Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master’s Thesis Nr. 32

Automatic Software Repair of C Programs with Genetic Programming and Fix Patterns

Manuel Borja

Course of Study:	Software Engineering
Examiner:	Prof. Dr. Lars Grunske
Supervisor:	Prof. Dr. Lars Grunske

Commenced:	2015/03/24
Completed:	2015/09/23
CR-Classification:	D.1.2, D.2.4, D.2.5, D.2.9, I.2.8

Abstract

Between 50% and 90% of the software costs are invested in repairing errors. The nature of this activity makes difficult its partial or complete automation through software-intensive systems. Nevertheless, a vast amount of research has been made over the last years in the pursuit of novel, effective, sound and scalable solutions towards the automation of software repair. Works like GenProg, PAR, ASTOR, JAFF, which use Genetic Programming to detect and repair software errors, demonstrate this fact. However, there are still a bunch of challenges to overcome in the field of Search-Based Software Engineering (SBSE) metaheuristics, group to which Genetic Programing belongs [Le Goues et al., 2013]. One of them is to improve the low quality of the software repairs. Since the basic idea of the Genetic Algorithms in automatic software repair is to evolve systematically a set of genotypes (i.e. programs) through a series of mutation operations until an optimal fix is found, nonsensical fixes can be presented as a solution.
Thus, generating meaningful and human-like repairs is an essential topic to cover in this area in order to improve the patch quality. Since human knowledge and expertise have played and play currently an important role in the bug fixing process, we must transport this intangibles to the automatic software repair process. In this work, similar to the work made by Kim [Kim et al., 2013] we extract this knowledge from different sources of information and structure them in a set of patterns which, along with the Genetic Programming operators, will transform buggy program systematically to find an optimal patch. To our knowledge, this is the first work which applies fix templates to repair C programs. At the end, we will demonstrate that the use of fix templates not only overcome other approaches to fix C programs, but the generated patches tend to be more similar that the ones made by a human.

Zusammenfassung

Zwischen 50% und 90% der Software-Kosten werden für Bug-Fixing ausgegeben. Aufgrund der Natur dieser Aktivität, erschwert sich die Möglichkeit, sie durch Software-Intensive Systeme durchzufüren. Nichtsdestotrotz ist die Recherche, um neu, effektiv, fundierte und skalierbare Methoden zu finden, in den letzten Jahren stark ausgestiegen. Ansätze wie GenProg, PAR, ASTOR und JAFF, die Genetic Programming verwenden, bestätigen diesen Fakt. Es gibt aber immer noch eine Menge von Herausforderungen bei der Search-Based Software Engineering (Such-bassierte Software Technik SBSE), Kategorie zu der Genetic Programming gehört [Le Goues et al., 2013]. Niedrige Qualität der Sofware Fixes ist eine von diesen Challenges. Da die grundsätzliche Idee genetischer Algorithmen bei der automatischen Software-Reparierung darin besteht, eine bestimmte Menge von Genotypen, also Programmen, durch verschiedene Mutationsoperatoren zu evolvieren bis eine optimalle Reparatur gefunden wird, es könnten unsinvolle Fixen vorkommen.
Es ist folgendermaßen notwendig, dass die Tools sinnvolle bzw. menschenähnliche Patches generieren, um die Qualität der Reparaturen zu erhöhen. Da menschliche Kenntnisse und Erfahrung hat gespielt bzw. spielt immer noch eine wichtige Rolle bei der Behebung von Software-Fehler, soll man sie in den Software-Reparatur Prozess inkludieren. In dieser Masterarbeit, analog zu dem Ansatz von Kim [Kim et al., 2013], extrahieren wir diese Kenntnise aus verschiedenen Quellen und struktutieren sie im Form von Fix-Muster die, neben den Genetic-Programming Operatoren, transformiren das fehlerhafte Programm, um einen optimalen Patch zu finden. Zu unserem Wissen ist diese die erste Arbeit, die Fix-Muster verwendet für die automatische Reparatur von C Programmen. Zum Schluss werden wir beweisen, nicht nur dass bei der Verwendung eine größere Menge von Bugs repariert werden könne, sondern auch dass solche Reparaturen menschenähnliche sind.

Contents

List of Tables

Listings

Chapter 1

Introduction

Costs in software-bugs repair represent a big part of the software maintenance budget. Up to 90% of the software expenses [Le Goues et al., 2013] are invested to fix software defects. This is an activity that consists on many tasks including fault localization, patch generation and implementation and execution of test cases. Many of them are in general expensive, difficult and in a big percentage of the cases human work is required, despite of the large amount of tools that nowadays support this work. Taking into account the growing and enormous worldwide software market, fixing software bugs more efficient and effective is an urgent necessity. In the last few years the community research has been focused on the automation of the software reparation activities. Works performed by Le Goues et al. [2012b] and Perkins et al. [2009] show meaningful results towards the complete automation of bug fixing. Nevertheless, there are still many challenges that need to be overcame in order to reach higher scalability, efficiency and quality.
Although it has been shown that certain amount of bugs can be detected and repaired in a considerable amount of time with a certain number of resources [Le Goues et al., 2012a], this solution is not attractive to be used in big and complex systems. Thus, it is necessary to explore intensively related alternatives which can bring better results i.e. better bug-finding-accuracy, wider variety of bugs to be detected, reduced amount of time-to-produce a patch and less use of hardware resources. The purpose of this work is precisely to focus in one of the most critial aspects of automatic software repair i.e. patch generation. It is based on the tool GenProg [Le Goues et al., 2012b] which uses a genetic programming algorithm to produce software repairs.

In the most recent version of GenProg, fix templates can be introduced as if they were genetic programming operands i.e. besides swap, delete and insert new code, another possible mutation is the application of a predetermined template. The purpose of this work is to demonstrate that by using fix templates in automatic software repair, even more bugs can be repaired with human-like patches. We will use the new feature of GenProg to apply fix templates by repairing C programs. The set of fix templates or fix patterns are extracted mainly from [Kim et al., 2013] along with other couple of works. Afterwards, the templates are implemented and additional adjustments to GenProg need to be made. Finally, we run a considerable amount of experiments to prove our hypotheses. At the end will be demonstrated that, despite of the lack of strong statistical evidence, we can consider that

our approach outperforms in certain grade quantitative and qualitative the current version of GenProg.

Chapter 2

Goals

2.1 G1: Discover repair-patterns for C

To elicit repair-patterns for C programs is the base of this work. Questions like which are the typical programming mistakes or which are the most language-related aspects to take into account to avoid failures, give us a more accurate perspective about real software errors and its respective solution. Thus, the systematic software transformations in order to build up a patch, which in the bottom is somehow random, will be closer to the reality, increasing the possibility to find a reparation in a more efficient way and, especially, with a higher quality.

Very similar works have been made to accomplish that. Nevertheless, to our knowledge, this is the first one made to C programs.

This goal will be achieved performing the following actions:

- Extract from literature typical repair actions for C programs.
- Classify and group repair actions.
- Represent the classified repair actions in templates.
- Describe the actions to be made in order to apply a template.

2.2 G2: Extend GenProg with repair templates

GenProg v3 offers the possibility to declare repair templates as part of the gamma of mutation operations. The appliance of a repair template is done as if were another mutation operator like swap two lines of code or delete/insert a line of code.
In order to achieve that, two main tasks need to be done:

- Adjust GenProg. Despite that with the last version of GenProg different kind of templates can be implemented, not of all the foreseen templates can be written and applied in the current status of the tool. It needs to be extended to the new program

repairs found in the Section 2.1. Specially, the modules Template and Search need to be adjusted.

