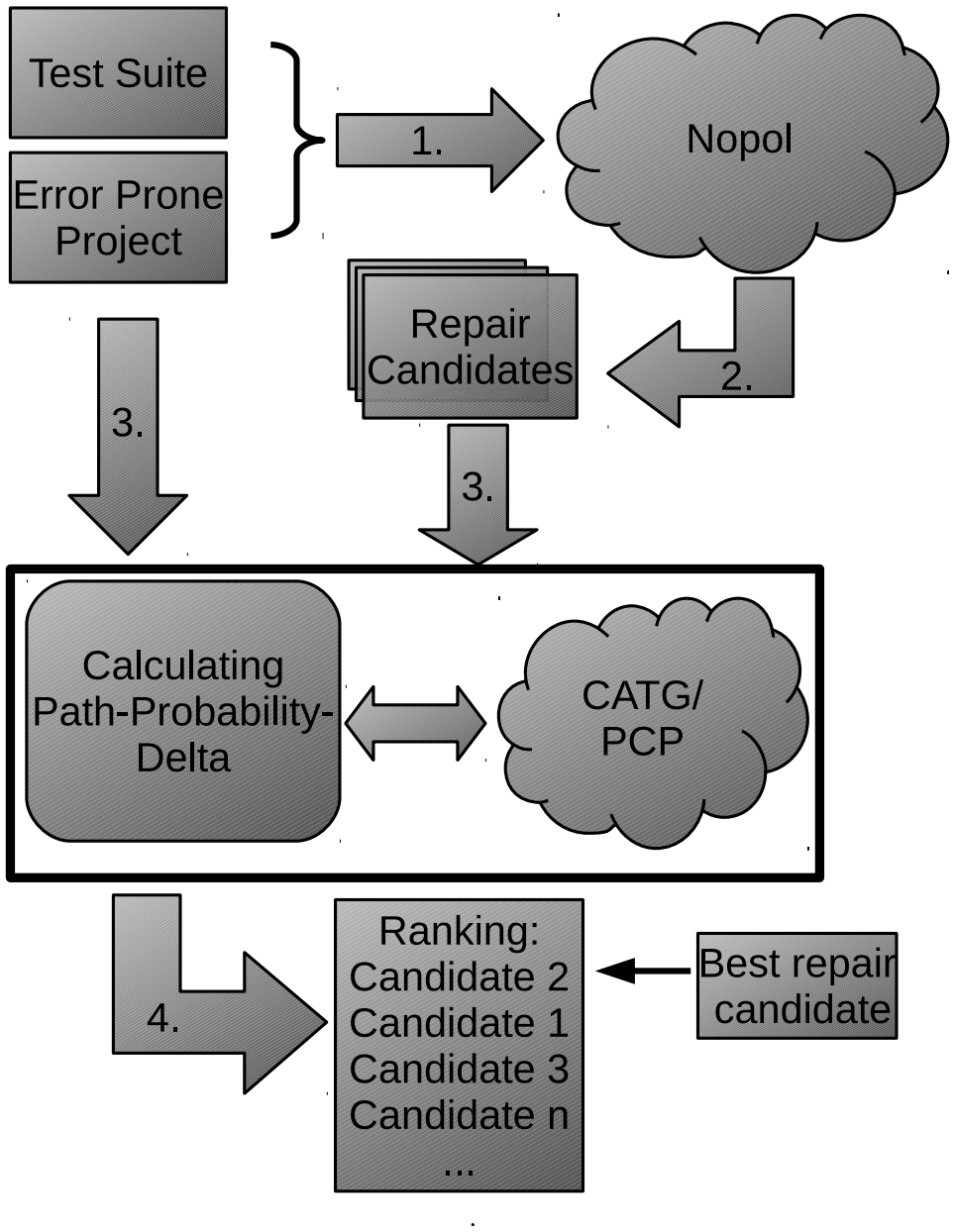

Figure 4.1. Overview ProRepair

4.2 Adapting Nopol

Nopol is used as foundation for ProRepair to generate automatic repairs from an error prone project under the consideration of given test cases. In order to be able to benefit from probabilistic symbolic execution and get the best possible repair according to the PPD some adjustments within Nopol have been done.
The main adjustment which has been done within the mechanisms of Nopol is enhancing the amount of created repairs. Nopol creates only one repair and uses this repair to fix a bug. Since ProRepair needs multiple repair candidates to rank these candidates after their PPD, the amount of created repair candidates is enhanced. A high amount of repair candidates ensures a better repair since more valid candidates are analysed according to their PPD. On the other hand the creation of more repair candidates leads to a higher effort in calculation these candidates. Therefore a trade-off has to be done to get the best number of repair candidates. The amount of created repairs can be assigned manually.

Nopol creates the repair candidates by calculating how to replace a branch condition by using a SMT solver. It could happen that the SMT solver crates solutions which results to branch conditions which are syntactically equal. Syntactically duplicate branch conditions are filtered by ProRepair. This could result into a lower amount of created repair. Solutions which are syntactically different but are logical the same are not filtered yet.
After creating those repair candidates the LoC probability of every branch is calculated and the PPD between repaired projects with these candidates and the original project is calculated, before choosing the repair with the best quality and finally creating a repair.

