TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology

Marina Nekrassova 153070IAPM

APPLICATION OF THE DELTA DEBUGGING ALGORITHM TO FINE-GRAINED AUTOMATED LOCALIZATION OF REGRESSION FAULTS IN JAVA PROGRAMS

Master's thesis

Supervisor: Juhan Ernits
PhD

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Marina Nekrassova 153070IAPM

AUTOMATISEETITUD SILUMISE RAKENDAMINE VIGADE LOKALISEERIMISEKS JAVA RAKENDUSTES

Magistritöö

Juhendaja: Juhan Ernits
PhD

Tallinn 2018

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Marina Nekrassova

08.01.2018

Abstract

In software development, occasionally, in the course of software evolution, the functionality that previously worked as expected stops working. Such situation is typically denoted by the term *regression*. To detect regression faults as promptly as possible, many agile development teams rely nowadays on automated test suites and the practice of continuous integration (CI). Shortly after the faulty change is committed to the shared mainline, the CI build fails indicating the fact of code degradation. Once the regression fault is discovered, it needs to be localized and fixed in a timely manner.

Fault localization remains mostly a manual process, but there have been attempts to automate it. One well-known technique for this purpose is delta debugging algorithm. It accepts as input a set of all changes between two program versions and a regression test that captures the fault, and outputs a minimized set containing only those changes that directly contribute to the fault (in other words, are failure-inducing). In previous studies developing this approach, only coarse-grained changes produced by an ordinary textual differencing tool was used as a basis for experiments, which led to performance issues and worsened the accuracy of localization. The goal of the current thesis is to substitute textual differencing with abstract syntax tree differencing and investigate the effects of such replacement on time behavior and output of delta debugging process.

As a result of this thesis, a prototypical AST differencing-based implementation of delta debugging tool has been built and evaluated on a set of real regressions collected from a large enterprise information system written in Java language. The evaluation shows that switching to AST differencing brings improvement in terms of effectiveness, performance, accuracy, and plausibility of the output.

This thesis is written in English and is 52 pages long, including 4 chapters, 14 tables, and 5 figures.

Annotatsioon

Automatiseeritud silumise rakendamine vigade lokaliseerimiseks Java rakendustes

Tarkvara evolutsiooni käigus juhtub aeg-ajalt, et mingi funktsionaalsus, mis varem töötas korralikult, enam ei tööta. Sellist olukorda nimetatakse tarkvara regressiooniks. Et avastada regressioonivead võimalikult kiiresti, paljud agiilsed arendusmeeskonnad kasutavad tänapäeval automaatteste ning pidevat integratsiooni (*Continuous Integration* ehk *CI*). Vahetult pärast seda, kui vigane muudatus on integreeritud ühiskasutatavasse keskkonda, CI ülesanne nurjub, mis viitab koodi kvaliteedi halvenemisele. Kui regressiooniviga on avastatud, tuleb see lokaliseerida ja õigeaegselt parandada.

Vea lokaliseerimine jääb enamasti manuaalseks protsessiks, kuid seda on püütud automatiseerida. Seoses sellega on delta-silumise algoritm üks hästi tuntud meetod, millele antakse sisendina ette hulk muudatusi sama programmi eri versioonide vahel ning ebaõnnestunud regressioonitest. Algoritmi väljundiks on minimeeritud hulk, mis sisaldab ainult neid muudatusi, mis otseselt põhjustavad regressiooni. Varasemates uuringutes kasutati muudatuste saamise eesmärgil ainult lähtekoodi failide harilikku tekstilist võrdlust, mille tõttu saavutatavad jõudlus ja täpsus ei olnud optimaalsed. Käesoleva magistritöö eesmärgiks on tekstilise võrdluse asendamine abstraktsete süntaksipuude võrdlusega ning sellise asendamise mõju uurimine delta-silumise protsessi ajalisele käitumisele ja väljundile.

Käesoleva töö tulemusena on loodud delta-silumise tarkvara prototüüp, mis põhineb süntaksipuude võrdlemisel. Prototüübi hindamiseks on kasutatud reaalsed regressioonivead, mis on kogutud ühest Java keeles kirjutatud suuremahulisest ettevõtte infosüsteemist. Hindamine näitab, et üleminek süntaksipuude võrdlusele mõjub positiivselt; paranenud on nii tõhusus, jõudlus, täpsus, kui ka väljundi usutavus.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 52 leheküljel, 4 peatükki, 14 tabelit, 5 joonist.

List of abbreviations and terms

AST	Abstract Syntax Tree
CI	Continuous Integration
DBMS	Database Management System
IoC	Inversion of Control
RT	Regression Test
SIR	Software-Artifact Infrastructure Repository
TBD	Trunk-Based Development
TDD	Test-Driven Development
VCS	Version Control System

Table of contents

List of figures

List of tables

Introduction

In recent years, numerous approaches were proposed for automatic test-based software fault localization. Specifically, considerable effort was exerted to develop effective methods to automatically isolate source code changes that induce test-detectable *regression* faults. One of the techniques actively utilized for this purpose is a delta debugging algorithm, developed by A. Zeller (Saarland University) and presented to the scientific community in his article from 1999 entitled “Yesterday, my program worked. Today, it does not. Why“? [1]. A variation of this algorithm relevant in the context of the current thesis accepts a set of changes in the source code as an input and produces a 1-minimal set of failure-inducing changes as its output. It belongs to the divide-and-conquer family of algorithms and guarantees linear worst-time complexity.

Dozens of studies have been conducted since the publication of Zeller’s paper with the aim to evaluate the effectiveness, correctness and performance of delta debugging. In addition several tools were built to prove the concept, but none of those tools have reached maturity and grown into a commercial product. The main reason for that seems to be insufficient performance due to typical presence of large number of unresolved test cases after initial splitting iterations. Another concern that is of interest to thesis author is lack of research targeted at solving the problem of effective automated regression fault localization for programs written in the Java language. Therefore, the goal of this work is to find optimized solution to the stated problem for Java programs. To the author’s best knowledge, there exists only one alternative approach (DARWIN [2]) which is principally different from delta debugging, but this method’s scalability is limited to that of SMT solvers and this complicates its application in a fully automated mode on industrial-scale applications.

To give the reader a better understanding of the context of this work, we should mention right away that the primary motivation driver for this study is the presumed potential of weaving technology for automatic fault localization into continuous integration software with the purpose of reducing total time spent on repairing regressions. An intriguing side-effect of the developed method lies in the increased precision of the localizing source of regression. The latter implies gaining better prospects for using this method in conjunction with automated software repair techniques. Chapter 1 provides the complete overview of the motivation.

The main contribution of this work is a prototype implementation of a regression fault localization tool which is targeted at achieving better performance in the above sketched main use case scenario, and providing a well-grounded answer to the question of practical applicability of this optimized version in enterprise-scale Java software development. A prototypical approach described in Chapter 2 of the current thesis combines delta debugging with abstract syntax tree differencing. Unlike previous research, which used the change sets produced by ordinary textual diff tool as an input for delta debugging procedure, this work attempts to perform code manipulations on a more fine-grained level and operates with changes detected between abstract syntax trees of the two program versions. We hypothesize that raising the granularity level will promote the consistency of randomly composed configurations and decrease the number of unresolved configurations, thus improving the overall performance of the algorithm. On the other hand, raising the granularity theoretically leads to a larger number of configurations to assess, therefore the achievable performance gain is not immediately obvious. The practical experiment conducted in the scope of this work shall clarify these concerns.

The prototype is evaluated on a data set of real regression faults collected from a large Java-based enterprise information system; the observations are documented in Chapter 3. During working on this thesis, the author faced the problem of absence of ready-to-use benchmarks for evaluation of regression defect localization tools. Neither broadly known in Java world Defects4J [3] nor Software-artifact Infrastructure Repository (SIR) [4] contained the required test data; therefore the only viable option was to gather it manually from the project well-familiar to the thesis author. Using the information extracted from a real history of source code changes done by a professional team during the ordinary course of software development throughout a year brings additional credibility to the evaluation process.

Finally, in Chapter 4 we discuss the results and give an assessment of the developed technique. This chapter also provides some insights into directions of future work.

1. Background and related work

1.1. Motivational drivers

1.1.1. CI-related motivation for the automated debugging of software regressions

During the last decade, continuous integration (CI) has become a mainstream practice in professional agile software development. Its main purpose is to facilitate early detection of integration problems by frequent merging of developer individual working copies to a shared mainline located at the integration server. To support this approach, numerous tools are available; they largely differ in characteristics and capabilities [5], but share common operating principles. Normally CI software is configured in such a way that it polls periodically the project's VCS repository to determine changes on specific branch. If new source code revisions are found, CI server updates its local working directory and immediately triggers a build job. A build is a complicated process that typically involves compiling, packaging, deployment, database schema migration, as well as running various automated tests. Intermediate results are constantly communicated back to the developers, most usually through some sort of visual representation, sometimes called 'build light indicator'. The exact form of such indicator representing current state of the build may range from web-accessible dashboard displayed on a separate flat screen monitor mounted near the ceiling to more exotic things like colored lava lamps [6]. The information about failing build might also be conveyed in the form of audial warning. Ultimately, the most important requirements are that this indicator is at any time accessible to entire team and each team member is aware that if the build is failing, then commits are disallowed.

Constant feedback on the actual state of the build enables to reduce the time between introducing problematic changes and the moment when the team discovers the problem and starts to solve it. Combined with the disciplined adherence of the team members to the policy of not committing code to remote repository in case the mainline is broken, such feedback allows preventing further build degradation and reduces the effort needed to localize and eliminate the cause of the failure. However, the flip side of this policy is that team members who are ready to commit their changes have to wait until the responsible person resolves the problem. Switching to other development activities during this time is oftentimes not desirable because it is a well-known fact that human context switching is associated with significant loss of productivity [7] [8].

Depending on the build configuration, sense of responsibility of individual team members, established practices, and current stage of release life cycle, the ‘commit window’ during the day might be as small as a couple of hours. Given the above facts, it is natural that fixing the build is considered one of the highest priority tasks in most development teams that use continuous integration. It was also recognized as a priority activity by such major agile proponents as Martin Fowler and Kent Beck [9]. Reducing the time needed to repair the build is thus of key importance to improving team’s overall productivity.

There are a multitude of reasons why a build can fail; therefore, handling the particular situation with unacceptably frequent failures should begin with gathering statistical data about causes of failures and costs of fixing. Although, in order to achieve the best possible results, one should collect the data using utility integrated into CI software itself (for example, Build Failure Analyzer plugin for Jenkins [10]), prioritize the most critical problems using some formal method, and develop effective countermeasures, in the presence of evident predominant reason the procedure might be simplified down to treating this concrete reason. Author’s personal observations, made on large enterprise software projects that utilize continuous integration, show that, in a typical CI pipeline configuration, given a project with reasonably well-developed test suite, the most influential cause of build degradation is failure of automated tests, contributing to the largest amount of total time spent on mainline recovery.

One obvious reason for this is that often developers refuse to verify their changes by running all tests in their local environment before they make a commit to a remote repository. Not always is that a sign of insufficient discipline and development culture – in large teams practicing trunk-based development, where commits to mainline occur very often, it is generally not possible to run the whole test suite locally between each two consecutive commit attempts. If the first commit attempt of developer *A* failed because head revision had already been updated by another developer *B*, developer *A* needs to merge *B*’s changes to the local copy before reattempting the commit. Like every codebase update, merging may introduce new bugs and, with a good test suite, automated tests are likely to catch at least some of them. However, if it takes relatively long time to run all the tests locally, developer *A* is tempted to skip this step and reattempt the commit with unverified changes – otherwise, there is a high chance that in the meanwhile the local copy becomes outdated again. Therefore, there always remains a risk that the problem is discovered only when some tests fail on a CI build pipeline.

Above described scenario is probably the most prevalent way to break the build, but there are also many other widespread causes for that to happen. Particularly noteworthy among those are differences between execution environments. The same test might pass successfully on a local development environment, but fail when being run during CI build. Sometimes this might indicate a problem with the test itself, as is the case with time-dependent tests. Yet another common mistake is writing a test case which relies on the particular order of elements inside dataset. For example, according to the query specification of SQL, in order to guarantee a specific order inside a returned result set, one must use ORDER BY clause. However, most relational DBMSs, including Oracle, tend to fetch the same data in the same order, provided that the query execution plan did not change. In an another environment, the execution plan for the same query might be completely different, which will lead to producing results in another order. To the less experienced developer, who is accustomed to looking only for functional mistakes within the main codebase, understanding that the real culprit is the incorrectly written test can take quite a while. Finally, randomly failing tests, especially those revealing issues rooted in concurrency violations, constitute a separate major class of test failures; reproducing and fixing the underlying cause possesses unique challenges.

Although a test might be failing on CI environment since it was firstly added to the source code repository, the much more usual scenario is that this test had successfully passed previously and the failure started to occur due to the lately made code changes (*here and further by 'code changes', if not explicitly stated otherwise or clearly inferred from context, we mean changes made in main codebase, and not in test code*). In this case, assuming that the test itself is correct, it is said that software regression was introduced. Apart from the most trivial cases, when the cause is immediately obvious from inspecting the exception stack trace or a developer is well familiar with project's codebase, handling the regression defect involves manual inspecting of source code modifications between last-known-good and broken revisions. Code diffs are a valuable source of information which guides a developer towards fault localization and issuing a fix.

The fastest approach to reconciling regressions is arguably reverting commits made since the last-known-good (in the context of particular failing test) revision from the mainline, reproducing the problem locally, fixing it, and committing the fixed version to the remote repository. Thus, trunkbaseddevelopment.com, a notable web resource about the same name approach, recommends this policy, stating that “The best implementations are going to perform automatic rollback of a broken commit that lands in the trunk. The developer gets notified and they get to fix it quietly on their workstation” [11]. Contrary to the latter recommendation, we claim that such setup cannot be universally adopted, because in certain settings it is not scalable enough. With a large automated test base and high rate of commits to mainline, it is highly probable that by the time the CI build gets broken because of newly failed test case and the team becomes aware of this event, another developer(s) has already made their commit(s) with unverified changes, therefore reverting ‘guilty’ revision would be disruptive for them (what we are ultimately pursuing to avoid).

