



DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS,
STOCKHOLM, SWEDEN 2021

Evaluating template-based automatic program repair in industry

Gunnar Applelid

Evaluating template-based automatic program repair in industry

GUNNAR APPLELID

Master's Programme, Software Engineering of Distributed
Systems, 120 credits
Date: August 30, 2021

Supervisor: Martin Monperrus
Examiner: Benoit Baudry

School of Electrical Engineering and Computer Science
Host company: Saab AB
Swedish title: Utvärdering av mallbaserad automatisk
programreparation inom industrin

Abstract

Automatic Program Repair (APR) is a field that has gained much attention in recent years. The idea of automatically fixing bugs could save time and money for companies. Template Based Automatic Program Repair is an area within APR that uses fix templates for generating patches and a test suite for evaluating them. However, there exists many various tools and datasets, and the concept has not widely been evaluated at companies or tried in production. Critique of current research is that the bug datasets are gathered from only a few projects, are sparsely updated and are not representative of real-world projects. This thesis evaluates, TBar, Template based automatic program repair tool for Java on a large open-source bug dataset Bears and a small company dataset. Further, TBar is modified to kBar to be used for experiments. The results show that kBar presents Plausible patches to 35% (19/54) of the selected bugs and 13% (7/54) of them is Correct. Finally, a prototype were implemented at Saab, waiting for the developers to submit the first real-world bug to fix.

Abstract

Automatisk programreparation (APR) är ett område som har fått mycket uppmärksamhet under senare år. Att reparera buggar automatiskt kan spara både tid och pengar för företag. Mallbaserad automatisk programreparation är ett område inom APR som använder färdiga mallar för att laga buggar och tester för att utvärdera dem. Men det finns många olika verktyg och dataset och konceptet har inte blivit storskaligt utvärderat på företag eller testat i produktion. Kritik av nuvarande forskning är att dataset med buggar är insamlade från ett fåtal projekt, sällan uppdaterade och inte representerar verkliga projekt. Det här arbetet utvärderar, TBar, ett mallbaserat automatiskt programreparationsverktyg för Java på ett stort open-source bug dataset Bears och ett litet dataset från företaget. Vidare modifieras TBar till kBar för att användas i experiment. Resultaten visar att kBar presenterar möjliga lagningar till 35% (19/54) av de utvalda buggarna och 13% (7/54) av dem är korrekta. Slutligen implementerades en prototyp hos Saab, som är redo för att utvecklarna ska skicka in den första buggen att reparera.

Acknowledgements

I would like to thank my Examiner Professor Benoit Baudry and Supervisor Professor Martin Monperrus for introducing me to Automatic Program Repair and many other interesting concepts in their DevOps course at KTH 2020.

I would also like to thank Martin Monperrus for providing interesting literature, tools, feedback, good advice and ideas which serve as a base for the work in this thesis.

Furthermore, I would like to thank my Industrial supervisor Olle Nordin at Saab, who has spent a large amount of time helping with answering crucial questions for the company integration and experiments.

Finally, I would like to thank my girlfriend Caroline and children Vera and Valter for supporting me during these five years of hard work at KTH and enduring my absence.

Stockholm, June 2021,
Gunnar Applelid

Contents

1	Introduction	1
1.1	Problem	2
1.2	Stakeholders	2
1.3	Research Questions	3
1.4	Contributions	4
1.5	Outline	5
2	Background	6
2.1	Automatic Program Repair	6
2.2	Datasets for Program Repair	11
2.3	Continuous Integration	14
3	Related work on Automatic program repair tools	17
3.1	Template-based Automatic Program Repair	17
3.2	Sketches, Hooks and Bytecode Mutation	29
3.3	Other Abstract Syntax Tree Based Repair Tools	33
3.4	Repair Tools with Machine learning	37
4	Technical contribution	44
4.1	Design of TBar	44
4.2	Extensions of TBar in kBar	49
5	Experimental Research Methodology	54
5.1	RQ1: What is the effectiveness of template-based automatic program repair on different bug datasets?	54
5.2	RQ2: What are the characteristics of the generated patches by kBar?	58
5.3	RQ3: How could template-based automatic program repair be integrated into the existing CI pipelines used at Saab?	59

6	Experimental Results	59
6.1	RQ1: What is the effectiveness of template-Based Automatic Program Repair on different bug datasets?	59
6.2	RQ2: What are the characteristics of the generated patches by kBar?	75
6.3	RQ3: How could template-based automatic program repair be integrated into the existing CI pipelines used at Saab?	86
6.4	Discussion of Results	89
7	Discussion	91
7.1	Benefits, Ethics and Sustainability	91
7.2	Threats to Validity	91
7.3	Future Work	92
7.4	Template-based Repair Tools in the Industry	93
8	Conclusion	95
	References	96

1 Introduction

Automatic Program Repair (APR) [1] is an area within Computer Science that automatically repairs software bugs. It does exist tools for locating the bug location, which is commonly used. However, the actual repair is still mainly conducted by the developer [2]. APR aims to automate and fill the gap between automatic bug location and manual bug fix. There exist many different APR tools with different approaches such as machine learning, template-based and static analysis. Furthermore, some research shows that 50% of the developers' time is done bug fixing [3].

Two of the areas [4] in Automatic Program Repair are Generate and Validate and Test-suite based Automatic Program Repair, which is one of the most common approaches for Automatic Repair [5]. They try to produce software patches that make failing tests pass. However, there is no guarantee that the actual bug is fixed by passing the tests, but it could instead introduce new errors not covered by the test cases. Therefore, manual inspection of the generated patches could be needed. TBar, Template Based Automatic Program Repair tool [6] is a Java tool that leverages test-suite based repair together with the approach of fix templates. A fix template is a predefined modification of the code according to specific rules, such as inserting a null check before a buggy statement or updating or removing a conditional expression. Further, it combines fix templates from other APR-tools, making it a suitable representative of template-based Automatic Program Repair in general. The overall goal of this thesis is to evaluate the effectiveness of TBar in an industrial context.

1.1 Problem

The problem is that there is a lack of evaluation of Automatic Program Repair, and many of the current tools have been evaluated on a few open-source bug datasets and not been used in production. Many of the evaluations do mainly focus on the repair algorithms and not the entire chain of APR. According to Monperrus et al., it is necessary to “*study the design and implementation of an end-to-end repair toolchain that is amenable to the mainstream development practices*” [5], to show the actual value of APR in industry.

Some attempts to use APR tools have been tried out at Facebook, and Ubisoft has made attempts to identify templates of risky commits [7]. However, there is no large-scale usage of template based-repair tools in the industry. Also, according to Liu et al. [6], the area of template based approaches show promising results on research bug datasets such as Defects4j, but there is a lack of presentation of the contributions of the individual templates on the datasets. They highlight the need of evaluation on more bug datasets such as Bears. Therefore, the overall goal of this thesis is to provide an evaluation of the usage of Template-Based Automatic Program Repair in an Industrial context.

1.2 Stakeholders

This master’s thesis has been done at Saab within the department of Surveillance, Combat Systems & C4I Solutions Järfälla, Stockholm. Saab [8] is a Swedish defence company that was founded in 1937 to supply the Swedish air force with aircraft. Their goal is to supply the global market with products and services for the military defence area and civil security. They have over 17000 employees worldwide and product segments of Air, Land, Naval and Security. The department is developing combat systems and solutions for naval platforms such as

combat boats, patrol boats, frigates, aircraft carriers and submarines. The development process leverages DevOps and Continuous Delivery, and in order to shorten the release cycles, there is an interest to evaluate techniques for Automatic program repair.

The Saab bug dataset is a collection of 14 bugs from two different projects within the department Surveillance. The author of this thesis manually collected the bugs according to specific criteria described in section 5.1.2. Two of the criteria are that it is supposed to be a bug that has occurred in production in real life and that it is reproducible with the latest project version with a verified git commit. The collection took several weeks, and many potential bugs were discarded due to the criteria.

1.3 Research Questions

In order to provide a sound and thorough assessment of Template-based automatic program repair in a real-world industrial context, there is a need to investigate different angles of the tools and the environment. There is also a need to investigate how the tool itself is implemented to gain technical understanding. This investigation should gain insight to what extent the tool should be modified for the industrial context. Nevertheless, to gain this knowledge, industrial domain knowledge is needed as well. Before implementing the tool in a real-world scenario, it should be evaluated on real-world bug datasets to get information about the behaviour and performance to serve as a knowledge base. The research questions contain both Quantitative and Qualitative methods [9], as the answers are given both with numbers and statistics, but also with a deeper analysis and answering of the questions. Therefore, the following research questions are suggested:

RQ1: *What is the effectiveness of template-based automatic program repair on different bug datasets?* This research question will provide an extensive and detailed evaluation of TBar on different bug datasets such as Bears and Saab bug dataset. The evaluation will not only present the number of bugs fixed but performance evaluation as well. This research question should serve as a base for the understanding of the behaviour of TBar and if it is suitable for use in an industrial context.

RQ2: *What are the characteristics of the generated patches by TBar?* The research question will provide a further understanding of the actual patches that TBar generates. A detailed evaluation of plausible and correct patches, overfitting, and the behaviour of fix templates.

RQ3: *How could template-based automatic program repair be integrated into the existing CI pipelines used at Saab?* With the received knowledge of the industrial context and CI pipelines, this research question will answer how Template-Based Automatic Program Repair can be integrated into Saab's existing environment. Hopefully, with at least one working solution, which could be used outside the scope of the thesis.

1.4 Contributions

- The design and implementation of kBar, an extended version of TBar suitable for standalone usage, and with other datasets than Defects4j.
- The industrial evaluation of kBar at Saab, on 14 inhouse bugs.
- The evaluation of kBar on the Bears bug dataset.

1.5 Outline

The rest of this thesis is structured as follows, chapter 2 contains a background of Automatic program repair, bug datasets used in research and examples from Continuous integration in a large scale industry project. Chapter 3 contains Related work on Automatic program repair tools and a description of them. Chapter 4 contains a detailed description of the tools and implementation of TBar and the modifications in kBar. Chapter 5 describes the experimental research methodology of this thesis and chapter 6 the results of the experiments. Finally, chapter 7 contains discussions, future work, threats to validity and chapter 8 a conclusion.

2 Background

This section will provide a background of Automatic Program Repair, Datasets for program repair and an example of Continuous Integration in the industry.

2.1 Automatic Program Repair

Automatic Program Repair (APR) [10] is a field that has gained much attention in recent years. The idea of repairing software bugs automatically, could save both time and money for software developers and companies. In the bibliography [1] by Martin Monperrus, one definition of Automatic Program Repair is described as *"Automatic program repair is the transformation of an unacceptable behaviour of a program execution into an acceptable one according to a specification"* [1]. A test suite decides the specification, and the result of executing the tests decides if the behaviour is acceptable. It is also essential to see if the repair does not cause new bugs to appear. The two main areas are presented as behavioural repair and runtime repair.

Behavioural repair - is used in this thesis and modifies the source code or binary code, both online or offline. The typical examples of online repair are within the IDE or Continuous Integration. The offline variants are repair attempts on software that are already used in production. One of the subjects that are mentioned in Behavioral repair is "repair templates", also called repair strategy, fix schema [1] but also fix pattern or fix template [6]. A simple example of a repair template could be inserting a null pointer checker. Multiple templates used together are called repair models [1].

Test-suite based repair is one of the areas in behavioural repair, and one of the first attempts of this was made already in the 1990s. The idea

of a program with at least one failing test case, the bug patch should make all test cases pass. Examples of approaches are modifying the Abstract Syntax Tree, where nodes are replaced with another node, arithmetic operators are replaced with operators from the same class or using repair templates [1].

Runtime repair - or state repair changes the execution state of a program. Some examples of this could be *"changing the input, the heap, the stack or the environment"* [1]. Input modification tries to change the input that fails the software, and Environment perturbation changes the memory or scheduling. More approaches mentioned are restarting the software, creating checkpoints and going back to them, having alternative program versions, reconfiguration by replacing services and other approaches such as creating objects to avoid null pointer exceptions [1].

2.1.1 Fault Localization in Automatic program repair

One crucial part of Automatic Program Repair is Fault Localization, which locates the error location in the code to provide it to the APR system. Fault Localization is conducted with the help of the software tests and could generate suspicious localizations at both file, method or line level. If the wrong line is selected, the APR tool could generate a plausible patch to a location with no bugs. Spectrum-based Fault Localization calculates a ranking metric and uses the *"execution traces of test cases to calculate the likelihood (based on suspiciousness scores) of program entities to be faulty"* [11]. The Ochiai ranking metric is *"one of the most effective techniques in localizing the root cause of faults in object-oriented programs"* [11], and the suspiciousness score is calculated with the help of the number of failed test cases and passed test cases [11].

Fault Localization investigation in kPar

Many new APR systems are presented and are using the same datasets. However, they are not using the same setup of Fault Localization. Therefore, Kiu et al. [11] investigates the different Fault Localization configurations in Automatic Program Repair together with the Defects4j benchmark. This is done by comparing performance and providing more understanding and real-world usefulness of Fault Localization in Automatic Program Repair. The investigation was conducted with kPar, a Java implementation of the PAR [6], described in Section 3.1.4. Many APR tools use the workflow of Fault Localization, Patch candidate generation and Patch validation. However, many APR approaches use different approaches for Patch candidate generation but similar methods for Fault Localization and patch validation. It is often conducted with test suites with GZoltar [12] as a testing framework and Ochai spectrum-based Fault Localization [13]. Patches could be considered valid when the tests pass, but the issue of overfitting patches is discussed, where non "semantically correct" patches pass the tests. Therefore, the terminology of plausible patches that pass the tests and correct patches equivalent to the original patch is presented [11].

The authors [11] highlight the importance that the Fault Localization techniques are considered when comparing different APR tools. One example mentioned is that SimFix and ACS were compared with different versions of the same Fault Localization tool without discussing the version's impact. Also, some comparisons are assuming Perfect Fault Localization, where the buggy code location is provided. The boost in performance of these assumptions should be considered when comparing with other tools. Therefore, the authors investigate different Fault Localization techniques, which bugs that could actually be located in Defects4j and how performance is affected by Fault Localization.

Experiments

The experiments was conducted with kPar, with six repair templates. Two different versions of GZoltar was used and different ranking metrics [11] such as Tarantula, Ochiai, DStar2, Barinel, Opt2, Muse and Jaccard. The results show that GZoltar version 1.6.0 finds 58 more bugs than GZoltar version 0.1.1, with the Line granularity level. Further, the ranking of Top-1 positions and Top-10 positions of different ranking metrics shows that Ochai performs well, and the results are similar. There is also a correlation between correctly fixed bugs and correct localization.

An in-depth investigation of APR tools jKali [10], jMutRepair [10], HDRepair [14], Nopol [15], ACS [16], ELIXIR [3], JAID [17], ssFix [18], CapGen [19], SketchFix [20], FixMiner [21], LSRepair [22] and SimFix [23] and Fault Localization was done. The results show that some of the APR tools report fixes for bugs, at line level, that their setup of Fault Localization and ranking metric does not correspond to. Some of the reasons for this could be improved versions of Fault Localization, assumptions that the faulty methods are known, and the usage of method level granularity. However, SketchFix, JAID and ELIXIR fixes some bugs which do not correspond to the configuration, which could be caused by a bug in the APR tool.

Further investigations by the authors [11] show that the test cases in Defects4j are not enough because changes in non-buggy code locations could lead to all test cases passing. When adjusting the results and only focusing on localizable bugs, the ranking of correctness of APR tools is adjusted, and the top five are FixMiner, CapGen, SketchFix, ACS and Elixir on Defects4j. Finally, kPar itself was evaluated with a three-hour repair timeout on the Defects4J dataset, and it did correctly repair 36 bugs.

Conclusions

The authors [11] drawn conclusions were that creators of APR tools should clearly state the protocol of Fault Localization and desirable come together with some standard. Alleged State-of-the-art tools must "qualify the performance gain" of their tool. There should be a distinct line, in research, between the Patch generation and Repair pipeline. Patch generation should use Perfect Fault Localization in the Repair pipeline. Normal Fault Localization should be used in order to prepare for a real-world scenario. Also, some tools focus on the context around the suspicious statement, heuristics and more which should be clearly stated.

2.1.2 Compile-time

A problem with the generate-and validate repair techniques presented within Template-based automatic program repair is the number of generated patches. The patch candidates are generated from a fix template, compiled and validated against a test suit. This process could take an extensive amount of time for certain projects, as the compilation of some projects could take up to one minute together with thousands of patch candidates. In the worst case, this could lead to many days for the repair process to go through all candidates [20].

2.1.3 Comparing Automatic program repair tools

There are many different techniques mentioned within automatic program repair, some closely related to template-based repair described in chapter 3. But there is difficulty deciding which of them are the best performing as they perform differently in execution time, number of patches, correct patches and more. Also, they are sometimes evaluated on different bug datasets with a non-equivalent setup as described in

Section 2.1.1 Fault Localization. Before TBar, SimFix was the best performing tool [6] on the Defects4j dataset. According to the evaluation in section 2.1.1 by a manual review, FixMiner, CapGen, SketchFix, ACS, and Elixir perform better than SimFix. So it is difficult to decide which tool is the best and there should be more extensive and identical evaluations conducted.

One of the new emerging techniques is Astor [24][10] which is used in research and facilitates equal comparisons and combines several state-of-the-art generate-and-validate repair techniques. Astor is a Java repair framework, which is based on repair approaches such as jGenprog, jKali, jMutRepair, DeepRepair, Cardumen and 3Sfix. The framework is publicly available, and researchers are encouraged to use it for *”setting up comparative evaluations and for exploring the design space for automatic program repair in Java”* [24]. Astor facilitates the implementation of new program repair approaches and to extend the existing ones. It has shown promising results on the Defects4J bug dataset, where it repairs more bugs than many other repair systems and 11 bugs that have never been repaired by another APR system.

2.2 Datasets for Program Repair

The publicly available bug datasets are a key ingredient in the research of Automatic Program Repair. With high-quality datasets, it is possible to compare and evaluate different tools for Automatic Program Repair and Fault Localization. Some of the considered datasets for this research are described in this section.

2.2.1 Defects4j

The Defects4j [25] Java bug dataset with, when the work of TBar was started, 395 bugs, consists of bugs and developer fixes and has been

widely used in research. There were 101 bugs that had been correctly fixed, and SimFix was the leading APR tool with 34 correct fixes [6]. The first version of Defects4j consisted of 357 bugs collected from five different real-world projects [26]. In addition to the actual bug and the developer fix, there is also a test suite that clearly demonstrates the bug, and the requirement is that at least one test case should fail on the bug, and it should succeed with the fix. The creators have provided a "high-level interface" to ease the process of compile, test, and download the bugs [26]. The approach of Defects4j is not to include other code changes such as refactoring and modifications, but only the bug itself. The following requirements were used when collecting the bugs of Defects4j:

- The bug is related to source code, where the bug commit is strictly labelled as a bug fix, not related to the build system, configuration or tests.
- The bug is reproducible, where at least one test fails and passes with and without the bug and can also run with the project build system and Java 7 runtime environment with OpenJDK.
- The bug is isolated; it does not include other changes than the bug itself.

Some of the projects initially collected were JFreeChart, Closure Compiler, Commons Math, Joda-Time and Commons Lang. The more current version of 2.0.0 consists of even more bugs and contains 835 bugs from 17 different projects, including Java 8 [27].

2.2.2 Bugs.jar

The large-scale dataset of Bugs.jar [28] contained 1158 bugs from 8 different Java open-source projects and was developed by the creators of Elixir [3]. One of the reasons for creating Bugs.jar was that, at the time,

there existed many diverse bug datasets for C, but Defects4j was the only major one for Java. Also, the projects in Defects4j were not considered to be diverse enough. One main criteria was that the Bugs.jar dataset should not contain bugs from Defects4j. Others were that the projects should be large and active, there should be no test cases with random behaviour for reproducibility. Also, the selected projects should have good development practices. The specific requirements when collecting the bugs from the projects were that:

- the buggy source code was available
- a description and a report of the considered bug
- a test suite that generates one failing test case with the bug and at least one passing test case.
- the patch from the developers does make all tests pass.
- the selected project should represent a typical Java project

2.2.3 Codeflaws

Codeflaws [29] is an extensive Bug dataset for C with 3902 defects and collected from over 7000 projects with 39 specific classes of bugs. One of the reasons for creating Codeflaws was that the existing bug datasets did not consider the classes of the bugs, also called defect classes. The authors propose a set of criteria for evaluation of the bug datasets for automatic program repair comparisons, which has been used in Codeflaws. There are five criteria:

- C1: High diversity of the defects
- C2: A large number of defects
- C3: A large number of programs
- C4: Programs that are algorithmically Complex

- C5: A large test suite

Over 10000 web pages were crawled, and 5544 bugs were collected, which further were manually reviewed and removed. The dataset and all scripts are available for future experiments for other researchers.