4.3 Measurement of LoC Probability and Calculation of the Path Probability Delta

As defined in Definition 1 the LoC probability is the probability of a line to get passed during all possible executions of a method. To measure the LoC probability probabilistic symbolic execution is used. The symbolic probabilistic analysis is executing within ProRepeir through a combination of CATG and PCP. CATG is used to cover all possible paths, PCP to calculate the probability of every possible path.

As explained in Chapter 2.2.2 some preparation is needed to be able to instrument CATG. In order to be able to start such an analyses a main method within the class of the analysed method has to be added. As shown in Listing 4.1 such a main method has to include a method call of the method which is the starting point of the symbolic analysis and all input parameter of this method marked as symbolic parameter. The main method is automatically injected by ProRepair into the class during execution. The SPOON library is used to insert the main method into the project. At this development state, an other main function is not allowed within the class of the tested function.

```
1 public static void main(String[] args){
2 System.out.println("$$TARGET$$");
3 int a= catg.CATG.readInt(0);
4 functioncall(a); }
```

Listing 4.1. Preperation of symbolic analysis

After running a probabilistic symbolic execution on the project, the probability of every possible path is given as shown in Listing 4.2.

```
1 [quantolic] domain coverage for this path: 0.75000000005820766 at 1459368093571
2 [quantolic] domain coverage for this path: 0.24999999994179234 at 1459368094793
```

Listing 4.2. Output without measurement points

The output of CATG/PCP contains the probability of every path, but does not contain information about the probability of reaching a certain line of code. In order to be able to get the probability of reaching a certain line of code some adjustment have to be made in the analysed project. ProRepair insert at every line where the LoC probability is measured a measurement. The measurement point indicates if the specific line of code is part of the path which is covered by a probabilistic symbolic analyses. A measurement point is a print method which prints a specific ID which appears in the output of the analyses if the measurement point is passed during the execution of a path. To generate an unique ID, the ID consists of the identifier $$P : which contains the information that this ID is part of a measurement point, the class name and the line which is to be analysed via probabilistic symbolic execution. An example of such a measurement point is shown in listing 4.3. Whenever such a measurement point is covered during probabilistic symbolic execution this ID is shown in the results of CATG/PCP. The SPOON library is used to insert such a measurement point.

Measurement points are inserted at every branch of the project, considering some exception which a discussed in Chapter 3. The project is scanned with the SPOON library and scanned after each if respectively else statement a measurement point is inserted.

```
1 if()
2 System.out.println("$$P:TestClass6") ;
3 else
4 System.out.println("$$P:TestClass8") ;
```

Listing 4.3. Inserting measurement points in the program code

Whenever a measurement point is passed during the probabilistic symbolic execution the ID of the measurement point occurs at the output of the analyses. To get the overall probability of passing a specific measurement point, all results where the ID occurs have to be summed up. An example of a results which include measurement points is shown

in listing 4.4. In this example two measurement points are recognized during execution. The measurement point with the ID $$*P* : *TestClass*8 occurs at all discovered branches during probabilistic symbolic execution and has a probability of 100%. The measurement point $$*P* : *TestClass*6 only occurs in the second execution which has a probability of 75.000000005820766%

```
1 $$P:TestClass6
2 $$P:TestClass8
3 [quantolic] domain coverage for this path: 0.75000000005820766 at 1459368093571
4 $$P:TestClass8
5 [quantolic] domain coverage for this path: 0.24999999994179234 at 1459368094793
```

Listing 4.4. Output with measurment points

It could happen that an if statement does not contain an else branch like shown in Listing 4.5. With the technique presented above the probability of the else branch would not be covered. To be able to cover the probability of the missing else branch as well an else branch is added to the if statement. In order to add a else statement which includes an measurement point the SPOON framework is used.

```
1  if {
2          if {
3          $$P:TestClass6
4          }
5          new : else{
6          $$P:TestClass6
7          }
8          }
9  else{
10 $$P:TestClass10
11 }
```

Listing 4.5. Inserting else branch

4.4 Limitations

The extension of Nopol, ProRepair, has some limitations which are mostly caused by Nopol, CATG and PCP.

Nopol is the automatic software repair approach which is used within ProRepair. All limitations which apply to Nopol are also limitations of our approach.

- Nopol can only repair bugs by changing one line of code. Repairs over multiple lines are not possible. Other automatic software repair approaches like GenProg are able

to change through the generic programming approach to manipulate multiple lines of code,

- ▻ Condition which include a method which has one or more input parameters as can not be repaired with Nopol.
- ▻ Limitation are caused by the quality of test cases which are used to find the course of a bug and evaluate a repair candidate. Like all automatic software repair tools which are based on test cases, Nopol is probably not able to repair all bugs in a code. As described earlier, Nopol tries to make all test cases pass. If Nopol is successful, all test cases pass. But this does not necessary imply that the code Nopol was applied is error free. Therefore Nopol's power is dependent on the quality of test coverage.