- Implement templates. The discovered templates in Section 2.1 must be coded in C.

2.3 G3: Benchmark

The comparison of the results obtained by GenProg with the results obtained in this work is of vital importance. We pretend to evidence that our approach outperform GenProg in the number of repaired bugs and the quality of the repair. In order to formalize this assessments, an extended description of the problems, along with the research method and hypotheses are given in the Chapter 5.

2.4 Document Structure

This master thesis is organized as follows. Chapter 3 presents the foundations and technologies related with automatic software repair with fix templates. In chapter 4 we expose the state of the art regarding automatic software repair and the use of fix-templates to board the software repair problem. In chapter 5 we state the problem that this thesis intend to solve. The hypotheses and research questions that guide this research work are presented there as well. In chapter 6 we expose the research method utilized as guideline of this master thesis. Chapter 7 contains the description of discovered templates. Chapter 8 present the implementation details i.e. the adjustments needed in GenProg and the implementation of the templates. The details about the experiments and their results are presented in the chapter 9. Finally, chapter 10 concludes the master thesis and future work is suggested.

Chapter 3

Foundations and Technologies

3.1 Automatic Software Repair

Automatic software repair has gained special attention in the research community in the last few years [Le Goues et al., 2013] [Shafiq and Arshad]. The main goal is to offer the possibility to fix software defects effectively and at a low cost. Besides, it is pretended that the software repairs must follow certain guidelines. For example, the repair must not deteriorate the program's quality, the expected program's behavior must not be altered or there must be traceability on the process followed for fixing the defect. Automatic Software Repair comprise a set of activities in order to accomplish its objective. In order to achieve a complete automation, the research has been focused in the automation of each one of the tasks performed to repair a bug.

- Fault localization (debugging): it consists on the code exploration in order to realize the precise place or places in the code where the failure is generated. To do that exist many techniques, for example
  - Delta debugging: pieces of code are systematically added and removed conforming a set of chunks of code which are repeatedly tested. The chunks which throw positive results and the ones which throw negative results are compared in order to realize in which line or lines of code the error takes place.
  - Log analysis: in this methodology, as the program runs an execution trace is written in a file. When a failure occurs, the trace is processed in order to determine where the error resides.
  - Static code analysis: with static code techniques possible points of failure can be detected. A static code analysis is performed on a program. Following certain rules, the analysis tool determines where a program can eventually fail.
  - Software verification: consist on the verification of a program based on a contract.
  - Test-based: the technique consists on the generation of two types of test: positives and negatives: In the positive test cases the program does not fail. On the other side, when a negative test is run, the program fails. The execution path of the program

when the positive and negative test cases are run is traced. Then, this traces are compared: the statements executed by when the negative test cases were run but not when the positive test cases did, are candidates to be the points of failure.

- Patch generation: once the bug is localized, a repair needs to be generated. There are mainly two techniques to accomplish that:
  - Patch generation based on contracts
  - Fix on the same code/package
  - Semantic patch generation
- Testing: Once the patch is generated and applied in the code, the program is tested to confirm that the bug has been removed. Here again techniques like positive and negative test cases are used. The negative test cases are used to confirm that the failure does not appear anymore; and the positive test cases confirm that the program normal behavior was not altered.
  - Oracles: an oracle describes the expected result of a program. Generally it is not sufficient to confirm that the program does not fail, but also that the expected results were not altered by the appliance of a patch. The positive test cases are run and then the result is compared with the oracle. If they differ, the patch is rejected.

3.2 Search-based Software Engineering

The term of Search-based Software Engineering was first used by Harman and Jones [2001] in order to describe a group of metaheuristics intended to solve typical Software Engineering problems like testing, cost estimation, software verification, requirements elicitation among others. SBSE has been also applied in many fields like mechanical, chemical, medical and biomedical, civil and electronic engineering. It is an approach in which, instead of finding a exact solution for a certain problem, an optimal or almost-optimal solution is found among of a set of possible solutions. These are normally grouped in a structure which exposes all possible solutions of the problem and the relation between them. This structure is called search space and normally is constituted by a large number of solution candidates. This cases are very suitable for SBSE metaheuristics since finding a solution with analytical techniques in such scenarios can be suboptimal and the provided results might be biased or its quality might not be high [Harman and Jones, 2001]. On the other hand, SBSE metaheuristics do not pretend to find the perfect solution, but an optimal or almost-optimal one. Thus, this approach is presented as very interesting alternative, which empirically has been demonstrated that can throw satisfactory results. The SBSE metaheuristics describe the way in which this search space has to be walked through in order to find the best solution in the given search space. That is the reason why the word

"search" is used, which should not be confused with search from another contexts. Search, in SBSE context, refers to the metaheuristic Search-Based Optimization (SBO) techniques [Harman et al., 2012a]. They describe as well how must be determined that the optimal solution was found. In order to accomplish that, it must be possible that every solution candidate can be evaluated with a so-called fitness function. This fitness function produces a numerical value (fitness) depending on the properties of the solution candidate and the problem's environment. At the end of the process, the candidate with the highest fitness value represents the optimal solution of the problem. Why is SBSE suitable to solve Software Engineering problems? The problems from this area are normally declared in the way that a optimization-based solution is required. For example, questions that frequently appear are [Harman et al., 2012b]:

- What is the smallest set of test cases to achieve the highest level of coverage?
- What is the best way to design the architecture of a system?
- How must be the requirements, so that exists a balance between cost and customer satisfaction?
- Which is the best sequence of steps to refactor this system?

Furthermore, SBSE benefits include [Harman et al., 2012b]:

- Generality
- Robustness
- Scalability through parallelism
- Re-unification
- Direct fitness computation

In order to apply SBSE techniques to solve Software Engineering problems, they need to be reformulated in order to apply SBSE techniques. To do that, it is necessary to define [Harman and Jones, 2001]:

- a representation of the problem, which is suitable to abstract symbolic manipulation
- a fitness function (in terms of this representation)
- a set of manipulation operators

SBSE techniques are:

- Simulated annealing [Kirkpatrick, 1984]
- Ant Colony Optimization (ACO) [Dorigo and Blum, 2005]
- Tabu Search [Glover, 1989]

3.3 Genetic Programming

The nature is full of examples about how different activities - simple and complex ones - can be performed more efficient and in a more sophisticated and pragmatic way. Approaches like biomimicry look for solutions for different humans' problems that have been already solved in the nature. The most interesting design and process are studied and then imitated. Efficient energy use of energy learn from leaves and conditioned structures with efficient use of materials learn from bees are a couple of examples how actually some knowledge gathered from the nature is applied to solve certain problems [Benyus, 1997].
One of the most remarkable examples is the appliance in the computer science of the Darwinian theory of evolution which stands that the biological systems are product of natural selection [Mitchell, 1998]. In the nature, a specie evolve through a series of events that result into the appearance of new "better" organisms. Given a population i.e. a group of individuals, they combine with each other following certain rules producing a new set of organisms, i.e. a new population. Through another activity called selection, the best individuals of the new generation are chosen to be part of the next parent generation. In order to do that, a fitness evaluation is performed in every specie in order to determined if it is good enough or not. If it is, it will be taken as a part of the new population to again produce a new set of individuals. If not, it is rejected.
The same principle is applied in computer science. This approach is called Genetic Programming and its dynamics is reflected in the so called Genetic Algorithms. The basic idea consists on establish a population i.e. a set of individuals (genotypes) which are representations of some computational unit. The initial population of individuals is in general generated randomly, although it must not so be. From this population, a set of individuals is selected with help of a fitness function. A fitness function evaluates numerically how "good" each individual is in a given environment. This score constitutes the basis of the selection operation. Individuals with the lowest scores are rejected and the ones with the highest scores are used to create a new generation. This is accomplished through different operations like mutation (change of a single piece of the computational unit) and crossover (exchanging two pieces of two individuals to obtain an offspring) the population is evolved i.e. new individuals, with a high possibility to be good as well, are generated.