Another interesting alternative to consider is the idea of “pending head”, or “delayed commit”, described by Martin Fowler in his 2007 web article [12], and widely adopted by many development teams today. It is proposed to use short-lived private feature branches for the regular development, have a continuous integration server to perform an integration build, and, if successful, automatically commit the changes to mainline. Fowler claims that “this way you never got broken code into the mainline of the project”, but nothing is said about realistic throughput achievable with this technique. The method implies a strictly sequential workflow, with queueing commits in the processing chain, so the downside of reduced throughput is evident. It is also unclear how the problem with arising merging conflicts is solved when a pending-head branch gets automatically updated with the true project head – not every source code merge can be performed without manual intervention. Fowler himself admits that he was not enough motivated to introduce this method in his company because of its relative complexity and concludes the article by stating: “As usual the people-issue is often a more important issue to deal with before introducing more complicated technology“. Last but not least, another blog article with a brief overview of build pattern in question contains a valuable remark about its potentially detrimental effect of forming a non-healthy attitude when a single person is made exclusively responsible for the mistake [13].

Having described two possible strategies, we are now ready to shortly discuss the most common option. The preferred practice is that a dedicated team member, usually the one who introduced the failure, tries to fix the regression locally without reverting the mainline. It means that during that time the build remains broken and other developers are prohibited to commit to mainline in order to not complicate the situation further. Therefore, there is a need to find a correct fix as soon as possible. Having a reliable technique that would allow to at least partially automate this process by localizing the fault could significantly shorten the time needed to produce a correct patch and be thus of a great practical value. The domain of execution is not very relevant: the fault localization tool might be installed on a local development machine (for example, bundled as IDE plugin) or directly on a CI server (be a part of CI software). Important is the speed-up gained by such semiautomatic regression resolution process.

1.1.2. Automatic defect localization in the context of automated software repair

A more ambitious perspective that drives the motivation for automated fault localization is the ability of the latter to serve as an input for automatic bug-fixing. The idea of automated software repair is relatively new; it started being actively explored only about 15 years ago. One possible definition of automated repair is as follows: „Automatic repair is the transformation of an unacceptable behavior of a program execution into an acceptable one according to a specification“ [14]. Specification can be defined in a multitude of different ways, ranging from formal specification – with the most notable example arguably being the design-by-contract approach popularized by Bertrand Meyer in his Eiffel language, – to the most implicit forms such as a natural language phrase. With the emergence of Extreme Programming (XP) and Test-Driven Development (TDD) paradigm, in the vast majority of modern object-oriented software projects, a collection of test cases validating a set of software program behaviors started to act as an implicit specification of this program. Such specification is sometimes informally called Specification by Example, whereas a collection of test cases is commonly known as a test suite.

A narrower term is ‘oracle’, which denotes a part of specification that captures acceptable output. Within the test suite, oracles take the form of assertions, which compare actual output to the expected at the end of test cases. In the context of automated repair, one may draw a distinction between bug oracles and regression oracles: the former serve to reveal an incorrect behavior, while the latter are required to preserve the existing correct behavior and guard against introducing new regressions. To skip ahead, most of the existing test-suite based fault localization and automated repair techniques operate on the more coarse-grained test case level rather than individual assertion level; in the role of bug oracles are failing test cases, while the passing test cases serve as regression oracles. Based on the above, the problem of test-suite based automated repair can thus be roughly formulated as follows: „given a program and its test suite with at least one failing test case, create a patch that makes the whole test suite passing“ [14]. Note that this definition does not tell anything about actual correctness of the produced patch as it is perceived by experienced developer. The only measure of patch plausibility is test-adequacy; therefore, availability of highly effective test suite is a key precondition for performing automatic repair attempt.

Automated repair techniques heavily rely on accurate fault localization. Identifying the precise location where a fault occurred is crucial to repairing the fault. Many fault localization techniques have been developed during recent years; according to the survey made by Wong et al. [15], all of them can be classified into 8 distinct categories. The most prominent of them is the spectrum-based category, contributing to the largest fraction of recently published papers. This is a group of relatively simple methods united by the same basic idea. A test suite with at least one failing test case which exposes a bug is executed against a program. The statistics on the number of failed and passed test cases for each program unit (most commonly, a statement), as well as total number of failed and passed tests, is collected during test execution. For every unit, a suspiciousness score indicating the degree to which execution pattern of the unit is related to the failure pattern is then calculated using heuristics. The units are then ranked according to the suspiciousness level – the higher priority for repair is given to the actually faulty statement, the better the result of fault localization is considered. Different spectrum-based techniques use different heuristic formulas and the effectiveness of those techniques is not equal, as is shown by controlled experiment conducted by Assiri and Bieman in 2016 [16]. Another study conducted in 2009 by Santelices et al. [17] shows that effectiveness of localizing various kinds of faults is closely related to concrete coverage type – statement, branch, or du-pairs, – used in each case to calculate suspiciousness rank and no single type of coverage performs universally well for all types of faults.

The next logical step after identifying a faulty statement or block is to attempt to replace it with the correct one. A number of various generic methods have been proposed recently, most notably relying on generate-and-validate technique. Statement with the highest suspiciousness rank is taken from the queue and a random mutation operator, optionally, parameterized with context information taken from elsewhere in the program, is applied to this statement, thus producing a new version of the program. Modified statement is called a candidate patch. This patch is validated against all test cases in the available test suite. Failure of at least one test leads to abandoning patch under validation and generating a new one. If no suitable patches are found, the process is repeated with the next most suspicious statement. The procedure terminates if a validated patch is found or if predefined time limit is exceeded.

A variation of this technique, called GenProg [18], was proposed in late 2000s as a general method for fixing software defects. Its main novelty is that it relies on genetic algorithm for finding a suitable patch. At the beginning of each generation a set of random candidate patches represented as the ordered list of abstract syntax tree edits is produced via mutation. A fitness function (defined as weighted average of passing and failing test cases) is then applied to select the best parent individuals from the set. Next, a crossover operation is applied pairwise on the chosen parents, so that a set of edits corresponding to the second parent is appended to the first parent’s set, and then each element is removed with probability ½. The resulting offspring individuals together with parents are mutated again and each candidate patch is evaluated against the whole test suite. If no valid candidate patch is found, the generation cycle is repeated, with the result of previous iteration serving as incoming population. As usual, the procedure is over when a validated patch is found or the resources are exhausted.

Despite still being generally considered a state-of-the-art approach, the effectiveness of GenProg remains controversial. Although the results reported by the authors of GenProg are very promising – in a study conducted in 2012 by Le Goues et al [19] it is claimed that 55 out of 105 bugs were fixed by their system – the later study by Qi et al [20] shows that most of the produced patches are incorrect. Qi et al theorize that in many cases the results could have been better if the search space, in principle, contained successful patches (which in case of GenProg are synthesized from existing source code taken elsewhere from the program) or the space itself was narrower. They also blame weak test suites as a major impediment to generating acceptable patches.

Another study conducted by Smith et al. in 2015 [21] confirms that the quality of the produced patches largely depends on the coverage of the repair test suite. Weak test suite used as an input for patch generation leads to producing a fix that passes all available tests, but fails to generalize and is therefore functionally incorrect. They call such phenomenon ‘overfitting’, drawing analogies with similar problem frequently occurring in machine learning.

Analysis of the existing body of knowledge concerning application of GenProg shows that at the time of writing this document (Autumn 2017), presumably, none of the previous studies have considered using this method in the specific domain of the automated repair of *regression* faults, although some of the authors make assumptions that a history of program modifications might be utilized for improving accuracy of the results. For example, of great interest is the article of Martinez et al [22] in which they claim that „as many as 52% of commits are composed entirely of previously existing tokens“. Given the earlier discussed context of continuous integration environment, where differences between two consecutive program versions are relatively small compared to the total codebase size and regressions are detected promptly, one may hypothesize that reducing GenProg’s fault space to contain only a minimal set of failure-inducing changes represented on a fine-grained level may substantially improve the overall efficiency of the method. Combined with adaptation of fix ingredient selection strategy in order to explore the search space containing relevant ingredients from both program versions (reference program and buggy program), this gives better prospects for ability of GenProg to synthesize a correct patch – which is indirectly supported by findings of Martinez et al [22].

To conclude this section, we should mention that actual adaptations of GenProg implementation are out of scope of this work due to time considerations. However, the improved method of localizing regression faults developed in this thesis can serve as solid foundation for proposed GenProg modifications and contribute to developing alternative methods targeted specifically at automatic fixing of regressions. As we concentrate our efforts on finding a solution for programs written in Java language, it is also worth noting here that there already exists a publicly available open source reference implementation of GenProg in Java, called Astor [23] (https://github.com/SpoonLabs/astor), so the results of this work can be directly applicable for its further development.

1.2.Existing approaches for localizing regressions

Speaking of possible solutions to the formulated problem – localization of regression faults using two program versions – studying the existing literature on the topic revealed two principally different methodologies: DARWIN (Dawey Qi et al [2]) and delta debugging (Zeller [1]).

DARWIN is a method developed in 2009 by a group of researchers from National University of Singapore. Essentially, it is a combination of enhanced version of symbolic execution called concolic execution and constraint solving. Its main idea is to generate an alternative input *t'* for a buggy program *P'* which would satisfy the following rules:

- In reference program *P*, input *t'* is following the same execution path as given input *t* that passes for program *P*, but fails for buggy program *P'*;
- In buggy program *P'*, input *t'* and given input *t* follow different execution paths [2].

The differences of traces obtained by executing *P'* with both input *t* and *t'* are then compared and observed distinction is translated into the cause of failure.

According to the evaluation conducted by authors of the method, DARWIN has a number of significant strengths, compared to delta debugging:

- Ability to discover so-called *unmasking regressions*. Those are types of defects that existed already in version *P*, but were unhidden by the changes expressed as difference between *P* and *P'*. Zeller's delta debugging, by design, is not capable to reveal such defects, because in this scenario the actual faulty statement is not contained within the set of changed statements.
- Ability to tolerate large amount of changes between reference and buggy program. Contrary to this, Zeller's method works well only with relatively small amount of modifications. However, it is not a substantial impediment for us, given the circumstance that we intend to use our developed method in a scenario that presumes close similarity of the two versions.

Nonetheless, the above mentioned strengths of DARWIN are outweighed by its weaknesses:

- Method is not suited well for localizing faults which are not triggered by changes in control flow of the program. In particular, it means that regression defects that are caused by wrong assignments can generally not be diagnosed by this approach. To overcome this problem, authors utilize instrumentation of the program with predicates in order to artificially create branch conditions and alter control flow, but they admit that: (1) there remains an uncertainty about applicability of this solution to all possible cases, (2) the instrumentation negatively affects the performance (overhead is ~20%).
- Since DARWIN is grounded on the SMT solving, its scalability largely depends on that of SMT solvers. Also, path condition size grows exponentially with the size of the program – a phenomenon known as 'path explosion' takes place [24]. Although authors claim that they were able to successfully tackle this issue using heuristics and evaluated their method with several real-world examples, the number of these examples is too small to generalize results. In the absence of other controlled experiment studies, we consider this drawback too serious to employ this method for our purposes.

Taking into account these considerations, the selection of delta debugging approach as a ground for experiments appears to be more promising for the purpose of solving the stated problem. Let us have a detailed look at how this technique works.

1.3. Delta debugging

1.3.1. Operating principle

Delta debugging is a powerful technique which allows to automatically isolate a cause of failure. Multiple typical applications of this algorithm exist, such as narrowing down the failure-inducing program input or even failure-inducing sequence of user interactions (Zeller [25]) (which may be viewed as a variation of the former case), but in its most classical form it operates on the changes to the program code. Its basic idea is to obtain a difference between reference and faulty version of the program, in a form of independent chunks serving as units of change ('deltas'), and perform systematic testing of selected subsets of changes until a minimal failure-inducing change set is found. In order to determine the outcome of each trial, a regression test (RT) capturing a fault of current interest, is used as a bug oracle. A trial consists of four distinct phases (which closely resemble standard Four-Phase testing pattern [26]):

- **Setup**. A subset of delta chunks is applied to the reference program version; the constructed version is recompiled.
- **Exercise**. A RT is executed against the new version and outcome is captured.
- **Teardown**. A working area is restored; all modifications to the reference version are undone.
- **Verify**. An outcome of the RT is evaluated and, depending on the result, a delta debugging algorithm follows one or another execution path.

The process stops when there is detected a change set that satisfies the following condition: removing of any individual change from this set and applying such reduced set to the baseline will cause the RT to not fail anymore. In this case, we say that a regression fault is *localized*.

From the description provided above, it is not clearly understood, how many combinations of changes is supposed to be tested in order to achieve the stated goal. The naïve approach to solving this challenge through a brute force method is to test all 2^n possible combinations, which renders inadequate any attempt to apply such automated debugging procedure to a real-size problem. The optimizations proposed by Zeller in his paper [1], which serve as a basis for his optimized version of delta debugging algorithm called dd^+, allow to substantially reduce the number of combinations to be tested and thus devise a much more effective way of delta debugging. At the ground of enhanced technique lies the observation that proper decomposition of difference descriptor into delta chunks allows for making informed assumptions regarding the possibility that particular selected subset of chunks contains a minimal failure-inducing set. The subsets themselves are not chosen arbitrarily, but rather according to the certain pattern that takes into account the intermediate results obtained at the previous stages of the algorithm execution. On each step, algorithm evaluates the current subset and, depending on the outcome, gradually reduces the size of the search space, until no further reduction is possible. At this point, the search space contains the minimal set we are looking for, and the algorithm ends.

The next logical question is how exactly a solid representation of difference between two program versions could be split in order to be suitable for conducting an optimized search? What properties should the delta chunks satisfy?