Besides the limitations of Nopol, there are also some limitations caused by CATG/PCP

- ▻ The internal counting is done with integer values. Therefore double values can not be used for the counting engine.
- ▻ Because of the same reasons as the limitation of double, character and string data types are not supported by the counting engine.
- ▻ An other limitation is that the modulo operator is not supported by PCP.

Limitations which have been added during the development of ProRepair:

- ▻ Ternary operator like $x?y : z$ can not be recognized as a branch condition. To be recognized as a branch condition, it has to be changed to a regular Java if/else statement as shown in Listing 4.6.

```
1 if (x)
2         return y;
3 else
4         return z;
```

Listing 4.6. Acceptable if statement

- ▻ Within the class of the function which is to be repaired should not contain a main function.

Chapter 5

Evaluation

In this chapter the evaluation of ProRepair and the newly defined PPD metric is described. As shown in chapter 4.4 limitations of ProRepair, currently an evaluation against real world programs is difficult. The best way to evaluate ProRepair would be using the bugs of the Defects4J framework which have been fixed by Nopol. Nopol is able to repair 34 bugs, with four repairs being correct [Durieux et al., 2015]. Unfortunately these results can not be used with ProRepair mainly due to the limitation of PCP not being able to count double values. In order to evaluate the correctness of ProRepair the example of Chapter 3.3 is used. Two case studies have been created in order to show that ProRepair rank repair candidates according to the PPD and choices the candidate which violate the behaviour of the smallest amount of possible scenarios. Finally within a real project, errors are injected and repaired by ProRepair.

5.1 Evaluation of the Correct Calculation of the Path Probability Delta by ProRepair

The key novelty of ProRepair is ranking repair candidates after their PPD to determine the repair candidate with the best quality. In order to evaluate the correct measurement of LoC probability and the correct calculation of the PPD the Tcas snippet which is know from chapter 3.3 is repaired with ProRepair. The goal of this evaluation is to investigate whether the measurement of the LoC probability of every important branch is done correctly as well as the correct calculation of the PPD.

The project and the test case of the Tcas snippet which are used within this evaluation are the same as shown in Chapter 3.3. The repairs which have been found with ProRepair $((0) != (up_sep))$ and $((up_sep) != (0))$. Trough the strictness of the given test cases ProRepair is not able to create more than these two repair candidates. The repair candidates are obviously not semantically but logically equal.

Listing 5.1 shows the instrumented Tcas snippet which had done been prepare by ProRepair for the symbolic analyses. Within every *if* and *else* branch a marker has been added. This is needed to calculate the LoC probability at this line of the code. Also a main method consisting of a function call and input parameters prepared for symbolic analysis

have been added within the instrumented Tcas snippet.

```
1  public static void main(String[] args){
2
3  boolean inhibit0 = inhibit0= janala.Main.readBool(true);
4  janala.Main.MakeSymbolicBool(inhibit0);
5  int up_sep1 = up_sep1= catg.CATG.readInt(0);
6  int down_sep2 = down_sep2= catg.CATG.readInt(0);
7  is_upward_preferred(inhibit0, up_sep1, down_sep2);
8   }
9      static int is_upward_preferred(boolean inhibit, int up_sep, int down_sep) {
10         int bias;
11         if (inhibit) {
12 System.out.println("$$P: 10") ;
13             bias = up_sep;
14         } else {
15 System.out.println("$$P: 12") ;
16             bias = up_sep;
17         }
18         if ((0) != (up_sep)) {
19 System.out.println("$$P: 14") ;
20             return 1;
21         } else {
22 System.out.println("$$P: 16") ;
23             return 0;
24         }
25     }
```

Listing 5.1. Instrumented Tcas snippet

The results which are produced by ProRepair are shown in Table 5.1. These results are compared with the results calculated in chapter 3 are shown in Table 5.2. Within both calculation the range of integers was set to $-2147483648 \leqslant x \leqslant 2147483647$ and the range for boolean is $true, false$. The results shown in both tables are almost identical. The small difference of these results is caused by a different amount of floating points. If the results of Chapter 3.3 are rounded up to the next floating point, the results are the same. The LoC probability of all four measured branches are identical compared to the results reported by ProRepair. Also the calculation of the PPD is the same. With these results the correctness of the measurement of the LoC probability as well as the correctness of calculating the PPD is shown.

Table 5.1. Results of Tcas Snippet calculated by ProRepair

Results by ProRepair			
LoC	**original**	**repaired**	**delta**
10	0.5	0.5	0
12	0.5	0.5	0
14	0.2499999999417923	0.9999999997671694	0.7499999998253771
16	0.75000000005820766	2.328306E-10	0.74999999982537706
PPD:			**1.49999999965075416**

Table 5.2. Results of Tcas Snippet from chapter 3

Results calculated manualy			
LoC	**original**	**repaired**	**delta**
10	0.5	0.5	0
12	0.5	0.5	0
14	0.24999999994179234	0.99999999976716935	0.74999999982537701
16	0.750000000058207661	2.3283065E-10	0.74999999982537701
PPD:			**1.49999999965075402**

5.2 Choosing the Best Repair Candidate

In the previous section the correctness of the calculation of ProRepair has been evaluated. In this section ProRepair choice of the best repair candidate according to the PPD is discussed. For this evaluation projects of the benchmark IntroClass [] are used. IntroClass is a collection of projects which are used to evaluate automatic program repair approaches for C written code. For the evaluation of ProRepair some projects of the benchmark are transformed to Java to be able to get repaired by ProRepair.