When this cycle of evaluation, selection and genetic operations is repeated many times, a overall population with a high fitness is obtained, which represents an optimal "solution" to a problem. It has been proved that the genetic algorithms converge [Jiang et al., 2007]. Also, leaving apart many details and variations of the algorithm, in general terms genetic programming has been presented as an cutting edge approach in order to solve problems. Poli et al. [2008] expose many fields in which Genetic Programming has been used with satisfactory results. For example, in environments where the interrelationships of relevant variables is unknown or poorly understood, or where conventional mathematical analysis does not or can not provide analytical solutions.

3.4 GenProg

GenProg [Le Goues et al., 2012b] is one of the most remarkable works in automatic software repair. It is based on the implementation of a Genetic Algorithm to evolutionarily generate software patches.

Listing 3.1. GenProg algortihm

```
Input: Full fitness predicate FullFitness : Patch → B
Input: Sampled fitness SampleFit : Patch → R
Input: Mutation operator mutate : Patch → Patch
Input: Crossover operator crossover : Patch 2 → Patch 2
Input: Parameter PopSize
Output: Patch that passes FullFitness
1: let Pop ← map mutate over PopSize copies of
2: repeat
3:       let parents ← tournSelect(Pop, Popsize, SampleFit)
4:       let offspr ← map crossover over parents, pairwise
5:       Pop ← map mutate over parents ∪ offspr
6: until ∃ candidate ∈ Pop. FullFitness(candidate) return candidate
```

The process begins with a sanity check to prove that the program and additional artifacts compile. It receives as input the program to repair and a set of test cases (positive and negative). The latest are used perform the bug localization and are the basis of the fitness function. Once the defect is located, a first patch is built based on the assumption that the fix of the bug can be found in the program's code itself. The process continues with the construction of a set of patch candidates which afterwards are evaluated using the fitness function. This construction is performed using selection, crossover and mutation operations. Once a fix candidate passes the fitness function, it is later minimized in order to eliminate useless code. This constitutes the final repair.

Chapter 4

Related Work

4.1 Automatic Software Repair

The research on Automatic Software Repair has been increased significantly in the last few years [Le Goues et al., 2013] [Shafiq and Arshad]. The empirical results in many scenarios have demonstrated that this is a promising approach, yet a long road has to be walked until it can be used in the real world. Le Goues et al. [2013] groups the current challenges in the automatic software repair in two big categories

- Real-world practicality
- Theory of unsound repair methods

The investigation in Automatic Software Repair can be divided into categories, depending on:

- The type of bugs to be repaired
- The software-repair-activity (fault localization, patch generation, testing) to automate
- The strategy to fix a bug

Thus, inspired by the study made by [Shafiq and Arshad] and adding more recent work on this area, in the following will be listed the research works son automated software repair following the described categorization:

4.1.1 Software repair based on formal methods

Automatic software repair based on contracts performs fault localization and patch generation with help of the program's formal specification. To this category corresponds AutoFix-E [Wei et al., 2010], which is a tool that automatically repairs Eiffel programs. Eiffel classes are accompanied with contracts, which are constituted of pre-conditions, post-conditions, intermediate assertions and class invariants. This contracts are the base artifact to determine the correct execution of a program. Test cases are generated automatically and run with AutoTest. Once a failure is discovered, the origin of the failure is

analyzed using object state assessment with help of boolean queries. Then, a fault profile - an abstract description of what went wrong - is generated based on the object's states of both passing and failing runs. After that, a normal behavior profile is generated based on the passing runs. This two characterizations are used to determine possible mutations that can transform the wrong behavior to a correct one in term of state transitions. With help of fix schemes and the possible mutations between fault and correct states, fix candidates are generated an evaluated with test suites. High ranked fix candidates correspond to a patch that eliminates the bug.
In the work of Pei et al. [2011] an extended version of AutoFix-E called AutoFix-E2 was implemented. It includes the novel approach evidence-based software repair. In this approach, not only the value of the queries are monitored, but also the values taken by any expression element in the program. During the fault localization, every expression and its values are ranked according to the possibility that there resides the error. The highest-rated expressions are the first candidates to generate fixes. AutoFix-E2 evidenced a remarkable improvement with respect to AutoFix-E: while AutoFix-E fixed 82% of precondition-faults and none of the postcondition-faults, AutoFixE2 repaired the 100% and 25% of the studied cases respectively.
He and Gupta [2004] developed an approach in which the fault localization is based on pre- and postconditions - expressed in terms of first order formulas – and positive and negative test cases. The process of bug fixing is similar to AutoFix-E2 i.e. also based on evidence tracking and model evolution. The evidence-track is analyzed and corresponding transformation are performed on this spots in order to remove the evidence. Afterwards, the modified program is tested and the pre- and postconditions are corroborated. Additionally, the concept of path-based weakest precondition is introduced. It serves as a base to define the hypothesized program state and an actual program state in every execution point, which in turn supports the fault localization, program transformation as correctness test. In the work of Gopinath et al. [2011] rich behavioral specifications written in the Alloy specification language is used. Using this specification language allows the fully automatic analysis of Alloy formulas. The Alloy tool-set permits to find the minimal input that yields in an output structure which violates a postcondition. Given that and a list of possible faulty statements, every statement is parametrized with a set of variables which take non-deterministic values from the domain of their respective types. Te set of fixed precondition, non-deterministic "faulty" code and postcondition is given to a SAT-tool (Satisfiability tool) to prune non-determinism. The SAT-tool performs the necessary transformation so that the postcondition is satisfied given a fixed precondition. The results given by the SAT tool are abstracted in program expressions, which in principle constitute the correction of the bug. Each of this expressions are then validated with every input in order to filter out wrong candidates.
Contract-based approaches are specially preferred to automatic detect and repair errors in data structures. The approach presented by Demsky and Rinard [2003] looks for constraint violations in data structures. Each violated constraint is converted into a normal disjunctive

form, which is a disjunction of conjunctions of basic propositions. Each of them has a repair action which makes it a true proposition. Repetitively a violated constraint is selected, a conjunction in normal form is selected and repair actions are performed in the basic propositions in that conjunction that are false. A repair action may cause that another constraint be violated. To avoid that, a pre-analysis is performed to avoid cyclic repairs that may turn into infinite loops. In [Malik et al., 2009] and [Malik et al., 2011] a methodology is presented in which a runtime technique – for data structures – and a compile-time technique – testing and debugging - are combined to repair Java programs. The approach consists on three steps: first, data structure repairs are used to transform erroneous output into a program state that satisfies the constraints. Then, it abstracts a sequence of repair actions within the context. Finally, heuristically is determined in which place the sequence applies. This approach is supported by Juzi [Elkarablieh and Khurshid, 2008] which is a powerful tool to automatically repair complex data structures in Java programs. Juzi performs repair actions consisting on mutating a faulty structure. At the end, such structure must satisfy a set of given constraints. To accomplish that, the Java program is instrumented in order to monitor data field accesses. Juzi performs repair both in reference as well as in the data fields of the structure. To repair data fields, Juzi uses symbolic execution supported by a solver for difference constraints and a general integer constraint solver; both to solve path conditions that can arise. SemFix, a tool developed by Nguyen et al. [2013] performs statistical fault localization, which consists on the location of the root-cause of the failure by means of correlating the execution of faulty statements and the program failure. A list of ranked suspicious fault locations is presented to the user. The repair is generated using symbolic execution and receiving as an input a set of repair constraints which are derived from a test suite. Using component-based program synthesis, a set of components which satisfies the repair constraints is built. Then, the final solution is constructed by combining the components obtained in the previous step.