Before proceeding to the listing of properties and explanation of the algorithm itself, we should provide some definitions to facilitate the understanding. The following is a succinct digest of the information provided in the Sections 2 and 3 of the original paper [1] combined with some additional clarification:

- **Configuration**. Given that $C = \{\Delta_1, \Delta_2, \dots, \Delta_n\}$ is the set of all possible chunks Δ_i , a subset $c \subseteq C$ is called a *configuration*. Configurations are to be applied to the reference program version.
- **Baseline**. An empty configuration (i.e. such that $c = \emptyset$) is called a *baseline*. Baseline applied to the reference program version is the reference version itself.
- **Testing function** is a function defined in the form $2^C \rightarrow \{✘, ✔, ?\}$, where C is the set of all possible changes and $\{✘, ✔, ?\}$ are the encoded test outcomes. (✘, or FAIL) stands for the outcome when a test failed in the same way as in the faulty program version; (✔, or PASS) means that test passed successfully, and (**?**, or UNRESOLVED) corresponds to the situation when test resulted in an indeterminate result (such as compilation failure or it produced a failure different to the original one). A testing function is to be applicable to any configuration $c \in 2^C$. We will refer to this function later in this work by an identifier *test*.
- **Failure-inducing change set** is a set $c \subseteq C$ that satisfies the following: $\forall c'(c \subseteq c' \subseteq C \rightarrow test(c') \neq ✔)$. In other words, applying any superset of a failure-inducing change set leads to a non-successful test outcome (either FAIL or UNRESOLVED).
- **Minimal failure-inducing change set** is a set $B \subseteq C$, for which the following holds: $\forall c \subset B(test(c) \neq ✘)$. In other words, applying any proper subset of a minimal failure-inducing change set leads to a non-failing test outcome.

According to the provided definitions, preconditions regarding a reference and a faulty program version to be used as an input for delta debugging could be formulated in the following way: $test(\emptyset) = ✔$ and $test(C) = ✘$.

When decomposing a differential descriptor into chunks, the following properties must be fulfilled by the resulting complete configuration C (consisting of all changes):

- **Monotony**. If applying some subset of changes leads to a failing test outcome, applying any configuration that includes this subset will lead to a non-successful test outcome (FAIL or UNRESOLVED). Using a definition given above, in case of monotone complete configuration, any superset of a failure-inducing change set is also failure-inducing. More formally:

  $$\forall c \subseteq C \; (test(c) = \text{✘} \rightarrow \forall c' \supseteq c \; (test(c') \neq \text{✔}))$$

  This works the other way around as well. If applying some subset of changes leads to a successful test outcome, applying any configuration that is a subset of the given subset will lead to a non-failing test outcome. Formally:

  $$\forall c \subseteq C \; (test(c) = \text{✔} \rightarrow \forall c' \subseteq c \; (test(c') \neq \text{✘}))$$

  In practice, this property allows us to narrow down the search space.

- **Unambiguity**. We assume that only one change set causes a failure and not several disjoint change sets independently. Formally:

  $$\forall c_1, c_2 \subseteq C \; (test(c_1) = \text{✘} \wedge \mathrm{test}(c_2) = \text{✘} \rightarrow test(c_1 \cap c_2) \neq \text{✔})$$

  In practice, this means that once a failure-inducing change set is found, there is no need to search the complement for other failure-inducing sets.

Another property which is not strictly mandatory, but is highly desirable, is **consistency**. Consistency means that within a given complete configuration, any randomly combined configuration produces a deterministic test result:

$$\forall c \subseteq C \; (test(c) \neq ?)$$

Failing to fulfil this property leads to considerable decrease of algorithm efficiency, as will be discussed below.

1.3.2. *dd*$^+$ algorithm

To begin with, Figure 1 displays the formal definition of Zeller's delta debugging algorithm dd^+:

```
$dd^+(c) = dd_3(c, \emptyset, 2)$, where
$dd_3(c, r, n) =$
    let $c_1, \dots, c_n \subseteq c$ such that $\bigcup c_i = c$, all $c_i$ are pairwise disjoint, and $\forall c_i (|c_i| \approx |c|/n)$;
    let $\overline{c_i} = c - (c_i \cup r)$, $t_i = test(c_i \cup r)$, $\overline{t_i} = test(\overline{c_i} \cup r)$,
        $c' = c \cap \bigcap\{\overline{c_i} \mid \overline{t_i} = \text{✘}\}$, $r' = r \cup \bigcup\{c_i \mid t_i = \text{✔}\}$, $n' = min(|c'|, 2n)$,
        $d_i = dd_3(c_i, \overline{c_i} \cup r, 2)$, and $\overline{d_i} = dd_3(\overline{c_i}, c_i \cup r, 2)$
```

$$
in \begin{cases} c & if\ |c| = 1 \text{ ("found")} \\ dd_3(c_i, r, 2) & else\ if\ t_i = \text{✘ for some i ("found in } c_i\text{")} \\ d_i \cup \overline{d_i} & else\ if\ t_i = \text{✔} \wedge \overline{t_i} = \text{✔ for some i ("interference")} \\ d_i & else\ if\ t_i = ? \wedge \overline{t_i} = \text{✔ for some i ("preference")} \\ dd_3(c', r', n') & else\ if\ n < |c| \text{ ("try again")} \\ c' & otherwise \text{ ("nothing left")} \end{cases}
$$

Figure 1. A. Zeller's delta debugging algorithm dd$^+$ [1]

For any smaller case of dd_3, a recursion invariant $test(r) \neq \text{✘} \wedge test(c \cup r) \neq \text{✔} \wedge n \leq |c|$ holds.

We will omit non-essential details for the sake of brevity and outline only main ideas necessary for understanding the subsequent material.

Dd$^+$ is, by definition, a recursive divide-and-conquer algorithm. The base case of the recursion is reached when the current configuration consists of only a single element ("*found*" case), or when there are no further recursion steps possible and the currently processed configuration becomes the only candidate for being the failure-inducing one ("*nothing left*" case). Otherwise, we start splitting the current configuration into n subsets $c_1, \dots, c_n$ (initially, $n = 2$) and test each subset and its complement separately. The most useful case here is "*found in* c_i", since it allows to immediately reduce the search space to a proper subset c_i of a current configuration. "*Interference*" denotes a situation when testing both subset and its complement separately produces in each case a positive test result, so this indicates that a failure is caused by a combination of some changes from both subsets. "*Preference*" case happens when testing a subset produced an indeterminate result, but the complement of this subset passed the test; we assume that the first subset contains failure-inducing changes (possibly in combination with some changes from the complement subset – that is why it remains applied in the subsequent invocations). Finally, if no other choices are possible, we increase the granularity of search by splitting the current configuration into [twice] more subsets (“*try again*” case), in the hope that it will raise the chances of getting a consistent configuration.

One final remark before we move on to the concrete example is that through the recursive calls certain “safe” changes may remain applied; those changes are denoted by a literal r.

Let us now have a look at the example demonstrating the mechanics of *dd*$^{+}$. Consider the Table 1, where every line represents a configuration. ‘⊙’ stands for a change that is included in the configuration, and ‘.’ (a dot) represents an excluded change. In the analyzed example, a combination of changes Δ_4 and Δ_5 is a minimal failure-inducing change set, and Δ_5 depends on Δ_4 (i.e. it cannot be applied without applying Δ_4). Cells with a light-green background correspond to the changes that remain applied in the current trial.

Step	c_i	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7	Δ_8	***test***	**Progress**
1	c_1	⊙	⊙	⊙	⊙	.	.	.	.	✔	
2	c_2	.	.	.	.	⊙	⊙	⊙	⊙	?	⇨ prefer c_2
Decision: “*preference*”											
3	c_1	⊙	⊙	⊙	⊙	⊙	⊙	.	.	✘	⇨ c_1 fails
4	c_2	⊙	⊙	⊙	⊙	.	.	⊙	⊙	✔	
Decision: “*found in* c_i”											
5	c_1	⊙	⊙	⊙	⊙	⊙	.	.	.	✘	⇨ Δ_5 found
6	c_2	⊙	⊙	⊙	⊙	.	⊙	.	.	✔	
Decision: “*preference*” – search the other half now											
7	c_1	⊙	⊙	.	.	⊙	⊙	⊙	⊙	?	
8	c_2	.	.	⊙	⊙	⊙	⊙	⊙	⊙	✘	⇨ c_2 fails
Decision: “*found in* c_i”											
9	c_1	.	.	⊙	.	⊙	⊙	⊙	⊙	?	
10	c_2	.	.	.	⊙	⊙	⊙	⊙	⊙	✘	⇨ Δ_4 found

Result: $\{\Delta_4, \Delta_5\}$.

Table 1. Example of searching for a minimal failure-inducing change set using dd^{+}

As shown, it took only 10 trials to find a set of changes causing a failure for a configuration consisting of 8 changes (*compare with 2^8 = 256 trials required in a brute-force approach!*). Now let us show a more complex example (see Table 2 and Table 3), where more than two changes (Δ_2, Δ_5, Δ_7) imply each other and are at the same time failure-inducing.

Step	c_i	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7	Δ_8	***test***	**Progress**
1	c_1	⊙	⊙	⊙	⊙	.	.	.	.	**?**	
2	c_2	.	.	.	.	⊙	⊙	⊙	⊙	**?**	⇨ split again
Decision: "*try again*", n = 4, $\lvert c' \rvert = 8$											
3	c_1	⊙	⊙	.	.	.	.	.	.	**?**	
4	c_2	.	.	⊙	⊙	.	.	.	.	✔	
5	c_3	.	.	.	.	⊙	⊙	.	.	**?**	
6	c_4	.	.	.	.	.	.	⊙	⊙	**?**	
7	$\bar{c}_1$	.	.	⊙	⊙	⊙	⊙	⊙	⊙	**?**	
8	$\bar{c}_2$	⊙	⊙	.	.	⊙	⊙	⊙	⊙	✘	
9	$\bar{c}_3$	⊙	⊙	⊙	⊙	.	.	⊙	⊙	**?**	
10	$\bar{c}_4$	⊙	⊙	⊙	⊙	⊙	⊙	.	.	**?**	⇨ split again
Decision: "*try again*", n = 6, $\lvert c' \rvert = 6$											
11	c_1	⊙	.	⊙	⊙	.	.	.	.	✔	
12	c_2	.	⊙	⊙	⊙	.	.	.	.	**?**	
13	c_3	.	.	⊙	⊙	⊙	.	.	.	**?**	
14	c_4	.	.	⊙	⊙	.	⊙	.	.	✔	
15	c_5	.	.	⊙	⊙	.	.	⊙	.	**?**	
16	c_6	.	.	⊙	⊙	.	.	.	⊙	✔	
17	$\bar{c}_1$	.	⊙	⊙	⊙	⊙	⊙	⊙	⊙	✘	
18	$\bar{c}_2$	⊙	.	⊙	⊙	⊙	⊙	⊙	⊙	**?**	

… to be continued on the next page

Table 2. More complex example of dd+ usage, involving increase of granularity

Step	c_i	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7	Δ_8	*test*	Progress
19	$\overline{c_3}$	⊙	⊙	⊙	⊙	.	⊙	⊙	⊙	?	
20	$\overline{c_4}$	⊙	⊙	⊙	⊙	⊙	.	⊙	⊙	✘	
21	$\overline{c_5}$	⊙	⊙	⊙	⊙	⊙	⊙	.	⊙	?	
22	$\overline{c_6}$	⊙	⊙	⊙	⊙	⊙	⊙	⊙	.	✘	⇨ nothing

Result: $\{\Delta_2, \Delta_5, \Delta_7\}$ – mutual intersection of failed complements $\overline{c_1}, \overline{c_4}, \overline{c_6}$.

Table 3. More complex example of dd+ usage, involving increase of granularity (continued)

To summarize, in theory, behavior of *dd*$^+$ is tolerable even in presence of fair amount of inconsistency. Still, compared to the case when configuration is completely consistent, the negative impact of inconsistency on performance turns out to be significant.

1.3.3. Inconsistent configurations and their influence on efficiency

In the ideal case, when failure is caused by only a single change and each set of arbitrarily taken chunks forms a consistent configuration, the complexity of the algorithm is logarithmic. If there are multiple changes constituting a minimal failure-inducing change set, the complexity degrades down to linear – consider the extreme case when each pair of testable subsets causes interference, because every single change in the complete configuration is failure-inducing, and only a combination of all of them forms a minimal change set responsible for regression.

Things change when inconsistency comes into play, which is very common in practice. Indeed, given the sufficiently large subset of ordinary text chunks representing the modification done between two program versions, the possibility that, when applying it to a reference version, the resulting code will be compilable is not very high. Of course, as was shown, handling the inconsistency is the integral part of *dd*$^+$ and proposed tactics of coping with it (see cases "preference" and "try again" of the formal definition) will eventually lead to the correct result. Splitting to more subsets of the smaller size works reasonably well, since it leads to less difference between known consistent configuration (either baseline or complete) and a current configuration under test, hence the chances of successful compilation are higher. However, with more subsets, more trials have to be performed, and as we know, each trial is relatively expensive, because it involves compilation. Furthermore, despite that Zeller claims the worst-case complexity of the complete version of his algorithm to be still linear (only "requires twice as many tests" [1]), we argue that, according to the provided definition, the time complexity of dd^{+} due to "try again" maneuver is more likely on the order of $O(n \cdot \log n)$. All this moves us further away from the theoretical examples shown earlier in this section and brings us closer to reality, in which proper strategy for *reducing* the degree of inconsistency prior to delta debugging, or *avoiding* inconsistency altogether, becomes essential.

This problem is acknowledged already in the seminal paper by Zeller [1] and some strategies are proposed there for tackling inconsistency. Author specifically advises to try grouping mutually related fragments on the basis of certain common characteristics – example of this are statements involving definition and usage of the same variable. Another suggestion is to try ordering the fragments in a way that will allow predicting the outcome of the test without actually conducting it. Both solutions imply, to a certain extent, structural analysis of the program and tie the implementation to the concrete programming language. The case studies given in the publication display a remarkable positive effect of such optimizations; however, the exact methodology used to implement them is not explained. Without this knowledge, the usefulness of reported results is compromised, because it is extremely difficult to reproduce exactly the same preconditions and repeat the experiment with another data set, let alone generalize the technique to other real-world examples.