5.2.1 Repairing: Finding the Smallest Number

The first project, shown in Listing 5.2 which is taken from IntroClass [Le Goues et al., 2015] is a small function which searches the smallest number out of four integers. The test cases which are used for the repair process are also taken from IntroClass are shown in Table 5.3.

```
1         public static int smallest(int A, int B, int C, int D)
2 {
3         int smallest = A;
4         if (A > B)
5                 smallest = B;
6         if (B > C)
7                 smallest = C;
8         if (C > D)
9                 smallest = D;
10        else if (A > D)
11                smallest = D;
12        return smallest;
13                }
14    }
```

Listing 5.2. Choosing the smallest integer

Table 5.3. Test cases for program smallest

#	IN				OUT	
	A	**B**	**C**	**D**	**Expected**	**Observed**
1	0	0	0	0	0	0
2	0	0	1	2	0	0
3	0	0	1	0	0	0
4	0	0	3	1	0	1
5	0	1	0	0	0	0
6	0	1	1	1	0	0
7	0	1	1	0	0	0

The amount of created repair candidate is set to eight. During the repair process line 8 of the median function is manipulated in order to repair the bug. The eight created repair candidates are shown in Table 5.4. Besides the repair candidate the normalized PPD of every candidate is shown in this table.

Table 5.4. Repair candidates of program smallest and their PPD

#	**repair candidate line**	**normalized PPD**
1	(C >D) && ((D == -1) || (!((C) != (1))))	0.14283839366078124
2	(C >D) && (!((1 == D) || (1 == D)))	7.141428696451545E-6
3	(1 == -1) || ((C >D) && (C <= 1))	0.1071285718749911
4	**(!(D == 1)) && (C >D)**	**7.141428696439649E-6**
5	(C >D) && (!(1 <= D))	0.03570714348209823
6	(smallest == D) && (C >D)	0.14284285785712503
7	(A <C) && (D == smallest)	0.14284285821421427
8	!((C <= D) || (((1) != (C)) && (C >D)))	0.1428428575

The best repair which is chosen by ProRepair out of the eight suggested repair candidates is $((b > c))\&\&(b == 0))||(((b > a)\&\&(b < c))||((b < a)\&\&(b > c))$. This repair candidate has the lowest PPD compared to the other repair candidates. This repair candidate should affect the lowest amount of scenarios which are not covered by the test cases shown in Table 5.3. The other repair candidates except repair candidate number two which has a similar PPD have a significant higher normalized PPD.
In order to evaluate that the chosen candidate has the lowest effect of scenarios which are not covered by the seven test cases, 5000 random test cases have been created. All these test cases pass for the original function. The goal of using the PPD metric as a quality metric is to ensure that the behaviour of less as possible scenarios are affected by a repair. The repair which is chosen by ProRepair should pass more test cases than the other

repair candidates would. The result of this experiment is shown in Table 5.5. The repair candidate 4 which is chosen by ProRepair validates none of the 5000 randomly created test cases. This is obviously the smallest amount of validation within the 8 repair candidates. Repair candidate 2 has the same amount of failing test cases as repair candidate 4, because the difference of the PPD of those two repair candidates is very small. Also the repair candidates 1, 6, 7, 8 have the same amount of failing test cases due to their small differences of their PPD.

Table 5.5. Comparing the results of table 5.4 with 1000 random scenarios

#	failing random generated test cases	normalized PPD
1	841	0.14283839366078124
2	0	7.141428696451545E-6
3	744	0.1071285718749911
4	**25**	**7.141428696439649E-6**
5	440	0.03570714348209823
6	841	0.14284285785712503
7	841	0.14284285821421427
8	841	0.1428428575

5.2.2 Repairing: Finding the Median

The second example is taken from the same benchmark as the function median. The function searches the median out of three integers. The function shown in Listing 5.3 is taken from the same framework like the example above. Most of the test cases are taken from the benchmark InnerClass. In order to get a better test case coverage, some test cases are added for this evaluation to the test cases provided by InnerClass. The whole test suite is shown in Table 5.6. Applying these test cases on this function show that some of these test cases fail. This leads to the assumption that the median function is not error free.

```
1  public static int median (int a, int b, int c)
2  {        if(((a>b)&&(a<c))||((a<b)&&(a>c)))
3                  return a;
4          else if(((b>a)&&(b<c))||((b<a)&&(b>c)))
5           return b;
6       else
7          return c;
8          }
9  }
```

Listing 5.3. Calculating the median of three integer

Table 5.6. Test cases for median program

#	IN			OUT	
	a	b	c	Expected	Observed
1	2	6	8	6	6
2	2	8	6	6	6
3	6	2	8	6	6
4	6	8	2	6	6
5	8	2	6	6	6
6	8	6	2	6	6
7	8	8	8	8	8
8	6	-6	2	2	2
9	0	0	-2	0	-2
10	700	0	600	600	600

The amount of the created repair candidate is set to eight. During the repair process of Nopol line 4 of the median function is manipulated in order to repair the bug. These crated repair candidates are shown in Table 5.7. Besides the repair candidate the PPD of every created repair candidate is shown in this table.