2.2.4 Bears

Bears [30] is a project which aim was to collect a large amount of Java bugs for APR studies. It was developed as an answer to the problems of existing benchmarks such as Defects4J or Bugs.jar. Some of the problems mentioned are that the bugs are collected from only a few projects, are sparsely updated and are not representative of real-world projects. BEARS uses Continuous Integration to identify bugs from open-source projects from GitHub and consists of 251 bugs from 72 different projects.

2.3 Continuous Integration

At Saab, the host company of this thesis, a study was conducted in a large-scale industry project by Mårtensson et al. [31] to determine which factors were affecting the developers when delivering code in Continuous Integration. Therefore the conclusions and the gained knowledge are important in the specific industrial context. The authors mention that a time-consuming build process should be avoided, as the developers could forget their implemented changes. One suggestion is that the build process should not take more than ten minutes, including testing. A long build and integration time could lead to increasing costs with a longer feedback loop.

Background

The study was conducted at a company developing a fighter aircraft before and after a new build system was introduced. The introduction of the new system took around six months up to three years, depending on the team. The pre-study did show that the old build system was slow, up to several hours, and all teams could not deliver software within one three-week sprint. The old build system was very complex and consisted of computers with customized hardware, partition communication was handled in specific time slots, and all processes needed to follow execution deadlines, i.e. hard real-time systems. The build system was considered the reason for not delivering software more continuously. The problems mentioned were a lack of transparency, problems with dependencies, and merge conflicts. Also, the developers had trouble understanding the build process, and it was referred to as "magic". The entire system was rebuilt even with minor modifications, and a lot of manual steps were needed, which led to many build failures and merge conflicts.

New build system

A new build system was developed to let all teams deliver software within one sprint, provide more transparency, and handle dependencies in a more predetermined way. The testing was included in the build system and no longer seen as a separate part. Only the affected parts of the system were rebuilt, and separate makefiles for each module were removed.

After the integration of the new build system, findings were that if a developer finds the system complicated, he or she will deliver less frequently. If the developers deliver less frequently, this will also lead to more problems due to unfamiliarity. Also, the less frequent delivery could lead to longer integration time, as more software from other de-

velopers may be needed to be rebuilt. Also, when encountering integration problems, the new system could be considered as problematic as the old one.

However, the authors argue that the build system should not be seen as the single motivation factor for developers to deliver more frequently. Other factors as salaries and working conditions should also be considered. Introducing the new build-system should be considered a success as it is no longer considered the main bottleneck but instead one of several factors affecting the delivery process. Future work of conducting studies at multiple companies is suggested and also investigate differences in technical and cultural factors.

3 Related work on Automatic program repair tools

As many Automatic program repair tools use or are closely related to repair templates, this section will go through some of them. Many of the tools are mentioned in the papers of "Template-Based Automated Program Repair tool" [6] and from the "The Living Review on Automated Program Repair" [32].

3.1 Template-based Automatic Program Repair

In template-based automatic program repair, there are many different approaches using sketches, hooks and bytecode mutation. Other approaches are the use of Machine learning and probabilistic models, collecting templates from GitHub or Stack Overflow, manually collected fix templates and the leverage of Abstract Syntax Trees. Many of them are described in the rest of section 3. In this section, the TBar tool is described.

The description of the TBar tool will be more extensive than other tools, as it is the main focus and is investigated in this thesis. Also, FlexiRepair, which is built on the idea of template-based program repair but targets the limitations of TBar, such as limited real-world usability, is described. Further, the API repair benchmark is described as it is a large-scale APR study, including TBar and Bears. It will also go through another template-based approach (PAR) and its critique, as it has been widely referenced in research and alleged a better result than another well-known tool GenProg. Finally, SOFix is described, as it tries to target the limitations of few fix templates in APR by mining StackOverflow for new templates.

3.1.1 Template-based automated program repair tool, TBar

TBar [6] is a *”Template-Based Automated Program Repair tool”* based on 15 different categories of fix templates. Examples of templates could be inserting a null check before a buggy statement, updating or removing a conditional expression or moving a buggy statement to another position. TBar is originally evaluated with the Defects4J [25] dataset, including buggy Java programs and fixes.

TBar was created to gain more knowledge of fix templates, fault localization and donor code retrieval. The authors conducted a survey of existing fix templates used by other tools and tried to summarize them in a presentable manner. The effectiveness of the templates is investigated, and how well they do generate patches. Also, the impact of Fault localization on repair effectiveness is investigated, as Perfect Fault Localization, where the bug location is known and Normal Fault Localization, where it is calculated with a probability.

The fix templates of TBar were manually collected from conferences, journals, a program repair website and literature related to Automatic Program Repair. The considered tools were focused on Java and were based on fix templates or closely related, i.e. using code change templates or rules. The selected tools were PAR, jMutRepair, NPEfix, Genesis, S3, ELIXIR, SketchFix, SOFix, FixMiner, REVISAR, AVATAR, HDRepair, ssFix, CapGen and SimFix and were released between 2009 to 2018. Some of the templates were also derived from previous work of manually gathering templates and mining static analysis violations. The authors manually inspected over 500 fix templates which resulted in choosing 35 Code Change Templates, gathered into 15 categories of fix templates shown in Table 3.1. The majority of fix templates do change a single statement, and a few can change multiple statements. The workflow of TBar is as following:

Template Name	Description	Template id
Insert Cast Checker	Insert a instanceof check before a statement containing a unchecked cast expression	FP1
Insert Null Pointer Checker	Insert a null check before a statement.	FP2
Insert Range Checker	Insert a range check for an access of an array or collection.	FP3
Insert Missed Statement	Insert a missing statement before, after or surrounding a statement. Statements are method invocation with an expression, return, try/catch or if.	FP4
Mutate Class Instance Creation	Replace a class instance creation with a super.clone() method invocation.	FP5
Mutate Conditional Expression	Mutate conditional expressions that return true or false by updating it, remove one condition or inserting a condition.	FP6
Mutate Data Type	Replace the data type in a variable declaration or cast expression with another.	FP7
Mutate Integer Division Operation	Mutates an integer division expression to return a float value.	FP8
Mutate Literal Expression	Mutates a boolean, number or string to another relevant literal or corresponding expression.	FP9
Mutate Method Invocation Expression	Mutates a method invocation expression by: 1) Replace the method name with another method with compatible return type and parameter types. 2) Replace at least one method argument with another compatible data type. 3) Remove arguments if the method has suitable overridden methods.	FP10
Mutate Operators	Mutate an operation expression by: 1) Replace one operator with another from the same class. 2) Change the priority of arithmetic operators. 3) Replace instanceof with inequality operators.	FP11
Mutate Return Statement	Replace the expression in a return statement with a compatible expression.	FP12
Mutate Variable	Replace a variable with a compatible variable or expression.	FP13
Move Statement	Move a statement to a new position.	FP14
Remove Buggy Statement	Removes a statement or the entire method.	FP15

Table 3.1: The details of Fix templates used in TBar [6].

Fault Localization - To find suspicious code locations where a repair attempt can be made, TBar needs to use Fault Localization. The GZoltar framework automates each program's testing, together with the Ochai ranking metric, to provide a ranked list of suspicious code locations. It does exist other methods for Fault Localization. Also, there are complementary techniques to improve further the fault localization results used in SimFix and ssFix.

Fix Template Selection - The list of suspicious code locations are iterated sequentially. TBar tries to match a fix template with each state-

ment in the list. This is done by traversing each node of the Abstract Syntax Tree (AST) of the suspicious statement and comparing it with the fix templates AST. If a fix template matches with a node, the template will be used for patch generation. Further, one statement can be matched with multiple templates when iterating through the child nodes.

Patch generation and Validation - The code is changed according to the matched fix templates and run against the test suit. The donor code for the templates is collected from the same file as the buggy project, and potential patches are ordered by the donor code's distance in the Abstract Syntax Tree. If all tests pass, a patch is considered plausible, and no more patches are generated for this bug. Other approaches do continue generating patches, but TBar tries to be suitable for a real-world Automatic Program Repair scenario. Finally, if the patch is validated as semantically equivalent to the patch of the bug dataset, it is considered correct. If a program contains multiple bugs and a patch candidate does pass a subset of the failing tests, it is considered a plausible sub-patch. TBar do then continue to evaluate more candidates until all patches are validated, the three-hour repair timeout is reached, or a plausible patch is found.

3.1.1.1 Evaluation

The evaluation of TBar was conducted on the Defects4j Java bug dataset with 395 bugs, consisting of bugs and developer fixes, because of its wide use in research. When the study was conducted, 101 bugs had been correctly fixed, and SimFix was the leading APR tool with 34 correct fixes.

TBar is first evaluated with Perfect Fault Localization, where the buggy

positions are known, to assess the contribution of fix templates. The measurements show that most of the bugs are correctly fixed by only one template, and a few bugs could be fixed by multiple templates. The templates Insert Range Checker, Insert Missed Statement with try/-catch, Mutate Class Instance Creation and Mutate Operators with instanceof to equality could not generate one plausible patch. The most common Change action was Update, Change granularity was Expression, and Change spread was Single line.

The conclusions drawn is that many fix templates can generate plausible patches. Further, it is crucial to choose a fix template that does not stop the execution with a plausible patch, where the correct one can be found with another template. The results show that TBar generates 74 correct and 101 plausibly fixed patches with Perfect Fault Localization. The plausible patches were manually examined in order to find the correct ones.

Further, TBar was evaluated in a real-world scenario with Normal Fault Localization. This resulted in 81 plausible patches, of which 43 were considered correct. The impact of Fault Localization led to some of the patches being plausible but in the incorrect position. This result outperforms other APR-tools such as SimFix that correctly fixes 34 bugs. The probability of generating plausible patches to be correct is lower than other tools such as CapGen, 84%, where TBar performs 53.1%. However, the lowest performing tool of jKali shows 4.5% probability.

3.1.1.3 Conclusion

Further improvements to TBar [6] could be collecting more fix templates, as some bugs cannot be repaired with the existing ones. Also, the search-space could be increased in order to find more donor code

and the right ingredients to the dictionary to produce more patches. This could be done both from the current project or other projects. However, the large search-space must be iterated in an effective way in order to save time. There is also a problem with choosing the wrong donor code, which could produce a plausible but not correct patch. The fix-templates must be prioritized and ordered in a way that the correct patch is found first.

The authors have demonstrated that many fix templates do contribute to fixing more bugs in APR. However, it does come with side effects as plausible fixes on the incorrect position. More work in the area is suggested, such as research in a database for fix templates, repair precision, fault localization, donor code retrieval and search optimization. The work is considered to be adaptable to other languages due to the use of Abstract Syntax Trees. Further, the tool should be evaluated on larger bug datasets such as Bears. TBar, and its promising results, should be seen as a baseline tool for helping other researchers to create even better approaches for Automatic Program Repair.

3.1.2 FlexiRepair

The need to *"express fix templates in a standard and reusable manner"* [7] resulted in the development of FlexiRepair [7], which is a repair framework that uses generic or equivalent terms of semantic patches. The idea is to explore the concept of generic patches and mentioned as fix templates in this work: a *"specification of transformation rules that can be given as a input to Coccinelle"* [7], Coccinelle is a code transformation tool that is included in the Linux kernel developer toolbox. FlexiRepair is built on the idea of template-based program repair but targets the limitations in current tools such as TBar as lack of support of adaption, extensions and real-world usability. The usage of Coccinelle sends a strong signal that it is ready to be used in the industry.

The individual user should have control over patch generation steps. The authors investigate where the repair transforms should be mined, how the fix templates should be inferred, and how the patches should be generated. The FlexiRepair pipeline is built on the steps of Miner, FlexiRepair, Inferrer and Generator.

Miner evaluates how similar the supplied patches are and creates clusters. The clustering is conducted with the help of Abstract Syntax Tree edit scripts to identify similar code changes. FlexiRepair uses the clusters to let the user decide how many code transformations should be used in the patch generation. Inferrer creates fix templates, generic patches from the clusters and makes them available for inspection and modification. This is done by finding the similarity in code fragments and control flows within the identified similar patches. Generator does create the actual patches for the buggy program by matching fix templates with the buggy code. The Coccinelle engine takes a generic patch and source code as input, matching the code locations with patches. This process should have an advantage over other approaches as it does not need specific bug localization tools with line localization of bugs; file-level is enough. Also, the generated patches are more likely to compile.

The user can easily decide which code source should be used with FlexiRepair for the mining of fix templates. The authors decided to use repositories manually and systematically collected from sources like GitHub and build activities in Travis. The dataset finally consisted of 351 repositories where the major ones have over 50 000 commits.

FlexiRepair was evaluated on the IntroClass and CodeFlaws bug datasets and did show a similar result on the IntroClass dataset and a significantly lower result on Codeflaws than other repair state-of-the-art tools such as GenProg. However, the authors do not aim to outperform cur-

rent state-of-the-art tools, but FlexiRepair should demonstrate the potential in generic patches.

3.1.3 API repair

There are few evaluations of TBar and Bears conducted in research. However, a Large-scale study was conducted, APIREPBENCH [33], with multiple tools and a mixture of datasets, including TBar and Bears. However, the combination of TBar and Bears was not investigated and chosen to match. Instead, there were results of time for TBar presented and fixed bugs from Bears dataset with other tools.

APIREPBENCH was created to be the first large-scale study on API-based Automatic Program Repair, where 14 different Java repair tools based on test-suites were evaluated on bugs with API misusages. Some of the selected tools were TBar, SIMFix, NPEFix, Avatar, NPEFIX and Astor. API misusages could be a missing method call sequence such as "java.io.InputStream.close" or missing an if condition for null check. The importance of automatic tools for API misusages is highlighted since the documentation of APIs is often not good enough to avoid misusages.

The authors did create their own benchmark APIREPBENCH for API misuse bugs and were collected from BEARS, Bugs.jar and MUBench bug datasets. The criteria for selecting bugs were that they should be Java-related, documented, publicly available, have an available test-suite, buildable, contain API misuse and be under active development. The collection from Bears.jar and Bears were done with MUBench as an inspiration, and in total, 101 buggy project revisions were collected, and 89 of them were unique. An execution framework was set up and called APIARTy and configured to work with the different Java repair tools. The experiments were conducted with Avatar, TBar and SIMFix

on 12 bugs, not including Bears and Bugs.jar, and the remaining 11 tools on all of the 89 bugs. The repair timeout on Avatar, TBar and Simfix was set to 4 hours, and all tests took 36 days to run. The results show that when summarizing all tools, they generate patches for 25 of all 89 bugs. There are 1015 repair attempts made, and 80 of them generates patches, where 52 is plausible, and 20 are semantically correct. The median and average time of all repair attempts were 240 and 166,19 minutes of Avatar, TBar and Simfix, where TBar did show values of 240 and 203,5 minutes. In contrast, the 11 other tools did show a median of 3,87 and an average of 30,79 minutes, whereas the DynaMoth did show the lowest result with 2.08 and 21.61 minutes.

The reasons for not generating patches are discussed. It could be that the repair tools are not designed to handle the specific bug, i.e. importing an API or bugs that are located in multiple locations. The fault localization could be incorrect, and the specific bug position is not detected, and more precise fault localization is suggested. Also, a higher repair timeout value could increase the number of patches. The configuration of projects with dependencies and compilation errors could also be a source to failure, and 90 repair attempts failed due to these reasons. Finally, other reasons such as exceptions in GZoltar fault localization, OutOfMemoryErrors, processes that hang, and these do occur in 175 repair attempts.

3.1.4 Another template-based approach, PAR

Pattern-based Automatic program Repair (PAR) [34] was created in order to target the limitations of approaches such as GenProg, which *“relies on random program mutations such as statement addition, replacement, and removal”* [34]. The limitations are that the random behaviour could generate fixes that are not accepted by developers and completely wrong, even though they pass the tests.

The authors of PAR manually generated ten fix templates from over 60 thousand fixes from open source projects. The process of generating templates was done by using a *"graph-based model for object usage"* to find semantic differences between the nodes before and after the fix. The patches with corresponding differences were grouped together. Finally, the groups were manually inspected to find the actual fix templates to use in PAR. The fix templates replace parameters, replace methods, add and remove parameters, replace expressions, add and remove expressions, check for null pointers, initialize objects, check range, check the size of collections and check class-casting. These fix templates could be connected to around 30% of the collected patches.

The workflow of PAR starts with Fault localization, where suspicious code locations are found and ranked with the help of the test suite. Fix template and patch candidate generation, where suitable fix templates are found by comparing the AST-tree of the code and then applied. The Patch Evaluation is done by a fitness function that compares different versions of the program with including the help of the number of passing test cases.

PAR was evaluated on over 100 bugs from open source projects such as Apache log4j, Rhino and AspectJ. The evaluation was manually done by students and developers, with the question if they would accept the fix. On the same dataset PAR created 27 patches and GenProg only 16.

3.1.4.1 Critique to PAR and Fix acceptability

In the paper, A Critical Review of Automatic Patch Generation Learned from Human-Written Patches: Essay on the Problem Statement and the Evaluation of Automatic Software Repair [35]. There is a respectful critique given to the PAR approach by Monperrus et. al. . One exam-

ple mentioned is the lack of information about the defect classes PAR is supposed to address. A defect class could be null pointer exceptions, changing a method call and more. There is no clear principle of the selection of fix templates. A major contribution to the field of APR could be providing templates for common defect classes, but not for uncommon ones. Further, the conclusion that PAR generates more fixes than GenProg is negligible since it is unclear if they target the same defect classes. Also, the evaluation done by students and developers is considered weak, as some bugs would need extensive domain knowledge of the environment to be considered acceptable or not.

Evaluation criteria of Automatic Program Repair such as Understandability, Correctness and Completeness is discussed. The criteria should be handled differently in a Fully Automatic System, which repairs bugs without human interaction, and Recommendation Systems, where a human conducts the repair.

Understandability - In the automatic system, a generated patch does not need to be fully documented as the patch is generated to fix the bug without human interaction. On the other hand, the recommendation system needs to be understandable by the human that is supposed to implement it.

Correctness - In order to be an automatic system, it needs to generate a correct patch. However, the recommendation system could present a partially correct patch that the human could modify and then apply.

Completeness - Which is similar to Correctness, where a patch should be totally complete in the automatic system. Even an incomplete patch with a sketch could give the developer insights to produce a patch in the recommendation system.

Further, Fix Acceptability is discussed, and it is difficult to decide if one fix is better than another. If an Automatic Program Repair tool produces $x==2$ and $x<=2$ as patches and both pass the tests, it could be challenging to decide which one is the better.

3.1.5 SOFix

SOFix [36] was created as further refining of Automatic Program Repair with fix templates. The authors describe the critique and limitations of real-world usability of current automatic program repair approaches with few fix templates. They highlight the fact that the successful and well worked out approach PAR only generates ten fix templates. Also, better methods of fault localization are desirable due to the relationship between fault ranking and repair time.

With SOFix, the authors try to mine more fix templates from Stack Overflow to leverage the large community and codebase. However, there are challenges when collecting data from a large dataset of 30 million posts and choosing the correct code, with small-scale granularity at the variable level. As a response to the challenges, only Stack overflow threads with bugs and correct patches were chosen. Stack overflow posts are stored in XML format, and only the question and accepted answer were collected. Further, an Abstract Syntax Tree was used to help mine templates from the differences between the modification and original code.

The mining resulted in 136 templates, which generated 13 repair templates after manual evaluation, of which two were completely new in APR. The following Templates was derived:

- BinaryOperator Replacer: replace operators in an if, for or while statement.
- Variable Replacer: Changes one or several many equal variables

to another of the same type.