4.1.2 Stochastic methods to automatic software repair

The work of Forrest [1993] is one of the first that presents Genetic Programming to repair software bugs. It presents the first stone of the automatic software repair's model used in GenProg. The experiments exposed in this work, which present the statistical behavior of genetic programming operators, served to the posterior development and refinement of GenProg. Schulte et al. [2010] present a framework to automatically repair assembly code taking as an input the assembly code of the program and a set of positive and negative tests. Through genetic operators, it generates patch candidates which are evaluated with a fitness function. The approach presented by Kessentini et al. [2011] performs fault localization based on examples and quality metrics. Examples are sample representations of defects. With this inputs, and using genetic programming, a set of detection rules are generated which in turn serve as a basis to detect errors and automatically generate correspondent fixes.

In this category we can list a significant number of works. Qi et al. [2014] developed RSRRepair. This is a tool that, instead of using genetic algorithms to search a bug fix, it performs a pure random search. They demonstrated that, since random search does not use a fitness function, ergo it is no necessary to run test cases, it outperforms GenProg in terms of time to find a repair and success rate (i.e. number of repaired bugs). Arcuri [2011] presents an extensive study regarding SBSE algorithms and Search Space Classification. He introduces a novel search operator which, unlike random search an genetic programming operators, takes into account information of the search space in order to guide the search in a more sophisticated manner. In order to perform a mutation, the search operator selects the highest-ranked nodes from a list of biased nodes instead of selecting them in a random manner. His results, obtained by implementing the tool JAFF (Java Automatic Fault Fixer), evidence a significant improvement on the automatic software repair when this search operator is used. Besides, evidence that Genetic Programming is superior in terms of accuracy and efficiency to other techniques like Hill Climbing or Random Search.

4.2 Templates in Automatic Software Repair

The work presented by Kim et al. [2013] serves as a base guideline to this master thesis. They developed PAR, which is a tool to automatically repair Java programs using fix templates. Initially this fix templates were elicited by mining the Eclipse JDT project repository. The process begins with statistical fault localization based on test cases. Given a set of weighted suspicious fault locations, the fix patterns are applied following an evolutionary computing process. In order to apply a fix pattern, the context of the fault location is evaluated i.e. it is determined whether the template can be applied or not depending on the characteristics of both pattern and statement at the fault location. The process finishes when the maximum number of generations or time limit is reached, or when a solution is found i.e. a modified program pass all positive and negative test cases. ASTOR [Martinez and Monperrus, 2014] is also a tool to automatically repair Java programs. ASTOR combines a self-implemented Java version of GenProg, a subset of PAR's templates and a mutation-testing based repair approach.
It is worth to include again in this category the work of Wei et al. [2010] which was previously review in the section (cite te section). In order to generate candidate fixes, AutoFix-E selects a fix schema i.e. a template that abstract common statement patterns. After that, this template is instantiated with actual condition which are extracted from the fault profile. The set of possible instantiations constitute the set of candidate fixes. Cfix [Jin et al., 2012] is intended to repair concurrency bugs. Its approach consists on apply a certain fix strategy for when a concurrency bug is detected. The types of bugs that Cfix is able to detect are atomicity and order violations, data-race and abnormal inter-thread data-dependence.

Chapter 5

Problem statement and hypotheses

Automatic software repair has been presented as an interesting strategy to reduce software's maintenance costs. Nevertheless it has been hardly criticized for many reasons. One of them, is the quality of the repairs [Arcuri, 2011] [Kim et al., 2013]. The patches generated by tools like GenProg [Le Goues et al., 2012b] diminishes in some cases the functionality of the program by, for example, replacing an statement in the code for other extracted from other location in the same program or package, though the bug is indeed removed i.e. all the negative and positive tests pass. Besides, these fixes are qualified as "ugly" i.e. they are not easy to read or understand, or worse, make no sense.
Time to find a fix is not concern of this work. In advance we infer that the use of fix-templates diminishes the performance or the tool. That is because GenProg with templates includes more mutation operators, many of the with a high level of randomness at the moment of apply a template e.g. select randomly a boolean operator. Thus, the amount of fix candidates increases considerably in every generation. The more fix candidates the program generates, the longer it takes to find a patch. And, the more mutation operators exist, the higher is the likelihood of generate not-compilable fix candidates.
PAR [Kim et al., 2013] is a tool that repairs software bugs using fix patterns. This strategy is presented as favorable, basically since the nature of the software patches are more acceptable. Besides, PAR is capable to repair more bugs than GenProg. Nevertheless, Monperrus [2014] criticizes strongly the results presented on Kim's work. The lack of description of the defect classes and the explicitly statement of what templates repair which type of bugs, represent a difficulty for future work. Despite of that, subsequent works [Martinez and Monperrus, 2014] encourages the use of patterns to address automatic software repair.
In order to answer the questions presented by Monperrus, this work pretends to establish which are the typical templates used to repair C programs.

- H_0: *Pure genetic programming and genetic programming along with fix patterns repair the same amount of bugs*
- H_1: *Automatic software repair with fix patterns can fix more bugs than pure genetic programming*
- *RQ1: Which type of bugs can be repaired using fix templates with Genetic Programming?*

Chapter 6

Discovered Templates

6.1 Methodology

Three different sources of information were taken to elicit fix templates for C programs:

- Templates derived from PAR,
- Templates elicited from the literature

6.1.1 Templates derived from PAR

As described in Section 4.2, PAR is a pattern-based tool for automatic software repair. The fix patterns used in PAR were mined out from the Eclipse JDT project's repository which contains a history of more than 10 years. The fix patterns are listed as follows:

- **Altering method parameters**: This pattern fixes the bug by changing one parameter that is given in a method invocation
- **Calling another method with the same parameters**: This pattern fixes the bug by invoking another method solving a wrong method invocation bug
- **Calling another overloaded method with one more parameter**: This pattern fixes the bug by adding one parameter to an overloaded method
- **Changing a branch condition**: This pattern modifies a branch condition in conditional statements by adding or removing a term from a predicate.
- **Adding a null checker**: This pattern adds a control flow statement, by which a object is checked if it is null or not.
- **Initializing an object**: This pattern inserts a object initialization before the object be used in an statement.
- **Adding an array bound checker**: This pattern adds a if-condition that checks that the array's position to be accessed be in the bound limits of it.

▻ **Adding a class-cast checker**: This pattern adds an if-condition before a cast take place. It checks that the castee be an object of the same type of the casted object.