Upon reading the paper and analyzing the flow of discussion, one circumstance immediately attracts attention. Namely, the input on which Zeller's implementation operates, are the ordinary text chunks produced by a text differencing tool. Not only does this induce a lot of noise to the obtained diff – since it makes it hardly possible to filter out the changes clearly irrelevant to the regression, – but it also jeopardizes the precision of fault localization. Even if the result of delta debugging run against such input consists of only a single (textual) chunk, this chunk may itself span dozens of source code lines, so that browsing it for a particular statement causing a failure still requires considerable manual effort. The more correct way seems to perform differencing between reference and faulty version on the abstract syntax tree (AST) level, so that each chunk was represented by a descriptor of a comparatively small modification done to AST node. The operation of applying configuration to the reference version can thus be expressed as rolling the selected set of AST node-level changes to the reference program's abstract syntax tree. In addition to the advantages already mentioned, the alleged effect of switching to high-granularity mode is the reduction of inconsistency, since the finer-grained modifications shall improve the chances of successful compilation.

Searching through the scientific literature did not reveal any publications elaborating on the idea presented. Most probably this is due to the fact that the comprehensive libraries for performing tree differencing emerged only recently. Specifically, the active development of these tools for Java language began about 10 years ago; the following chapter will give a brief overview over existing products. At the time of Zeller’s first publication on the subject [1], presumably, there were no such tools available for any of the other widespread OOP languages.

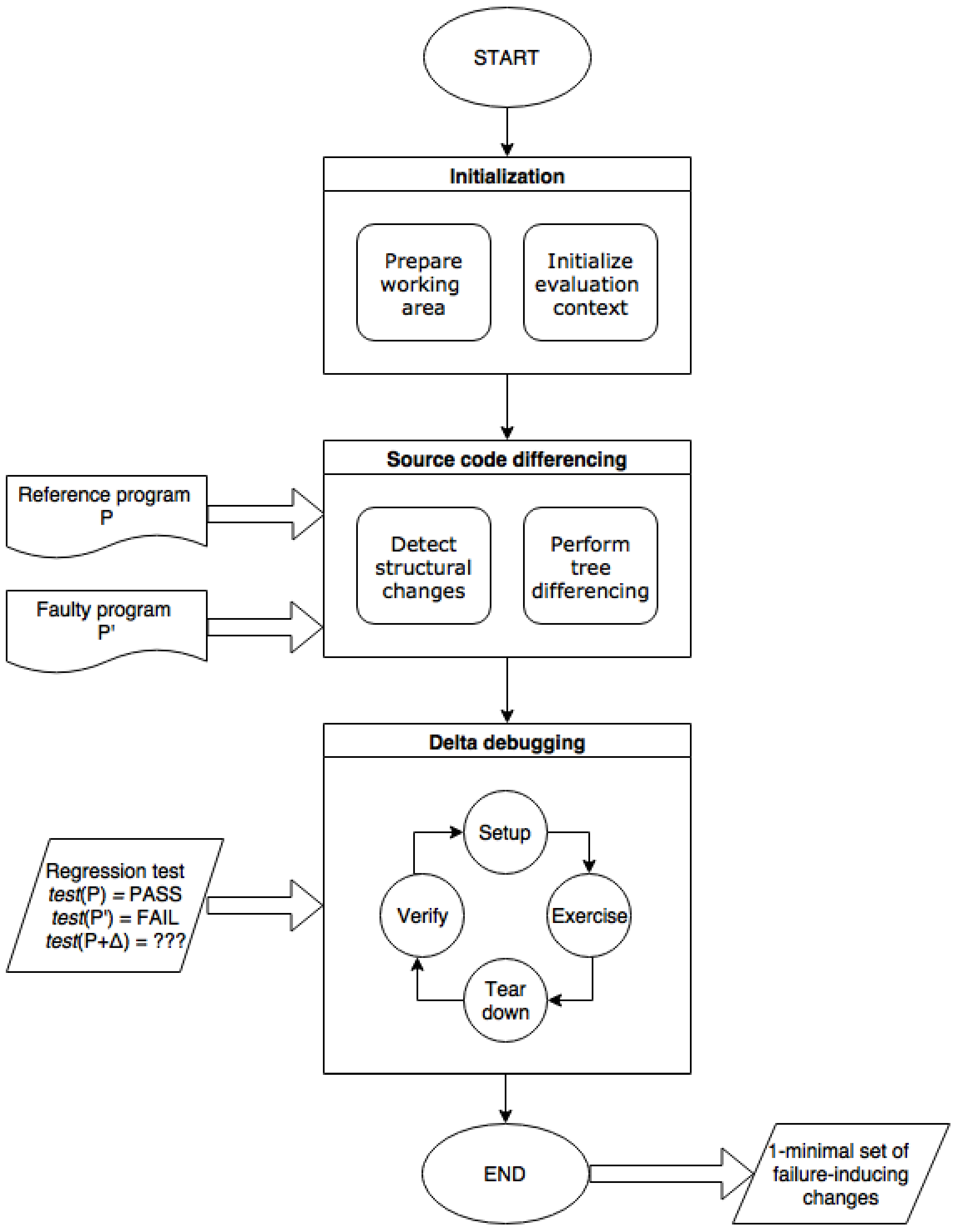
To conclude the chapter, we formulate the objective of the current work as follows: The goal is to develop an improved method for localizing regression faults, which aims to increase the performance and accuracy by utilizing the output of AST tree differencing as input for delta debugging. A secondary objective is to evaluate the suitability of developed method in the typical scenario of trunk-based development, where a regression is introduced by a new commit and is promptly detected by a continuous integration tool. Since, unlike textual diff of the source code, complete tree differencing solution is not a language-agnostic technique, our implementation is tied to the particular programming language, which affects the choice of particular tools. As a basis for experiments we choose Java language, because it is the most familiar programming language to the author of the thesis.

The next chapter presents the prototypical implementation of a tool that utilizes the proposed approach.

2. Methodology

2.1. Overall workflow

Figure 2 displays the simplified workflow of our prototypical implementation called DDFine.



START
Initialization
Prepare working area
Initialize evaluation context
Source code differencing
Reference program P
Faulty program P'
Detect structural changes
Perform tree differencing
Delta debugging
Regression test test(P) = PASS test(P') = FAIL test(P+Δ) = ???
Setup
Exercise
Tear down
Verify
END
1-minimal set of failure-inducing changes

Figure 2. Overall workflow of DDFine application

The minimally required input is a test that executes a regression fault to be localized, and two program versions: a reference and a faulty. The whole process consists of three phases: initialization, source code differencing, and delta debugging. In the first phase, a working directory is prepared and evaluation context initialized. The second phase calculates the difference between the source code of a reference and a faulty version of the program and produces the set of minimal non-intersecting changes that could be applied to the reference version. Finally, in the delta debugging phase, the main action takes place – subsets of changes detected at the previous step are systematically applied to the reference version, with test re-execution and anticipation of the next candidate subset to try (as described in 1.3.1). The process stops when no further progress can be made or when a time limit of 60 minutes is exceeded.

2.2.Requirements and assumptions

In order to assess the genuinely achievable potential of the presented technique for the purposes of incorporating it into CI software, the prototype must simulate the characteristics of the final product as closely as possible. Therefore, as a first step of the prototype design process we elicit key functional requirements for the system and identify the realistic assumptions concerning the nature of the input. For the reading convenience and ease of reference, every requirement is given a short ID and data is organized in a table format.

Req. ID	Requirement Definition
REQ.01	Source code differencing must support modifications done both on structural level (adding, removing, moving, renaming of files and directories) and on the level of a single file.
REQ.02	On the level of a single file, changes must be identified with preciseness of AST tree node. Consequently, if the failure can be attributed to a set of changes done on a file level, the result of localization is a minimal set of modified AST nodes.
REQ.03	If the failure is caused by a new or deleted structural unit (file or directory), the localization is repeated multiple times with increasing the granularity to narrow down to minimally detectable AST node modification(s).
REQ.04	A mechanism used for source code differencing must produce a set of non-overlapping changes, since monotony of configuration is a prerequisite for conducting delta debugging.

REQ.05	Changes produced by the source code differencing mechanism must be in a format that allows applying them to the reference version.
REQ.06	The differencing module must support all types of source code modifications that are considered syntactically and semantically correct for the given language (Java).
REQ.07	The system should not differentiate between input test cases that fail in faulty version with assertion failure or due to unexpected exception; both kinds of faults must be treated uniformly.
REQ.08	When executing a regression test and verifying results, in case of test failure the system must be able to compare error message and stack trace against the original message and original stack trace. If during a trial the regression test failed for a different reason, the result of the trial must be considered as UNRESOLVED.
REQ.09	It must be possible to configure the maximum time limit for a single run. The practically reasonable limit for an automated localization is 60 minutes since the regression was detected on CI; after this period the negative cumulative effect of impeding the normal merging process arguably outweighs the effort spent by a developer to find the cause of the problem.
REQ.10	In case of terminating an attempt in the middle of delta debugging phase, the intermediate results are reported to the user.
REQ.11	The system must be run in command-line mode.
REQ.12	The system should not require user interaction of any kind; it must be completely autonomous.

Table 4. Requirements specification of the system

Certain assumptions are made regarding the input provided to the system, given the restrictions imposed by the delta debugging technique itself and the intended usage scenario. Here is the list of the assumptions that are not checked by the system and are expected to be true when a new execution is triggered:

- Both reference and faulty version provided as input are compilable. All external dependencies required for compilation are provided.
- Provided the regression test correctly captures the fault to be fixed in the faulty version and does not need to be adjusted due to modifications introduced between the two versions. It passes successfully in the reference version.
- The regression test exists in both versions and is unchanged between them.
- There are no changes in external dependencies of the debugged application.

- Test outcome is consistently repeatable between invocations.
- Test failure is not related to the execution environment setup. The necessary setup for executing the test is prepared.
- The provided reference version corresponds to the source code commit which is an immediate predecessor of a commit that introduced a regression in the given faulty version. The fact that the two versions are consecutive implies the relatively low number of source code modifications to evaluate.
- In a faulty version, there exists only one reason for a failure of the provided test case. The failure is caused by a single set of changes and not by multiple sets independently. In other words, the complete configuration is unambiguous.
- There can be multiple unrelated regression bugs introduced in the faulty version. Provided test case captures the bug to be localized during particular execution.

In addition, some of the requirements not strictly essential for the evaluation are relaxed or even dropped altogether in order to save effort and time required for the prototype development. The table below shows the adjusted requirements, together with summarizing the scope and impact of the appropriate modification. Numeric part of identifiers corresponds to the same part of matching original requirement's identifier.

Req. ID	**Difference to original requirement**
pREQ.01	Changes related to moving and renaming of files and directories are not identified as such by the prototype; they are reflected simply as additions and deletions of structural units. This impacts potentially achievable performance, because it makes impossible to apply those 'harmless' changes to the reference version in advance (as a kind of pre-optimization step).
pREQ.04	It is not completely verified whether a differencing mechanism to be used produces strictly disjoint set of changes in all the cases. The accuracy of the result might be compromised to a certain degree.
pREQ.05	For the prototype, it is enough that ~50% of all detected changes could be applied.

Table 5. Relaxed requirements for the prototypical implementation

Requirement **REQ.03** is dropped for the prototype. As a consequence, in some cases the fault might be localized with the granularity of a structural unit, and not an AST node.

Since source code differencing mechanism is one of the central parts of the designed system, the justification of a choice of the proper tool that satisfies the related requirements deserves closer attention. We skip the details concerning the detection of changes on the structural level (files and directories) since it is a trivial programming task and does not need further explanation, and move straight to the more engaging aspect: the internals of the method used to identify fine-grained changes on the single-file level.

2.3.Selection of a tree differencing algorithm

2.3.1. Abstract syntax trees and AST differencing

Abstract syntax trees are a common form of representing structure of computer programs. In computer science literature, they are often contrasted with concrete syntax trees, otherwise called parse trees. The difference between the two is that parse tree reflects the exact syntactic structure of a program written in a language according to its context-free grammar [27], whether abstract syntax tree enables a more succinct view on the structure, in a way that simplifies conducting program analysis and perform program transformation [28]. Probably the most well-known application of abstract syntax trees is compilers; however, it is not the way of usage we are mainly interested in for the purpose of this work.

Abstract syntax tree differencing, or AST differencing, is a compelling idea that began to evolve in the early 2000s. As the name implies, it is a procedure that computes the difference between ASTs of two versions of the same program. Output produced by this procedure is commonly called an edit script; in essence, it is a sequence of edit actions made to the first AST in order to obtain the second one. The eventual goal is to reflect the actual change made by developer as clearly as possible; therefore, the existing tree differencing realizations strive to find a minimal edit script. Unlike its textual counterpart, most notably represented by Myers algorithm [29], tree differencing works at a considerably higher level of granularity than a whole text line and is able to deal not only with insert and delete edit actions, but also with updates and moves. The latter makes AST differencing irreplaceable for analyzing just source code changes, since refactorings involving moving of statements naturally occur in the course of software evolution. Besides, being inherently a structure-centric method, tree differencing effectively dismisses mere formatting changes and thus serves as a first filter for retaining the potentially regression-relevant changes only.

There exist several competing algorithms for performing AST differencing. They vary in details, but share a common working principle. First, they traverse both trees to identify matching pairs of parent nodes, with the restriction that each node may be included into only one pair. Additionally, the nodes are considered as matching based on their labels. On the second step, the actual generation of edit script takes place, based on the mappings determined at previous step. For this part, there are already developed optimal algorithms of complexity O(n^2) (see, for example, the work by Chawathe et al. [30]), so that ongoing research concentrates on finding the optimized solution for the first problem.

It turns out that even if considering only three types of edit actions – insertions, deletions and updates, – the best exact algorithm for computing mappings between ASTs has cubic complexity, as proven by Pawlik et al. [31]. Addition of the fourth operation – move – makes the problem NP-hard. In practice, tree differencing algorithms resort to heuristics to tackle excessive complexity. The optimizations can be targeted at particular edit action type, such as move-actions thoroughly surveyed by Dotzler et al. in 2016 [32], or be generic. The recent articles on the topic mention only three general-purpose AST differencing algorithms which have a significant impact: GumTree, ChangeDistiller, and RTED.