- **Type Replacer:** Modifies the type in a local variable declaration.
- **Arguments Adder, Mover, Replacer and Remover:** adds, moves, replaces or removes variables in a method call.
- **Invocation Replacer:** change a method call to another with compatible arguments.
- **BinaryOperator Inversion:** changes the priority of two connected operators.
- **Variable to invocation:** changes a method call within a variable read by another similar method.
- **Return Statement Adder:** Inserts a return statement.
- **Statement Remover:** Deletes a statement.
- **If Checker Adder:** Inserts an if statement for checking null pointers.

SOFix was evaluated on the Defects4J dataset, where it successfully repaired 23 bugs.

3.2 Sketches, Hooks and Bytecode Mutation

3.2.1 NPEFix

There was considered a lack of program repair techniques focusing specifically on null pointer exceptions. Therefore, the NPEFix [37] technique is proposed, which has nine repair strategies or fix templates for handling null pointer exceptions. The main idea that separates NPEfix from other template-based approaches is that it does find values for patches during runtime. The implementation uses metaprogramming or code transformation in order to apply all repair strategies.

The two main approaches are replacing the null value and avoiding the execution of the specific code. In more detail, the repair strategies based on null replacement could be as follows, Reuse: a variable is replaced with another global variable or replace the null variable without changing it, i.e. with an if/else statement. Creation: a new global or local object is created. The repair strategies of avoiding execution are: Line skipping, where a value is checked to be not null with an if statement in order to skip the line otherwise. Return Null, New Object or Variable: if a method should return a value null, another variable or a New Object is returned if a null check is true. Vanille return: Similar to Return Null, but if a null check is true, a "return;" is returned.

A simple Template Based version of NPE, TemplateNPE is presented, which follows the structure of ordinary template base repair. It uses source code transformation to go through all possible templates for patches. The drawbacks are that the search space is not as large as the metaprogramming version, and it generates one compiled file for each patch. The metaprogramming version, NPEfix, uses hooks, a boolean that is not activated by default. A hook controls the modification of the code without creating new files for each template. The phases of NPEfix, are the generation of the metaprogram, compilation and running the failing test cases with different alternatives with hooks.

The results show that the NPEfix version finds 68 more patches than the TemplateNPE version on a bug dataset; however, the TemplateNPE is faster in most cases. The evaluation was done on 16 real-world bugs with null pointer exceptions.

3.2.2 SKETCHFIX

SKETCHFIX [20] was introduced to avoid the time-consuming compilation and testing of generate-and-validate repair techniques. The

solution presented here was to avoid the repeat of compilation and to insert "sketches" instead. One sketch is possible to correspond to thousands of patch candidates. With the help of a test suite, "holes" are created in the suspicious code locations. Further, a sketch engine is used to fill in the holes with sketches which is suitable for the corresponding code. Minor code modifications or ones that are closer to the original code are prioritized, and changes are done at the Abstract Syntax Tree node level. This is done in order to make the human developer more likely to accept the modification. The patches are generated during runtime, and when it "encounters runtime exceptions or test failures, it backtracks immediately and fetches for the next choice". When a generated program passes all tests, it is considered as repair for the buggy program.

On the JFreeChart project, it would take 15 days to test all patch candidates with ordinary compilations; SKETCHFIX finds a solution with one compilation in 40 seconds. On the Defects4j benchmark, it finds 19 correct and seven plausible fixes of the 357 with an average of 23 minutes, 9 minutes for sketch generation.

3.2.3 PraPR

PraPR (Practical Program Repair) [38] is an Automatic Program Repair tool developed to investigate bytecode mutation in APR. A simple bytecode mutator is changing $a > b$ to $a < b$. Currently, there has been no APR research focusing only on bytecode mutation. Due to the JVM bytecode mutation, PraPR does not need to compile to validate patches, which is an advantage against many other APR techniques. PraPR is up to 26 times faster than another tool CapGen, on the same dataset. On one bug, PraPR validated almost 36 thousand patches in one hour. Other benefits of modifying at the JVM-level are that the tool could be used to generate fixes for other JVM-based languages such as Kotlin,

Scala and Groovy.

PraPR *”supports Maven-based Java and Kotlin projects with JUnit, or TestNG, test suites”* [38] and uses the Ochai framework for fault localization. It generates patch candidates for all suspicious localizations and validates with the tests. Further, it does check if a generated patch is covered by the failing tests. If the control shows that it is not covered, the patch will not be considered. Only the passing tests covering the patched location are checked to save more time and finally generate a Plausible patch. PraPR uses the mutators of the state-of-the-art mutation engine PIT and further extends some of them with more alternatives, such as replacing `>` with `=`. The mutators are:

- Arithmetic Operator: replace an arithmetic operator such as `+` to `*` and more operators as deletion, which is conducted by removing operands from the JVM stack.
- Conditional: replace conditional operators such as `<` and `!=`.
- Dereference Guard: tries to prevent `NullPointerException`s by returning default values, values of local variables, values of compatible fields and more.
- Method Guard: adds a check to a method to avoid `NullPointerException`s.
- Pre/Post Condition: adds `NullPointerException` checks for objects and method returns.
- Field Name: mutates field access instructions by selecting another field with the same type, such as `static` or `public`.
- Method name: selects another method.
- Argument list: selects a method with the same name but with different parameters.

- Local Variable: replaces a variable with another local variable.
- Accessor: replaces read and write access to fields in order to avoid race conditions in concurrent programs.
- Case Braker: tries to fix a missing break or return statements in switch-case statements.

PraPR was evaluated on the Defects4j dataset, and the results show that with the basic bytecode mutators, it produced correct fixes for 18 bugs, and with extended mutators it generated 43. The authors also motivate the plausible patch candidates of PraPR to support developers in fixing the bug manually.

3.3 Other Abstract Syntax Tree Based Repair Tools

Many of the described tools in section 3 includes Abstract Syntax Trees. Genesis and Avatar and FixMiner are other tools which gathers fix templates and generates templates with the help of AST.

3.3.1 Avatar

Avatar [39] is an Automatic Program Repair system that uses fix templates from static bug detection tools to generate patches. *“Static analysis tools help developers check for common programming errors in software systems. The targeted errors include syntactic defects, security vulnerabilities, performance issues, and bad programming practices. These tools are qualified as “static” because they do not require dynamic execution traces to find bugs”* [39]. The reason for using static tools are the challenges in collecting bug fixes from repositories in order to create fix templates. Some challenges mentioned are finding varying bugs and excluding irrelevant bug fixes.

Also, the dangers of using test suites to decide both bug localization and

if a patch is correct are discussed. This is due to the varying quality of test cases, as a poorly written test case could lead to the generating of patches that seem to be correct but introduce new errors instead. By using bugs fixed by static analysis violations, Avatar aims to get more consistent fix templates. Static analysis tools could have a hard time finding other bugs in, e.g. Defects4j dataset. However, their derived fix templates could be useful in Automatic Program Repair.

The Avatar project does not gather fix templates itself but uses 13 fix templates from different projects. However, the process of mining fix templates is done with the first step of Data Collection from open source projects. This is done by running static detection tools on each revision of a program and collecting fixes related to the violations. The second step of Data Preprocessing is done with the help of Abstract Syntax Trees or Git diffs to generate edit scripts. Finally, the Fix Template Mining is done by gathering scripts similar to each other and generating a fix template.

Avatar itself consists of Fault Localization with GZoltar and Ochai ranking metric, followed by Fix template matching with an Abstract Syntax Tree. Patch Generation is done by applying the actions from the fix templates. Finally, the patch is validated with a test suite. Avatar was evaluated on the Defects4j bug dataset and generated 34 correct patches, some of which had never been fixed before.

3.3.2 FixMiner

FixMiner [21] is a systematic approach of automatically mining fix templates to use in Automatic Program Repair with an *“iterative and three-fold clustering strategy, to discover relevant fix templates automatically from atomic changes within real-world developer fixes”* [21]. Further, it uses *“Rich Edit Script which is a specialized tree data struc-*

ture of the edit scripts that captures the AST-level context of code changes. To infer templates, FixMiner leverages identical trees, which are computed based on the following information encoded in Rich Edit Scripts for each round of the iteration: abstract syntax tree, edit actions tree, and code context tree.” [21].

The idea is that it should be usable with other patch generation systems, and the authors mined fix templates from over 11 thousand patches from different Java open source projects. The top 50 fix templates of FixMiner were compared to the templates in TBar, and 16 templates were compatible.

The results show that FixMiner, with 31 fix templates integrated with PAR, PARFixminer, did create correct fixes for 26 bugs in the Defects4J dataset, where six had never been fixed before. The correctness of the plausible patches was 81% which could be compared to around 60% of Elixir and SimFix.

3.3.3 Genesis

Genesis [32][40] is another APR system that uses repair schemas and Abstract Syntax Tree to generate new patches. It *”works with hundreds to over a thousand candidate transforms to obtain productive search spaces generated by tens to over a hundred selected transforms — many more transforms than any previous generate and validate system”*. *”In comparison with previous manually developed transforms, the Genesis transforms are more numerous, more diverse, and target a wider range of defects more precisely and tractably” [40] .*

Genesis was compared to PAR and generated correct patches on a dataset of Null Pointer Exceptions, Out of bounds and Class cast bugs collected from 356 open source applications. The results show that Genesis generates correct patches to 21 of 49 defects and PAR only 11.

3.3.4 DevReplay

DevReplay [41] is a Static analysis tool that is developed to fix source code violations in general and check specific conventions of the current project. DevReplay is described as *"generate code change templates by mining the code change history, and we recommend changes using the matched templates. Using DevReplay, developers can automatically detect and fix project/API-specific problems in the code editor and code review."* [41]. There is no need for test cases, as in traditional Template Based Repair tools, and it does create fix templates from previous git commits and targets one line changes. The tool is easy to use and generates a string as a suggested patch but does not check if new errors are created. The algorithm is inspired by Elixir, Avatar, PAR and other APR and static analysis tools. Typical examples of usages are "maintaining source code consistency" and "language migrations fixes".

The general workflow of DevReplay is Extracting, Matching, Prioritize, and Suggestion. Extracting code change templates is conducted by creating an Abstract Syntax Trees of previous git commit history of one month. The ASTs are created from different git versions of the code, as pre-changed and changed. Further, a changing template is created from different versions with the help of TextMate. TextMate is a syntax suitable for auto-complete in programming editors or writing change templates. The Matching of a template with code contents is done by comparing the source code and generated change templates and saving all matching templates for the next step. Further, the step of Prioritize templates by changing the date is done by ordering templates, as the correct one is more likely to appear closer in time. Finally, the Suggestion of patches is done by a command line user interface, code editor plugin or GitHub code review bot.

The difference between DevReplay and other tools, such as FixMiner, is that it does also consider non-buggy code when creating change templates. DevReplay was evaluated on two C benchmark data sets of CodeFlaws and IntroClass and compared to state-of-the-art tools of Angelix, CVC4, Enum and Semfix. However, it is not evaluated on a large bug dataset such as Defects4j. The results show that DevReplay does repair more bugs than the state-of-the-art tools, and its proposed real-world open-source patches were accepted by 80%.

3.4 Repair Tools with Machine learning

As Machine learning is an emerging technology within Artificial Intelligence, this section will go through some APR tools that leverage machine learning techniques.

3.4.1 R-hero

Techniques based on machine learning are showing promising results on different bug datasets. A new interesting technique is called R-Hero [42], which makes use of Continual Learning and analyzes and collects commits from Continuous Integration, which makes a build pass. These commits are single-line changes and could both be a bug fix or other changes. The changes in the commits are used as training data to a machine learning model, which generates suggestions for new patches from buggy code. The repair in R-hero is conducted by monitoring the Continuous Integration and checking out a failing build. Further, the Fault Localization process generates possible faulty locations, and the machine learning model generates one or many patches that makes the build pass. The generated patches are checked with an Overfitting detection system (ODS) based on a probabilistic model with supervised learning to avoid overfitting. When a suitable patch is de-

tected, a pull request is created to the corresponding GitHub project with a description of the patch and build failure.

The current state of R-hero is that it collected 550000 commits that have been used for training the machine learning model. Further, it has been evaluated on 44002 failing builds. However, to evaluate if R-hero did fix the bug, the final evaluation had to be conducted manually. The results show that it has already submitted a correct patch to an open-source variant of the AlphaGo Zero project, which the developers accepted. It has also been evaluated on the CodRep4 bug dataset with around 4000 bugs, producing over 600 correct patches. However, some limitations are that R-hero does focus on one-line bug fixes, it has not yet produced fixes for a large amount of continuous integration builds and the machine learning model could forget previously learned fixes. R-hero should be seen as a *”milestone in showing that developers and bots can cooperate fruitfully to produce high-quality, reliable software systems”* [42] and hopefully lead as an inspiration to future contributions with APR bots based on Continuous Learning.

3.4.2 Elixir

Elixir: Effective Object-Oriented Program Repair [3], was developed to respond to the fact that most program repair tools were developed for non-object-oriented programming languages such as C and sparse or no utilization of method calls. In an object-oriented language as Java, it is common to use method calls to objects, and one study conducted by the authors [3], did show that up to 57% of the statements in a program of some projects were method calls. Also, around 77% of the one-line bug fixes changed or inserted code in a method call. A similar study, conducted by the authors [3], did show that analyzed C code only had around 33% method calls. The authors discuss that other APR-techniques, such as PAR, change the method calls but only replace the

call with code from the same method. The reason for this is the large search space for exploring candidates outside the method.

Elixir uses method calls extensively and tries to leverage "*local variables, fields and constants*" [3] to generate more patches. In order to reduce the search space of patch candidates, a machine learning model is used to assess the generated patches. Elixir consists of four different parts: Bug localization, Generation of Candidate Patches, Ranking and Selection, and Validation. Bug localization is conducted using the Ochai bug localization framework and collecting statements to rank suspicious code locations. Generation of Candidate Patches uses eight different Program Transformation Schemas which are:

- Widening type: replaces variable type in a declaration such as an int to double.
- Changing expression in Return Statement: replace a return expression with a compatible type
- Checking Null Pointer: adds an if guard to an object reference to avoid null object access,
- Checking Array Range and Collection Size: adds an if guard to check that accesses are within a valid range,
- Changing Infix Boolean Operator: Changes a boolean operator such as $a > b$ to $a < b$ and other variants.
- Loosening and Tightening Boolean Expression: removes or adds predicates in an if condition or return statement.
- Changing Method Call: replace object references with other compatible expressions, replacing method name, replacing method arguments with compatible types, replacing the entire method with an overloaded or synthesized method call, which corresponds

to a sort of adapter class for a given replacement class [43],

- Insertion of a Method Invocation: Synthesizes a method call which is inserted as an expression or statement.

So, *"ELIXIR extracts all the local variables and literals in scope, fields in the same class, and all the public fields in other classes that are relevant to the buggy class"* [3] and these items serve as a base for patch generation. The patches are generated according to the transformation schemas, where the ones with method calls usually result in the most considerable amount of candidates. The Ranking and Selection of Candidate Patches are conducted by logistic regression. This common machine learning technique uses the program context and bug report to generate a probability score of the selected patch candidate.

The results of the evaluation of Elixir shows that it generates more correct patches than other state-of-the-art approaches such as PAR. On the Defects4J and Bugs.jar datasets, it correctly fixes 26 and 22 patches, respectively.

3.4.3 Cardumen mode of Astor

The Cardumen [44] mode of Astor is a generate-and-validate repair approach which mines templates from the code which is being repaired. This approach of generating new templates from the buggy code and inserting variables during repair is unique. A probability model is used to order and speed up iteration through the generated patch candidates.

The first step of Cardumen is to reduce the search space of code by using GZoltar Fault Localization to find and calculate values for suspicious code locations. The code localizations above a specific value is chosen and ordered. Modification points are code identified in the Fault Localization that could be suitable for repair. The Cardumen ap-

proach of code Modification is flexible in which code element it could change. The specific code has a code type with a corresponding root node in an AST tree. Further, the modification specifically considers the return types of code elements such as Integers or Booleans. The creation of modification points is done by filtering the AST nodes and creates one point per node. In the example: *"(a > b) && ((d - e) > 3)*, Cardumen creates four modification points: one for reference to the whole Boolean expression, the other for the Boolean expression *(a > b)*, the third one for a Boolean expression *(d - e) > 3*, and the last one for the Integer expression *(d - e)*." [44]. When a list of Modification points is gathered, the actual template is created by traversing the AST and replacing variables with a placeholder. In the following example, *a > b* will be transformed to *int1 > int2*.

The repair process is conducted by selecting a random modification point with the calculated suspicious value from Fault Localization. A list of compatible templates is generated with the help of a compatibility filter and location filter. The compatibility filter chooses templates with the same return type as the original code, and the location filter does select templates from the same file, files in the package or from all statements. In this way, the search space could be reduced, and a template is then selected with random weighted selection. The template is then instantiated, and this could result in thousands of different instances. To reduce the number of instances, a probabilistic model on the variable number of occurrences is used to calculate the probability of variables appearing together in a statement. Also, a localness probability value is added to the model and instances with high probability are chosen. Finally, one instance is selected with weighted random selection and inserted to the modification point. The modified program is then evaluated with the help of the test-suite.

The results of Astor-Cardumen repair shows that it generates almost

9000 plausible patches on 77 bugs of the Defects4j dataset, which is more than any other repair system. It can generate patches at different locations in a bug and does so in over 50% of the plausible patches. Also, eight new patches which have never been identified before were created. The generated patches were made publically available to the community of Automatic Program Repair for further research.

3.4.4 RITE a type error reporting tool

One problem with programming languages that uses "Hindley-Milner" style is the compiler's complicated error messages. To ease the interpretation process, attempts have been made to locate the error, but there is a lack of support for the repair tools. Sakkas et al. introduce RITE [45], "a type error reporting tool", for the programming language OCaml. The used repair strategy is called Analytic Program Repair and uses supervised learning instead of manually collected templates used by many other APR tools.

It extracts potential repair templates from a training dataset, predicts suitable templates by training a multi-class classifier and finally creates and ranks patches with the help of the templates. The tool is suitable for type errors, as the repair space, in general, is large for these errors, and the goal *"is to use historical data of how programmers have fixed similar errors in their programs to automatically and rapidly guide novices to come up with candidate solutions"* [45].

RITE does represent fixes using Generic Abstract Syntax Trees (GAST), which collects data from Abstract Syntax Trees, but removes information at a specific depth and then replaces it with holes. This will give *"information about a fix's structure rather than the specific changes in variables and functions"* [45] and one example mentioned is when `var [a * var b]` becomes `[_ binary operator _]`.

The dataset of fixes was gathered from programs fixed by students, and especially the interaction traces was utilized. A special compiler gathers traces in all the students' sequence of fixes. It checks the differences in their Abstract Syntax Trees, finally creating a fix label: "the smallest sub-tree that changed between the correct and ill-typed attempt of the program". Fix templates are selected by using the GAST to identify the most general fix templates by their similarity. To create predictions of error location and which fix template to use, two different machine learning approaches of Binary classification and Supervised Multi-Class Classification are used. Further, candidates are created for the generated error locations, and a ranked list is returned to the user. Also, RITE does not only support single line errors but multiple location errors as well by combining their score.

The solution is evaluated on 4500 OCaml programs, from a programming course, with errors. When using three repair templates, RITE chooses the correct template in 69% of the cases and 80% when increasing to six templates. The repair time is up to 20 seconds in 70% of the cases, and in a user study, RITE shows better results than the state-of-the-art tool SEMINAL [46].

4 Technical contribution

This section will describe the original implementation of TBar and the modifications that were made in kBar.