From this list we must select those which are applicable for C programs. For the nature of the both programming languages (Java and C), is impossible that many of the bugs that this patterns pretend to solve, appear in C programs. Thus, from this list the following are pre-selected as Template Candidates (TC):

▻ **TC1: Parameter replacer**

▻ **TC2: Expression replacer**

▻ **TC3: Expression adder and remover**

▻ **TC4: Null-pointer checker**

▻ **TC5: Range checker**

6.1.2 Templates derived from literature

Dex [Raghavan et al., 2004] is a tool for mining CVS repositories. It was tested to analyze bug fixes from two open-source software development projects: the Apache HTTP server and the GCC compiler suite. From there, the following fix patterns were extracted:

▻ **TC6: Altered existing function bodies**

▻ **TC7: Inserted conditional statements**

▻ **TC8: Into existing functions**

▻ **TC9: Inserted function calls**

▻ **TC10: Altered existing function calls**

▻ **TC11: Altered existing if conditions**

▻ **TC12: Altered existing assignment statements**

Besides, the experiments evidenced the type of variables added when an if-predicate is modified. These are listed as follows:

▻ Local variable

▻ Field

▻ Function Parameter

▻ Global variable

The last source of information to elicit fix patterns is the study performed by [Chang et al., 2008]. The results show the major classes of problems that can origin an error in C programs, namely:

- Invalid method parameters.
- Wrong method usage.
- Faulty method implementation.
- Environment fault.

Based on this, we can infer which kind of fix templates can be applied in case one of this errors is made.

- Invalid method parameters: **TC13: Alter method parameters**
- Wrong method usage: **TC14: Alter method parameters**
- Faulty method implementation: Multiple solutions possible. Not unique template applicable.
- Environment fault: Out of the context.

From the above mentioned sources of information, we select those patterns which can be applicable for C programs. Besides, we must factorize some of them because of their similitude. One important aspect to take into account, is the assumption that many of the fix patterns for Java programs are applicable in C programs. If we compare the fix patterns of PAR and the ones of the other works, we can notice that they are practically the same except for some tightly coupled with the Java programming language. The following list presents the selected templates

- **T1: Parameter replacer**
- **T2: Expression adder**
- **T3: Expression remover**
- **T4: Null-pointer checker**
- **T5: Variable inicializer**

In the following sections, details about each template is presented. The presentation's format is inspired in the work of Kim et al. [2013].

6.2 T1: Parameter replacer

Listing 6.1. Template parameter replacer

```
<ID>
T1

<Description>
Modifies one argument of a function call by replacing one of them
by some variable in the same scope.

<Preconditions>
P ← program
B ← fault location in P. Must be a function call with arguments

<AST Analysis>
p: randomly selected function's argument from B
V: list of variables on the scope of B with the same type of p

<Program Editing>
if V is not empty
        v ← randomly variable selected from V
        replace p for v
```

6.3 T2: Expression adder

Listing 6.2. Template expression adder

```
<ID>
T2

<Description>
Adds a new term to an existing predicate

<Preconditions>
P ← program
B ← fault location in P. Must be an if-predicate
O ← list of logical operators: {=,!=,<,>,<=,=>}
N ← list of defult numerical values: {0}
S ← list of defult char values: {''}

<AST Analysis>
V: list of variables on the scope of B

<Program Editing>
if V is not empty
        v ← randomly selected variable from V
        o ← randomly selected operator from O
        if v is numeric
                n ← randomly selected value from N
                p ← buildpredicate (v,o,n)
        else
                s ← randomly selected value from S
                p ← build_predicate (v,o,n)

        ADD p TO B USING AND LOGICAL OPERATOR
```

6.4 T3: Expression remover

Listing 6.3. Template expression remover

```
<ID>
T3

<Description>
Removes a term from an existing predicate.

<Preconditions>
P ← program
B ← fault location in P.
    Must be an if-predicate

<AST Analysis>
B must have at least two terms.

<Program Editing>
p ← randomly selected predicate from B
REMOVE p
```

6.5 T4: Null-pointer checker

Listing 6.4. Template null-pointer checker

```
<ID>
T4

<Description>
Adds a if-predicate surrounding a variable to check that
it is not null

<Preconditions>
P ← program
B ← fault location in P

<AST Analysis>
V ← variables in the same scope of B

<Program Editing>
v ← randomly selected variable from V
o ← '!='
n ← NULL
p ← build_predicate (v,o,n)
INSERT p AS PARENT NODE OF B
```

6.6 T5: Variable initializer

Listing 6.5. Variable initializer

```
<ID>
T5

<Description>
Adds a initialization statement for a variable

<Preconditions>
P ← program
B ← fault location in P

<AST Analysis>
V ← variables in the same scope of B

<Program Editing>
v ← randomly selected variable from V
o ← '←'
if v = NULL
    if v is numeric
      p ← build_predicate(v,o,0)
    else if v is char
      v ← build_predicate(v,o,'')

INSERT p AS PARENT NODE OF B
```

Chapter 7

Implementation

In this project, the implementation consisted on two parts: First, GenProg v3 had to be adjusted to support the fix templates elicited in this work. Second, the templates itself had to be coded. After that, the changes needed to be tested and GenProg had to be many times adjusted until a stable version were ready for the experiments. Besides, since the version 3 of GenProg is the most recent, there were a significant amount of errors that needed to be repaired in order to be capable of perform the repairs. For these reasons, this part of the project was one of the most time-consuming.

7.1 How do templates work in GenProg?

A GenProg template consists on three parts:

- Name: describes uniquely the template.
- Holes: describe, through a series of attributes, the conditions that a code unit has to accomplish to be electable to apply a template.
- Actions to performed: state how the templates are to be applied in terms of the holes previously described.

Listing 7.1. Example of a template included in the source code of GenProg v3

```
void swap() {
    hole __hole1__ __attribute__((holetype ("stmt"))) __attribute__((constraint ("fault_path")));
    hole __hole2__ __attribute__((holetype ("stmt"))) __attribute__((constraint ("fault_path")));
    { __blockattribute__(__hole1__)
        USE(__hole2__);
    }
    { __blockattribute__(__hole2__)
        USE(__hole1__);
    }
}
```

The listing Listing 7.1 shows an example of a template included in the source package of GenProg v3. The name of this template is swap and its purpose is to interchange two nodes of the AST tree. The kind of nodes that can be interchanged are given by the holes' attributes. In the example, both hole1 and hole2 look for statements which are present in

the fault path. After the holes' declaration, it is established the actions to be performed with the selected units of code. The __blockattribute__ directive indicates that the unit of code given as parameter must be replaced by the portion of code that is contained in this block. We can describe in plain English the actions in the example as follows: "the unit of code selected by hole1 must be replaced for USE(_hole2_) and the unit of code selected by hole2 must be replaced for USE(_hole2_)". USE(hole) is a macro which injects the portion of code described by a hole.

GenProg v3 defines four types of hole, namely:

- Statement: a statement in the program
- Expression: a block consisting of a set of statements
- Lvalue: a LValue
- Argument: an argument

and the following constraints, which serve as selection criteria

- Fault_path: The unit of code must be selected from the fault locations.
- Fix_path: The unit of code must be selected from the fix locations.
- Ref (String): The unit of code is referenced in the hole named by the string.
- InScope (String): The unit of code must be in the scope of the hole named by the string.
- ExactMatches (String): The unit of code must match exactly with the expression given in the string.
- FuzzyMatches (String): The unit of code must "almost" match with the expression given in the string.
- HasType (String): The unit of code (Lvalue or Argument) must be of the type given by the string.
- IsLocal: The unit of code must be local.
- IsGlobal: The unit of code must be global.
- HasVar (String): the unit of code contains a variable with the name given by the string.

