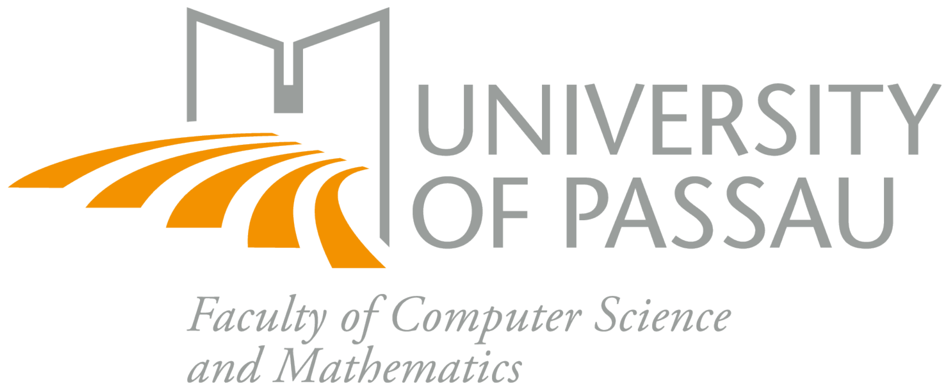


MASTER'S THESIS

FLORIAN HECK

Comparing merge stratgies regarding build and test conflicts

CHAIR:

Chair of Software Engineering I
Faculty of Computer Science and Mathematics
University of Passau

Advisor: Georg Seibt
Supervisor: Prof. Dr.-Ing. Sven Apel
2nd corrector: Prof. Dr. Gordon Fraser

SUBMITTED:

2019-09-11

ABSTRACT

An essential part of software development is changing code. Large projects require multiple developers working on the software and often their changes are located in the same parts of the source code. Those different changes need to be merged together without breaking their individual functionality or a bug fix needs to be applied to different versions.

Merging versions often results in conflicts. While the development of tools for merging focuses on reducing the number and size of those conflicts, other aspects, in particular the correctness of the resulting code, might be neglected.

Automated test suites can help the developer to affirm that individual changes, as well as merges, keep working code intact. In this thesis, we utilize automated test suites to evaluate JDime, a structural merge tool for Java, regarding the correctness of its results and show that, while reducing the number of syntactic conflicts, the number of introduced delayed, semantic errors is not significantly increased compared to default line-based merge algorithms.

CONTENTS

1 INTRODUCTION 1
1.1 Motivation 1
1.2 Goal 1
1.3 Overview 2
2 BACKGROUND 3
2.1 Git, a distributed version control system 3
2.1.1 Merging versions 4
2.1.2 GitHub, a collaboration git hoster 6
2.2 Unit testing 6
2.2.1 Flaky test cases 8
2.3 JDime 10
2.3.1 JDime merge strategies 11
2.3.2 JDime integration 12
3 AUTOMATED MERGE QUALITY ANALYSIS WITH TEST SUITES 13
3.1 MergeProfiler 13
3.1.1 Structure 14
3.2 Evaluating merge test suite data 19
3.2.1 Visualizing merge scenarios 19
3.2.2 Visualizing results from the whole project . . . 21
3.2.3 Statistics 23
3.2.4 JDime data 24
4 EVALUATION 25
4.1 Research questions 25
4.1.1 Finding bugs caused by a merge 25
4.1.2 Automatic merging with advanced tools does not introduce delayed defects 26
4.1.3 Differences between regular merges and pull requests 26
4.2 Analyzed projects 27
4.3 Results 28
4.3.1 Project commons-math 28
4.3.2 Project dropwizard 29
4.3.3 Project fastjson 29
4.3.4 Project ghbrp-plugin 30
4.3.5 Project github-api 31
4.3.6 Project javaparser 31
4.3.7 Project jedis 32
4.3.8 Project okhttp 32
4.3.9 Project ontop 33
4.3.10 Project openmrs-core 34
4.4 Discussion 34
5 CONCLUSION 39