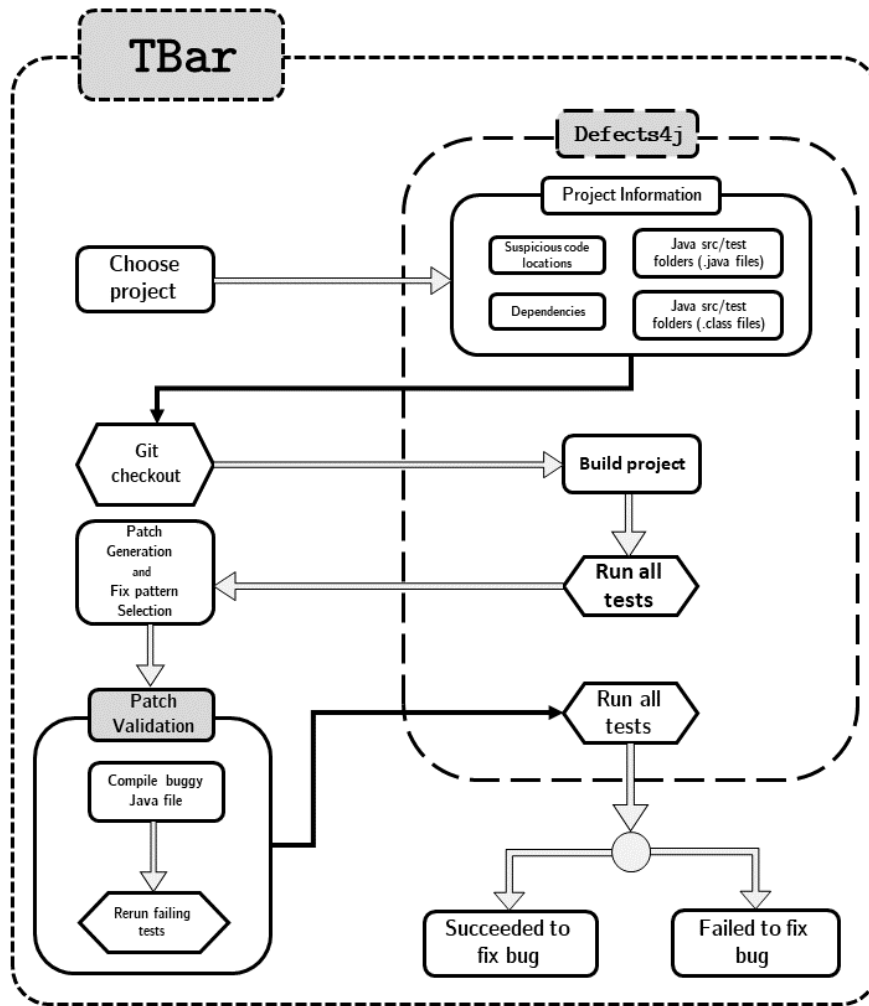


Figure 4.1: A simplified and generic flowchart of the original implementation of TBar.

4.1 Design of TBar

The original implementation of TBar consisting of over 12 000 lines of code was closely integrated into the Defects4j bug dataset. The integration is shown in Figure 4.1 and described in this chapter.

4.1.1 Initialization, build and test project

The user selects the desirable project in the command line, and TBar initializes the needed project information from Defects4j. Project information includes Suspicious code locations, Java classpaths, Java source and test folder, Project location and more. The buggy project is then checked out with git and compiled with the Defects4j default compiler. Further, the tests of the buggy project are executed with the Defects4j default test command, and the failure or error tests are counted and collected. The tests are needed in the future step of Patch validation.

4.1.2 Patch generation and Fix template selection

Suspicious code locations and Context information

TBar reads the suspicious code positions from a pre-generated text file by the suspected line number in the .java file shown in Figure 4.2, provided by the GZoltar framework.

```
org.apache.commons.lang3.ClassUtils@903  
org.apache.commons.lang3.ClassUtils@904  
org.apache.commons.lang3.ClassUtils@905  
org.apache.commons.lang3.ClassUtils@906
```

Figure 4.2: Pre-generated text file from the GZoltar framework with suspicious code locations.

```

if (parser.hasNext(4)) {
    position.setNetwork(new Network(CellTower.from(
        parser.nextInt(), parser.nextInt(),
        parser.nextInt(16), parser.nextInt(16),
        parser.nextInt())));
}

```

Figure 4.3: The entire if statement identified as the Suspicious code node by TBar, where the original bug Bears-98 is marked in red.

```

25 = IfStatement
32 = MethodInvocation
42 = SimpleName
34 = NumberLiteral

```

Figure 4.4: Gathered Context Information from the Suspicious code node in Figure 4.3, with corresponding Integers.

The java files with the suspicious code are then parsed into a List of suspicious code nodes shown in Figure 4.3. Context Information, shown in Figure 4.4, is then gathered and corresponds to an Integer matched with IfStatements, MethodInvocations and more, which is used by the fix templates.

Matching with Fix templates

The Context Information of each node is iterated through and compared to each fix template. If a fix template matches the Context Information of the suspicious code node, the template is used for the generation and validation of patches. In the example in Figure 4.5, the Fix Template ConditionalExpressionMutator matches with the Context Information IfStatement and generates 12 Patch Candidates.

Dictionary of fix ingredients

A Dictionary of fix ingredients is created to serve the fix templates with ingredients for Patch Generation, where examples are shown in Fig-

```

- if (parser.hasNext(4)) {
+ if ((parser.hasNext(4)) || (parser.hasNext())) {
+ if ((parser.hasNext(4)) && (parser.hasNext())) {

```

Figure 4.5: Bug Bears-98 , two of the 12 generated Patch Candidates when Context Information number 25 IfStatement is matched with fix template ConditionalExpressionMutator.

Figure 4.6. In detail, a Dictionary in TBar is a class containing multiple HashMaps where a key corresponds to the classpath and the values of Variables, Dependencies, Methods or Superclasses. The Dictionary is generated from the .java file where the bug is located to reduce the search space. However, there would be minor required changes in the code to expand the search space. The Dictionary also contains helper methods for the fix templates to get access to the desired values.

```

Fields with variables:
PRIVATE Pattern PATTERN
PRIVATE Pattern PATTERN_ITEM
PRIVATE Pattern PATTERN_OLD

Imported dependencies:
org.traccar.protocol.NavigilFrameDecoder
org.traccar.protocol.H02FrameDecoder
org.traccar.protocol.SkypatrolProtocolDecoder
.
.

Methods:
GoSafeProtocolDecoder
decodePosition
decode

Superclasses:
org.traccar.BaseProtocolDecoder

```

Figure 4.6: Examples of the 338 Dictionary ingredients for Bears-98 bug.

4.1.3 Patch validation and Run all tests

The validation of the generated patches is conducted by executing only the failing tests from the step of "Run all tests", this is done by first compiling only the modified.java file. Then if the compile succeeds, only the individual failing test is executed with java, and finally, if the test succeeds, all tests are rerun. The idea of avoiding the execution of all tests saves time. If all tests pass and no new failures are generated, TBar generates the message "Succeeded to fix bug" and a plausible patch is presented, as shown in Figure 4.7.

```
--- a/src/org/traccar/protocol/GoSafeProtocolDecoder.java
+++ b/src/org/traccar/protocol/GoSafeProtocolDecoder.java

-156,7 +156,7 public class GoSafeProtocolDecoder extends
    BaseProtocolDecoder {
        position.set(Position.KEY_HDOP, parser.next());
-       if (parser.hasNext(4)) {
+       if ((parser.hasNext(4)) || (parser.hasNext())) {
            position.setNetwork(new Network(CellTower.from(
                parser.nextInt(), parser.nextInt(),
                parser.nextInt(16),
                parser.nextInt(16),
                parser.nextInt())));
        }

TBar: Finish off fixing == Succeeded to fix bug Bears-98
```

Figure 4.7: The output of TBar when "Succeeded to fix bug" and a plausible patch is presented.

4.2 Extensions of TBar in kBar

As a response to the close integration with the Defects4J bug dataset and using TBar with other projects, TBar was modified into a standalone version kBar - “kungliga TBar” as shown in Figure 4.8. This was done in the scope of this thesis in order to use the tool for experiments. Some parts of the close integration in TBar were that the Fault Localization process in TBar had already been conducted and stored in text files, which was read by TBar. It was not integrated into the workflow and the Fault Localization needed to be executed before TBar. Further, the compilation of projects and running of all tests were only adapted for the Defects4j dataset. The classpaths and dependencies were also prepared for Defects4j and the interpretation of tests results. The general modifications are available online ¹. A short description of the company specific modifications of kBar are described in Section 6.3.

4.2.1 Domain knowledge

In order to configure kBar for different projects, there is a need for domain knowledge about the project and its context. The setup process does assume knowledge of classpaths, dependencies, class folders, java src folders, test folders and some knowledge about the configuration options in kBar and the output of the specific buggy project.

¹<https://github.com/gynther-k/kbar/>

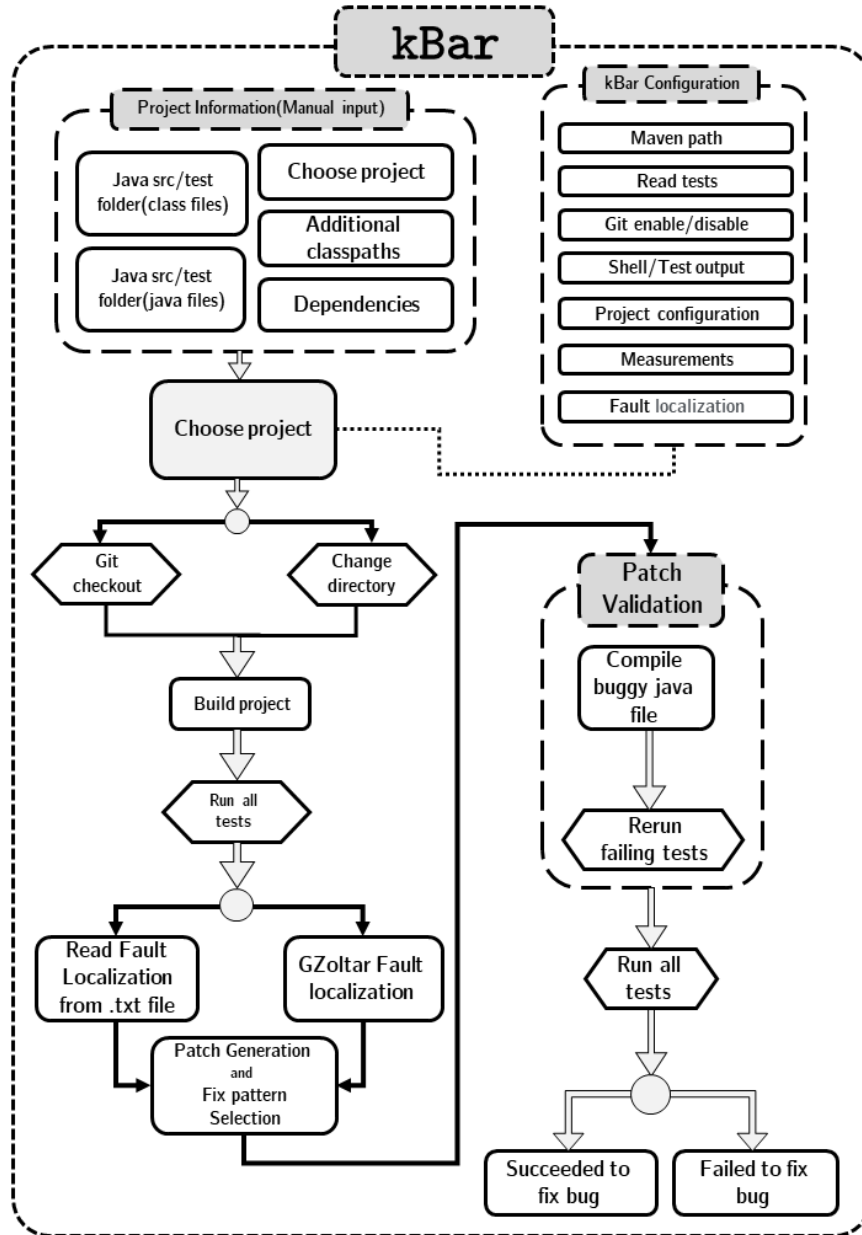


Figure 4.8: A simplified and generic flowchart of the modified version of kBar - “kunliga TBar”.

4.2.2 Modifications and arguments

Modifications were done in order to remove the hardcoded paths and specific Defects4j bug dataset configurations, and as a result of these

modifications, the following parts to kBar were added.

Project Information

The arguments of *"bug data path"*, *"bug id"* and *"defects4j project"* folder were the original input arguments to TBar. In the work of this thesis, other arguments as the *"Java src/test .class and .java"* folders were added, for java to be able to compile and run the individual test cases and the buggy code. Also, the argument of *"Additional classpath file"* for other specific classpaths needed by the project is read from a file, separated by a colon, if known beforehand. Another argument of *"Dependencies"* folder was added, which corresponds to the root folder that should be scanned for .jar files and its subfolders for dependencies not covered by the *"Additional classpath file"*. The *"Project configuration"* argument decides if kBar should be run with maven or java test and compile commands specific for the Defects4j dataset or with the Bears default maven and java commands. The Bears java and maven arguments are easily modified in the code to make TBar easily adapted to other bug datasets or company-specific projects. This modification is especially useful as all companies or projects do not use maven or the same java version as defects4j or Bears datasets.

Read tests

The *"Read tests"* argument decides how the failing test output should be read by TBar, as the output of the complete package and class name is needed for test and patch validation. Originally the interpretation of test output was adapted for the Defects4j specific test and compile outputs. The output of failing test cases can differ in different versions of maven or other company-specific java build and test tools.

Git configuration and Maven

The *"Git Enable/Disable"* alternative decides if the git checkout and the presentation of the patch should be done with or without git. There

could be specific reasons for this, in some instances, if a buggy project folder is copied without git or other scenarios. Finally, the "*Maven path*" argument decides if the default maven version in the system should be used or another folder with maven, which could be useful when reproducing experiments.

Measurements and shell output

For the experiments, a "*Measurements*" argument provides an alternative to deciding what parts of TBar to measure, such as Fault Localization, the entire workflow or no measurements. The original version of TBar did show important outputs as generated patch candidates, patch process and result. However, there was no output of the java and maven or project-specific compile and test procedure, and therefore the "*Shell/Test output*" option was added. It could be important to see the output of some tests or the compile procedure to find out if missing dependencies or classpaths should be added.

4.2.3 Fault Localization

As the original version of TBar did not have Fault Localization implemented in the workflow, GZoltar Fault Localization was implemented in kBar, which is described in section 2.1.1 Fault Localization. This was done after the step of "Run all tests" to be able to collect the failing tests to GZoltar. In order to make the usage of GZoltar realistic in a custom scenario in the industry, the GZoltar Command Line Interface was used. There are other suitable alternatives, such as the GZoltar maven plugin; however, it was discarded as there is no guarantee of the use of maven in an industry or open-source project. There are two modes implemented *All* and *Fail*, to get alternatives to predict the most accurate Fault Location. *All*, GZoltar will run all tests in the entire project. *Fail*, if maven or company build tool finds a failure in

`org.springframework.data.mapping.model.ExampleTest`, GZoltar will run tests in `org.springframework.data.mapping.model.*` to limit the scope.

4.2.4 Other modifications

Other modifications were adding a rerun of failing tests to the Defects4j dataset alternative, as some of the tests in projects of Defects4j could act as "flaky tests", which fails sometimes but passes if re-executed. This was previously handled by ignoring the flaky tests with a "Fake Failed Test Cases" text file. Flaky tests should be considered in a real-world scenario, as the correct patch could be falsely discarded. Another modification was that the interpretation of shell output from the tests was modified to use Java `BufferedReader`¹ and `InputStreamReader`² instead of the provided `ReadShellProcess` from the TBar project. In this way, some problems with projects with extensive output and getting stuck on empty lines were avoided.

¹<https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html>

²<https://docs.oracle.com/javase/7/docs/api/java/io/InputStreamReader.html>

5 Experimental Research Methodology

This section will describe the experimental Research Methodology with the protocols of Research Questions. The explanation will be in a detailed level and contain descriptions such as experimental setup, bug collection, performance evaluation, fault localization and more.

5.1 RQ1: What is the effectiveness of template-based automatic program repair on different bug datasets?

This Research Question is answered through an extensive and detailed evaluation of kBar on the different bug datasets from Saab described in section 1.2, and Bears, chosen due to its ease to work with and publicly availability, described in section 2.3.4. The evaluation will not only present the number of bugs fixed but performance evaluation as well. The evaluation should serve as a base for the understanding of the behaviour of kBar and if it is suitable for use in an industrial context.

This Research Question aims to answer what is the effectiveness of kBar, described in section 4.2, on the considered bug datasets. It is answered by collecting suitable bugs from a set of selected criteria, described in section 5.1.2. Then kBar is executed on the selected bugs, and the performance and effectiveness are evaluated with specific criteria. The result also provides further understanding to the other Research Questions of the use of kBar in an industrial context.

5.1.1 Experimental setup

The tests were executed in parallel at the Kebnekaise¹ supercomputer at HPC2N, High-Performance Computing Center North, with 432 nodes of Node type Compute. Further, the node has an Intel Xeon E5-2690v4

¹<https://www.hpc2n.umu.se/resources/hardware/kebnekaise>

CPU, 2x14 Cores and Memory of 128 GB/node. The test configuration was one task per node, exclusive access to the node and 8 cores per job. The executions at Saab were run in serial at one of the standard developer computers, accessed by only one developer and no other jobs running. The source code and test results executing on Kebnekaise were made publicly available². The reason for separate executions of experiments is due to confidentiality reasons at Saab.

5.1.2 Bug collection

The bugs were collected from an open-source bug dataset Bears, and the history of git commits at Saab. In order to limit the scope of potential bugs, they were collected according to a number of criteria:

Bears dataset

From the Bears dataset, the criteria for bug collection were as following:

1. The bug is available and described at the Bears official website¹.
2. The bug produces at least one failed or error test case.
3. It corresponds to a one-line bug or a bug with few lines.

Saab dataset

In order to investigate if the tool can be used at Saab, it is important that the collected bugs correspond to real-world test failures that occurred in production and has a verified git commit. The tool must be able to execute with the latest version of the project and repository. At Saab, the criteria were as follows:

1. The bug is located in the project's history of git commits.

²<https://github.com/gynther-k/TBar/tree/test-kebn/>

¹<https://bears-bugs.github.io/bears-benchmark/>

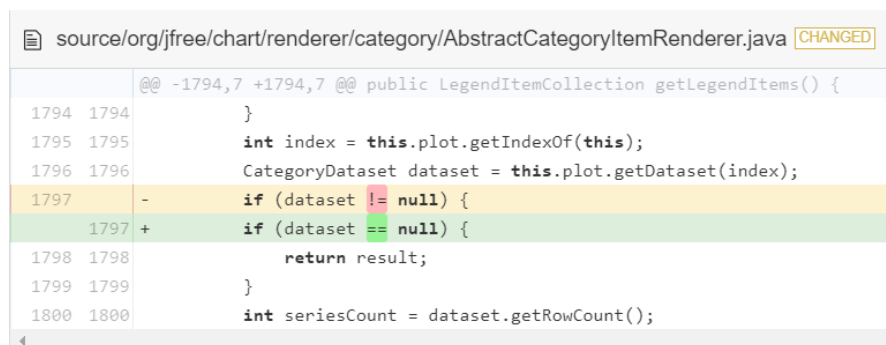
2. The description contains fix, test, fail, bug or is interpreted as a bug.
3. It is corresponding to a one-line bug or a bug with few lines.
4. The bug should be able to produce at least one failed or error test case with the latest version of the Saab project.

Also, in the company-specific bugs, tangled or mixed bugs are used. Tangled or mixed bugs contain unrelated changes to the bug, such as refactoring or documentation and 11% to 39% of all the fixing commits used for mining archives are tangled [21].

5.1.3 Performance and Effectiveness evaluation

When the bugs had been collected, the kBar tool was evaluated, focusing on four areas:

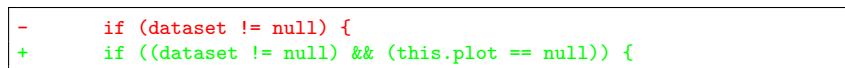
5.1.3.1 Patches and fixes and Fault Localization



```

source/org/jfree/chart/renderer/category/AbstractCategoryItemRenderer.java [CHANGED]
@@ -1794,7 +1794,7 @@ public LegendItemCollection getLegendItems() {
1794 1794     }
1795 1795     int index = this.plot.getIndexOf(this);
1796 1796     CategoryDataset dataset = this.plot.getDataset(index);
1797 -    if (dataset != null) {
1797 +    if (dataset == null) {
1798 1798         return result;
1799 1799     }
1800 1800     int seriesCount = dataset.getRowCount();
  
```

Figure 5.1: The official Human patch of bug Defects4j C-1.



```

-    if (dataset != null) {
+    if ((dataset != null) && (this.plot == null)) {
  
```

Figure 5.2: A Plausible patch, that passes all tests, presented by kBar when fixing Defects4j bug C-1.


```
-    if (dataset != null) {  
+    if (dataset == null) {
```

Figure 5.3: A Correct patch, that passes all tests and is equivalent to the official Human patch, presented by kBar when fixing Defects4j bug C-1.