Table 5.7. Repair candidates of program median and their PPD

repair candidate	normalized PPD
(((b >a) && (b <c)) || ((b <a) && (b >c))) || (a == 0)	9.378280781305315E-6
((((b >a) && (b <c)) || ((b <a) && (b >c))) || (0 == a)) && (-1 <0)	9.378280781305315E-6
(((b >a) && (b <c)) || ((b <a) && (b >c))) || (!(1 <a))	0.12505156429648437
(a <1) || (((b >a) && (b <c)) || ((b <a) && (b >c)))	0.12504218601560935
(((b >c)) && (!((b >a)))) || (((b >a) && (b <c)) || ((b <a) && (b >c)))	1.8748125140577443E-5
(((b >c)) && (c <= 0)) || (((b >a) && (b <c)) || ((b <a) && (b >c)))	0.0625234361718906
(((b >c)) && (b == 0)) || (((b >a) && (b <c)) || ((b <a) && (b >c)))	**4.688202984381995E-6**
(((b >a) && (b <c)) || ((b <a) && (b >c))) || (a <= 0)	0.12504218601560935-5

The results of the calculation of the PPD in Table 5.7 show that the first two condition terms of every repair candidates are the same. Only the last terms are different. Out of the eight repair candidate the best repair candidate which is chosen by ProRepair is $((b > c))\&\&(b == 0))||(((b > a)\&\&(b < c))||((b < a)\&\&(b > c))$. This repair candidate has the lowest PPD compared to the other repair candidates. This repair candidate should affect the lowest amount of scenarios which are not covered by the test cases shown in Table 5.6.

In order to evaluate that the chosen candidate affects the less of scenarios which are not covered by the seven test cases with which the repair was created, 5000 random test cases have been created for the original function. The result of this experiment is shown in Table 5.8. The repair candidate 7 which is chosen by ProRepair violates only 2 test cases. This is the smallest amount of violations within the 8 repair candidates.

Table 5.8. Comparing the results of table 5.7 with 1000 random scenarios

#	failing random generated test cases	normalized PPD
1	0	1.1641532182693481E-10
2	0	1.1641532182693481E-10
3	792	0.1111569460413195
4	792	0.1111486097916528
5	0	1.66650001249824E-5
6	368	0.0555763877083472
7	**0**	**4.16729154171606E-6**
8	792	0.1111486097916528

In this section two small projects have been repaired with ProRepair. For every project eight repair candidate has been created. The best candidate have been chosen by ProRepair

according to PPD. It has been shown that the repair candidate with the lowest PPD changes the outcome of the lowest scenarios which there not covered by the test suite which had been used during the repairing process. This has been evaluate through random 5000 test cases which outcome were compared.

5.3 Repairing a Lager Version of TCAS

In order to evaluate ProRepair with a real life project, the whole project of Tcas is to be repaired with ProRepair. For this evaluation the version zero of Tcas has been taken from the SIR project. As first step some test cases are created. For creating test cases the Symbolic Java Pathfinder [Anand et al., 2007] has been used. Symbolic Java Pathfinder uses symbolic execution to find all possible paths through the analysed project. Instrumenting Java Pathfinder with Tcas, 69 different paths have been found. Out of these output we generate 69 test cases which cover every possible path of Tcas.

Obviously Tcas let all of these 69 test cases pass. In order to be able to repair Tcas, some errors has been inserted. The insertion of bugs has been done with the major mutation framework [Just, 2014]. Major manipulates condition and operator of a project. After applying Majors manipulation on Tcas, 141 different mutations has been produced, with one manipulation per mutation. All those manipulated versions are applied on ProRepair. Out of theses 141 versions ProRepair/Nopol was able to generate for 38 repairs. This is a coverage of 26.95%. The amount of generated repair candidates was set to eight. In this thesis two out of those 38 repairs are investigated in detail. The focus of the investigation will be the PPD and if using this metric helps to get the best possible repair in terms of the change the code.