7.2 Adjustments in GenProg

One of the new features included in the most recently version of GenProg is the possibility of include templates to repair a bug. This templates serve as another mutation operator i.e.

in addition to swap, add and remove we have the “apply template” operator. The changes in GenProg v3 consisted in modifying two modules:

- Template module:
- A new hole type was added, namely Predic_hole. It was included to explicitly select and manipulate predicates. In this work only “if” predicates are selected and manipulated.
- A new constraint was added, namely Type. It was included to apply certain templates to certain types of variables.
- For the template “Expression adder/remover” a list of default values and logic operators was added in order to build predicates based on them.

Chapter 8

Experiments

8.1 Setup

The table Table 8.1 presents the programs included in the experiments of this work.

Table 8.1. Dataset used in our experiments. LOC=lines of code, bugs=number of bugs

Program	LOC	Description	bugs
fbc	97,000	Free basic compiler	3
gmp	145,000	Mathematic library	2
gzip	491,000	Compression tool	12
libtiff	77,000	Image processing	24

Because of time and resource limitations, lighttpd, php, python and wireshark were not included in the experiment. The same parameter configuration used by the experiments performed in [Le Goues et al., 2012b] were used. The listing Listing 8.1 shows an example of a GenProg's configuration file. The only remarkable difference, is the fact that in our experiments the parameter template is set and points to the template file. This file contains the list of templates described in Chapter 6.
H_0 is of error type I it is rejected if it is true. The significant level we choose is 0.05 and the number of runs is established to 10 for each data sample i.e. we run 10 GenProg trials for each bug. Despite the advised number of runs to obtain reliable results is 30 or "as much as possible" [Arcuri and Briand, 2014], because of time and resource limitations it was reduced to 10. The amount of time that took to run every experiment and the totals are shown in the table Table 8.2. We must also take into account the time that took to build and adjust every bug's scenario, which in some cases implied a work of many hours. Each trial consisted on repair perform of GenProg with giving as a parameter a seed to the pseudo-random generator.

Since the ultimately purpose of this work is to demonstrate that genetic programming along with fix templates can find repair more bugs than mere genetic programming, we choose the metric Number of Repaired Bugs as baseline to prov e this statement.

The results obtained in this work were statistically compared with those obtained by Le Goues [2013]. In order to statistically compare the results obtained by two experiments, we need to run the experiments enough number of times in an independent way [Arcuri and Briand, 2014]. Besides, for fairness sake, In this set of experiments

Nevertheless, for sake of utilization of up-to-date data and tools, unlike the experiments performed by LeGoues in this work were used the benchmark-data of ManyBugs [Le Goues et al., 2015] and GenProg version 3, which is the latest version of this tool. ManyBugs is to our knowledge the most complete benchmark repository to automatic software repair of C programs. It presents a set of C programs with defects. For every program, a test suite is provided which allows to reproduce the bugs described. Besides, ManyBugs contains the results of the experiments performed by GenProg v2.0 and other tools. It provides as well a set of virtual machines, which are appropriate to perform experiments in an homogeneous setup. GenProg v3.0, in comparison with version 2.0, presents a list of improvements including many bug fixes, many improvements like more compact and comprehensible code, support to templates, among others. Support to templates constitutes the main feature for this work, since it basically contains the logic of parsing and appliance of templates to repair bugs.

Since the main goal of this work is to demonstrate that using genetic programming along with fix templates more bugs can be repaired, we select the metric Number of Repaired bugs.

Listing 8.1. Example of GenProg experiment setup

```
—multi-file
—search ga
—compiler-command perl compile.pl \_\_EXE\_NAME\_\_ > /dev/null
—test-command \_\_TEST\_SCRIPT\_\_ \_\_TEST\_NAME\_\_ >& /dev/null
—crossover subset
—rep cilpatch
—suffix-extension .c
—describe-machine
—pos-tests 144
—neg-tests 1
—program bugged-program.txt
—prefix preprocessed
—seed 5
—test-script ./test.sh
—popsize 40
—generations 10
—promut 1
—mutp 0
—fitness-in-parallel 2
—rep-cache default.cache
—templates templates.c
```

In the last line of the configuration file the parameter *templates* is introduced which referenced the file *templates.c* which contains the set of implemented templates in C.

8.1.1 Software

- The Fedora release 13 (Goddard) Virtual Machine, provided by GenProg's team was used as platform to run the experiments
- Since the GenProg's experiments of ICSE2012 were performed with the version 2 of it, for sake of fairness, this experiments were performed once again with GenProg v3.

8.1.2 Hardware

- Quad-core 2.53GHz Processor
- 2519 MB of RAM Memory

8.2 Results

The table Table 8.3 shows the number of repairs performed by GenProg v3 and our modified version of GenProg. The results evidence that our modified version of GenProg in 2 of the 4 cases studied outperform GenProg v3, namely by repairing 2 of 3 bugs in fbc and 2 of 2 in gmp. In gzip the same mount of repairs where generated. In the case of libtiff the number of fixes by GenProg v3 was superior than the ones of GenProg with fix templates. Analyzing the nature of the bugs which could not be repaired of both gzip and libtiff, we can notice that bugs, like *libtiff-bug-2007-07-08-bd2f947-ccc10c7*, *libtiff-bug-2007-07-08-c766cb7-0cc95fb* required the modification by humans of many files. Approaches like the implemented of this work are not able to perform such complex reparations. For that reason, we present the Table 8.4 which contains the number of number of repaired bugs without including those bus that we marked as complex.

The listing Listing 8.2 presents an example of a repair performed by GenProg with templates. In this case, the template Expression adder was the one that was applied. In comparison to the human-performed bugfix, we can notice that the repair are very similar. However, most of the fixes performed by GenProg and GenProg with templates are nonsensical. The lack of context knowledge disables the capacity of generate more appropriate patches. It is important to remark that the appliance of the templates in the way they were designed, takes in a certain way more context information in comparison with GenProg. In this approach the template-based mutation operators are more aware about the structure of the program than GenProg.

Listing 8.2. Example of an applied template

```
// Faulty line in FBC R5458

- if( (start > 0) && (start <= dst_len) )

// Repaired line with the fix template "Expression Adder"

+ if( (start > 0) && (start <= dst_len) && (len > 0) )

// Repaired line by a human in R5459

+ if( (start > 0) && (start <= dst_len) && (len != 0) )
```

As we could notice in the table Table 8.2, the time to find a repair for GenProg with templates is 43,1% more in comparison with GenProg without templates. Nevertheless, efficiency is not a concern in this work. As we stated in Chapter 5, we know in advance that the use of fix templates diminishes the performance of the algorithm.

The Table 8.5 presents the applied templates to the fixed bugs in our experiments. In the following section we will analyze the results obtained in this sections.

8.3 Analysis of results

In the following, the results will be analyzed based on the stated problem and hypotheses Chapter 5.

8.3.1 H_0: Pure genetic programming and genetic programming along with fix patterns repair the same amount of bugs

Because of the nature of the experiments (few sample bugs for every program), it is difficult notice a significant difference between the results of GenProg with templates and GenProg. The minimum *p-value* obtained was 0.576087, which is the result of comparing the results of the most distanced "number of repaired bugs"-values, namely the number of repairs found for fbc and lighttp. The rest of p-values are equal to 1. We used the Fisher exact test to evaluate whether there is statistical difference between dichotomous results. This *p-value* is clearly too high for the stablished significance level (=0.05).
However, this sort of comparisons can be contraproductive Arcuri and Briand [2014]. The value of significance level is a heritage with more than 100 years of antiquity applied in the context of natural sciences. In the context of our concern, it is neccesary to redefine such parameters in the face of the nature of our experiments and the subjects we analize. Taking into account the recommendation of Arcuri and Briand [2014], we limit ourselves to report

the p-value obtained in this work and let the reader to make his/her own conclusions based on the context.