The evaluation of the effectiveness with respect to the number of fixed bugs focused on the effectiveness of kBar itself, not on the accuracy of Fault Localization. Therefore, the first evaluation assumed Perfect Fault Localization, where the buggy location is provided in a pre-generated text file. The results are the number of generated patches, plausible patches and correct patches. A Plausible patch does make all tests pass, and a Correct patch does make all tests pass and is equivalent to the original human patch. Examples of a Human official, Plausible and Correct patch are shown in figure 5.1, 5.3 and 5.2.

The protocol of correctness analysis is followed by the author of this thesis and is manually conducted by the following criteria:

- **Plausible patch**, a patch that passes all tests.
- **Correct patch**, a patch that passes all tests and is equivalent to the official human patch.

Normal Fault Localization must also be considered in a real-world scenario, and we investigate the accuracy of Fault Localization. This process includes GZoltar Fault Localization and presents Fault position, Total position, Suspicious score, time and if the Suspicious position is detected.

5.1.3.2 Dictionary

We will investigate the size of the Dictionary of fix ingredients, described in section 4.1.2.3, and the impact on the time and number of generated patch candidates. The Dictionary is collected from the buggy

.java file and this will gain understanding to extend the search space in the future.

5.1.3.3 Time

Time consumption of the different parts of kBar, such as Total execution time, patch generation, validation of patches and dictionary generation. This gives knowledge of which part is the most time consuming and where we can make future improvements.

5.2 RQ2: What are the characteristics of the generated patches by kBar?

This research question provides an evaluation of the behaviour of kBar on the selected bugs from research question 1. According to Monperius et al., there is a problem with “*Fix Acceptability*” [35], and it could be challenging to decide if one patch is better than another if they both do pass the tests. In order to determine whether a patch is correct, there could be extensive knowledge needed about the project and its behaviour. Ideally, the original developers review the patches. However, in this Research Question, the details and usefulness of some patches are evaluated with the arbitrary knowledge of the author of this thesis.

In order to limit the scope, only three patches or bugs of each class of Correct, Plausible and Failed are selected and examined in detail. Due to confidentiality reasons, no Saab bugs are chosen, but only bugs from the Bears bug dataset. Bugs with interesting behaviour are manually selected by the author of this thesis, which clearly demonstrates the characteristics of kBar. Their behaviour is investigated, and the possible reasons for not fixing the actual bug. The Research Question provides a further understanding of the actual patches that kBar generates, and a detailed evaluation of the behaviour of the considered fix

templates and suggestions for improvements.

5.3 RQ3: How could template-based automatic program repair be integrated into the existing CI pipelines used at Saab?

This Research Question aims to answer how a prototype of the kBar tool can be integrated into the existing Continuous Integration pipelines at Saab. This is necessary, as the default version of TBar is not configured to be used in production as other tools, i.e. Repairnator [5]. The answer to the question focuses on a general point of view of the integration rather than a detailed description. The answer contains at least one working solution and suggestions for future integrations. Therefore, the evaluation of the implementation is considered a proof of concept and serves as a base for future and more extensive evaluations of Automatic program repair tools in the production environment at Saab.

6 Experimental Results

This section will describe the results of the experiments with answers to the protocols of Research Questions. The results will be presented in a detailed level as described in section 5.

6.1 RQ1: What is the effectiveness of template-Based Automatic Program Repair on different bug datasets?

Table 6.1: kBar results on the 54 collected bugs, with the template that produces a Plausible or Correct patch in **bold**. The numbers corresponding to each Fix Template are the number of generated Patch Candidates, an empty cell equals a non-matching of fix template. A Correct patch is marked **C**, Plausible patch **P** and patches which break the tests with - . Dictionary size is the total number of fix ingredients generated for the bug.

Bug-Id	FP2 Null Pointer Checker	FP3 Range Checker	FP4 Statement Inserter	FP6 Conditional Expression Mutator	FP7 Data Type Replacer	FP8 Mutate Integer Division Operation	FP9 Literal Expression Mutator	FP10 Method Invocation Mutator	FP 11 Operator Mutator	FP12 Return Statement Mutator	FP13 Variable Replacer	FP14 Statement Mover	FP15 Statement Remover	Total Patch Candidates	Dictionary Size	Correct/Plausible
Bears-2	6		0	70		0			16		4	2	4	102	1300	-
Bears-3	5	0	6					16	0		13	0	2	42	603	-
Bears-5	14	0		20				0	0		2	11		47	413	C
Bears-8	7	0		4				0	0		4			15	660	P
Bears-19	7	0	0					0	0		0	4	4	11	924	-
Bears-87	12	0	0					0	0	0	1	0		14	918	-
Bears-88	16	0	0		0			11	0	0	0	0	5	32	646	-
Bears-95				0		0			2					2	650	P
Bears-90	2	0	0	4		0		0	1		78	10	2	97	1092	-
Bears-98				12										12	338	P
Bears-109	17	1	3					2	0		34	20	2	79	3540	-
Bears-121				6		0			1					7	208	P

Bears-125	18	0						2	0		27			47	2085	P
Bears-127	11	0						2	0		0	0	2	15	2961	P
Bears-129	0	0	0				0	0	0		0	0	1	1	3780	-
Bears-130	0	0	0				0	0			0	1		1	3528	-
Bears-132	0	0	0				0	0	0		0	0	1	1	4300	-
Bears-133									5					5	387	C
Bears-136	15	0				0		0	3		4			22	2772	C
Bears-139	0			0							0	0	3	3	2080	P
Bears-144	5		0							0	5	0	0	11	234	-
Bears-151	7			0					0		0	0	3	10	54	C
Bears-154	11	0	4				0	19	0		10	5	2	51	540	-
Bears-160	16	0	4	788			0	0	0		14	0	6	828	2376	-
Bears-163	9	0	2	782		0	0	0	1		0	0	3	797	1980	-
Bears-166	7	0	0					10	0		12	2	6	37	774	-
Bears-169	15	0	2					5	0		11	0	2	35	1232	-
Bears-180	8	0	0				2	0	0	0	21	0	1	32	460	-
Bears-183	11	0	0					6	0	0	4	0	1	22	666	-
Bears-195	11	0	2	0		0	0	0	3		10	0	2	28	533	-
Bears-184	46	0	0	0			1	2	0		4	0	1	54	190	-
Bears-198	2	0	0					4	0		0	2	2	10	369	-
Bears-199													2	2	77	-
Bears-200	5	0	0					0	0	0	0	0	1	6	0	-
Bears-202	7	0	0					21	0		28	0	2	58	1008	-
Bears-232	0	0	0	0		0		0	5		0	4	2	11	247	-
Bears-233	0	0	0				0	0	0		0	4	2	6	189	-
Bears-238				24		0			1					25	125	P
Bears-249	2		0					2	0		0	2	2	8	328	-

Bears-251	0							1	0		0			1	892	C
Saab1.0				36					1					37	172	P
Saab1.1		0						3	0		13			16	344	C
Saab1.2				28					1					29	172	P
Saab2.1													2	2	35	-
Saab3	0		0				0				0	0	1	1	726	-
Saab4	22	0	6				2	161	0	2	40	0	1	234	1400	-
Saab5	19		0		0			31			0	0	2	52	440	-
Saab6.1									1					1	41	-
Saab6.2									1					1	41	-
Saab7	4	0	16		0		1	2	0		23	6	6	58	1408	-
Saab8									5					5	154	C
Saab9	14	0						0	0		12	5		31	1232	P
Saab10	5	0	4		0			0	0		7	0	2	18	846	-
Saab11				8										8	43	P
Total Number	356	1	49	1782	0	0	6	300	47	2	381	78	80	3080		C: 7 P: 12
P/C	2	0	0	5	0	0	0	1	5	0	1	2	3			

Total number of Patch candidates	3080
Average number Potential patches/bug	57
Median number of Potential patches/bug	15,5
Average size of Dictionary	972
Median size of Dictionary	603
Percent of Plausible patches of all bugs	35%
Percent of Correct patches of all bugs	13%

Table 6.2: Summarized data from Table 6.1

6.1.1 Patches and fixes

The results from Table 6.1 show that kBar generates 3080 patches on the 54 collected bugs from the considered bug datasets. kBar finds 19 Plausible¹ and Correct² patches and presents "Succeeded to fix the bug" and then stops the execution. Of these 19 patches, 7 are considered Correct or equivalent to the original human patch, per the correctness analysis protocol described in Section 5.1.3.1.

Table 6.2 shows that the median number of potential patches per bug is 15,5 and that kBar generates plausible patches to 35% of the selected bugs and 13% of them is correct.

The fix templates that generated the most Potential patches are FP6 Conditional Expression Mutator with 1782 and FP13 VariableReplacer with 381. The fix templates that produce the most Plausible and Correct patches are FP6 Conditional Expression Mutator and FP11 Operator-Mutator, both with 5. However, the fix templates of FP5 Mutate Class Instance Creation and FP1 Insert Cast Checker are never considered matching any of the bugs. The templates of FP8 Mutate Integer Division Operation and FP7 Data Type Replacer are considered matching multiple bugs, but no patches are generated.

Compared to the execution of TBar on the Defects4j bug dataset, which generates almost 26% Plausible patches and 19% Correct of (74/101) 395 bugs with Perfect Fault Localization, our work with a different setup with kBar generates 35% Plausible patches and 13% Correct patches.

¹<https://github.com/gynther-k/kbar/tree/main/ThesisResults/Patches/Plausible>

²<https://github.com/gynther-k/kbar/tree/main/ThesisResults/Patches/Correct>

The difference in experiment setup is that in the original TBar with Defects4j execution, the comparison is done with Perfect Fault Localization and generates all possible patch candidates with all fix templates. In this thesis, when a matching fix template has found a Plausible patch, the execution stops.

However, the results in the original TBar show that FP3, FP4.3, FP5, FP7.2 and FP11.3 cannot generate any Plausible patches. This corresponds to our findings, where FP5 is never considered matching and FP7 is considered matching, but no patches are generated. FP3 is generating 1 Plausible patch in our work, and when reviewing in detail, the FP4.3 and FP11.3 are not generating any plausible patches. In our work, templates that generate the highest number of Potential patches are FP6 Conditional Expression Mutator and FP13 Variable Replacer. With TBar and Defects4j, the FP6 Conditional Expression Mutator and FP13 Variable Replacer generate the most number of Plausible and Correct patches, so it is possible that these templates also generates the most number of Potential patches, similar to our findings. So the conclusion when comparing to the original TBar results, there are both differences and similarities.

6.1.2 Dictionary size of fix ingredients

The median size of the Dictionary of fix ingredients, described in section 4.1.2, of the fix templates is 603 items, as shown in Table 6.1 and Table 6.2. The largest Dictionary is generated by bug Bears-132 with 4300 items. However, there is no clear correlation between increasing Dictionary size and Total execution time, as shown in Figure 6.1. There is also no apparent connection between an increasing Dictionary size and the total number of Patch candidates as shown in Figure 6.2.

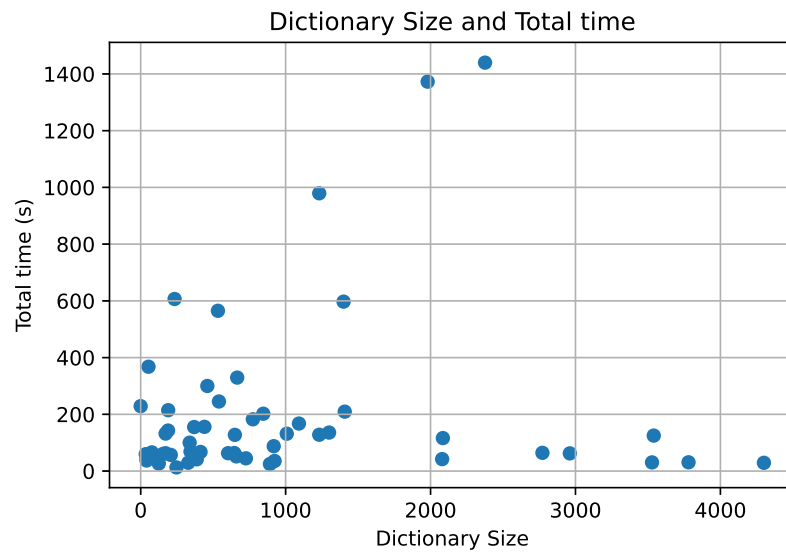


Figure 6.1: Correlation between Dictionary size and Total execution time (s)

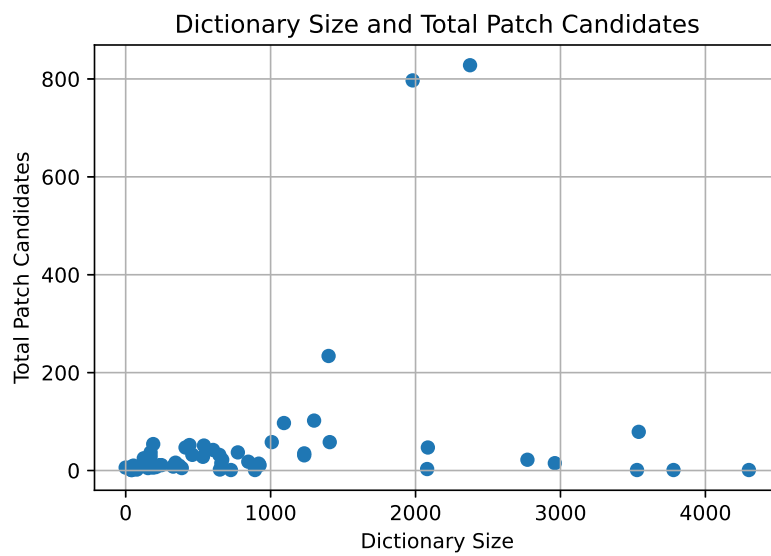


Figure 6.2: Correlation between Dictionary Size and Total Number of Patch Candidates

6.1.3 Time

Table 6.3: Results of kBar (time), with Perfect Fault Localization.

Bug-Id	Patch generation (ms)	Dictionary generation (ms)	Patch validation (s)	Total time (s)
Bears-2	12	10	84	136
Bears-3	19	15	34	63
Bears-5	127	17	43	67
Bears-8	29	21	25	52
Bears-19	109	17	8	36
Bears-87	4	12	20	88
Bears-88	5	3	30	64
Bears-95	5	29	38	128
Bears-90	9	6	125	168
Bears-98	3	18	7	100
Bears-109	101	8	77	125
Bears-121	5	11	19	57
Bears-125	64	47	86	116
Bears-127	34	10	27	62
Bears-129	23	21	1	31
Bears-130	30	14	1	31
Bears-132	24	39	1	29
Bears-133	2	11	12	41
Bears-136	29	7	35	64
Bears-139	5	19	9	42
Bears-144	2	9	8	607
Bears-151	4	4	155	368
Bears-154	13	10	62	245
Bears-160	27	18	1322	1440
Bears-163	38	22	1287	1373
Bears-166	15	3	16	183
Bears-169	6	19	51	979

Bears-180	6	7	48	300
Bears-183	4	10	69	330
Bears-195	6	8	54	565
Bears-184	9	6	50	215
Bears-198	9	5	10	155
Bears-199	0,4	20	9	66
Bears-200	2	0	12	229
Bears-202	5	18	69	132
Bears-232	5	6	5	13
Bears-233	2	4	4	143
Bears-238	5	10	19	26
Bears-249	22	8	5	29
Bears-251	8	82	9	25
Average	21	15	99	223
Median	9	11	26	108
Saab1.0	13	71	27	63
Saab1.1	8	47	27	69
Saab1.2	12	46	88	132
Saab2.1	1	60	22	60
Saab3	2	41	2	45
Saab4	46	36	554	597
Saab5	14	26	112	156
Saab6.1	3	2136	2	37
Saab6.2	3	25	2	42
Saab7	19	16	166	209
Saab8	4	112	17	60
Saab9	85	37	84	128
Saab10	9	31	59	202
Saab11	5	50	14	70
Average	16	195	84	134
Median	9	44	27	69

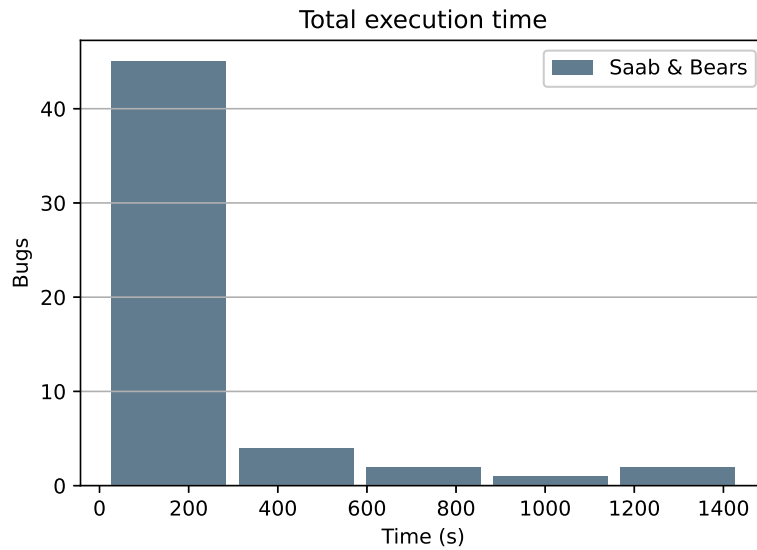


Figure 6.3: Histogram of kBar(time), with Perfect Fault Localization.

In the Saab environment, as shown in Table 6.3 and Figure 6.3, the median Total Execution time is 69 seconds. The lowest Total Execution time was with bug Saab4 in around 37 seconds and the highest in 597 seconds. The validation of patches has a median execution time of 27 seconds at Saab, which corresponds to approximately 39 percent of the total execution time.

With the Bears bug dataset, the median Total Execution time is 108 seconds. The lowest Total Execution time was 13 seconds with Bears-232 and the highest with Bears-160 in 1440 seconds. The validation of patches has a median execution time of 26 seconds, which corresponds to around 24 percent of the total execution time.

All results from the "generate patch process" and "generate dictionary time" are executed in milliseconds with both datasets, except one Saab bug, which consumes around 2 seconds. There are differences in the median Total execution time of Saab and Bears experiments of 43 seconds. The reason for faster execution time at Saab could be the hardware setup and differences in project complexity and test coverage. At Saab, only two different projects were chosen, in contrast to more than 10 different with Bears. However, the median

time of validation of patches is around 26 to 27 seconds in both projects and corresponds to 24 to 39 percent of the total execution time. The results show that the most significant time-consuming part in both environments except for initialization, build process and first-time testing, and more is the validation of patches.

6.1.4 Fault Localization

The GZoltar Fault Localization with the alternatives *all* or *fail* did find the correct location in 38 of the 54 bugs shown in Table 6.4. The correct positions are located between the first and 12401 with a median position of 85 and the top 2% of the generated suspicious positions. A lower top percentage value is good. The suspiciousness percentage value presented by GZoltar is in the median 6%, where 100% corresponds to most suspicious. The top percentage and suspicious value are calculated with only the bugs where GZoltar finds the faulty location, excluding the others. Finally, the median time for execution is 7 seconds at Saab and with the Bears bug dataset.

When reviewing only the Bears dataset, the GZoltar Fault Localization did find the correct location in 32 of the 40 bugs. The correct positions are located between 1 and 12401 with a median position of 60 and the top 2%. The suspiciousness percentage value presented by GZoltar is in the median 92%.

When reviewing only the Saab dataset, the GZoltar Fault Localization did find the correct location in 6 of the 14 bugs. The correct positions are located between 11 and 1087 with a median position of 179 and the top 8%. The suspiciousness percentage value presented by GZoltar is in the median 6%.