2.3.2. Comparison of available tree differencing tools

Detailed evaluation of above mentioned state-of-the-art algorithms is clearly out of scope of this work, so in order to justify the selection of concrete tool to build the prototype on, it is reasonable to rely on the lately published comparative articles on the matter. Among the articles that appeared within the last 5 years, the best cited paper that presents the results of empirical evaluation of the most promising tree differencing techniques is “Fine-grained and accurate source code differencing” by Falleri et al. (2014) [33]. Although the reported research findings cannot be considered absolutely trustworthy – the authors of the paper conduct the comparison in the context of evaluating their own developed technique (GumTree), – the described experiment setup and public availability of research dataset convincingly demonstrate the effort applied to provide the unbiased information. Based on experimental findings by Falleri et al, at least one technique can be ruled out from further consideration immediately: RTED.

Compared to other two, this algorithm has too high time and space complexity and is not practically applicable to real data. Additionally, it is not able to identify move-actions, which is a crucial drawback for our target scenario.

This leaves us with two choices: GumTree or ChangeDistiller. Both algorithms have reference implementations available in public domain (see [34] and [35], correspondingly), and both have an integrated AST parser for Java language. In the above-referred study, the authors show the evidence of superiority of GumTree over ChangeDistiller, in terms of performance and edit script size. In spite of this, there is another factor that becomes definitive when deciding between two differencing implementations to be used in the designed fault localization tool: the degree to which it satisfies the requirements **REQ.04**, **REQ.05**, **REQ.06**. As stated before, requirements **REQ.04** and **REQ.05** have less strict versions **pREQ.04** and **pREQ.05** devised specifically for building the prototype; on the other hand, requirement **REQ.06** is absolutely essential both for the final product and the prototype. As for **REQ.05**, data structure describing the single modification done to AST node must contain all the necessary information to "replay" the arbitrary set of changes on the original AST. Likewise, data structure representing the abstract syntax tree extracted by parser should support operation of applying given set of changes.

Surprisingly, neither GumTree's nor ChangeDistiller's APIs were designed with out-of-the-box support of rolling up selected modifications to original tree in mind. As of autumn 2017, available reference implementation of ChangeDistiller uses internally a standalone version of Eclipse compiler called ECJ, which contains an AST parser module that provides only a read-only view on the tree. Contrary to this, GumTree's implementation for Java utilizes standard Eclipse JDT compiler which allows manipulating the tree [36], but ties its users to Eclipse platform and is thus intended for development of Eclipse plug-ins, and not standalone applications. Even worse, in both GumTree and ChangeDistiller implementations, the data structures used as descriptors of detected AST node changes are purely representational, in the sense that they do not hold a reference to the instance of modified node itself or its parent node. What they are is an abstract representation of a change that contains sufficient information for a developer to visually identify the proper place in the code, but not enough to apply the change to the tree directly and trigger recompilation.

Searching for a way to quickly overcome this impediment for the prototyping purposes leads to a solution in which the subject of manipulation during delta debugging phase is, again, source code of the reference program version. Figure 3 shows the descriptor (class `SourceCodeChange`) of a fine-grained change detected by ChangeDistiller, together with some of the related classes.

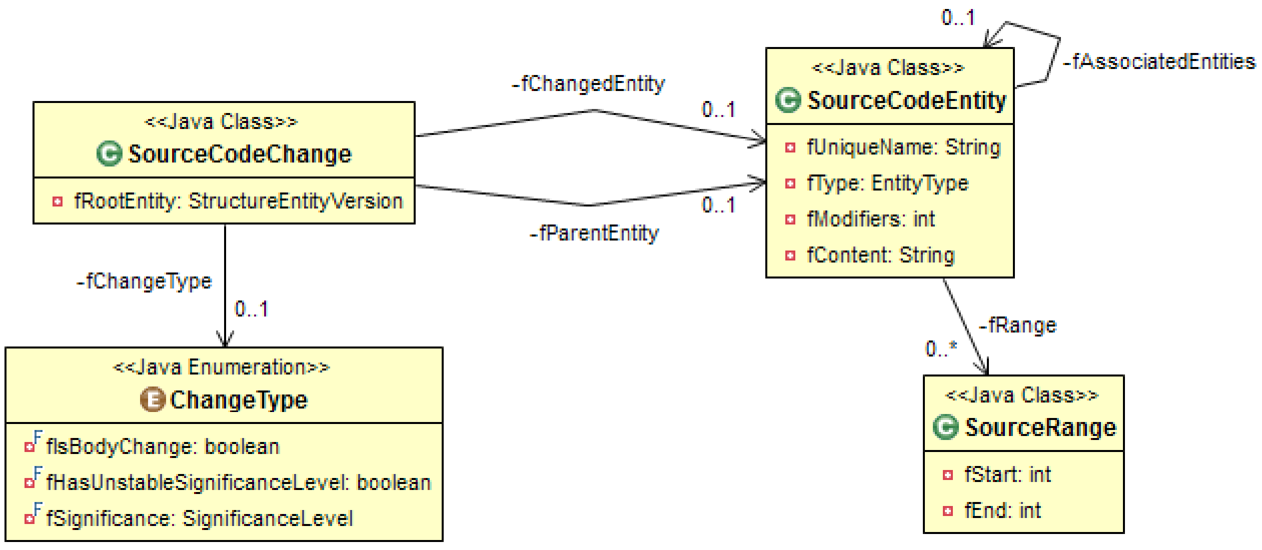


Figure 3. ChangeDistiller's modification descriptor and related entities

As is seen from the class diagram above, class `SourceCodeChange` references through a composition relationship class `SourceRange`, which stores start and end position of a changed entity and its parent in the source document. Furthermore, the changes are classified according to taxonomy, with each change type being assigned a significance level – exactly as explained in an article by Fluri et al [37]. In total, ChangeDistiller distinguishes between 48 types of changes (defined as constants in `ChangeType` enumeration, not shown on the diagram). Compared to this, the descriptor of GumTree does not provide any reference to the location of the change in the source code document. Given the other restrictions described earlier and the roughly estimated effort of adapting GumTree to satisfy the requirements, it was decided to exclude it from the further consideration.

2.4. DDFine: A prototypical implementation

This section presents a prototype DDFine created as a practical part of the thesis. The source code of the prototype is publicly hosted on GitHub and is accessible via URL http://bit.ly/2zCnTZe.

The prototype implements the workflow described in 2.1. It is written in Java 8 and leverages the Spring Framework ver. 4.3.11 as an IoC container and a provider of core application services. For smooth dependency management and easy configuration, Spring Boot 1.5.7 is used. The project is built with Maven; instructions on how to set up the project locally and run the example can be found on this page: http://bit.ly/2AagbT8. Application is launched from the command line (**REQ.11**) and does not require user interaction (**REQ.12**).

As a part of Spring application context startup, an initialization of evaluation context for the current execution takes place. To minimize performance overhead, almost no specific validation of the input, besides basic checking for presence of mandatory arguments, is done at this stage; it is expected that all the assumptions listed in 2.2 hold. The sequence of the actions taken during initialization phase can be summarized as follows:

1. Using paths to project root of reference and faulty version, build in-memory representations of both projects' structure. The prototype works with Maven projects only and expects that they follow standard hierarchical layout for multi-module Maven projects.
2. Prepare working directory for conducting a delta debugging session. Reference version is copied over to the temporary directory and compilation is triggered via Maven Invoker plugin. Like in the previous step, an in-memory representation of the project's clone in the working directory is created and stored in the evaluation context for the future reference.
3. Using a test method given as input, obtain the original stack trace of the failure. The faulty version is compiled and a test is executed; the result (instance of `Throwable`) is stored in the evaluation context in order to perform later the exact comparison, as described in **REQ.08**.
4. Finally initialize a statistics tracker, which is used to gather various metrics during particular execution. The tracked metrics and a method used to collect them will be explained in Chapter 3 (Evaluation).

The source code differencing phase begins with comparing reference and faulty project hierarchies to determine possible structural changes. Only changed directories and .java files are considered. Consistent with requirement **pREQ.01**, moving and renaming of structural entities is not fully supported by the prototype. Instead, all structural changes are initially classified into 5 groups: added files, removed files, added directories, removed directories, and modified files. The first four types are represented by data structure `MinimalStructuralChange`, which is one kind of chunks serving as input to delta debugging function. Modified files are run through ChangeDistiller (**REQ.02**, **REQ.06**); the detected fine-grained changes are represented by instances of `MinimalChangeInFile` – another kind of chunk for delta debugging. No exact validation for monotony is performed (**pREQ.04**). However, as a tiny pre-optimization step, certain types of insignificant changes that cannot possibly contribute to regression and are not required for compilation are filtered out at this point. Such changes are, for example, changes in ordinary comments and Javadoc comments.

The central part of the whole solution is, of course, a delta debugging phase. Searching the Internet for available open-source implementations of delta debugging algorithm brought up several competing realizations in Java, but upon closer investigation none of them deemed to be compliant with dd^+ definition given in 1.3.2. Ultimately, the JAR library containing the seemingly proper implementation of dd^+ (confirmed by decompiling the library) was found in an archive containing the sources for Eclipse plugin called DDState: https://www.st.cs.uni-saarland.de/eclipse/ddstate_sources.zip. The .jar file is extracted and included in `/lib` folder of prototype sources (ddcore.jar).

As recalled from Section 1.3.1, each delta debugging trial starts with applying selected subset of chunks to the reference version stored in working directory. Section 2.3.2 explains why direct manipulation of reference version AST is too difficult at the moment, and why, as a workaround, it was decided to apply modifications to the source code of the reference version instead. To meet the requirement **pREQ.05**, a preliminary support of 25 types of changes was implemented (out of 48 detectable by ChangeDistiller). The types of changes, for which modification operators were created, are selected on the basis of estimated frequency of occurrence in the real projects and ease of implementation. Here is a table of changes supported by DDFine:

Group	**Change type**	**Significance**
INSERT	STATEMENT_INSERT	LOW
	REMOVING_CLASS_DERIVABILITY	CRUCIAL

	REMOVING_METHOD_OVERRIDABILITY	CRUCIAL
	REMOVING_ATTRIBUTE_MODIFIABILITY	HIGH
	INCREASING_ACCESSIBILITY_CHANGE	MEDIUM
	DECREASING_ACCESSIBILITY_CHANGE	HIGH
	ADDITIONAL_FUNCTIONALITY	LOW
	ADDITIONAL_OBJECT_STATE	LOW
	ADDITIONAL_CLASS	LOW
	PARAMETER_INSERT	HIGH
	PARENT_INTERFACE_INSERT	CRUCIAL
UPDATE	STATEMENT_UPDATE	LOW
	INCREASING_ACCESSIBILITY_CHANGE	MEDIUM
	DECREASING_ACCESSIBILITY_CHANGE	HIGH
	PARAMETER_RENAMING	MEDIUM
	PARAMETER_TYPE_CHANGE	HIGH
	METHOD_RENAMING	MEDIUM
	CONDITION_EXPRESSION_CHANGE	MEDIUM
	ATTRIBUTE_TYPE_CHANGE	HIGH
	RETURN_TYPE_CHANGE	HIGH
DELETE	REMOVED_FUNCTIONALITY	HIGH
	STATEMENT_DELETE	MEDIUM
	ALTERNATIVE_PART_DELETE	MEDIUM
	REMOVED_OBJECT_STATE	HIGH
	PARAMETER_DELETE	HIGH

Table 6. Modification operators supported by DDFine

Due to high complexity and limited time available, no modification operators for move-related changes were created for the prototype. Minor adaptations were made to ChangeDistiller to allow for preserving information about positions of inserted nodes.

The next step is to compile the resulting code in the working directory and execute a regression test. Since a full build was already performed during initialization phase, the subsequent builds are done incrementally. Again, a Maven Invoker plugin is used to do so. Method `execute()` of interface `org.apache.maven.shared.invoker.Invoker` returns a result of invocation that contains an exit code. A non-zero value indicates build failure, so that the result of the trial can be immediately marked as UNRESOLVED. Otherwise, the trial proceeds to test execution.

To execute a failing test case, a JUnit API class `JUnit4TestAdapter` is used. Both the definition of a test class and JUnit runner classes have to be loaded with the same class loader that does not delegate to the system class loader; the good explanation of why this is so is given in [38]. The class loader is initialized with URLs of classpath resources which consist of paths to JUnit library, paths to `target/classes` and `target/test-classes` directories of the project under test, and paths to all external dependencies needed for building and test execution. To minimize running time, the paths for external dependencies that are required for the project on which the evaluation is carried out were collected once using Maven Dependency plugin (`mvn dependency:build-classpath`), so that during initialization they are loaded from the prepared file.

After the test is executed, the working directory is restored to match the reference version. The result of test execution is compared against the original `Throwable` obtained from the evaluation context. In conformance with requirement **REQ.07**, there is no differentiation between tests failing due to assertion error or due to the exception.

For the delta debugging phase duration there is established a time limit of 60 minutes (**REQ.09**). If, by that time, an algorithm did not complete, the process is interrupted. In either case, the report with results is available to the user (**REQ.10**). The report from execution that was terminated due to timeout may contain the failure inducing change set which is not truly minimal, but it still could be useful enough to guide the developer to the right place in the code.

The prototype was thoroughly assessed using a dataset collected by the author of the thesis. The next chapter gives the insights into the details of evaluation process.

3. Evaluation

3.1. Research questions

In the process of evaluation, answers to the research questions listed below are sought. To avoid repetitive statements, each question is assumed to be pertinent in the context of localizing regression faults under specific circumstances outlined in Section 1.1.1.

- **RQ1**. *What is the effectiveness of the developed technique in comparison with the textual diff-based approach? Using either of those approaches, how strong is the possibility to get the 1-minimal failure-inducing change set within at most 60 minutes of automated debugging?*

  Answer to this question will give a good understanding of the practical applicability of delta debugging for our purposes, as well as clarify whether the modification proposed in this thesis gives any advantage over 'conventional' version.
- **RQ2**. *What is the average degree of advancement achievable during unsuccessful execution caused by timeout? Which approach better reduces failure-inducing change set within 60 minutes of debugging phase?*

  This question tackles the problem of applicability at a slightly different angle. The answer to it shows how useful, in general, delta debugging is. Even if the attempt was interrupted due to timeout and the returned change set is not minimal, it is probably still small enough to help conveniently localize the faulty code.
- **RQ3**. *How performant is the developed solution compared to the one based on the textual differencing? Does switching the source code differencing and patching technique alone bring a benefit in terms of statistically significantly reduced fault localization time?*

  As was stated earlier, one of the main objectives of this thesis is to develop a more efficient fault localization technique than that relying on textual differencing, so answering this question is essential to assess the result of the effort taken.
- **RQ4**. *Does switching to AST differencing help to promote the consistency of complete configurations? Does it lead to less unresolved trials?*

Lack of consistency was identified before as primary reason of delta debugging performance degradation [1], therefore the percentage of unresolved cases is another measure of solution's efficiency.