5.1 Summary 39
5.2 Related work 39
5.3 Future work 39
A APPENDIX 41
A.1 Global merge scenario results 41
A.2 Merge scenario results per projects 44
A.3 Test case results 48
A.4 Global JDime merge statistics 51
A.5 JDime merge statistics per project 52

BIBLIOGRAPHY 59

LIST OF FIGURES

Figure 1 Example git graph 4
Figure 2 Example of virtual ancestors in git merge . . . 5
Figure 3 MergeProfiler analysis process 13
Figure 4 Simplified class diagram of the git library . . . 15
Figure 5 Simplified class diagram of the MergeProfiler . 17
Figure 6 Example of the merge scenario visualzation . . 20
Figure 7 Example of a merge result visualization in a sankey plot 23
Figure 8 Example of a merge result visualization in a line plot 24
Figure 9 Examples merge scenarios showing bug introduction 35
Figure 10 The data from Table 3 visualized. 37
Figure 11 Comparison between merged pull requests and regular merges 38
Figure 12 Comparison of filtered and unfiltered merge scenario results 41
Figure 13 Merge scenario results as a Sankey plot 42
Figure 14 Merge scenario results as a Sankey plot 43
Figure 15 The result category plot for project commons-math. 44
Figure 16 The result category plot for project dropwizard. 44
Figure 17 The result category plot for project fastjson. . . 44
Figure 18 The result category plot for project ghprb-plugin. 44
Figure 19 The result category plot for project github-api. 44
Figure 20 The result category plot for project javaparser. 44
Figure 21 The result category plot for project jedis. . . . 45
Figure 22 The result category plot for project okhttp. . . 45
Figure 23 The result category plot for project ontop. . . . 45
Figure 24 The result category plot for project openmrs-core. 45
Figure 25 The global JDime merge statistics. 52
Figure 26 The JDime merge statistics for project commons-math. 52
Figure 27 The JDime merge statistics for project dropwizard. 53
Figure 28 The JDime merge statistics for project fastjson. 53
Figure 29 The JDime merge statistics for project ghprb-plugin. 54
Figure 30 The JDime merge statistics for project github-api. 54
Figure 31 The JDime merge statistics for project javaparser. 55

Figure 32 The JDime merge statistics for project jedis. . . 55
Figure 33 The JDime merge statistics for project okhttp. . 56
Figure 34 The JDime merge statistics for project ontop. . 56
Figure 35 The JDime merge statistics for project openmrs-core. 57

LIST OF TABLES

Table 1 Summery results after filtering 22
Table 2 List of analyzed projects. 27
Table 3 Combined results from all merge scenarios. . . 37
Table 4 Results from all merge scenarios that could be identified as pull requests. 41
Table 5 Combined results from all remaining merge scenarios. 41
Table 6 The result category data for project commons-math. 45
Table 7 The result category data for project dropwizard. 46
Table 8 The result category data for project fastjson. . 46
Table 9 The result category data for project ghprb-plugin. 46
Table 10 The result category data for project github-api. 46
Table 11 The result category data for project javaparser. 46
Table 12 The result category data for project jedis. . . . 47
Table 13 The result category data for project okhttp. . . 47
Table 14 The result category data for project ontop. . . 47
Table 15 The result category data for project openmrs-core. 47
Table 16 Results of all test case scenarios. 48
Table 17 Global JDime merge conflict statistics. 51

LIST OF LISTINGS

Listing 1 Example JUit tests. 7
Listing 2 An example pom.xml 9
Listing 3 Excerpt from a maven surefire report. 10
Listing 4 Excerpt from a generated xml file. 18
Listing 5 Exceprt from commons-math source code. . . 29

1 INTRODUCTION

This chapter will give an introduction to our thesis and we will describe the aim of this work.

1.1 MOTIVATION

Software development consists of writing code to create computer programs. For larges projects, this process often involves many developers working on millions of lines of code. This requires that code is shared between multiple developers. One way to achieve this is the use of version control systems, which bring many additional advantages like keeping record of older versions of the codebase or allowing developers to maintain different configurations of the software in parallel.