From the results, GZoltar performs better on the Bears dataset where it finds the correct location on 80% of the bugs where Saab only finds around 43%. The suspicious score is in median 86% higher with Bears, and the top percent is 2% against 8% at Saab. Also, the fail mode did not succeed on all bugs at Saab, leading to a low suspicious score. Possible reasons for the differences could be different test coverage between Bears and Saab, different Java versions and test environment. Also, the fail mode of kBar presents a higher

suspicious value, but there are too few bugs to state this clearly.

In the TBar paper, the Fault Localization accuracy is not reviewed in detail per bug. Instead, an average value of the Fault position of the Plausible or Correct patches of each Fix template is presented, eg. where position 1 corresponds to the first found suspicious code location by the Fault localization tool. The average positions of the Correct patches are varying between 1 and 62, and the Plausible patches between 1 and 191. When they run the experiments without Perfect Fault Localization and replicate a "standard and practical APR pipeline", the number of correct bugs is almost 40% lower than with Perfect Fault Localization, so the Fault Localization is introducing errors as in our work. Also, the original TBar setup was unable to find some buggy locations. The key take away point is that unfixed bugs *"are generally more poorly localized than correctly fixed bugs"* [6].

Table 6.4: Results of kBar with respect to Fault Localization.

Bug-Id	Fault position	Total position	Mode	Top percent (%)	Time (s)	Suspicious score
Bears-2	2035	21690	fail	9%	18	17%
Bears-3	4186	20059	fail	21%	6	17%
Bears-5	12401	18126	fail	68%	7	0%
Bears-8	-	-	fail	-	13	-
Bears-19	28	18819	fail	0,1%	7	38%
Bears-87	1	591	fail	0,2%	2	100%
Bears-88	169	835	fail	20%	3	32%
Bears-95	44	1759	fail	3%	10	9%
Bears-90	94	1629	fail	6%	9	4%
Bears-98	108	12711	fail	0,8%	5	100%
Bears-109	18	13796	fail	0,1%	7	100%
Bears-121	1	14780	fail	0%	7	100%
Bears-125	338	15486	fail	2%	7	100%
Bears-127	24	15594	fail	0,2%	7	100%
Bears-129	22	15734	fail	0,1%	7	100%
Bears-130	32	15742	fail	0,2%	8	100%
Bears-132	169	16305	fail	1%	8	100%
Bears-133	8	16450	fail	0,05%	7	100%
Bears-136	22	16568	fail	0,1%	8	100%
Bears-139	262	16962	fail	2%	8	100%
Bears-144	-	-	fail	-	-	-
Bears-151	9	26	fail	35%	2	100%
Bears-154	148	342	fail	43%	3	29%
Bears-160	208	2177	fail	10%	5	8%
Bears-163	191	2068	fail	9%	4	4%
Bears-166	31	330	fail	9%	1	71%
Bears-169	-	-	fail	-	-	-

Bears-180	-	-	fail	-	-	-
Bears-183	-	-	fail	-	-	-
Bears-195	-	-	fail	-	-	-
Bears-184	-	-	fail	-	-	-
Bears-198	13	852	fail	2%	3	100%
Bears-199	-	-	fail	-	8	-
Bears-200	1	410	fail	0,2%	13	100%
Bears-202	75	3046	fail	2%	2	71%
Bears-232	4	204	fail	2%	1	71%
Bears-233	76	125	fail	61%	1	100%
Bears-238	22	101	fail	22%	1	63%
Bears-249	117	1523	fail	8%	12	83%
Bears-251	313	12571	fail	2%	7	22%
Average	662	8669		11%	6	67%
Median	60	7809		2%	7	92%
Saab1.0	145	2149	fail	7%	7	13%
Saab1.1	11	2149	fail	1%	6	34%
Saab1.2	186	2149	fail	9%	6	24%
Saab2.1	-	3381	all	-	7	-
Saab3	171	3416	all	5%	7	0%
Saab4	-	3415	all	-	7	-
Saab5	-	3821	all	-	8	-
Saab6.1	-	3821	all	-	7	-
Saab6.2	-	3821	all	-	7	-
Saab7	613	1190	fail	52%	5	0%
Saab8	-	3821	all	-	7	-
Saab9	-	3821	all	-	7	-
Saab10	1087	2977	all	37%	21	0%
Saab11	-	1151		-	5	-

Average	369	2934		18%	8	12%
Median	179	3398		8%	7	6%

6.1.5 Conclusion

Patches and fixes

The results from kBar shows that it generates Plausible patches to 35% of the selected bugs and 13%, which is considered Correct. In total, 3080 patches were generated to the 54 bugs, and four fix templates are never considered matching or not generating any patches. This could be compared to the execution of TBar on the Defects4j bug dataset, which generates almost 26% Plausible patches and 19% Correct of 395 bugs (74/101) with Perfect Fault Localization. However, it would be interesting to investigate execution with all fix templates and without Perfect Fault Localization to get a more accurate view of the behaviour of fix templates and overfitting.

Compared to the original TBar paper results, our work presents 35% Plausible patches, which is 9% higher, and 13% Correct patches, which is 6% lower. However, our work has a less advantageous setup but more realistic, where kBar stops when finding the first Plausible patch, and the original TBar paper executes all possible fix templates. There are also similarities in fix template behaviour, where some templates do not generate Plausible patches. Based on the results in our work and original TBar, we assume that FP6 Conditional Expression Mutator and FP13 Variable Replacer generate the highest number of Potential patches.

Fault Localization

The Fault Localization is most accurate on the Bears dataset, where it finds the correct location in 80% of the bugs. The median Fault Localization top percentage of 2% and a suspicious score of 6% show that the top percentage would be a better value to cut the fault positions and limit the scope. The suspicious score is low and generates too many positions. Finally, the Fault Localization is introducing errors, and our work corresponds to the take away point from TBar paper, where "unfixed bugs are generally more poorly localized than correctly fixed bugs".

Time

The largest individual time-consuming part of kBar is the "Validation of patches", which does consume, in the median, between 39 and 24 percent of the time when using only one buggy location with Perfect Fault Localization. This part in kBar should be considered when running with Normal Fault Localization when the number of patch candidates increases. Due to time constraints, only one measurement was done with Saab1.1 bug with GZoltar and Normal Fault Localization, resulting in a total execution time of around 838 seconds (~14minutes). Of these ~14 minutes, the "Validation of patches" took 823 seconds or 98% of the total execution time. The single measurement was

conducted with a bug with the Fault Location calculated in position 11. Also, there seems to be no clear connection between the Dictionary size and Total Execution time or Total number of patches. However, the two most common Fix templates do not make use of the dictionary and other datasets may behave differently.

Theoretical assumptions

Theoretically, with the median values from the measurements as a base, a general best case scenario of "Validation of patches" with Normal Fault Location position 1 would take 27 seconds. The median Normal Fault location of 85 is theoretically assumed to execute in around 38 minutes($27 \cdot 85$) and a worst-case scenario of 93 hours($27 \cdot 12801$), not following the 3 hour recommended execution time of TBar. So, when considering the time, the most crucial factor that should be considered is the process of "Validation of patches".

Saab and Bears differences

When reviewing the results in detail between the Saab and Bears datasets, there are some differences. The median Total execution time is 43 seconds faster at Saab than with Bears. The validation of patches is still the most significant time-consuming part in both environments, 24 to 39% of the time. The Fault Localization is more accurate on the Bears dataset than at Saab, with a successful location of 80% against ~43%. The suspicious score is in median 86% higher with Bears, and the top per cent value is 2% against 8% at Saab. So the execution is faster at Saab, but Fault Localization is less accurate in this environment. Possible reasons for the differences could be hardware setup, the variation of programming languages at Saab, test coverage, project differences, test complexity, number of projects tested, Java versions and test environment.

6.2 RQ2: What are the characteristics of the generated patches by kBar?

With the results from RQ1 as a base, the following bugs and patches were selected and manually reviewed. There are three chosen classes of Correctly, Plausible and Failed to fix bugs.

6.2.1 Correctly fixed bugs

Detailed review of bug Bears-5

```
src/main/java/com/fasterxml/jackson/databind/deser/BeanDeserializer.java [CHANGED]
@@ -373,6 +373,10 @@ public class BeanDeserializer
373 373     for (; t == JsonToken.FIELD_NAME; t = p.nextToken()) {
374 374         String propName = p.getCurrentName();
375 375         p.nextToken(); // to point to value
376 +         // Object Id property?
377 +         if (buffer.readIdProperty(propName)) {
378 +             continue;
379 +         }
380
381 // creator property?
382 SettableBeanProperty creatorProp = creator.findCreatorProperty(propName);
383 if (creatorProp != null) {
384
@@ -405,10 +409,6 @@ public class BeanDeserializer
405 409     }
406 410     continue;
407 411 }
408 - // Object Id property?
409 - if (buffer.readIdProperty(propName)) {
410 -     continue;
411 - }
412 // regular property? needs buffering
413 SettableBeanProperty prop = _beanProperties.find(propName);
414 if (prop != null) {
```

Figure 6.4: The official Human patch of bug Bears-5

```
@@-374,7 +374,10 @@ public class BeanDeserializer
String propName = p.getCurrentName();
p.nextToken(); // to point to value
// creator property?
- SettableBeanProperty creatorProp = creator.findCreatorProperty(propName);
+ if (buffer.readIdProperty(propName)) {
+     continue;
+ }
+ SettableBeanProperty creatorProp = creator.findCreatorProperty(propName);
+ if (creatorProp != null) {
+     // Last creator property to set?
+     if (buffer.assignParameter(creatorProp,
@@-406,9 @@+409,7 public class BeanDeserializer
continue;
}
// Object Id property?
- if (buffer.readIdProperty(propName)) {
-     continue;
- }
+
+ // regular property? needs buffering
SettableBeanProperty prop = _beanProperties.find(propName);
if (prop != null) {
```

Figure 6.5: The patch presented by kBar when fixing bug Bears-5.

During the manual assessment, the Bears-5 bug was considered to be Correctly fixed by kBar. The Human patch is shown in Figure 6.4 and the patch presented by kBar in Figure 6.5. It was fixed by one of the Patch Candidates in the FP14 Statement Mover fix template which generated 11 Patch Candidates for this bug. The templates of FP2 NullPointerChecker and FP6 Conditional Expression Mutator did present the most number of Patch Candidates. In total, 7 Fix templates were considered matching the buggy code but not all of them resulting in a Patch Candidate. Some examples of patch candidates are shown in Figure 6.6.

```

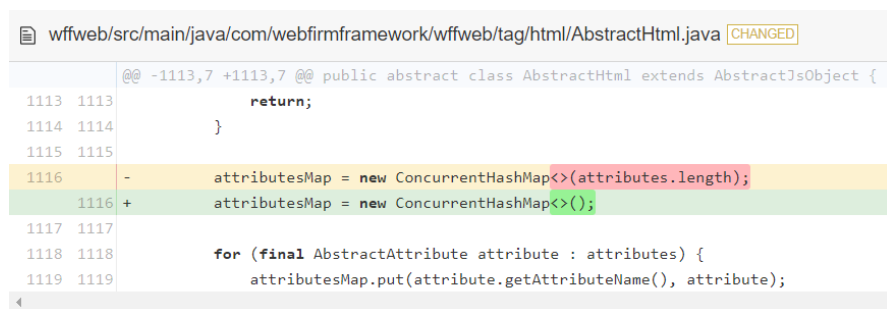
1. FP6: if ((buffer.readIdProperty(propName)) && !(ext.handlePropertyValue(p,ctxt,propName,null)))
2. FP13: if (asArrayDeserializer().readIdProperty(propName))
3. FP2: if (buffer != null)
4. FP2: if (buffer != null || buffer.readIdProperty(propName))

```

Figure 6.6: Examples of Patch candidates generated by kBar when fixing bug Bears-5

Some of the presumed behaviour in Figure 6.6 is the added AND condition in Example 1, which is gathered from another if condition in the same buggy .java file. Also, the condition in Example 2 is replaced with another compatible expression as described in section 3.1.1 of TBar. An interesting behaviour of the FP2 NullPointerChecker template is shown in Example 4, where a non expected OR condition is added in the expression. Another template that should be able to produce a patch is the FP10 MethodInvocation Mutator. The reason for not creating a patch could be that kBar has a list of previously tried patches and does not execute the generated patches from FP10. The idea of not executing old patches saves time.

Detailed review of bug Bears-251



```

wffweb/src/main/java/com/webfirmframework/wffweb/tag/html/AbstractHtml.java [CHANGED]
@@ -1113,7 +1113,7 @@ public abstract class AbstractHtml extends AbstractJsObject {
1113 1113         return;
1114 1114     }
1115 1115
1116 1116 -     attributesMap = new ConcurrentHashMap<>(attributes.length);
1116 1116 +     attributesMap = new ConcurrentHashMap<>();
1117 1117
1118 1118     for (final AbstractAttribute attribute : attributes) {
1119 1119         attributesMap.put(attribute.getAttributeName(), attribute);

```

Figure 6.7: The official Human patch of bug Bears-251

The manual assessment of the Bears-251 bug was considered to be Correctly

```

--- a/wffweb/src/main/java/com/webfirmframework/wffweb/tag/html/AbstractHtml.java
+++ b/wffweb/src/main/java/com/webfirmframework/wffweb/tag/html/AbstractHtml.java
@@ -1113,7 +1113,7 @@ public abstract class AbstractHtml extends AbstractJsObject {
    return;
}

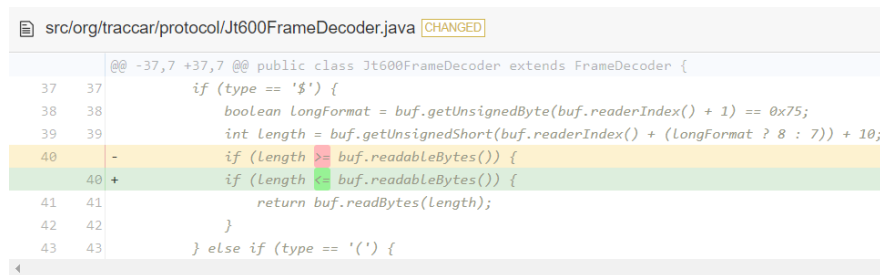
- attributesMap = new ConcurrentHashMap<>(attributes.length);
+ attributesMap = new ConcurrentHashMap<>();

```

Figure 6.8: The patch presented by kBar when fixing bug Bears-251.

fixed by kBar. The Human patch is shown in Figure 6.7 and the patch presented by kBar in Figure 6.8. It was fixed by the only generated Patch Candidate in FP10 Method Invocation Mutator by removing the buggy variable.

Detailed review of bug Bears-133



```

src/org/traccar/protocol/Jt600FrameDecoder.java [CHANGED]
@@ -37,7 +37,7 @@ public class Jt600FrameDecoder extends FrameDecoder {
    37 37         if (type == '$') {
    38 38             boolean longFormat = buf.getUnsignedByte(buf.readerIndex() + 1) == 0x75;
    39 39             int length = buf.getUnsignedShort(buf.readerIndex() + (longFormat ? 8 : 7)) + 10;
    40 -         if (Length >= buf.readableBytes()) {
    40 +         if (Length <= buf.readableBytes()) {
    41 41             return buf.readBytes(length);
    42 42         }
    43 43     } else if (type == '(') {

```

Figure 6.9: The official Human patch of bug Bears-133

```

--- a/src/org/traccar/protocol/Jt600FrameDecoder.java
+++ b/src/org/traccar/protocol/Jt600FrameDecoder.java
@@ -37,7 +37,7 @@ public class Jt600FrameDecoder extends FrameDecoder {
    if (type == '$') {
        boolean longFormat = buf.getUnsignedByte(buf.readerIndex() + 1) == 0x75;
        int length = buf.getUnsignedShort(buf.readerIndex() + (longFormat ? 8 : 7)) + 10;
-       if (length >= buf.readableBytes()) {
+       if (length <= buf.readableBytes()) {
            return buf.readBytes(length);
        }
    } else if (type == '(') {

```

Figure 6.10: The patch presented by kBar when fixing bug Bears-133.

During the manual assessment, the Bears-133 bug was considered to be Correctly fixed by kBar. The Human patch is shown in Figure 6.9, and the patch presented by kBar in Figure 6.10. It was fixed by one of the five Patch Candidates in the FP11 Operator Mutator. This was the only fix template considered,

as it was the first template executed, and some examples of patch candidates are shown below in Figure 6.11.

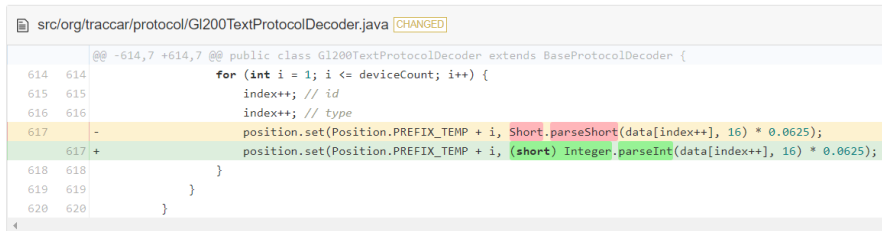
```
FP6: if (length==buf.readableBytes()) {
FP6: if (length!=buf.readableBytes()) {
FP6: if (length<buf.readableBytes()) {
FP6: if (length<=buf.readableBytes()) {
FP6: if (length>buf.readableBytes()) {
```

Figure 6.11: Examples of Patch candidates generated by kBar when fixing bug Bears-133

6.2.2 Plausible patches

The following discussed bugs were reported as Plausible and passed all tests by kBar but were found to be incorrect after manual analysis.

Detailed review of bug Bears-125



```
src/org/traccar/protocol/Gl200TextProtocolDecoder.java [CHANGED]
@@ -614,7 +614,7 @@ public class Gl200TextProtocolDecoder extends BaseProtocolDecoder {
614 614         for (int i = 1; i <= deviceCount; i++) {
615 615             index++; // id
616 616             index++; // type
617 -             position.set(Position.PREFIX_TEMP + i, Short.parseShort(data[index++], 16) * 0.0625);
617 +             position.set(Position.PREFIX_TEMP + i, (short) Integer.parseInt(data[index++], 16) * 0.0625);
618 618         }
619 619     }
620 620 }
```

Figure 6.12: The official Human patch of bug Bears-125.

```
--- a/src/org/traccar/protocol/Gl200TextProtocolDecoder.java
+++ b/src/org/traccar/protocol/Gl200TextProtocolDecoder.java
@@ -614,7 +614,11 @@ public class Gl200TextProtocolDecoder extends BaseProtocolDecoder {
    for (int i = 1; i <= deviceCount; i++) {
        index++; // id
        index++; // type
-        position.set(position.PREFIX_TEMP + i, short.parseShort(data[index++], 16) * 0.0625);
+        if (position == null) continue;
+        if (position.PREFIX_TEMP == null) continue;
+        if (data == null) continue;
+        if (data[index++] == null) continue;
+        position.set(position.PREFIX_TEMP + i, short.parseShort(data[index++], 16) * 0.0625);
    }
}
```

Figure 6.13: The patch presented by kBar when fixing bug Bears-125.

During the manual assessment, the Bears-125 bug was considered to be Plausible fixed by kBar. The Human patch is shown in Figure 6.12 and the patch presented by kBar in 6.13. It was fixed by one of the Patch Candidates in the FP2 NullPointerChecker fix template, which generated 11 Patch Candidates for this bug. The templates of FP2 NullPointerChecker and FP13 Variable Replacer Mutator did present the most number of Patch Candidates. In total, 5 Fix templates were considered matching the buggy code but only three of them resulting in a Patch Candidate. Some examples of patch candidates are shown below in Figure 6.14.

```
1. FP13 position.set(Position.ALARM_DOOR + i, Short.parseShort(data[index++], 16) * 0.0625);
2. FP13 position.set(Position.ALARM_OIL_LEAK + i, Short.parseShort(data[index++], 16) * 0.0625);
3. FP13 position.set(Position.KEY_ODOMETER + i, Short.parseShort(data[index++], 16) * 0.0625);
4. FP13 position.set(Position.KEY_BLOCKED + i, Short.parseShort(data[index++], 16) * 0.0625);
5. FP2 if (Position.PREFIX_TEMP == null) Position.PREFIX_TEMP = new null();
6. FP2 if (Position.PREFIX_TEMP == null) return new Object();
7. FP2 if (data != null) {
```

Figure 6.14: Examples of Patch candidates generated by kBar when fixing bug Bears-125

When manually inspecting the suggested Plausible patch in Figure 6.13, it is clear that it does not generate a behaviour close to the human patch. It is interesting to see that the correct Integer.parseInt() does exist in the same buggy .java file. However, the reason for it not being chosen as a suitable candidate for the patch generation could be that kBar cannot cast it to a short.