First Example repair of a Tcas manipulation

The fist repair process which was investigated is the repair of a manipulation of setting a return value of a function always to false within Tcas. This results into the manipulation shown in Listing 5.4. By running the 69 test cases at the major manipulation tool manipulated code, 8 out of the 69 test cases fail.

```
1   private static boolean Non_Crossing_Biased_Descend() {
2       static int  alt_sep_test(){
3           boolean enabled, tcas_equipped, intent_not_known;
4           boolean need_upward_RA, need_downward_RA;
5           int alt_sep;
6
7           enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) &&
8            (Cur_Vertical_Sep > MAXALTDIFF);
9           tcas_equipped = Other_Capability == TCAS_TA;
10          intent_not_known = Two_of_Three_Reports_Valid && Other_RAC == NO_INTENT;
11
12          alt_sep = UNRESOLVED;
13
14          if (enabled && ((tcas_equipped && intent_not_known) ||
15           !tcas_equipped)){
16                      need_upward_RA = Non_Crossing_Biased_Climb() &&
17                       Own_Below_Threat();
18                      need_downward_RA = Non_Crossing_Biased_Descend() &&
19                       Own_Above_Threat();
20       Modification:  need_downward_RA = false;
21                      if (need_upward_RA && need_downward_RA)
22                          alt_sep = UNRESOLVED;
23                      else if (need_upward_RA)
24                          alt_sep = UPWARD_RA;
25                         else if(need_upward_RA)
26
27                          alt_sep = DOWNWARD_RA;
28                      else
29                          alt_sep = UNRESOLVED;
30          }
31
32          return alt_sep;
33      }
```

Listing 5.4. First repair for Tcas

The eight repair candidates are shown in Table 5.9. All of these repair candidates replaces the bold line in Listing 5.4. The PPD of all these results are small and quite similar. This shows that most all possible scenarios outcome stays the same for all repair candidates. The best repair candidate seems to be repair candidate 3. Unfortunately the results can not be evaluated like the case studies before. Like the evaluation before 5000 random test cases have been created which pass the original project. But for every repair candidate these

5000 random test cases pass, too. The same results by crating 10000 random test cases. Though to the fact that Tcas is a safety function which seldom gives an other return value as 0, almost all of the possible scenarios return 0.

Table 5.9. Repair candidates for first the mutation with their PPD

#	generated repair candidates	normalized PPD
1	(NO_INTENT == OLEV) || (!(((Up_Separation) != (Own_Tracked_Alt_Rate - Down_Separation + Own_Tracked_Alt_Rate - Down_Separation)) && (NOZCROSS <Own_Tracked_Alt_Rate - Down_Separation)))	0.1640624999450768
2	(!((Alt_Layer_Value) != (MINSEP + NOZCROSS - Up_Separation))) && (Positive_RA_Alt_Thresh!=null)	0.17013888901146826
3	**((Other_Capability == Alt_Layer_Value) || (!(Own_Tracked_Alt <= Alt_Layer_Value))) && ((Up_Separation - MINSEP + Own_Tracked_Alt) != (NOZCROSS))**	**0.16102430572421675**
4	(MINSEP == Up_Separation - NOZCROSS) || (1 <= -1)	0.17013888901146826
5	!(((0) != (Other_Tracked_Alt)) || (Up_Separation <NOZCROSS + MINSEP))	0.1631944446322301
6	(Cur_Vertical_Sep + NOZCROSS <= TCAS_TA) || (!((NOZCROSS) != (Up_Separation - MINSEP)))	0.17013888901146826
7	Up_Separation == NOZCROSS + MINSEP	0.17013888901146826
8	(High_Confidence) && ((Positive_RA_Alt_Thresh.length <= Own_Tracked_Alt) || (Up_Separation - OLEV + MINSEP == NOZCROSS))	0.1635199654844354

Second Example repair of a Tcas manipulation

The second repair process which is investigated is the repair of a manipulation of a branch condition to always true as shown in Listing 5.5. By running the 69 test cases at the major manipulation tool manipulated code, 33 out of the 69 test cases fail.

```
1   private static boolean Non_Crossing_Biased_Descend() {
2       static int  alt_sep_test(){
3           boolean enabled, tcas_equipped, intent_not_known;
4           boolean need_upward_RA, need_downward_RA;
5           int alt_sep;
6
7           enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) &&
8            (Cur_Vertical_Sep > MAXALTDIFF);
9           tcas_equipped = Other_Capability == TCAS_TA;
10          intent_not_known = Two_of_Three_Reports_Valid && Other_RAC == NO_INTENT;
11
12          alt_sep = UNRESOLVED;
13
14          if (enabled && ((tcas_equipped &&
15           intent_not_known) || !tcas_equipped)) {
16                      need_upward_RA = Non_Crossing_Biased_Climb() &&
17                       Own_Below_Threat();
18                      need_downward_RA = Non_Crossing_Biased_Descend() &&
19                       Own_Above_Threat();
20                      if (need_upward_RA && need_downward_RA)
21
22                          alt_sep = UNRESOLVED;
23                      else if (need_upward_RA)
24                          alt_sep = UPWARD_RA;
25                       else if(need_upward_RA)}
26      MANIPULATION: else if(true)
27                          alt_sep = DOWNWARD_RA;
28                      else
29                          alt_sep = UNRESOLVED;
30          }
31
32          return alt_sep;
33      }
```

Listing 5.5. Second repair of Tcas

The eight repair candidates which would replace the bold branch condition in Listing 5.5 are shown in Table 5.10. The PPD of all these results are very small and quite similar. This shows that most all possible scenarios outcome stays the same for all repair candidates. The best repair candidate seems to be repair candidate number 7 because this candidate leads to the smallest PPD. Like discussed during the evaluation of the first Tcas repair, these results can not be evaluated by creating a huge amount of random test cases. But

trough the evaluation of the PPD in Chapter 5.2 it can be expected that the results are correct.