However, collaboration brings additional challenges for the software development process, which mostly relate to the coordination of work. This is especially important when integrating changes to the same code from different developers.

To ensure a high and constant quality of the created program, a key point is testing the software. Testing can uncover unintended changes to the behavior of the program. Therefore, the software development process should always include a testing phase. In general, the earlier a defect is found, the easier and less costly it is to fix.

As indicated before, one critical point for changing existing code is the merging of different versions. This is especially difficult since this process can include many changes made by various developers. To help reduce the work necessary to be done by the person performing the merge, tools exist that try to integrate as much of the changes from both sides as possible. While those tools can reduce the work, it is not immediately clear, that the result is the intended one.

1.2 GOAL

With the approach of combining information available through the version control system with the information provided by test suites, we aim to show that merge algorithms that reduce the number of conflicts, and therefore the work necessary to be done by a developer, do not result in more defects in the automatically merged code.

1.3 OVERVIEW

First, Chapter 2 gives an introduction to some tools and practices commonly used in the software development process and what they are used for in our analysis. In Chapter 3 our tool *MergeProfiler* is described and the gathered data is detailed. In Chapter 4 results for the analyzed repositories are reported. Finally, Chapter 5 contains the conclusion drawn from the findings and we look at suggestions to further improve the approach and related work by other researchers.

2 BACKGROUND

In this chapter, we look into some tools which are widely used in software development, specifically in the development of open-source projects written in the Java programming language. These tools provide functionality which is used in many different stages of the software development process. This includes software versioning, the software build process and testing the software project.

All of these tools are used in this thesis to gather valuable data that we use to compile feedback for the developer. These tools include *git*, a version control system, *Maven*, a build system for Java projects, and *JUnit* and *TestNG*, two unit testing frameworks for Java code.

2.1 GIT, A DISTRIBUTED VERSION CONTROL SYSTEM

Git is a version control system (VCS) that has gained popularity in recent years[1,2]. One of its advantages, which has helped its adoption especially in the world of open-source projects is its decentralized nature [16]. Each copy of a *repository* contains the whole history of the project and is capable of being updated with new versions, called *commits*. Each commit contains a set of changes made to the repository. Git saves those changes internally into so-called blobs and hashes them together with metadata about the commit like the author, a message, and a link to parent commit. These resulting SHA1 hashes are used to identify commits as well as any internal structure used by git. Due to the link to the parent(s) being part of the commits data, a chain of commits is created that represents the history which contains all commits, and thereby all incremental changes leading to a given version.

Git supports *branching* the history into diverging lines of development. Branches are separate chains of commits, usually with a shared part of the history. Branches are a useful tool in software development, as often new features of a software are developed in their own branch, for example, to limit interaction during development. To combine changes of two or more branches, git allows *merging* those branches together with special merge commits that have more than one parent.

1 A historical comparison on open source projects indexed by OpenHub and their used VCS can be found through the Wayback machine. http://web.archive.org/web/*/http://www.ohloh.net/repositories/compare, http://web.archive.org/web/*/https://www.openhub.net/repositories/compare (visited on 2019-07-15).

2 *StackOverflow Developer Survey 2018*. https://insights.stackoverflow.com/survey/2018 (visited on 2019-08-29).

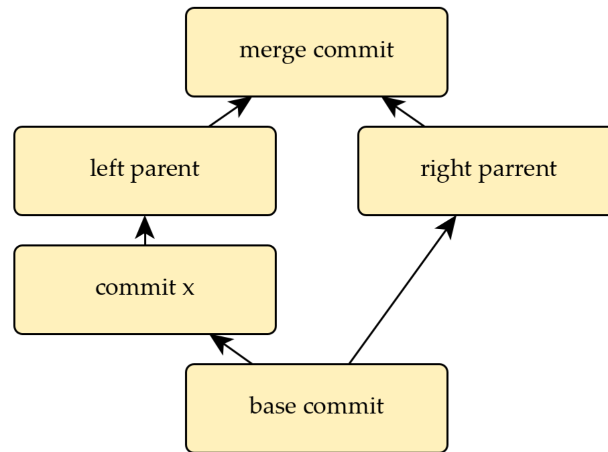


Figure 1: Example of a git revision graph. The commits are named according to the terminologies used here.