8.3.2 RQ1: Which type of bugs can be repaired using fix templates with Genetic Programming?

The table Table 8.6 present the properties of the fixed bugs of GenProg with templates in comparison with the human fixes. We can notice, that in 10 of 14 fixes the type of bug fix coincide, being the template Expression adder/remover the most used. We can notice as well, that in comparison with the other bugs of ManyBugs, the one repaired by GenProg with templates needed just a change in one line when were repaired by a human. We infer that the fix-templates approach are more suitable for small bugs i.e. bugs that only require a small change.

Table 8.2. Time of experiments in hours

Bug	GenProg3	GenProg3 + Templates
fbc-bug-5251-5252	46.158	47.681
fbc-bug-5458-5459	37.224	35.884
fbc-bug-5556-5557	43.552	39.458
gmp-bug-13420-13421	50.407	47.635
gmp-bug-14166-14167	42.006	38.436
gzip-bug-2009-08-16-3fe0caeada-39a362ae9d	13.424	12.592
gzip-bug-2009-09-26-a1d3d4019d-f17cbd13a1	8.86	8.763
gzip-bug-2009-10-09-1a085b1446-118a107f2d	9.666	9.008
gzip-bug-2010-01-30-fc00329e3d-1204630c96	12.35	13.252
gzip-bug-2010-02-19-3eb6091d69-884ef6d16c	9.666	9.318
libtiff-bug-2005-12-14-6746b87-0d3d51d	13.424	13.478
libtiff-bug-2005-12-21-3b848a7-3edb9cd	0.491	24.851
libtiff-bug-2005-12-27-6f76e76-5dac30f	0.003	22.909
libtiff-bug-2006-02-23-b2ce5d8-207c78a	0.519	22.486
libtiff-bug-2006-02-27-6074705-e6d0c32	26.184	24.901
libtiff-bug-2006-03-03-a72cf60-0a36d7f	0.484	26.719
libtiff-bug-2006-03-03-eec4c06-ee65c74	0.499	0.542
libtiff-bug-2007-07-08-bd2f947-ccc10c7	25.447	25.066
libtiff-bug-2007-07-08-c766cb7-0cc95fb	0.49	22.809
libtiff-bug-2007-07-13-09e8220-f2d989d	27.947	26.941
libtiff-bug-2007-07-19-ce4b7af-7d6e298	0.489	0.454
libtiff-bug-2007-08-24-827b6bc-22da1d6	0.48	0.516
libtiff-bug-2007-11-02-371336d-865f7b2	26.316	23.684
libtiff-bug-2007-11-23-82e378c-cf05a83	0.483	22.909
libtiff-bug-2008-04-15-2e8b2f1-0d27dc0	0.481	0.493
libtiff-bug-2008-09-05-d59e7df-5f42dba	0.505	0.468
libtiff-bug-2008-12-30-362dee5-565eaa2	0.537	25.872
libtiff-bug-2009-02-05-764dbba-2e42d63	0.517	0.496
libtiff-bug-2009-06-30-b44af47-e0b51f3	28.447	26.797
libtiff-bug-2009-08-28-e8a47d4-023b6df	0.499	25.524
libtiff-bug-2009-09-03-6406250-6b6496b	26.132	25.426
libtiff-bug-2010-06-30-1563270-1136bdf	0.549	0.591
libtiff-bug-2010-11-27-eb326f9-eec7ec0	0.497	26.794
libtiff-bug-2010-12-13-96a5fb4-bdba15c	0.495	0.485
Total	455.228	653.238
Total	**46.1 days**	

Table 8.3. Number of repaired bugs by GenProg with templates and GenProg

Program	Number of bugs	GenProg + Templates	GenProg
fbc	3	2	1
gmp	2	2	1
gzip	5	1	1
libtiff	24	9	12
Total	34	14	15

Table 8.4. Number of repaired bugs by GenProg with templates and GenProg without taking into account libtiff refactorings

Program	Number of bugs	GenProg + Templates	GenProg
fbc	3	2	1
gmp	2	2	1
gzip	5	1	1
libtiff	24	9	9
Total	34	14	12

Table 8.5. Templates applied to repair bugs

Bug	Repair found?	Templates
fbc-bug-5251-5252	No	
fbc-bug-5458-5459	Yes	Expression adder/remover
fbc-bug-5556-5557	Yes	Expression adder/remover
gmp-bug-13420-13421	Yes	Parameter replacer
gmp-bug-14166-14167	Yes	
gzip-bug-2009-08-16-3fe0caeada-39a362ae9d	No	
gzip-bug-2009-09-26-a1d3d4019d-f17cbd13a1	No	
gzip-bug-2009-10-09-1a085b1446-118a107f2d	No	
gzip-bug-2010-01-30-fc00329e3d-1204630c96	No	
gzip-bug-2010-02-19-3eb6091d69-884ef6d16c	Yes	Expression adder/remover
libtiff-bug-2005-12-14-6746b87-0d3d51d	No	
libtiff-bug-2005-12-21-3b848a7-3edb9cd	Yes	Expression adder/remover
libtiff-bug-2005-12-27-6f76e76-5dac30f	No	
libtiff-bug-2006-02-23-b2ce5d8-207c78a	Yes	Expression adder/remover
libtiff-bug-2006-02-27-6074705-e6d0c32	No	
libtiff-bug-2006-03-03-a72cf60-0a36d7f	Yes	Expression adder/remover
libtiff-bug-2006-03-03-eec4c06-ee65c74	Yes	Expression adder/remover
libtiff-bug-2007-07-08-bd2f947-ccc10c7	No	
libtiff-bug-2007-07-08-c766cb7-0cc95fb	No	
libtiff-bug-2007-07-13-09e8220-f2d989d	Yes	Parameter replacer
libtiff-bug-2007-07-19-ce4b7af-7d6e298	Yes	Expression adder/remover
libtiff-bug-2007-08-24-827b6bc-22da1d6	No	
libtiff-bug-2007-11-02-371336d-865f7b2	Yes	Expression adder/remover
libtiff-bug-2007-11-23-82e378c-cf05a83	No	
libtiff-bug-2008-04-15-2e8b2f1-0d27dc0	No	
libtiff-bug-2008-09-05-d59e7df-5f42dba	No	
libtiff-bug-2008-12-30-362dee5-565eaa2	Yes	Null-checker
libtiff-bug-2009-02-05-764dbba-2e42d63	Yes	Null-checker
libtiff-bug-2009-06-30-b44af47-e0b51f3	No	
libtiff-bug-2009-08-28-e8a47d4-023b6df	No	
libtiff-bug-2009-09-03-6406250-6b6496b	No	
libtiff-bug-2010-06-30-1563270-1136bdf	No	
libtiff-bug-2010-11-27-eb326f9-eec7ec0	No	
libtiff-bug-2010-12-13-96a5fb4-bdba15c	No	