Detailed review of bug Bears-238



```
src/main/java/com/json/ignore/JsonIgnoreFields.java [CHANGED]
@@ -130,7 +130,7 @@ public class JsonIgnoreFields {
130 130     }
131 131
132 132     private boolean fieldAcceptable(Field field) {
133 -         return field.getType().isPrimitive() || field.getType().isArray() || ignoredNames.contains(field.getName());
133 +         return field.getType().isPrimitive() || field.getType().isArray() || ignoredNames.contains(field.getName());
134 134     }
135 135
136 136     public void ignoreFields(Object object) throws IllegalAccessException {
```

Figure 6.15: The official Human patch of bug Bears-238

During the manual assessment, the Bears-238 bug was considered to be Plausible fixed by kBar. The Human patch is shown in Figure 6.15 and the patch presented by kBar in Figure 6.16. It was fixed by the only Patch Candidate in the FP11 Operator Mutator fix templates. The template of FP6 Conditional Expression Mutator did present the most number of Patch Candidates of 24. In total, 3 Fix templates were considered matching the buggy code but only


```

--- a/src/main/java/com/json/ignore/JsonIgnoreFields.java
+++ b/src/main/java/com/json/ignore/JsonIgnoreFields.java
@@ -130,7 +130,7 @@ public class JsonIgnoreFields {
    }

    private boolean fieldAcceptable(Field field) {
-   return field.getType().isPrimitive() || field.getType().isArray() || ignoredNames.contains(field.getName());
+   return field.getType().isPrimitive() && field.getType().isArray() || ignoredNames.contains(field.getName());
    }

```

Figure 6.16: The patch presented by kBar when fixing bug Bears-238.

two of them resulting in a Patch Candidate. Some examples of patch candidates are shown below in Figure 6.17.

```

FP6: return (field.getType().isPrimitive() || field.getType().isArray() || ignoredNames.contains(field.getName())) ||
(items.contains(field.getName()));
FP6: return (field.getType().isPrimitive() || field.getType().isArray() || ignoredNames.contains(field.getName())) &&
(items.contains(field.getName()));
FP6: return (field.getType().isPrimitive() || field.getType().isArray() || ignoredNames.contains(field.getName())) ||
!(items.contains(field.getName()));
FP6: return (field.getType().isPrimitive() || field.getType().isArray() || ignoredNames.contains(field.getName())) &&
!(items.contains(field.getName()));
FP11: return field.getType().isPrimitive() && field.getType().isArray() || ignoredNames.contains(field.getName());

```

Figure 6.17: Examples of Patch candidates generated by kBar when fixing bug Bears-238

When manually inspecting the Patch Candidates of FP6 fix template, all candidates only change the last element of the Conditional Expression. Also, the FP11 only generates one Patch candidate. One possible reason is that the considered Fix templates behave differently when handling a return statement instead of an if statement, and this behaviour should be further investigated. Finally, the Plausible patch presented to bug Bears-238 would probably be considered incorrect by the human behind the official patch. But it could be difficult to decide if the patch is not so different from the human patch, adding && instead of remove, and all tests pass.

Detailed review of bug Bears-139

```
src/org/traccar/protocol/MeitrackProtocolDecoder.java [CHANGED]
@@ -259,7 +259,7 @@ public class MeitrackProtocolDecoder extends BaseProtocolDecoder {
259 259         boolean started = data[0].charAt(0) == '0';
260 260         position.set("taximeterOn", started);
261 261         position.set("taximeterStart", data[1]);
262 -         if (!started) {
262 +         if (data.length > 2) {
263 263             position.set("taximeterEnd", data[2]);
264 264             position.set("taximeterDistance", Integer.parseInt(data[3]));
265 265             position.set("taximeterFare", Integer.parseInt(data[4]));
```

Figure 6.18: The official Human patch of bug Bears-139

```
--- a/src/org/traccar/protocol/MeitrackProtocolDecoder.java
+++ b/src/org/traccar/protocol/MeitrackProtocolDecoder.java
@@ -259,13 +259,7 @@ public class MeitrackProtocolDecoder extends BaseProtocolDecoder {
    boolean started = data[0].charAt(0) == '0';
    position.set("taximeterOn", started);
    position.set("taximeterStart", data[1]);
-    if (!started) {
-        position.set("taximeterEnd", data[2]);
-        position.set("taximeterDistance", integer.parseInt(data[3]));
-        position.set("taximeterFare", integer.parseInt(data[4]));
-        position.set("taximeterTrip", data[5]);
-        position.set("taximeterWait", data[6]);
-    }
+
}
```

Figure 6.19: The patch presented by kBar when fixing bug Bears-139.

During the manual assessment, the Bears-139 bug was considered to be Plausible fixed by kBar. The Human patch is shown in Figure 6.18 and the patch presented by kBar in Figure 6.19. It was fixed by one of the Patch Candidates in the FP15 Statement Remover which generated 3 Patch Candidates for this bug. In total, four other Fix templates were considered matching the buggy code but none of them resulting in a Patch Candidate. The only Patch Candidate of FP15 that includes code is the example below in Figure 6.20.

```

-         position.set("taximeterEnd", data[2]);
-         position.set("taximeterDistance", Integer.parseInt(data[3]));
-         position.set("taximeterFare", Integer.parseInt(data[4]));
-         position.set("taximeterTrip", data[5]);
-         position.set("taximeterWait", data[6]);

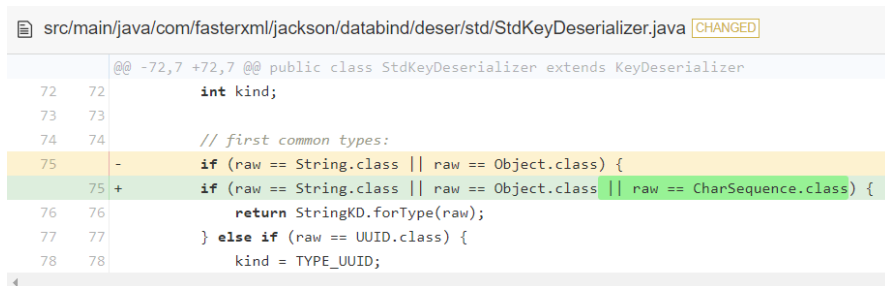
```

Figure 6.20: The only Patch candidate with code, generated by kBar when fixing bug Bears-139

As we can see in Figure 6.20, the patch candidate does remove the if statement but keeps the code inside it. The presented Plausible patch that does pass the tests does remove the entire if statement with all its content. There are no other patch candidates generated, and the other fix templates may try to match other booleans in the file with the original "boolean started". However, there are no more booleans in the buggy .java file. On the other hand, many method calls return a boolean, but they are not considered, which should be investigated. The correct patch should be `if (data.length > 2)`, and it does not exist in any of the other .java files in the entire project. Although, the `if(data.length > 1)` does exist in a nearby .java file, and it is possible that this would generate a more correct patch. So it could be interesting to increase the search space with bug Bears-139.

6.2.3 Failed to fix bugs

Detailed review of bug Bears-2



```

src/main/java/com/fasterxml/jackson/databind/deser/std/StdKeyDeserializer.java [CHANGED]
@@ -72,7 +72,7 @@ public class StdKeyDeserializer extends KeyDeserializer
72 72     int kind;
73 73
74 74     // first common types:
75 -     if (raw == String.class || raw == Object.class) {
75 +     if (raw == String.class || raw == Object.class || raw == CharSequence.class) {
76 76         return StringKD.forType(raw);
77 77     } else if (raw == UUID.class) {
78 78         kind = TYPE_UUID;

```

Figure 6.21: The official Human patch of bug Bears-2

During the manual assessment, the Bears-2 bug was considered to be not fixed by kBar, and 102 Patch Candidates were generated. The templates of FP6 Conditional Expression Mutator and FP11 Operator Mutator did present the most number of Patch Candidates. In total, 8 Fix templates were considered matching the buggy code but not all of them resulting in a Patch Candidate. Some examples of patch candidates are shown in Figure 6.22.

```

FP 11 OperatorMutator:
if (raw!=String.class || raw == Object.class)
if (raw<String.class || raw == Object.class)
if (raw > String.class || raw > Object.class)

FP6 ConditionalExpressionMutator:
if ((raw == String.class || raw == Object.class) || (raw == UUID.class))
if ((raw == String.class || raw == Object.class) || (raw == Integer.class))
if ((raw == String.class || raw == Object.class) && (raw == Character.class)) {

FP2 NullpointerChecker:
if (raw == null) {
    return null;
}

```

Figure 6.22: Examples of Patch candidates generated by kBar when fixing bug Bears-2

We can see that both operators are changed in the if statement from the FP11 OperatorMutator. The FP6 Conditional Statement Mutator does add additional conditions such as AND (raw = Character.class). So if the correct fix Charsequence.class would exist in the same .java file, a Correct patch would be suggested. When manually reviewing other .java files in folders within the project the CharSequence.class does exist, so expanding the search space would produce the correct patch.

Detailed review of bug Bears-129

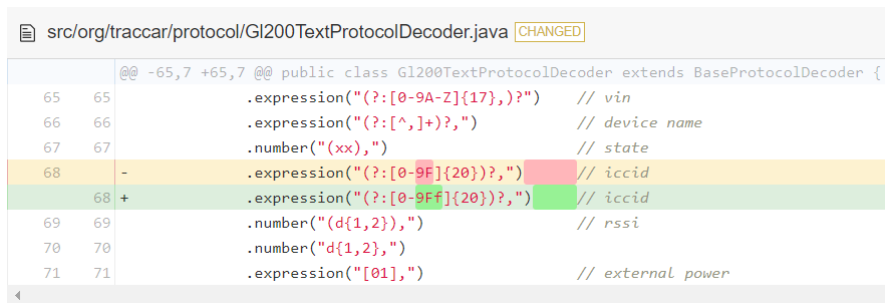


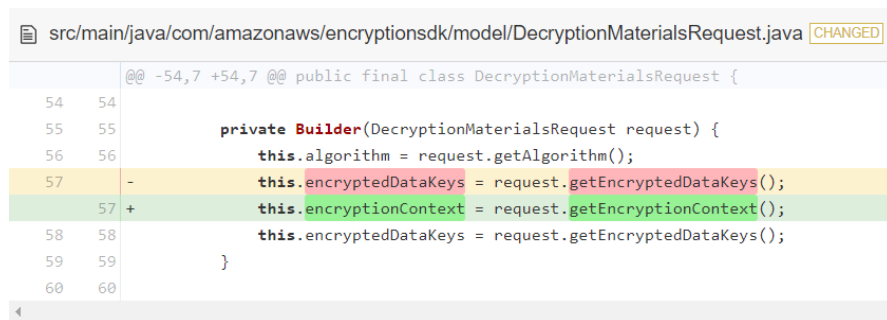
Figure 6.23: The official Human patch of bug Bears-129

During the manual assessment, the Bears-129 bug was not fixed by kBar, and only 1 Patch Candidate was generated. Only the FP15 StatementRemover could generate a Patch Candidate. Although a large Dictionary size of 3790 items is generated, it could not create one single Patch Candidate for the suspicious code shown in Figure 6.24. One reason could be that the fix templates cannot mutate a PatternBuilder with multiple variables as in the figure, which could be a future improvement.

```
private static final Pattern PATTERN_INF = new PatternBuilder()
    .text("+").expression("(?:RESP|BUFF):GTINF,")
    .number("[0-9A-Z]{2}xxxx,") // protocol version
    .number("(d{15}|x{14}),") // imei
    .expression("(?:[0-9A-Z]{17},)?") // vin
    .expression("(?:[^\,]+)?") // device name
    .number("(xx),") // state
    .expression("(?:[0-9F]{20})?,") // iccid
```

Figure 6.24: Full view of the code in bug Bears-129.

Detailed review of bug Bears-198



```
src/main/java/com/amazonaws/encryptionsdk/model/DecryptionMaterialsRequest.java CHANGED
@@ -54,7 +54,7 @@ public final class DecryptionMaterialsRequest {
54 54
55 55     private Builder(DecryptionMaterialsRequest request) {
56 56         this.algorithm = request.getAlgorithm();
57 -         this.encryptedDataKeys = request.getEncryptedDataKeys();
57 +         this.encryptionContext = request.getEncryptionContext();
58 58         this.encryptedDataKeys = request.getEncryptedDataKeys();
59 59     }
60 60
```

Figure 6.25: The official Human patch of bug Bears-198

During the manual assessment, the Bears-198 bug was considered to be not fixed by kBar, and only 10 Patch Candidates were generated. The FP10 Method Invocation Mutator template did generate the most Plausible patches of 4 shown in Figure 6.25. So we can see that in patch number 4, 50% of the correct human bug is located. So a future suggestion would be to combine multiple template on the same buggy row. However, the encryptionContext and encryptedDataKeys are of different types, List and String, therefore not suggested by the FP13 Variable Replacer. So another future suggestion to generate more patches is that the FP13 Variable Replacer should also consider non-compatible expressions.

```
1: FP10: this.encryptedDataKeys = request.newBuilder();
2: FP10: this.encryptedDataKeys = request.toBuilder();
3: FP10: this.encryptedDataKeys = request.getAlgorithm();
4: FP10: this.encryptedDataKeys = request.getEncryptionContext();
```

Figure 6.26: Examples of Patch candidates generated by kBar when fixing bug Bears-198

6.2.4 Conclusion

The Correct and Incorrect patches are easy to distinguish manually by a user of kBar, but it could be more difficult with Plausible patches that are close to the human patch and pass all the tests. If the human behind the original patch is not self-assured, then the Plausible patch considered incorrect could be Correct. Expanding the search space for dictionary ingredients could produce more correctly fixed bugs as the correct ingredients in some cases are located in a file close to the buggy one or the same project. However, there is no guarantee that a buggy code with thousands of dictionary items generates many patches. Suggestions for improvements and extensions in the fix template should also be considered, such as casting variables and changing a line at multiple places. An interesting time effective solution, which is used in TBar, is that if multiple templates generate the same patch, only one patch is used. It is possible that the first generated patch that does pass the test is not the most correct one. TBar or kBar still can generate more plausible patches with all Fix templates. Therefore, it would be interesting to introduce the overfitting detection system(ODS) suggested in R-Hero [42] and use all fix templates to see if more correct patches are generated.

Finally, it would be interesting to conduct a more extensive evaluation of all fix templates and their assumed behaviour on more bugs to produce even more insights into the fix templates and their behaviour.

6.3 RQ3: How could template-based automatic program repair be integrated into the existing CI pipelines used at Saab?

Two different approaches of kBar were integrated into the existing development environment at Saab. The implementation of the kBar tool described in Section 4.2 was used as a base. However, to be able to use kBar in the production environment, further modifications were made. The modifications were needed to use the company-specific build tools, read the shell output, change kBar JUnit version, and adapt to the specific behaviour in the complex company environment. Due to the time consuming manual configuration of implementing kBar, it was implemented as a proof of concept in one of the considered projects only.

Normal use cases

As a pre-study, two different use-cases were considered and investigated for the research question. The use cases were described by one of the experienced

developers. One case is where the developer downloads the entire workspace, dependencies and current project to their local computer. From the local computer, the “make build” and “make test” commands are run from the command line, before submitting code to the global repository. Many of the tests are executed here to avoid submitting code with failing tests to the global repository. The second case is where the developer submits code changes to the global repository and then makes use of the Continuous Integration pipeline to execute “make build” and “make test” in a remote environment.

Implementations

The following implementations and modifications of kBar were implemented at Saab. The patch proposal or *Recommendation system* presents a patch suggestion to the developer but conducts no changes. The *Automatic repair system* takes the first patch and inserts it into the buggy project without human interaction. Both of the alternatives share the same prerequisites.

Prerequisites:

1. Configure project-specific arguments to kBar.
2. Check if kBar already exists in the workspace.
3. If not 2, git clone kBar to project workspace.

Option 1: Command line, Recommendation system

The configured project was integrated with kBar as a Recommendation system in the following way:

1. Build the project (make build).
2. Run the project test command (make test).
3. kBar executes automatically - check if there are test errors.
4. If 3, kBar tries to repair the bug.
5. If successful, present a plausible patch as shell output for manual insertion.

Option 2: Command line, Automatic repair

The configured project was integrated with kBar, as an Automatic system in the following way:

1. Build the project (make build)

2. Run the project test command (make test)
3. kBar executes automatically - check if there are test errors.
4. If 3, kBar tries to repair the bug.
5. If successful, present a plausible patch as shell output.
6. kBar does automatically insert the plausible patch in the buggy project.
7. Rerun all tests.

Continuous Integration

Both of the two options do work by default in the Continuous Integration workflow and are easily enabled or disabled. As the Continuous Integration workflow is complex, there is a need to avoid making the integration of an Automatic Program Repair Tool add more complexity to the process. Therefore, the Automatic repair option was used by default in the Continuous Integration workflow. According to the answer of RQ1, for one bug, Saab.1.1, this would add an additional 14 minutes of execution time. When considering the theoretical assumptions from RQ1, the Continuous Integration would be delayed with 38 minutes in median considering the "Validation of Patches".

Conclusion

The modifications of kBar for the Saab environment should not be seen as specific to Saab. When using Automatic Program Repair tools in a real-world scenario, the complex environments in different companies or open-source projects should be taken into account. The implementation of kBar does need manual configuration for each project at the company, and one or more developers should have knowledge of the tool and its behaviour. Therefore, kBar was only implemented with one project, but it could be extended to other projects as well in the future. Possible future extensions to the kBar implementation at Saab could be a more automatic behaviour with the Continuous Integration such as Repairnator [5], a more user-friendly recommendation system, and include kBar by default for all workspaces.

The usefulness of a fully automatic integration to Continuous Integration could be discussed as the patch should be manually reviewed before used in production. Also, the time constraints of ~14 minutes and ~38 minutes for one project do extend the recommended values. According to the study in a large-scale industry project [31], the total time of Continuous Integration should not exceed ten minutes, including testing. However, in a complex company environment as the one mentioned in the study, the build of the entire system could take several hours. Then, the execution time of kBar is more acceptable. According to one of the experienced developers at Saab, the execution

time of 14 minutes is acceptable for daily use and several hours during the night.

Finally, before a large scale usage at the company, improvements should be made to Fault Localization, patch ranking, search space, execution time, and ease of the kBar configuration to leverage the full potential of template-based automatic program repair. Also, other APR tools should be investigated, and possible be combined with kBar. Nevertheless, the kBar tool is already working in Saab production and can be used to fix real-world bugs.

6.4 Discussion of Results

6.4.1 Generated patches

The results from kBar execution on the collected datasets in this thesis show that it is comparable to the original TBar paper concerning the ratio of Plausible patches and number of bugs. The original TBar paper presents a slightly better ratio of Correct patches, which could depend on the different setup. The setup in the TBar paper includes execution with all fix templates, and it does not stop execution when a Plausible patch is found as in kBar. This approach could be implemented in kBar as well, but then the execution time would increase. However, with future improvements of execution time, this should be seen as an alternative in a real-world scenario. But this raises the question of how to integrate the solution, with all fix templates and a patch ranking system, or a list of all Plausible patches to be chosen by the developer.

6.4.2 Search space

It is possible that increasing the search space of the Dictionary of fix ingredients would lead to more Plausible and Correct patches. The current setup of collecting fix ingredients in the same file does not correlate between Dictionary size against Total execution time and the number of patches. But it is possible that the correlation changes when expanding the search space. So an extensive search space should be weighed against that many Potential patches could be challenging to handle concerning time.