Updates in one copy of a repository can be shared with other copies by pushing the relevant data from the local repository to the remote copy or by pulling them from the remote.

Git is a collection of tools and protocols that comprise the whole version control system, its commands allow high-level operations, as well as full access to internal data structures. Most of its tools have a text-based interface that can be accessed under the single command-line interface `git`. The documentation for git and all its commands can be found online[3].

2.1.1 *Merging versions*

As described above, when two (or more) different versions need to be combined, git creates a merge commit. Therefore, all changes between the latest commit on each side of the merge and a common ancestor, a commit that can be reached from all versions in each part of the diverged history, will be calculated as a final changeset and used during the merge. Going forward, we will only consider cases, where there is a common ancestor and the merge commit contains exactly two merge partners. When combining those two changesets into one to build the final version that is part of the commit, there are different cases to consider. In the simple case, where no changed files on one side coincide with changed files on the other side, it can be resolved easily by combining the two changesets. When both sides change lines inside the same file, but those sections do not overlap, git can also combine those changes for each file.

The most complex scenario is the case where different changes in both versions overlap on the same lines. With the utilization of both parent commits and common ancestor (or base commit), git uses a

3 https://git-scm.com/doc (visited on 2019-05-15).

three-way merge algorithm by default. This technique can sometimes resolve these so-called *merge conflicts*. Often this is not the case, and the remaining conflicts are presented to the developer for manual resolution. The final version, including all resolved conflicts, is then committed to the repository.

Besides this default behavior of using its three-way merge algorithm, git can be configured to use a list of merge strategies which include simple strategies for using only one side of merge as the final result, using a two-way merge when no base version is available or complex strategies that can handle situations with ambiguous base versions. This so-called *recursive* strategy will recursively merge base versions until there is a virtual base version available that is unique to all parents. According to the documentation of git[4], this helps to reduce the number of merge conflicts. An example case, where the recursive strategy creates a virtual ancestor is shown in Figure 2. Here, the recursive strategy creates an intermediate merge of `ancestor b` and `ancestor c` (where `ancestor a` is used as the base) that is later used as the virtual base for merging `left parent` and `right parent`.

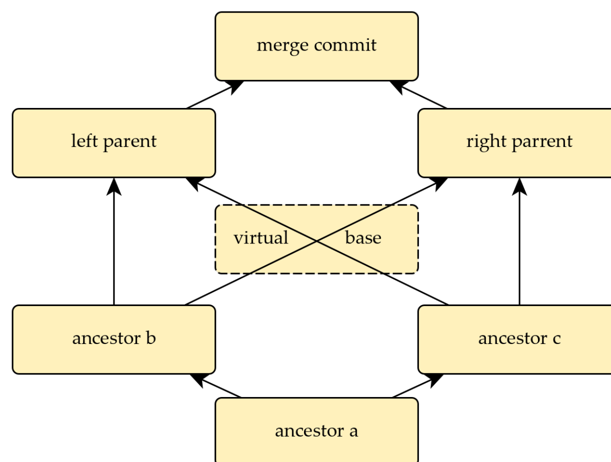


Figure 2: Example of a git revision graph where the recursive strategy creates an intermediate merge used as a virtual base for merging.

At the time of the analysis, this recursive strategy is the default used by git.

Besides different strategies, git can use external tools to perform the merge instead of the line based textual merge implementation provided by git itself. These so-called *merge drivers* can be configured for specific file types and have full control over the merge result while still being provided with the same (virtual) ancestors selected by the merge strategy if the *recursive* strategy is used.