Table 5.10. Repair candidates for first the mutation with their PPD

#	generated repair candidates	normalized PPD
1	(NOZCROSS == Up_Separation - MINSEP) || (!((Up_Separation - MINSEP + NOTPOSSIBLE) != (-1)))	0.16362847233217026
2	(Up_Separation == MINSEP) || (!((Up_Separation) != (MINSEP + NOZCROSS)))	0.16362847233217026
3	(Up_Separation == MINSEP) || ((!((Up_Separation) != (MINSEP + NOZCROSS))) && (!((Up_Separation) != (MINSEP + NOZCROSS))))	0.16362847233217026
4	(!((Up_Separation - MINSEP) != (NOZCROSS))) || (MINSEP == Up_Separation)	0.16362847233217026
5	(NOTPOSSIBLE <Up_Separation - MINSEP) && ((NOZCROSS == Up_Separation - MINSEP) || (Up_Separation <= MINSEP))	0.14973958199172768
6	((High_Confidence) && (!((NOZCROSS + OLEV - Up_Separation) != (MINSEP)))) || (MINSEP == Up_Separation)	0.16362847233904199
7	**((Up_Separation <= MINSEP) || (Up_Separation - MINSEP == NOZCROSS)) && ((Other_Tracked_Alt) != (Own_Tracked_Alt))**	**0.13031683892323714**
8	((!((NO_INTENT) != (NOZCROSS - Up_Separation + MINSEP))) && (((tcas_equipped && intent_not_known) || (!tcas_equipped)))) || (NOZCROSS - Up_Separation + MINSEP == NOZCROSS)	0.16362847233217026

5.4 Quality of Repair

The quality of the repairs which are created at the Tcas example of Chapter 5.3 seems to be at most of the times not meaningful. This is caused by the problem every test base automatic program repair approach has to face. The only information which can be used to create repairs are based on the knowledge available form test cases. The result of test based automatic software repair approaches is not a repaired project but a project where the given test cases pass. The approach which is proposed in this thesis does not affect the creating of repairs but the ranking of repair candidates. Therefore this issue can not be solved by neither the proposed PPD metric or the implemented extension of Nopol. With the PPD metric the quality in terms of affecting scenarios which are part of the test suite is improved. As a next step the quality in terms of meaningful fixes should be address to

improve automatic software repair.
The quality and amount of the test cases which are used during the automatic repair process affects directly the quality of the created repair candidates. A low test case coverage leads to a bad repair because only a low amount of scenarios are covered with such a test suite. A high amount of test cases leads to restrictions during the creation of repair candidates which are too strict. This problem was discovered in [Qi et al., 2015] and also observed during the the evaluation of ProRepair.
With the huge dependencies between the results of test based automatic software repair and the test cases which are used for finding the bug and evaluating the repair, a good choice of test cases is important. The use of other test cases could therefore lead to other results.

Chapter 6

Conclusions and Future Work

This chapter presents some concluding remarks on the results of this thesis and sketches some directions for future work.

6.1 Conclusions

The thesis started with an overview over state of the art automatic software repair approaches. An analysis of those approaches in terms of success rate and quality of the created repairs has been discussed. Neither of the two criteria is fully satisfied by current approaches.

Most of the current automatic software repair approaches use a given set of test cases as oracle to assess the correctness of the program. However, such test cases can provide, in general, only a partial specification of the program behavior. First, most of the possible execution paths of the program are usually uncovered; second, they do not provide enough information to quantify the impact of a repair on the behavior of the program, apart from checking whether or not all the tests pass.

In this thesis, Path Probability Delta (PPD), a new metric of the behavioral discrepancy between two software versions, is proposed in order to evaluate the impact a program repair has on all the possible execution paths of a program. PPD can be quantified using probabilistic symbolic execution. A repair candidate with a lower PPD is usually preferable than a repair candidate with a higher PPD because, besides passing all the tests, it has the minimal impact on the other behaviors of the the program, thus preserving its functionalities.

PPD is used to improve the test-based automatic software repair tool Nopol. ProRepair, the proposed extension of Nopol, uses PPD to choose the best repair candidate, therefore improving the quality of the created repair.

The evaluation performed for this thesis, showed that PPD is effective in quantifying the discrepancy of between the behaviors of the original program and the repaired one, allowing ProRepair to produce better repairs than Nopol. The cost of this improved quality is in terms of execution time, with ProRepair requiring the to quantify the PPD metric for all the repair candidates produced by Nopol.