6.4.3 Plausible patches and templates

When manually reviewing the Plausible patches, many of them are not close to the human patch, but in some Potential patches that are not so far away from the human patch. Expanding the search space to fix ingredients or slightly modify the behaviour of the template would produce the correct patch. Also,

an extensive investigation of the behaviour of the fix templates should be conducted to see if there are differences in the actual behaviour and described behaviour. There are tendencies towards this in our work, but the number of bugs was too small to clearly state this.

Some templates produce more Plausible patches than others, and the possibility of adding or removing other fix templates should be investigated. But fix templates that generate a small number of Potential patches also generate Plausible or even Correct patches. Also, some of the templates do not seem to generate some potential Patches at all. So removing templates should be carefully considered.

6.4.4 Time improvements

Based on the results from Table 6.3, Figure 6.3 with Perfect Fault Localization and the theoretical assumptions in 6.1.5, the total execution time is varying between seconds and many hours in a worst-case scenario. So, improvements in execution time should be conducted. There are already optimizations of removing duplicate patches, and similar optimizations should be investigated. One solution could be a distributed or parallelized version of kBar of the Validation of patches. This would drastically reduce the execution time in large projects as the Validation of patches is the main bottleneck.

6.4.5 Fault Localization

In theory, the Fault Localization accuracy does present a seemingly good median value of the top 2% of the generated code locations and does find the correct location in around 80% of the bugs with the Bears dataset. However, this number could correspond to a median line position of 85, which means that the execution time of Validation of patches could increase with the same number. Also, this increases the risk of false positives, which produce a Plausible patch that does pass the tests but in the wrong position. The lower accuracy at Saab could depend on multiple reasons such as hardware setup, test coverage, test environment and more. Therefore more work should be done investigating Fault Localization.

7 Discussion

7.1 Benefits, Ethics and Sustainability

Automatic Program Repair and Artificial Intelligence could impact the social aspect of ethics negatively by replacing tasks carried out by humans, resulting in unemployment and poverty. However, the results could also be more time for developers carrying out non-repetitive tasks such as bug fixing and reducing stress.

The environmental aspects do show that the share of ICT products, such as communication networks, personal computers, and data centers, increased from 3.9% to 4.6% of the total worldwide electricity consumption between 2007 and 2012 and continues to increase [47]. The use of data centers within Automatic Program Repair and Artificial Intelligence in industry, could increase this number even more. But hopefully, intelligent algorithms and solutions would decrease energy consumption instead.

Although Saab [48] produces products for military and surveillance combat systems, their vision is *"It's a human right to feel safe"* and they aim to keep people safe with technical systems and to increase security in the society. This thesis should be seen as a generic evaluation of the use of Automatic Program Repair in an industrial context.

7.2 Threats to Validity

Failed execution - Several bugs of the Bears dataset were discarded due to failed execution with kBar. Some examples of failures are multiple Exceptions with the INRIA-Spoon project in bug Bears-27 to Bears-83, invalid git directory with Bears-157, Bears-187 getting stuck during test execution and Bears-224 where kBar fails to read the test output. It is possible that modifying kBar to work with these bugs and a successful execution would give a different answer to the Research Questions.

Evaluation execution - Due to time constraints, the tests and executions of the Research Questions were run only once. With a strict experimental setup with multiple reruns, there could be different results concerning time. However, the aim of this thesis is not at strict and exact performance evaluation. The objective is to understand the behaviour of the tool on the considered bugs and environment.

Datasets - The evaluation was only conducted with 54 bugs from two different datasets, and a more extensive study conducted on thousands of bugs from

other bug datasets could present different results. This should be considered in future research.

False Positives - If Normal Fault Localization would have been run on all bugs instead of Perfect Fault Localization, possible false positives could have been found. Then the number of Correct patches could be lower than presented in this thesis.

7.3 Future Work

Detailed investigation of repair templates - The templates in kBar should manually be examined by comparing the code of the templates, description in TBar paper and examples of actual patches.

Examples from section 6.2.1 are when manually reviewing bug Bears-238, where the potential patches generated by the FP6 fix template does not change all operators, only the last operator of a return statement. Another example is the bug Bears-5, where a potential patch of the FP2 NullpointerChecker adds an unexpected OR statement to a null check.

These examples may correspond to the expected behaviour of the templates, but it is not clear from the TBar paper. Therefore, an extensive and detailed investigation of the fix templates should be conducted. The analysis should provide answers such as: should the FP2 fix template be extended with different behaviour? Is the behaviour with Bears-238 and Bears-5 reproducible with other bugs? Are there other divergent or interesting behaviours to fix templates and bugs?

This could lead to a better understanding of the fix templates and possible improvements, such as modifying more operators with the FP6 Conditional Expression Mutator template. These are only two examples, and there should be a large number of examples to consider.

Extension of kBar - Extend and further improve the kBar tool. Many of the fix templates used in the described tools in section 3 are used in kBar today. But there are other interesting techniques described in the section, not only fix templates. Machine learning techniques such as logistic regression to select patches or generating new templates from the buggy code or other sources with supervised learning or probability models. An Overfitting detection system(ODS) could be introduced and used to rank patches and avoid overfitting; this assumes configuring TBar to not stop with the first patch.

A more extensive performance evaluation of APR tools at Saab - With a larger dataset from industry and open source projects, conduct an evaluation of the

effectiveness of many tools to gain insight into which tool(s) is adequate to use in the Saab context.

This should be possible with the gained knowledge from conducting the work in this thesis. There was an extensive amount of time spent to understand the complex company environment and manual collection of bugs. Also, the modification of TBar into kBar was time-consuming. Therefore, a more extensive performance evaluation was not possible in the scope of this thesis. This work should serve as a base for future evaluations, and the collected bugs can be reused, possibly with other manually created bugs. The knowledge from the integration and configuration will make it easier to implement other APR tools.

A performance evaluation of Fault Localization tools at Saab - Compare and evaluate the effectiveness of different Fault Localization techniques such as GZoltar, which is used in this thesis. This could lead to insights into which tool is most accurate to use in the specific industrial context.

A distributed or parallelized implementation - To make kBar more effective, the main bottleneck of Validation of patches could be distributed in a large cluster in the cloud or parallelized on one node or a combination of both. This would drastically improve the execution time. However, one bug is estimated to cost 400\$ [5], and the cost of a distributed solution should be weighed against the profit.

7.4 Template-based Repair Tools in the Industry

As described in section 1.1, there have been attempts to evaluate APR tools at Facebook [49], which automatically presents patches to null method call bugs by mining fix templates and has presented a high accuracy on 53% of the bugs in one experiment. Ubisoft has attempted to identify templates of risky commits with Clever [50] to respond to the high number of false positives and non-user-friendly recommendations of current approaches. The Ubisoft approach manages to detect risky commits with 79% precision and recommend fixes in 66,7% of the cases within a Ubisoft experiment.

Another template-based approach of FlexiRepair [7], described in section 3.1.2, specifically targets the limitations of TBar, such as real-world usability. It leverages the Coccinelle code transformation tool, which is included in the Linux kernel developer toolbox. By using Coccinelle, it sends a strong signal that it is ready to use in the industry. However, it did only show lower or similar repair results than other repair state-of-the art tools.

The contribution of our work differs from other work, as TBar combines a

variety of different types of templates from multiple tools. Therefore it is a promoter of template-based automatic program repair in general. The integration into kBar makes it more suitable for industrial integration. Further, the difference in our work is that it provides an extensive evaluation of both repair efficiency and other factors such as time, Fault Localization and real-world usability in an industrial context.

Despite the small number of bugs evaluated, the contribution of this thesis shows that it is possible to use kBar in an industrial context and what improvements need to be done for future and wider usage of the tool in the industry. Without this work, it would have been difficult to know where improvements should be made and if it kBar at all would be suitable in the industry.

8 Conclusion

This thesis evaluates a template-based automatic program repair tool, kBar, within an industrial context, but also the general effectiveness. The tool presents Plausible patches to 35% (19/54) of the selected bugs and 13% (7/54) of the patches is Correct, comparable to the experiments with TBar and Defects4j. The Fault accuracy is lower in the Saab environment, and more work should be done with Fault Localization and execution time, repair templates, and possible extensions. However, a prototype of the kBar tool is integrated with one project at Saab and can already be used to repair real-world bugs.

The key take-away point is that template-based automatic program repair already is suitable to be integrated into an industrial context, apart from the discussed flaws. The work in this thesis has provided knowledge of effectiveness, patches and continuous integration and should serve as a base for future research in the area of APR.

Finally, Automatic Program Repair is a broad subject, and other tools separately or combined should be investigated before a final industrial integration is considered. Automatic Program repair shows a promising future, and APR tools will probably be integrated by default at many companies within a few years.

References

- [1] Monperrus, Martin. “Automatic Software Repair: A Bibliography”. In: *ACM Comput. Surv.* 51.1 (Jan. 2018). ISSN: 0360-0300. DOI: 10.1145/3105906. URL: <https://doi.org/10.1145/3105906>.
- [2] Nielebock, S. “Towards API-specific automatic program repair”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Oct. 2017, pp. 1010–1013. DOI: 10.1109/ASE.2017.8115721.
- [3] Saha, Ripon K. et al. “ELIXIR: Effective Object Oriented Program Repair”. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. ASE 2017*. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 648–659. ISBN: 9781538626849. DOI: 10.5555/3155562.3155643. URL: <https://dl.acm.org/doi/pdf/10.5555/3155562.3155643>.
- [4] Long, Fan and Rinard, Martin. “An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems”. In: *Proceedings of the 38th International Conference on Software Engineering. ICSE '16*. Austin, Texas: Association for Computing Machinery, 2016, pp. 702–713. ISBN: 9781450339001. DOI: 10.1145/2884781.2884872. URL: <https://doi.org/10.1145/2884781.2884872>.
- [5] Monperrus, Martin et al. “Repairnator Patches Programs Automatically”. In: *Ubiquity* (July 2019). DOI: 10.1145/3349589. URL: <https://doi.org/10.1145/3349589>.
- [6] Liu, Kui et al. “TBar: Revisiting Template-Based Automated Program Repair”. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. IS-

- STA 2019. Beijing, China: Association for Computing Machinery, 2019, pp. 31–42. ISBN: 9781450362245. DOI: 10.1145/3293882.3330577. URL: <https://doi.org/10.1145/3293882.3330577>.
- [7] Koyuncu, Anil et al. *FlexiRepair: Transparent Program Repair with Generic Patches*. 2020. arXiv: 2011.13280 [cs.SE]. URL: <https://arxiv.org/abs/2011.13280>.
 - [8] *Saab, Company in brief*. (accessed March 13, 2021). URL: <https://www.saab.com/about/company-in-brief>.
 - [9] Håkansson, Anne. *Portal of Research Methods and Methodologies for Research Projects and Degree Projects*. June 2013, pp. 67–73. URL: <http://kth.diva-portal.org/smash/record.jsf?pid=diva2%5C%3A677684&dswid=5465>.
 - [10] Martinez, Matias and Monperrus, Martin. “Astor: Exploring the design space of generate-and-validate program repair beyond GenProg”. In: vol. 151. Elsevier BV, May 2019, pp. 65–80. DOI: 10.1016/j.jss.2019.01.069. URL: <http://dx.doi.org/10.1016/j.jss.2019.01.069>.
 - [11] Liu, K. et al. “You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems”. In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 2019, pp. 102–113. DOI: 10.1109/ICST.2019.00020.
 - [12] *The GZoltar toolset - Automatic Testing, Debugging using Spectrum-based Fault Localization (SFL)*. (accessed Aug 28, 2021). URL: <http://www.gzoltar.com/>.
 - [13] Abreu, Rui, Zoetewij, Peter, and Gemund, Arjan J.C. van. “On the Accuracy of Spectrum-based Fault Localization”. In: *Testing: Academic and Industrial Conference Practice and Research*

- Techniques - MUTATION (TAICPART-MUTATION 2007)*. 2007, pp. 89–98. DOI: 10.1109/TAIC.PART.2007.13.
- [14] D Le, Xuan Bach, Lo, David, and Goues, Claire. “History Driven Program Repair”. In: Mar. 2016. DOI: 10.1109/SANER.2016.76.
 - [15] Xuan, Jifeng et al. “Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs”. In: *IEEE Transactions on Software Engineering* 43.1 (2017), pp. 34–55. DOI: 10.1109/TSE.2016.2560811.
 - [16] Xiong, Yingfei et al. “Precise Condition Synthesis for Program Repair”. In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE ’17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 416–426. ISBN: 9781538638682. DOI: 10.1109/ICSE.2017.45. URL: <https://doi.org/10.1109/ICSE.2017.45>.
 - [17] Chen, Liushan, Pei, Yu, and Furia, Carlo Alberto. “Contract-Based Program Repair without The Contracts: An Extended Study”. In: *IEEE Transactions on Software Engineering* (2020), pp. 1–1. DOI: 10.1109/TSE.2020.2970009.
 - [18] Xin, Qi and Reiss, Steven P. “Leveraging syntax-related code for automated program repair”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017, pp. 660–670. DOI: 10.1109/ASE.2017.8115676.
 - [19] Wen, Ming et al. “Context-aware patch generation for better automated program repair”. In: May 2018, pp. 1–11. ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180233.
 - [20] Hua, Jinru et al. “Towards Practical Program Repair with On-Demand Candidate Generation”. In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 12–

23. ISBN: 9781450356381. DOI: 10.1145/3180155.3180245. URL: <https://doi.org/10.1145/3180155.3180245>.
- [21] Koyuncu, Anil et al. “FixMiner: Mining relevant fix patterns for automated program repair”. In: *Empirical Software Engineering* 25 (May 2020). DOI: 10.1007/s10664-019-09780-z.
- [22] Liu, Kui et al. “LSRepair: Live Search of Fix Ingredients for Automated Program Repair”. In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. 2018, pp. 658–662. DOI: 10.1109/APSEC.2018.00085.
- [23] Jiang, Jiajun et al. “Shaping Program Repair Space with Existing Patches and Similar Code”. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 298–309. ISBN: 9781450356992. DOI: 10.1145/3213846.3213871. URL: <https://doi.org/10.1145/3213846.3213871>.
- [24] Martinez, Matias and Monperrus, Martin. “ASTOR: A Program Repair Library for Java (Demo)”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbrücken, Germany: Association for Computing Machinery, 2016, pp. 441–444. ISBN: 9781450343909. DOI: 10.1145/2931037.2948705. URL: <https://doi.org/10.1145/2931037.2948705>.
- [25] *Defects4J*. (accessed November 23, 2020). URL: <https://github.com/rjust/defects4j>.
- [26] Just, René, Jalali, Darioush, and Ernst, Michael D. “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*. ISSTA 2014. San

- Jose, CA, USA: Association for Computing Machinery, 2014, pp. 437–440. ISBN: 9781450326452. DOI: 10.1145/2610384.2628055. URL: <https://doi.org/10.1145/2610384.2628055>.
- [27] Gay, Gregory and Just, René. “Defects4J as a Challenge Case for the Search-Based Software Engineering Community”. In: *Search-Based Software Engineering*. Ed. by Aldeida Aleti and Annibale Panichella. Cham: Springer International Publishing, 2020, pp. 255–261. ISBN: 978-3-030-59762-7.
- [28] Saha, Ripon K. et al. “Bugs.jar: A Large-Scale, Diverse Dataset of Real-World Java Bugs”. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 10–13. ISBN: 9781450357166. DOI: 10.1145/3196398.3196473. URL: <https://doi.org/10.1145/3196398.3196473>.
- [29] Tan, Shin Hwei et al. “Codeflaws: a programming competition benchmark for evaluating automated program repair tools”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, pp. 180–182. DOI: 10.1109/ICSE-C.2017.76.
- [30] Madeiral, Fernanda et al. “BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies”. In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (Feb. 2019). DOI: 10.1109/saner.2019.8667991. URL: <http://dx.doi.org/10.1109/SANER.2019.8667991>.
- [31] Mårtensson, Torvald, Hammarström, Pär, and Bosch, Jan. *Continuous Integration is Not About Build Systems*. Aug. 2017, pp. 1–9. DOI: 10.1109/SEAA.2017.30.

- [32] Monperrus, Martin. “The Living Review on Automated Program Repair”. working paper or preprint. Dec. 2020. URL: <https://hal.archives-ouvertes.fr/hal-01956501>.
- [33] Kechagia, Maria et al. “Evaluating Automatic Program Repair Capabilities to Repair API Misuses”. In: *IEEE Transactions on Software Engineering* (2021), pp. 1–1. DOI: 10.1109/TSE.2021.3067156.
- [34] Kim, D. et al. “Automatic patch generation learned from human-written patches”. In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 802–811. DOI: 10.1109/ICSE.2013.6606626.
- [35] Monperrus, Martin. “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* (May 2014). DOI: 10.1145/2568225.2568324. URL: <http://dx.doi.org/10.1145/2568225.2568324>.
- [36] Liu, X. and Zhong, H. “Mining stackoverflow for program repair”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2018, pp. 118–129. DOI: 10.1109/SANER.2018.8330202.
- [37] Durieux, Thomas et al. *Dynamic Patch Generation for Null Pointer Exceptions using Metaprogramming*. Feb. 2017, pp. 349–358. DOI: 10.1109/SANER.2017.7884635.
- [38] Ghanbari, A. and Zhang, L. “PraPR: Practical Program Repair via Bytecode Mutation”. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 1118–1121. DOI: 10.1109/ASE.2019.00116.

- [39] Liu, Kui et al. “AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations”. In: Feb. 2019, pp. 1–12. DOI: 10 . 1109/SANER.2019.8667970.
- [40] Long, Fan, Amidon, Peter, and Rinard, Martin. “Automatic Inference of Code Transforms for Patch Generation”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 727–739. ISBN: 9781450351058. DOI: 10 . 1145/3106237 . 3106253. URL: <https://doi.org/10.1145/3106237.3106253>.
- [41] Ueda, Yuki et al. “DevReplay: Automatic Repair with Editable Fix Pattern”. In: May 2020. URL: <https://arxiv.org/pdf/2005.11040.pdf>.
- [42] Baudry, Benoit et al. “A Software Repair Bot based on Continual Learning”. In: *IEEE Software* (2021), pp. 0–0. ISSN: 1937-4194. DOI: 10 . 1109/ms . 2021 . 3070743. URL: <http://dx.doi.org/10.1109/ms.2021.3070743>.
- [43] Samak, Malavika, Kim, Deokhwan, and Rinard, Martin C. “Synthesizing Replacement Classes”. In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10 . 1145/3371120. URL: <https://doi.org/10.1145/3371120>.
- [44] Martinez, Matias and Monperrus, Martin. “Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor: 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings”. In: Jan. 2018, pp. 65–86. ISBN: 978-3-319-99240-2. DOI: 10 . 1007 / 978-3-319-99241-9_3.
- [45] Sakkas, Georgios et al. “Type Error Feedback via Analytic Program Repair”. In: *Proceedings of the 41st ACM SIGPLAN Con-*

- ference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 16–30. ISBN: 9781450376136. DOI: 10.1145/3385412.3386005. URL: <https://doi.org/10.1145/3385412.3386005>.
- [46] Lerner, Benjamin S. et al. “Searching for Type-Error Messages”. In: *SIGPLAN Not.* 42.6 (June 2007), pp. 425–434. ISSN: 0362-1340. DOI: 10.1145/1273442.1250783. URL: <https://doi.org/10.1145/1273442.1250783>.
- [47] Heddeghem, Ward et al. “Trends in worldwide ICT electricity consumption from 2007 to 2012”. In: *Computer Communications* (Sept. 2014). DOI: 10.1016/j.comcom.2014.02.008.
- [48] *Saab, Purpose and Values*. (accessed March 13, 2021). URL: <https://www.saab.com/about/company-in-brief/purpose-and-values>.
- [49] *Getafix: How Facebook tools learn to fix bugs automatically*. (accessed June 26, 2021). URL: <https://engineering.fb.com/2018/11/06/developer-tools/getafix-how-facebook-tools-learn-to-fix-bugs-automatically>.
- [50] Nayrolles, Mathieu and Hamou-Lhadj, Abdelwahab. “CLEVER: Combining Code Metrics with Clone Detection for Just-in-Time Fault Prevention and Resolution in Large Industrial Projects”. In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 2018, pp. 153–164.

TRITA -EECS-EX-2021:697