- **RQ5**. *Does AST differencing produce fewer chunks than textual differencing?*

  The lower the number chunks to process, the less iterations are needed to find the minimal set. The computation cost of delta debugging is linearly dependent on the problem size.

- **RQ6**. *What is the accuracy rate of the developed technique, as measured in average number of lines of code that a developer has to review manually after localization is completed? Is it better or worse, compared to the old version?*

  The point of this question is to clarify if finer-grained differencing leads also to significantly more precise localization, which is one of this thesis's objectives.

- **RQ7**. *Are the results produced by the developed solution generally more plausible to average developers than the results produced by the old technique?*

  Answer to this question will give an idea of how valuable the developed technique is for the goal of speeding up the process of fixing introduced regression faults.

3.2.Experimental setup

Already at the beginning of working on this thesis it became evident that samples for evaluation of the developed prototype will have to be collected manually. Author carefully inspected two widely known scientific databases containing real faults gathered from various open-source Java projects – Defects4J [39] and SIR [4] – none of them turned out to be suitable for experimentation. The main problem with both data sets is that neither of them stores the information about reference project versions, i.e. versions that are not yet affected by fault. At best, besides buggy project version, the test sample also points to the version in which the fault was fixed, but this information is not useful enough. Moreover, the buggy version provided with the sample is, in most cases, not the first version where the particular bug was introduced. To sum up, the existing benchmarks are not targeted for studies concentrating on regression faults.

The alternate way of obtaining sample data – to resort to automated fault seeding – was briefly considered but discarded because of insufficient evidence that, in the analyzed scenario, such replacement is equivalent to the real fault introduced by developer. This intuition is supported by the empirical study conducted by Just et al. in 2014 [40], where they conclude that due to the fact that a lot of categories of real faults cannot in practice be simulated by commonly used mutation operators, the results derived from evaluating fault localization technique on seeded faults do not generalize to real faults.

Eventually, the data suitable for analysis was collected from a large-scale project (500k LoC) the author is closely familiar with. The method used to select sample revisions consists of two parts. First, the reports from continuous integration tool were manually monitored over a period of 1 year and observed cases of regression were tracked down. These cases constitute around 30% of all gathered samples. Another 70% were retrieved through browsing the history of VCS commits in the project's source code repository. Several searches with different keywords were performed, targeted at finding the revisions with particular commit messages indicating that a commit author was attempting to deal with the noticed regression. Such indicative keywords are, for instance, "fix", "failing", "failed", "test", "tests", "stest", "revert", "rollback", "ignore", "temporarily", "regression", and various combinations thereof. Having found these 'base' revisions, it was nearly trivial to obtain the corresponding 'faulty' and 'reference' revisions.

The samples were prefiltered in accordance with assumptions specified in section 2.2; those not compliant were eliminated from the selection. Additionally, from the further consideration were excluded the samples where regression manifested itself in the failed automated customer acceptance tests, because these tests are too slow to be invoked repeatedly during delta debugging. Two samples were also removed for the reason that the AST diffs between reference and faulty version contained modifications not implemented in the prototype. No other form of preselection was performed.

The total sample size after eliminating unfit examples is 32. This includes both different faulty revisions and different failed tests within the same revision.

3.3.Statistics tracker and collected metrics

For each execution, the detailed statistical data is gathered using performance instrumentation via AOP. Data is recorded to the embedded HSQLDB database. Database is committed to project's GitHub repository and can be accessed through

DataManagerRunner application included into the main codebase. The evaluation-relevant parameters being tracked are listed in a table below:

	Parameter	**Relevant to research questions**
GENERAL EXECUTION INFO	duration of preparation phase	**RQ3**
	duration of change distilling phase	**RQ3**
	duration of delta debugging phase	**RQ3**
	total execution time	**RQ3**
	total number of detected structural changes	**RQ5**
	total number of detected fine-grained changes	**RQ5**
	number of detected significant changes	**RQ2, RQ5**
	number of lines to inspect after localization	**RQ6**
	number of delta debugging trials	**RQ2, RQ4**
	outcome (1/0 – *registered manually*)	**RQ1, RQ2, RQ6, RQ7**
DISTILL ED	type of distilled change	**RQ7**
	path to affected unit	**RQ7**
	location in the source document	**RQ7**
DELTA DEBUGGING TRIALS	outcome of the trial	**RQ1, RQ2, RQ4, RQ7**
	distilled changes used in trial	**RQ1, RQ2, RQ7**
	duration of preparing working area	**RQ3**
	duration of recompiling	**RQ3**
	duration of test execution	**RQ3**
	duration of restoring working area	**RQ3**

Table 7. Metrics collected for each execution and their relation to research questions

The resulting set is not registered separately; it is derived from the last failed trial.

3.4. Alternate implementation based on textual differencing (DDPlain)

One last point to note before proceeding to results is that for performing the comparisons suggested by questions in 3.1, the reference textual diff-based implementation has been built. It relies on the robust google-diff-match-patch library

[41] which, in turn, implements Myers difference algorithm. The prototype has undergone minimal modifications to switch to another source code differencing method; the core logic stays the same as presented in 2.4. Source code of the alternate implementation DDPlain is available at URL http://bit.ly/2ixaGta. For each of the samples collected for evaluation, two tests were performed: one with AST differencing version, another with version that utilizes plain textual differencing. Below is the report of experimental findings.

3.5.Results

3.5.1. Overview

All trials were carried out on a PC with Intel(R) Core(TM) i7-4600U CPU @ 2.10 GHz 2.70 GHz, 16 GB RAM, Windows 7 Professional 64 bit. Result of each trial was reviewed and assessed manually by the author of the thesis. As an aid in assessment of localization quality, where possible, the matching bug-fixing revisions found in VCS history were used.

In the subsequent material, the reference version will be shortly referred to as *P*, faulty version as *P'*, and bug-fixing version as *P''*.

Some of the key distinctive features of the data used for evaluation are:

- number of modified paths between *P* and *P'*: median=**6**, avg=**8**, min=**2**, max=**15**
- time between committing *P'* and *P''* to master branch (time spent by a developer to fix regression): median=**138**, avg=**270**, min=**14**, max=**1171** minutes
- number of failed tests per *P'*: **1** to **14**
- distribution by nature of failure: unexpected exceptions – **53**%, assertion errors – **47**%
- distribution by kind of regression tests: unit tests (all external dependencies are mocked) – **34,4**%, integration tests (“slow”, dependencies are not mocked, require Spring context initialization and a database with the test data) – **65,6**%
- average number of detected changes between *P* and *P'*: **27**
- average number of detected fine-grained changes per modified file: **16**
- distribution of fine-grained changes by change group: inserts – **44,7**%, updates – **33,7**%, deletes – **21,6**%
- distribution of fine-grained changes by exact change type:

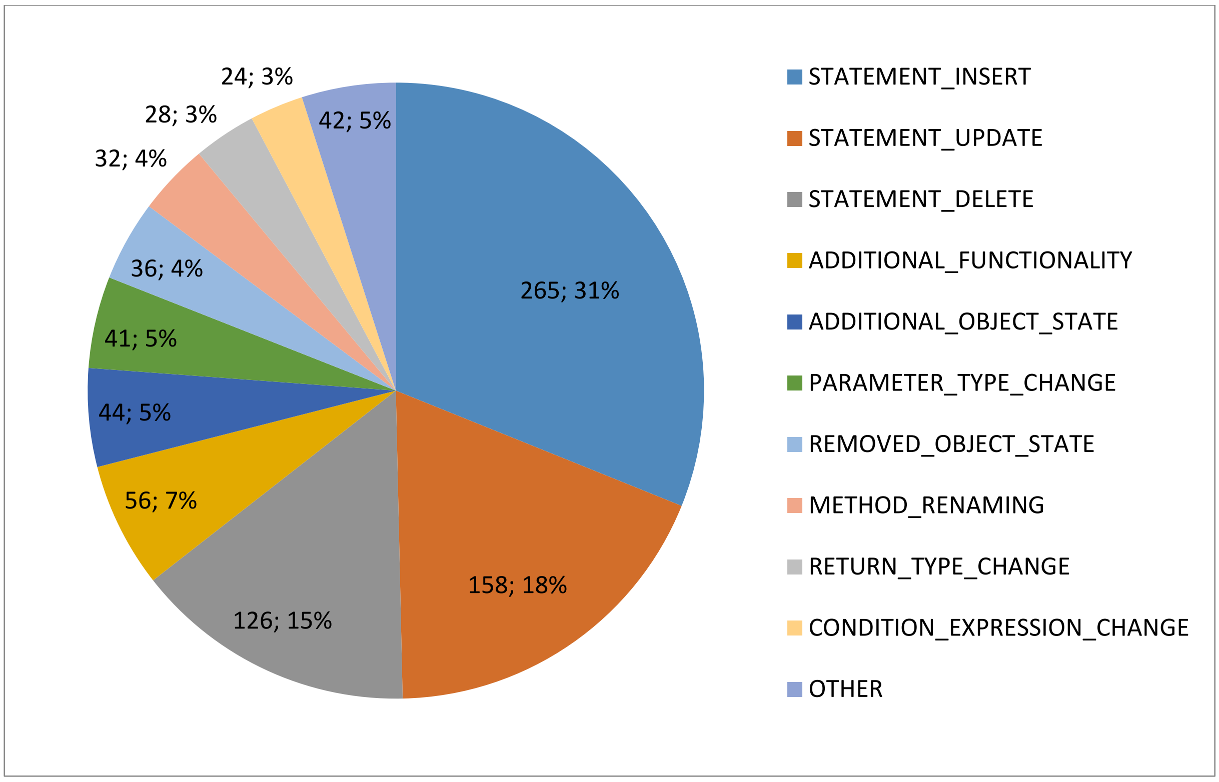


Figure 4. Distribution of fine-grained changes in the sample data (by type)

3.5.2. Effectiveness (RQ1, RQ2)

Table 8 and Table 9 summarize the combined data for **RQ1-** and **RQ2-**relevant metrics.

Sam-ple ID	***DDFine***					***DDPlain***				
	Outc.	**Time-out?**	**Perc. reduced**	**Targ. perc.**	**Red. per iter.**	**Outc.**	**Time-out?**	**Perc. reduced**	**Targ. perc.**	**Red. per iter.**
08d5e	✓		97	97	2.14	✓		99	99	3.67
20585	✓		83	83	0.47		✓	42	86	0.16
32c44	≅		87	93	0.51	✓		92	92	0.31
39e8a		✓	7	98	0.03		✓	13	99	0.13
3d20e	✓		85	85	0.56	✓		97	97	1.73
40dd0		✓	0	97	0		✓	0	92	0
48a20	✓		96	96	2.12	✓		99	99	4.40
48fb7	✓		82	82	0.56	✓		96	96	1.95
50e16	≅		87	93	0.62	✓		92	92	0.71

Table 8. Overall effectiveness of DDFine and DDPlain on the subject samples

Sam-ple ID	***DDFine***					***DDPlain***				
	Outc.	**Time-out?**	**Perc. reduc ed**	**Targ. perc.**	**Red. per iter.**	**Outc.**	**Time-out?**	**Perc. reduc ed**	**Targ. perc.**	**Red. per iter.**
543a9	≅		86	94	0.62	✔		92	92	2.71
5442e		✔	10	98	0.03		✔	13	99	0.13
59c1f		✔	0	98	0		✔	13	99	0.13
5d3fe		✔	4	98	0.04		✔	0	99	0
62b22	✔		97	97	2	✔		99	99	3.88
65f80	≅		89	92	0.59	✔		92	92	0.51
6659f	≅		87	91	1.20		✔	51	90	0.28
6ac2b	≅		87	93	0.68	✔		91	91	0.72
771c7	✔		83	83	0.47		✔	43	86	0.16
8699b	≅		82	90	0.62	✔		99	99	0.59
ad958	≅		87	93	0.71	✔		92	92	1.13
b0077	✔		94	94	1.71		✔	81	99	0.61
b4a6a		✔	38	92	0.08		✔	36	95	0.04
c2fef		✔	0	98	0		✔	0	99	0
c3998		✔	0	98	0		✔	13	99	0.13
c7c12	≅		79	81	0.60	✔		90	90	0.87
cc542	≅		87	93	0.69	✔		95	95	0.71
d10e4		✔	6	98	0.03		✔	13	99	0.14
d34c6	✔		94	94	1.81	✔		96	96	1.08
d3528	≅		89	93	0.80	✔		97	97	1.25
dca56	≅		87	93	0.70	✔		92	92	1.50
fb8bb	≅		89	90	0.72	✔		99	99	0.71
fdc9a	≅		86	91	0.73	✔		92	92	1.02

Table 9. Overall effectiveness of DDFine and DDPlain on the subject samples (continued)

In total, DDFine was able to find minimal failure-inducing change set in 9 cases out of 32. In 14 more cases (denoted by ≅ sign), the resulting set was quite close to minimal, only containing 1-2 excessive chunks. Further analysis shows that the cause of this is that output produced by ChangeDistiller sometimes has some overlapping chunks, i.e. the complete configuration is not completely monotonic. In the rest 9 cases, DDFine failed to complete delta debugging within 60 minutes and was terminated due to timeout.

For DDPlain, 19 trials ended successfully: the resulting set had no excessive textual chunks and could not be minimized further. In all other trials, the execution was terminated due to timeout. Unlike with DDFine, there were no trials where the resulting set had only some unnecessary chunks; the outcome was either a complete success or a complete failure. However, it should be noted here that in half of the "successful" DDPlain cases, the minimal set spanned more source code lines than the corresponding set computed by DDFine. This will be further discussed in section 3.5.4.