6.2 Future Work

There are several aspect worth to be investigated in the future. The new defined path probability delta (PPD) metric could be used as a new aspect in benchmarking automatic software repair approach. This would help to evaluate the quality of the generated repairs.

This thesis implemented PPD quantification on top of Nopol. However, many more tools for test-based automatic software repair could be improved with the use of PPD. Furthermore, PPD can be used to define the fitness function employed by evolutionary approaches for automatic software repair, like GenProg. In this case, the tool may look directly for solutions that not only pass the available test cases but also induce the smallest behavioral discrepancy.

In its current version, PPD does not support the deletion of a branch (which is not an option for the repair candidates generated by Nopol). It would however be straightforward to extend it to handle deleted branches. For example, if a nested branch is deleted, some new measurement points can be inserted into the code and used to compare the behavior of the different (repaired) versions of the program.

ProRepair could be further improved by collecting repair candidates generated by more than one tool. For example, consider the Tcas snippet reported in Chapter 3. Nopol's repair approach is to change the condition $bias > down\_sep$ to $up\_sep! = 0$. As discussed is the repair created by SemFix is better compared to the Nopol repair. A better fix would be a condition that includes at least one parameter that is part of the original branch condition like $bias > up\_sep + 100$. This repair is similar to the SemFix repair and would affect less scenarios which are not covered by the test cases .

Finally, being based on probabilistic symbolic execution, the computation of PPD is currently limited by the ability of this technique of handling complex constraints. The development of probabilistic symbolic execution will directly extend the applicability of the proposed approach to more complex software.

Bibliography

[Anand et al. 2007] S. Anand, C. S. Păsăreanu, and W. Visser. Jpf–se: A symbolic execution extension to java pathfinder. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138. Springer, 2007.

[Arcuri 2011] A. Arcuri. Evolutionary repair of faulty software. *Applied Soft Computing*, 11 (4):3494–3514, 2011.

[Debroy and Wong 2010] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 65–74. IEEE, 2010.

[DeMarco et al. 2014] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, pages 30–39. ACM, 2014.

[Do et al. 2005] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[Durieux et al. 2015] T. Durieux, M. Martinez, M. Monperrus, R. Sommerard, and J. Xuan. Automatic repair of real bugs: An experience report on the defects4j dataset. *arXiv preprint arXiv:1505.07002*, 2015.

[Elkarablieh et al. 2007] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based repair of complex data structures. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 64–73. ACM, 2007.

[Filieri et al. 2013] A. Filieri, C. S. Păsăreanu, and W. Visser. Reliability analysis in symbolic pathfinder. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 622–631. IEEE Press, 2013.

[Filieri et al. 2015] A. Filieri, C. S. Păsăreanu, and G. Yang. Quantification of software changes through probabilistic symbolic execution. 2015.

[Forrest et al. 2009] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954. ACM, 2009.

[Geldenhuys et al. 2012] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 166–176. ACM, 2012.

[Goues et al. 2012] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1): 54–72, 2012.

[Just 2014] R. Just. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 433–436, San Jose, CA, USA, July 23–25 2014.

[Just et al. 2014] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[Kim et al. 2013] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.

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

[Le Goues et al. 2015] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *Software Engineering, IEEE Transactions on*, 41(12):1236–1256, 2015.

[Long and Rinard 2015] F. Long and M. Rinard. Staged program repair with condition synthesis. In *10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2015.

[Martinez and Monperrus 2014] M. Martinez and M. Monperrus. Astor: evolutionary automatic software repair for java. *arXiv preprint arXiv:1410.6651*, 2014.

[Mechtaev et al. 2015] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 448–458. IEEE, 2015.

[Monperrus 2015] M. Monperrus. Automatic software repair: a bibliography. 2015.

[Nguyen et al. 2013] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781. IEEE Press, 2013.

[Qi et al. 2014] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265. ACM, 2014.

[Qi et al. 2015] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. 2015.

[Riboira and Abreu 2010] A. Riboira and R. Abreu. The gzoltar project: A graphical debugger interface. In *Testing–Practice and Research Techniques*, pages 215–218. Springer, 2010.

[Seacord et al. 2003] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing legacy systems: software technologies, engineering processes, and business practices*. Addison-Wesley Professional, 2003.

[Tanno et al. 2015] H. Tanno, X. Zhang, T. Hoshino, and K. Sen. Tesma and catg: automated test generation tools for models of enterprise applications. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 717–720. IEEE Press, 2015.

[Wei et al. 2010] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72. ACM, 2010.

[Weimer et al. 2013] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 356–366. IEEE, 2013.

[Xuan et al. 2016] J. Xuan, M. Martinez, F. Demarco, M. Clément, S. Lamelas, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering*, 2016. URL `https://hal.archives-ouvertes.fr/hal-01285008`.

Decleration

I declare that this thesis is the solely effort of the author. I did not use any other sources and references than the listed ones. I have marked all contained direct or indirect statements from other sources as such. Neither this work nor significant parts of it were part of another review process. I did not publish this work partially or completely yet. The electronic copy is consistent with all submitted copies.

---

place, date, signature