Table 8.6. Comparison between human fixes and fixes from GenProg with templates

Bug	Type of human fix	Template
fbc-bug-5458-5459	Change of conditional	Expression adder/remover
fbc-bug-5556-5557	Change arguments to a function	Expression adder/remover
gmp-bug-13420-13421	Change of condition Change arguments to a function	Parameter replacer
gmp-bug-14166-14167	Change statement	Expression adder/remover
gzip-bug-2010-02-19-3eb6091d69-884ef6d16c	Change of condition Change arguments to a function	Expression adder/remover
libtiff-bug-2005-12-21-3b848a7-3edb9cd	Change of condition	Expression adder/remover
libtiff-bug-2006-02-23-b2ce5d8-207c78a	Add 1 or more if-statements	Expression adder/remover
libtiff-bug-2006-03-03-a72cf60-0a36d7f	Change of condition	Expression adder/remover
libtiff-bug-2006-03-03-eec4c06-ee65c74	Change of condition	Expression adder/remover
libtiff-bug-2007-07-13-09e8220-f2d989d	Changes arguments to a function	Parameter replacer
libtiff-bug-2007-07-19-ce4b7af-7d6e298	Change of condition	Expression adder/remover
libtiff-bug-2007-11-02-371336d-865f7b2	Change of condition	Expression adder/remover
libtiff-bug-2008-12-30-362dee5-565eaa2	Change of condition Add new variable	Null-checker
libtiff-bug-2009-02-05-764dbba-2e42d63	Added function call	Null-checker

Chapter 9

Conclusions and Future Work

The results presented in this work allow us to understand which are the advantages of using fix-templates along with genetic programming to repair software bugs. We can assess that definitely the pattern-based approach generates patches similar to those made by humans. That is an important factor to take into account, since, in order to improve maintainability and traceability, the code must be as understandable and clean as possible. Many approaches have tried formal methods to solve the automatic software repair problem. The advantage with the solution presented in this work is that, despite of the high quota of randomness and unsoundness, it is significant to have the possibility of apply human knowledge and experience in the solution of this kind of problems. More efforts need to be made towards the elicitation of knowledge from repositories. They contain a enormous amount of information about how software engineers actually repair software errors. Martinez and Monperrus [2013] presents a interesting framework in order to mine software repositories.

Because of the increment of mutation operators, the performance of GenProg was negative affected. An effort towards more efficient algorithms need to be made as we every time increase the complexity of our algorithms to find software bugs. Alternative metaheuristics to Genetic Programming to board the automatic software repair problem have not until now been explored. New alternatives can easily being applied with the refinement of fitness functions and a better knowledge about the search space. Since metaheuristics such as Ant Colony Optimization, Simulated Annealing and Tabu Search have been proved that not only solve many engineering problems, but that they outperform some traditional approaches, it would be valuable to explore them towards soundness and more efficiency.

In this work it was demonstrated that is very important to count on gut benchmarking data. For the research community is of vital importance to have a homogeneous, wide-spread, stable and up-to-date benchmark data. GenProg Team makes a remarkable work in this sense. New researchers can easily get involved in the subject of investigation if they have, for example, ready-to-use virtual machines.

Bibliography

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

[Arcuri and Briand 2014] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014. ISSN 1099-1689. doi: 10.1002/stvr.1486. URL http://dx.doi.org/10.1002/stvr.1486.

[Benyus 1997] J. M. Benyus. *Biomimicry*. William Morrow New York, 1997.

[Chang et al. 2008] H. Chang, L. Mariani, and M. Pezzè. Self-healing strategies for component integration faults. In *Automated Software Engineering-Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on*, pages 25–32. IEEE, 2008.

[Demsky and Rinard 2003] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *ACM SIGPLAN Notices*, volume 38, pages 78–95. ACM, 2003.

[Dorigo and Blum 2005] M. Dorigo and C. Blum. Ant colony optimization theory: A survey. *Theor. Comput. Sci.*, 344(2-3):243–278, Nov. 2005. ISSN 0304-3975. doi: 10.1016/j.tcs.2005.05.020. URL http://dx.doi.org/10.1016/j.tcs.2005.05.020.

[Elkarablieh and Khurshid 2008] B. Elkarablieh and S. Khurshid. Juzi: A tool for repairing complex data structures. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 855–858, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368222. URL http://doi.acm.org/10.1145/1368088.1368222.

[Forrest 1993] S. Forrest. Genetic algorithms: Principles of natural selection applied to computation. *Science*, 261(5123):872–878, Aug. 1993.

[Glover 1989] F. Glover. Tabu search - part 1. *ORSA Journal on Computing*, 1(3), 1989.

[Gopinath et al. 2011] D. Gopinath, M. Z. Malik, and S. Khurshid. Specification-based program repair using sat. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software*, TACAS'11/ETAPS'11, pages 173–188, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19834-2. URL http://dl.acm.org/citation.cfm?id=1987389.1987408.

[Harman and Jones 2001] M. Harman and B. F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, Dec. 2001.

[Harman et al. 2012a] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11:1–11:61, Dec. 2012a. ISSN 0360-0300. doi: 10.1145/2379776.2379787. URL http://doi.acm.org/10.1145/2379776.2379787.

[Harman et al. 2012b] M. Harman, P. McMinn, J. T. De Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical Software Engineering and Verification*, pages 1–59. Springer, 2012b.

[He and Gupta 2004] H. He and N. Gupta. Automated debugging using path-based weakest preconditions. In *Fundamental approaches to software engineering*, pages 267–280. Springer, 2004.

[Jiang et al. 2007] H. Jiang, C. K. Chang, D. Zhu, and S. Cheng. A foundational study on the applicability of genetic algorithm to software engineering problems. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 2210–2219. IEEE, 2007.

[Jin et al. 2012] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *OSDI*, volume 12, pages 221–236, 2012.

[Kessentini et al. 2011] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni. Design defects detection and correction by example. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 81–90. IEEE, 2011.

[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*, ICSE '13, pages 802–811, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486893.

[Kirkpatrick 1984] S. Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of statistical physics*, 34(5-6):975–986, 1984.

[Le Goues 2013] C. Le Goues. *Automatic program repair using genetic programming*. PhD thesis, University of Virginia, 2013.

[Le Goues et al. 2012a] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, Piscataway, NJ, USA, 2012a. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337225.

[Le Goues et al. 2012b] C. Le 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, 2012b.

[Le Goues et al. 2013] C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[Le Goues et al. 2015] C. Le Goues, N. Holtschulte, E. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of c programs. 2015.

[Malik et al. 2009] M. Z. Malik, K. Ghori, B. Elkarablieh, and S. Khurshid. A case for automated debugging using data structure repair. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 620–624. IEEE, 2009.

[Malik et al. 2011] M. Z. Malik, J. H. Siddiqi, and S. Khurshid. Constraint-based program debugging using data structure repair. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 190–199. IEEE, 2011.

[Martinez and Monperrus 2013] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2013.

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

[Mitchell 1998] M. Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[Monperrus 2014] M. Monperrus. A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 234–242. ACM, 2014.

[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*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486890.

[Pei et al. 2011] Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer. Evidence-based automated program fixing. *CoRR, abs/1102.1059*, 2011.

[Perkins et al. 2009] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 87–102, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629585. URL http://doi.acm.org/10.1145/1629575.1629585.

[Poli et al. 2008] R. Poli, W. B. Langdon, N. F. McPhee, and J. R. Koza. *A field guide to genetic programming*. Lulu. com, 2008.

[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.

[Raghavan et al. 2004] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 188–197. IEEE, 2004.

[Schulte et al. 2010] E. Schulte, S. Forrest, and W. Weimer. Automated program repair through the evolution of assembly code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 313–316. ACM, 2010.

[Shafiq and Arshad] H. A. Shafiq and Z. Arshad. Automated debugging and bug fixing solutions: A systematic literature review and classification.

[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*, ISSTA '10, pages 61–72, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-823-0. doi: 10.1145/1831708.1831716. URL http://doi.acm.org/10.1145/1831708.1831716.

Declaration

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