When comparing two implementations directly, it turns out that if we consider ≅-cases as successful, too, then in 4 trials DDFine outperformed DDPlain by being able to produce positive outcome when DDPlain failed completely. There were no opposite cases. In the remaining trials, either both implementations succeeded, or both failed. Altogether, effectiveness rate of DDFine is ~72%, while for DDPlain it is about 59%. Therefore, the answer to **RQ1** is: *DDFine is generally more effective than DDPlain, and has slightly better chances to localize minimal failure-inducing change set within 60 minutes of automated debugging.*

As for unsuccessful trials, three derived characteristics come into play. "Percentage reduced" (3rd leftmost column) shows ratio by which it was possible in the given trial to reduce original set with all detected significant changes. "Target percentage (4th column) displays the ideally achievable reduction for particular implementation, from complete set to a minimal. The third characteristic, "Reduction per iteration" (5th column), is the ratio of eliminated chunks to the total number of iterations in the particular trial.

To answer **RQ2**, only the trials that were interrupted due to timeout will be considered. Among such trials, DDFine, in average, scored only 8% for degree of achieved advancement, while DDPlain showed an average of 26% reduction (target percentages shown in tables were normalized to 100%). Therefore, *DDPlain is presumably better at reducing original change set within 60 minutes of automated debugging*.

3.5.3. Performance (RQ3-RQ5)

Table 10 and Table 11 show the summary of measurements completed for comparing the performance of the two approaches. All time intervals are specified in minutes.

Sample ID	*DDFine*					*DDPlain*				
	Prep. + diff phase	**Debug. phase**	**Total exec. time**	**Perc. of unr. trials**	**Num. of sig. changes**	**Prep. + diff phase**	**Debug. phase**	**Total exec. time**	**Perc. of unr. trials**	**Num. of sig. changes**
08d5e	27.6	22.4	51.5	0	31	19.6	19.4	40.3	0	62
20585	25.1	43.8	70.2	57	44	25.5	–	–	72	42
32c44	22.8	15.2	39.4	30	15	19.9	24.5	45.6	41	31
39e8a	22.0	–	–	88	43	20.6	–	–	88	119
3d20e	22.3	12.4	36.0	44	12	32.1	14.0	47.3	0	27
40dd0	19.8	–	–	50	65	20.2	–	–	98	87
48a20	21.3	15.5	37.7	0	32	45.5	16.7	63.4	0	62
48fb7	21.6	12.9	35.8	42	12	24.9	20.4	47.1	0	27
50e16	22.1	13.7	37.3	33	16	20.5	24.6	46.4	43	33
543a9	22.5	13.5	37.5	31	14	28.7	26.6	56.7	38	26
5442e	22.1	–	–	87	53	21.0	–	–	88	109
59c1f	26.8	–	–	86	40	20.4	–	–	85	79
5d3fe	28.7	–	–	84	35	27.5	–	–	89	119
62b22	24.2	16.5	41.9	0	31	19.4	21.5	42.8	0	70
65f80	23.6	13.4	38.5	37	15	22.9	24.5	48.7	44	25
6659f	21.7	13.1	36.6	29	11	19.5	–	–	59	30
6ac2b	29.3	16.3	47.4	35	17	26.7	25.1	53.0	40	29
771c7	31.8	53.5	87.1	59	37	23.0	–	–	76	42
8699b	33.4	18.8	54.3	33	18	19.5	27.1	48.5	39	26
ad958	23.6	16.3	42.0	31	19	19.9	24.6	45.8	48	33
b0077	18.1	18.7	37.9	11	32	28.1	–	–	52	64

Table 10. Performance-relevant data of DDFine and DDPlain obtained on the subject samples

Sample ID	*DDFine*					*DDPlain*				
	Prep. + diff phase	**Debug. phase**	**Total exec. time**	**Perc. of unr. trials**	**Num. of sig. changes**	**Prep. + diff phase**	**Debug. phase**	**Total exec. time**	**Perc. of unr. trials**	**Num. of sig. changes**
b4a6a	25.5	–	–	82	26	22.0	–	–	80	50
c2fef	23.8	–	–	90	43	32.6	–	–	86	121
c399	33.6	–	–	86	41	20.2	–	–	88	85
c7c12	22.5	13.9	37.9	23	16	19.9	24.2	45.7	37	32
cc542	22.7	13.6	37.7	32	15	19.9	24.8	46.0	41	24
d10e	24.9	–	–	87	40	28.3	–	–	87	99
d34c	20.3	15.1	36.4	12	31	19.9	32.1	53.3	37	67
d352	22.2	14.1	37.7	24	20	19.6	24.7	45.6	33	23
dca56	24.9	13.9	40.2	36	13	27.4	33.2	62.5	41	19
fb8bb	23.0	13.4	37.9	33	21	28.3	34.2	64.5	39	29
fdc9a	22.7	14.1	38.3	45	12	27.4	24.9	53.6	52	28

Table 11. Performance-relevant data of DDFine and DDPlain obtained on the subject samples (continued)

Durations of three main phases – preparation, differencing, and delta debugging – were measured separately in order to gain better understanding of effort distribution. Unexpectedly, for source code differencing phase, the time duration was marginally small: in average, it took only ~10 seconds for DDFine (*ChangeDistiller*) and ~7 seconds for DDPlain (*Diff, Match, and Patch*) to obtain the difference between two given versions. Therefore, in the tables above, timings of this phase are merged with timings of preparation phase (1st column, "Preparation + differencing phase"), which is mainly comprised of I/O operations and compilation. Time duration of delta debugging phase (2nd column) and total execution time (3rd column) are shown only for trials which were not interrupted abnormally because of timeout. The total running time slightly exceeds the sum of phase durations; it also includes the time required for final clean-up of the working directory.

The mean time required to complete the first two phases for the subject project is ~24 minutes. In principle, these computations could be triggered in parallel by CI build, so that by the time a regression is detected, a large part of the fault localization work is already finished.

The most relevant values are definitely the durations of debugging phase. The data collected during executions has the following statistical properties:

Method	Percentiles					Mean	Std. deviation	Min	Max
	50%	**75%**	**90%**	**95%**	**99%**				
DDFine	14.1	16.5	22.4	43.8	53.5	18.0	10.0	12.4	53.5
DDPlain	24.6	26.6	33.2	34.2	34.2	24.6	5.1	14.0	34.2

Table 12. Statistical properties of DDFine and DDPlain delta debugging phase duration value sets

From this data it could be inferred that even that in 90% of the cases DDFine was able to efficiently minimize the failure-inducing change set within as little as 22.4 minutes, still, at least each 20th execution resulted in extreme duration of debugging phase. The degree of variability, expressed by sample standard deviation, was almost twice as high (10.0) as for DDPlain implementation (5.1). In general, DDPlain shows more uniform distribution for duration of debugging phase and tends to demonstrate more predictable performance.

Nevertheless, in case of DDFine, the debugging, on average, took considerably less time (18 minutes) than for DDPlain (24.6 minutes). Based on the statistical properties of the sample, it could be concluded that the observed difference is statistically significant (P-value is 0.0128). Consequently, the answer to **RQ3** is: *DDFine yielded ~27% statistically significantly better performance than DDPlain.*

To understand the cause of the difference, it makes sense to have a closer look at the gathered statistical information about delta debugging trials. The time taken by test re-execution does not depend on the particular technique – on average, it took ~12 seconds between all trials. The same holds for time needed to restore working area after each debugging trial; this only takes about 11 milliseconds. However, contrary to intuitive expectation, the mean time required to prepare and recompile working area for the next debugging trial did not differ significantly between techniques: ~59 seconds for DDFine and ~52 seconds for DDPlain. The real problem in case of DDPlain was the total number of trials, which is 63% more than for DDFine. Furthermore, the average percentage of unresolved trials was also higher for DDPlain (51%), compared with DDFine's 44%. So, for **RQ4** the answer is: *yes, switching to AST differencing has a positive effect on the consistency of complete configurations*.

The data reveals that the biggest impact on performance stems from the average problem size. After filtering out the insignificant changes, the complete configuration built by DDFine consisted, on average, of 26.9 chunks, whereas DDPlain's configuration had 54.0 chunks, i.e. twice more. In 7 cases out of 32, DDFine was able to filter out 1-2 chunks irrelevant for fault localization, but this did not affect the problem size by more than 0.2 chunks. All in all, the definitive answer to **RQ5** is that *AST differencing produces approximately twice fewer chunks than textual differencing, this contributing to the better overall performance of DDFine in comparison with DDPlain.*

3.5.4. Accuracy and plausibility (RQ6, RQ7)

For answering questions **RQ6** and **RQ7**, which both aim to assess practical usefulness of the output produced by the delta debugging technique, two kinds of evaluation, automatic and manual, were performed. As a basis for comparing the accuracy of localization, during automated evaluation, for the trials that ended normally the statistics tracker recorded the number of source code lines that corresponded to the found minimal failure-inducing set of chunks. After automatic evaluation was completed, a survey was conducted to determine the perception of results by 3 developers who are familiar with the project codebase. Each participant was asked to review pairwise the output of DDFine (**1**) and DDPlain (**2**) for 19 trials that ended successfully (i.e. without timeout) for both techniques, and choose the variant that, to their opinion, more clearly points to the place that has to be fixed (**1** or **2**, correspondingly). Optionally, the rater could decide that neither DDFine nor DDPlain point to the right spot to fix with enough preciseness (↓↓), or that both were equally good (↑↑). In a few cases where the output of DDFine and DDPlain was identical, only the last two answer options were made available, and the controls corresponding to the 'preferential' answers were disabled. The information about the nature of regressions, as well as the exact fixes, was provided to the raters beforehand. The assessment was conducted in a blind manner, i.e. a developer did not know which of the two outputs was originating from which source (DDFine or DDPlain). Despite that, in many cases this could be easily guessed from the context, since DDPlain's output tends to be more coarse-grained.

Table 13 below summarizes the data gathered for 19 trials that were efficient for both DDFine and DDPlain:

Sample ID	**Number of lines to review**		**Voting results by developers**		
	DDFine	**DDPlain**	**#1**	**#2**	**#3**
08d5e	1	1	↑↑	↑↑	↑↑
32c44	4	7	**1**	**1**	↓↓
3d20e	1	1	↓↓	↓↓	↓↓
48a20	1	2	↑↑	↑↑	**1**
48fb7	1	1	↑↑	↑↑	↑↑
50e16	8	14	**1**	↓↓	**1**
543a9	9	20	**1**	**1**	↓↓
62b22	1	1	↑↑	↑↑	↑↑
65f80	5	9	**1**	↑↑	↑↑
6ac2b	13	11	**2**	↓↓	↓↓
8699b	10	23	↓↓	**1**	↓↓
ad958	3	5	**1**	↑↑	↑↑
c7c12	4	9	**1**	**1**	**1**
cc542	6	16	**1**	↓↓	**1**
d34c6	1	1	↑↑	↑↑	↑↑
d3528	19	25	↓↓	↓↓	↓↓
dca56	3	4	**1**	**2**	↑↑
fb8bb	4	12	**1**	**1**	**1**
fdc9a	11	14	↓↓	↓↓	↓↓

Table 13. Accuracy of localization and results of the manual assessment by 3 developers

The mean number of source lines of code that developers had to review was 5.5 for DDFine and 9.3 for DDPlain, so it could be stated with confidence that *the accuracy of fault localization is significantly better when DDFine is used* (**RQ6**). As the questionnaire shows, this value correlates well with the developers' perceived satisfaction with the output to be assessed; if the minimal set spanned more than 10 lines, the raters never considered the result useful. On the contrary, if the output contained only 1-5 lines, as a rule, it was accepted by a rater as plausible. One remarkable exception to this is the output for sample `3d20e` which was commonly rejected by all raters in spite of being very short. The reason for this is that the failure-inducing code was not within the set of modified lines; rather the modification unmasked the previously existing bug. The behavior of the code under test was altered by adding an invocation to the buggy method that existed already in the earlier program version, and such unmasking regressions are not detectable in principle by delta debugging technique. Other than that, it is worth noting that in several cases the output of DDFine could be even more concise if ChangeDistiller did not fail to return completely disjoint set of chunks, so there is a potential for further improvement.

In total, the raters performed 3 × 19 = 57 evaluations and reviewed 2 × 19 = 38 outputs each. Table 14 shows the summary of raters' agreements for the question about output quality (based on the data from Table 13):

Answer	**Full**	**Majority**
"1"	2	4
"2"	0	0
↑↑	4	3
↓↓	3	2

Table 14. Agreements of the manual assessment by 3 developers

In 2 / 19 (10.5%) of evaluation items, the raters fully agreed that DDFine produced better output than DDPlain. In 4 / 19 (21.1%) more cases, the majority of the raters considered DDFine's output to be better than that of DDPlain. There were no items where at least the majority of developers would prefer the output of DDPlain. For (4 + 3) / 19 (36.8%) evaluation items, at least 2 of 3 developers decided that the output of DDFine and DDPlain is equally good, whereas in (3 + 2) / 19 (26.3%) items at least 2 of 3 raters considered the output to be equally bad. The overall agreement $\bar{P}$ between raters was 0.632, which indicates substantial degree of agreement. The Kappa coefficient κ (fixed-marginal multirater kappa), which shows the confidence level of the observed degree of agreement, was 0.464, which clearly indicates that the agreement is above chance. To conclude, there is statistically significant evidence that *the results produced by DDFine are, to a certain extent, more plausible to average developers than those produced by DDPlain* (**RQ7**).

4. Conclusions and future work

4.1. Discussion

Overall, the implementation of delta debugging algorithm that relies on the output of AST differencing convincingly demonstrated the superiority over conventional version that utilizes textual differencing. The improvements were achieved in terms of effectiveness, performance, accuracy of fault localization, as well as plausibility of the output. Unlike in previous studies on delta debugging technique, which took a broad view of its applicability, current work focuses on one particular use case scenario involving the localization of regression faults introduced during integration of individual developer's changes into a shared mainline. The comparative evaluation conducted using 32 randomly selected real examples of regression inducing commits to the source code repository showed promising early evidence of the practical applicability of the technique in question for the purpose of aiding developers in finding problematic changes. The proof-of-concept prototype built in this work may serve as foundation for creating elaborated framework targeted at efficient localization of regressions, and the findings reported in the Chapter 3 of this thesis justify the choice of tree differencing as the recommended source code differencing method to be used by such a framework.

It could be argued that instead of dealing with consequences that arise from tampering the main body of code with buggy changes, the development team could concentrate on preventive measures and switch to safer development and integration practices. One competitive alternative to trunk-based development (or, TBD) is feature-based workflow, which, in its simplest form, involves creating a separate branch for working on particular feature and initiating a pull request once the work is completed. Direct commits to mainline can be disallowed altogether; merging process might force mandatory preliminary integration of changes from the mainline into feature branch, running the complete test suite, and having someone to review and approve the changed code. More complex variations of this workflow, like GitFlow model [42], prescribe the exact branching scheme encompassing not only a development, but also a release management process. Although such course of action naturally influences the developers to put more effort into stabilizing the code before integrating their changes into the main development branch and significantly lessens the risk of code quality deterioration, it comes with its own cost and cannot be universally recommended for each type of project and every development team. Thus, for example, in one web article comparing and contrasting TBD and GitFlow, it is stated that GitFlow is ill-suited for teams consisting mostly of senior developers, since the infrastructure and overhead costs would probably outweigh the potential benefits [43]. In addition to that, thesis author's personal experience shows that introducing a new branching model in the large (>50 contributors) and diverse development team is associated with major initial loss of productivity and negative attitude of many team members who find themselves struggling with the learning curve of the new process and feel that this impacts their performance. In any case, switching the development workflow in order to get relief from the broken builds does not seem to be a decent strategy in a situation when the project is under high time pressure and development speed is of a main concern.

Another option to consider requires even more radical change to the established development process, and therefore would work best for 'greenfield' projects: the adoption of TDD practices. In conjunction with applying systematic regression test selection techniques, like those described in the 2011 article by Cibulski and Yehudai [44], this could allow to detect up to 90% of the bugs, while having to run locally only a tiny part of the whole test suite each time a code change is made. The main argument against this approach is that acceptance of TDD style alone demands considerable mind shift among the team who is not accustomed to using it daily, so it hardly could be envisioned as a quick and efficient solution to the stated problem.

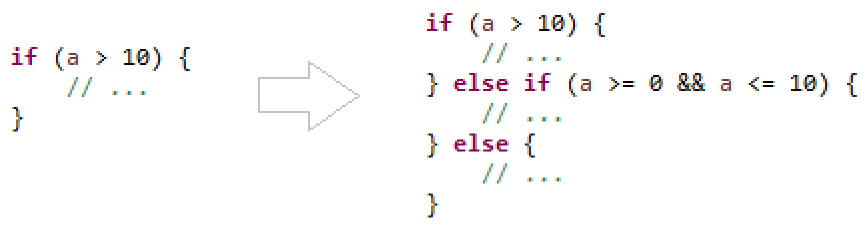
Referring back to the prototypical implementation created as a practical part of this thesis, there are many more concerns left to address before attempting to turn it into a ready-to-use product. There are also certain doubts regarding generalizability and correctness of the obtained evaluation results. The rest of this chapter's sections shortly summarize the factors that could have an effect on the derived conclusions and outline the possible directions for future research.

4.2.Threats to validity

The main obstacle to generalizability of presented results is the fact that the evaluation was performed using examples collected from a single closed source project and the data used is not available to the general public for review. Although the thesis's author recognizes the usage of trunk-based development and continuous integration in the analyzed project as typical, there remains a risk that the code integration practices materialized in evaluation items are to some extent affected by the team-specific habits and conventions, which might lessen the value of this work for the broad audience. The proper way to mitigate this risk would be, of course, to carry out extended evaluation involving examples taken from various open-source projects, but due to time limitation it is impossible to conduct additional tests for the purpose of this thesis.

Apart from concerns about general applicability of the drawn conclusions to other contexts, there also remains a certain level of uncertainty about the absolute correctness of both prototypes. As is mentioned in section 2.4, those interested in further development of automated debugging techniques currently face the problem of lack of openly available reference implementation of dd^+ algorithm. The library used in the built prototypes originates from the dd^+ author’s web page, but it is not confirmed to be fully compliant with the optimal workflow sketched in 1.3.2. It could be useful to re-implement the algorithm from scratch and redo the evaluation using the same test data. Although it is not expected that the outcome of direct comparison would be principally different (i.e. in favor of the version based on textual differencing), there might be some improvement in absolute measures of performance and effectiveness of both prototypes.

Similarly to what was stated above, another factor that could possibly affect the correctness and impact the measured values are the flaws in ChangeDistiller library and imperfection of realized modification operators. When evaluating DDFine, the author manually analyzed several cases characterized by unusually high number of unresolved debugging iterations. In all cases, the underlying cause of the problem was the inability of ChangeDistiller to discover changes in generic type arguments in the parameterized types used as method return types, which led to compilation failures. Yet another issue is related to specifics of manipulating changes in control flow statements. Consider the following simple example:



```
if (a > 10) {
    // ...
}
```

```
if (a > 10) {
    // ...
} else if (a >= 0 && a <= 10) {
    // ...
} else {
    // ...
}
```

Figure 5. Example of problematic handling of changes in if-then-else blocks

Provided that both if-then-else statements belong to the same parent node in the AST of the program and are correctly identified as matching nodes by ChangeDistiller, the latter will output two inserted nodes as the result of differencing: a node corresponding to *else-if* branch and a node corresponding to *else* branch. Both nodes are treated as independent chunks during debugging phase, and can end up in the different subsets. Now, if we try to insert only the *else* branch and do it in a naïve way, without considering the change in semantics of else-statement in the absence of preceding else-if block, we will inadvertently modify the control flow of the program and get a new version which is semantically not a result of incremental application of selected changes to the original version. The new *else* branch will be executed each time when `a <= 10`, whereas the original *else* was triggered when `a < 0`. As a consequence, delta debugging algorithm might not be able to discern the minimal set or could even include in the output the chunks totally irrelevant for the failure. All this suggests that, as was anticipated, the simplistic approach taken in the PoC prototype does not take into account all nuances of a complex OOP language, and for the final product some form of pre-normalization of 'distilled' chunks has to be implemented.

4.3.Future work

Due to the unexpected impediments described in 2.3.2 and lack of suitable ready-to-use benchmarks, the initially planned scope of this work had to be adjusted and a number of interesting optimization tricks was left for future experiments. Among the most promising ideas are to group interdependent changes into larger logical chunks (using libraries that provide automated refactoring support for Java) and exclude the chunks irrelevant for the failure by using code coverage information collected during execution of a failing test. Grouping promotes the consistency of configurations, leading to a smaller number of unresolved test outcomes, and pre-filtering of groups reduces the total problem size for delta debugging. Applying both measures together might have a tremendous positive effect on performance and effectiveness of the technique.

One of the most important challenges to overcome in the future DDFine versions is proper realization of applying tree edit operations to the abstract syntax tree. As was discussed in 2.3.2, neither of the suitable libraries for calculating tree edit scripts currently has out-of-the-box support for this functionality. Adaptation of a chosen library for that purpose is a relatively time-consuming task; for instance, the rough estimate of the total time required to adapt ChangeDistiller is 5000-7000 man-hours.

Finally, it would be useful to conduct a separate study devoted to investigating the real effects of using automated debugging tools on the overall productivity of development team. Contrary to intuitive belief that minimizing the number of changes to review to only a tiny subset will lessen the total time spent on fixing a regression fault, this might not always be the case. Fixing the regression bug rarely means simply reverting the affected code parts, the context of change is also important. Moreover, there are other factors influencing the success of applying fault localization aiding techniques, like developer’s level of experience, familiarity with project codebase, quality of automated tests, etc. Since most of those characteristics are not directly measurable, deriving an optimal strategy for making decision over suitability of delta debugging for the particular project is a good candidate for further research.

References

[1] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?," in *Software Engineering—ESEC/FSE'99*, Springer Berlin Heidelberg, 1999.

[2] D. Qi, A. Roychoudhury and Z. Liang, "DARWIN: An Approach to Debugging Evolving Programs," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 21, no. 3, p. [Article 19], 2012.

[3] R. Just , D. Jalali and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ACM, 2014.

[4] "Software-artifact Infrastructure Repository," [Online]. Available: http://sir.unl.edu/portal/index.php. [Accessed 6 May 2017].

[5] "Comparison of continuous integration software - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_continuous_integration_software. [Accessed 18 March 2017].

[6] K. W. Collier, in *Agile Analytics: A Value-Driven Approach to Business Intelligence and Data Warehousing*, Boston, Addison-Wesley, 2011, p. 281.

[7] "Multitasking: Switching costs," [Online]. Available: http://www.apa.org/research/action/multitask.aspx. [Accessed 19 March 2017].

[8] "The Invisible Problem Wrecking Your Productivity And How To Stop It," [Online]. Available: http://blog.trello.com/why-context-switching-ruins-productivity. [Accessed 19 March 2017].

[9] M. Fowler, "Continuous Integration," [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html. [Accessed 19 March 2017].

[10] "Build Failure Analyzer," [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Build+Failure+Analyzer. [Accessed 19 March 2017].

[11] "Trunk Based Development: Observed habits," [Online]. Available: https://trunkbaseddevelopment.com/observed-habits/#powering-through-broken-builds. [Accessed 20 March 2017].

[12] M. Fowler, "PendingHead," [Online]. Available: https://martinfowler.com/bliki/PendingHead.html. [Accessed 20 March 2017].

[13] "Build Pattern: Gated Commit," [Online]. Available: http://osherove.com/blog/2013/1/20/build-pattern-gated-commit.html. [Accessed 20 March 2017].

[14] M. Monperrus, "Automatic Software Repair: a Bibliography," Unitversity of Lille, 2015.

[15] W. E. Wong, R. Gao, Y. Li, R. Abreu and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering,* vol. 42, no. 8, pp. 707-740, 2016.

[16] F. Y. Assiri and J. M. Bieman, "Fault localization for automated progam repair: effectiveness, performance, repair correctness," *Software Quality Journal,* pp. 1-29, 2016.

[17] R. Santelices, J. A. Jones, Y. Yu and M. Harrold, "Lightweight fault-localization using multiple coverage types," in *Proceedings of the 31st International Conference on Software Engineering*, IEEE Computer Society, 2009.

[18] "GenProg | Evolutionary Program Repair," [Online]. Available: http://dijkstra.cs.virginia.edu/genprog/. [Accessed 3 March 2017].

[19] C. Le Goues, M. Dewey-Vogt, S. Forrest and W. Weimer, " A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *34th International Conference on Software Engineering (ISCE)*, IEEE, 2012.

[20] Z. Qi, F. Long, S. Achour and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ACM, 2015.

[21] E. K. Smith, E. T. Barr, C. Le Goues and Y. Brun, "Is the cure worse than the disease? Overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ACM, 2015.

[22] M. Martinez, W. Weimer and M. Monperrus, "Do the fix ingredients already exist? An empitical inquiry into the redundancy assumptions of program repair approaches," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ACM, 2014.

[23] M. Martinez and M. Monperrus, "ASTOR: A Program Repair Library for Java," in *Proceedings of ISSTA*, Demonstration Track, 2016.

[24] "Symbolic execution - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Symbolic_execution#Path_Explosion. [Accessed 22 March 2017].

[25] A. Zeller, "Simplifying and Isolating Failure-Inducing Input," *IEEE Transactions on Software Engineering,* vol. 28(2), pp. 183-200, 2002.

[26] "Four Phase Test at XUnitPatterns.com," [Online]. Available: http://xunitpatterns.com/Four%20Phase%20Test.html. [Accessed 24 October 2017].

[27] "Parse tree - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Parse_tree. [Accessed 14 November 2017].

[28] "Abstract syntax tree - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Abstract_syntax_tree. [Accessed 14 November 2017].

[29] "Diff utility - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Diff_utility. [Accessed 15 November 2017].

[30] S. Chawathe, A. Rajaraman, H. Garcia-Molina and J. Widom, "Change detection in hierarchically structured information," *ACM SIGMOD Record,* vol. 25, no. 2, pp. 493-504, 1996.

[31] M. Pawlik and N. Augsten, "RTED: a robust algorithm for the tree edit distance," in *Proceedings of the VLDB Endowment, 5(4), 334-345.*, 2011.

[32] G. Dotzler and M. Philippsen, "Move-optimized source code tree differencing," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering,* Singapore, 2016.

[33] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez and M. Monperrus, "Fine-grained and Accurate Source Code Differencing," in *Proceedings of the International Conference on Automated Software Engineering*, Västeras, Sweden, 2014.

[34] "GumTreeDiff/gumtree: A neat code differencing tool," [Online]. Available: https://github.com/GumTreeDiff/gumtree. [Accessed 19 November 2017].

[35] "sealuzh / tools-changedistiller / wiki / Home - Bitbucket," [Online]. Available: https://bitbucket.org/sealuzh/tools-changedistiller/wiki/Home. [Accessed 19 November 2017].

[36] "Eclipse Corner Article: Abstract Syntax Tree," [Online]. Available: http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html. [Accessed 19 November 2017].

[37] B. Fluri, M. Würsch, M. Pinzger and H. C. Gall, "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Transaction on Software Engineering,* vol. 33, no. 11, pp. 725-743, 2007.

[38] "java.lang.Exception: No runnable methods exception in running JUnits - Stack Overflow," [Online]. Available: https://stackoverflow.com/a/29865611. [Accessed 27 November 2017].

[39] "rjust/defects4j: A Database of Existing Faults to Enable Controlled Testing Studies for Java," [Online]. Available: https://github.com/rjust/defects4j. [Accessed 29 November 2017].

[40] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes and G. Fraser, "Are mutants a valid substitute for real faults in software testing?," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software*

Engineering, Hong Kong, 2014.

[41] "google-diff-match-patch - Google Code Archive," [Online]. Available: https://code.google.com/archive/p/google-diff-match-patch/. [Accessed 1 December 2017].

[42] "Introducing GitFlow," [Online]. Available: http://datasift.github.io/gitflow/IntroducingGitFlow.html. [Accessed 22 December 2017].

[43] "Git Flow vs. Trunk Based Development," [Online]. Available: https://www.toptal.com/software/trunk-based-development-git-flow. [Accessed 22 December 2017].

[44] H. Cibulski and A. Yehudai, "Regression Test Selection Techniques for Test-Driven Development," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, Berlin, 2011.