When we consider a merge, we include all additional relevant commits (parents, base commit) in what we call a *merge scenario*. Further,

4 `https://git-scm.com/docs/merge-strategies#Documentation/merge-strategies.txt-recursive` (visited on 2019-08-21).

when we speak of a *merge strategy*, not only the git merge strategy configuration is included but also potential configured merge drivers.

2.1.2 *GitHub, a collaboration git hoster*

GitHub[5] is a popular online hosting system for git repositories and has become the largest source code host in the world. Users can create their own repositories to share access to their projects. The web interface allows for easy browsing, contains stats, and has additional features we will discuss below. A GitHub repository can be used as a remote for a local repository and changes can thereby be synchronized with a central system.

A feature, that is especially popular in the open-source community is the ability to *fork* repositories. A fork is another copy that can be managed by the forking user, but still has a connection to the original repository. If the fork is updated, the user can create a so-called *pull (or merge) request* to the original repository to merge those changes back. Thanks to git's internal structure, this is very similar to merging branches. GitHub enhances this process by providing a simple interface and a connection to its issue tracker. Comments and integration to external tools allow developers a good workflow for code review using pull requests.

In addition to the web interface and the direct git access, GitHub provides a REST API[6] to access most of the data belonging to a repository. Issues and pull requests, including comments and information from the underlying git repository, can be accessed by requesting the corresponding HTTP URI. The response includes a machine-readable representation in the form of JSON objects.

2.2 UNIT TESTING

The software development lifecycle generally should include testing the created software. Different methods can be used for testing, and different stages and parts of the development require different approaches to testing. One of the fundamental methods of testing software is called *unit testing*.

Unit tests are the smallest category of test cases that target very specific functionality [5], in the Java world, often on the level of individual methods or even below. Due to this fine granularity of testing, many test cases are needed to check a large portion of the code and therefore have high coverage. Further, to directly target specific methods, it is often necessary to do setup before the actual test code can be executed. This includes prerequisites, which can be provided by

5 `https://github.com` (visited on 2019-06-30).
6 `https://developer.github.com/v3/` (visited on 2019-09-10).

```
1  import ...
2
3  public class HelloWorldTest {
4
5      public static class Example {
6          public static String test() { return "correct"; }
7      }
8
9      @Test
10     public void passTest() {
11         assertEquals("correct", Example.test());
12     }
13
14     @Test
15     public void failTest() { fail(); }
16
17     @Test
18     @Ignore
19     public void skipTest() { }
20 }
```

Listing 1: Example JUit tests.

mocking and thereby simulating the irrelevant behavior without interfering with the program in normal execution.

For efficient testing and test development, these challenges need to be handled. Unit test frameworks make it easier for the test developer to generalize test cases, help to simplify the setup and even provide a straightforward interface for test execution. Two of the most popular frameworks for unit testing Java code are *JUnit* and *TestNG*. Both follow the "xUnit" convention on how to structure test suites [15]. Since both frameworks are reasonably similar in features and tooling support, we will not differentiate unless necessary. Details information on the latest versions of JUnit[7] and TestNG[8] can be found online.

Tests are structurally or logically grouped into test classes that contain test methods and setup/teardown methods which are called before after the test methods, respectively. Annotations tell the frameworks (see Listing 1, line 17f), how and when to execute certain methods. The test methods contain one or more calls to assertion methods of the framework (see Listing 1, line 12), which will determine the test result. Individual tests can either result in success, if all assertions pass, or fail if at least one of the assertions does not hold or, if the execution fails with some exception, it may be counted as an execution error. The test results are usually displayed on the command line or exported to a report.

7 https://junit.org/junit5/ (visited on 2019-08-06).
8 https://testng.org/doc/index.html (visited on 2019-08-06).

While there are efforts for automatically creating test cases for Java programs (for example, as proposed by Fraser et al. [8] with their tool *EvoSuite*), in this thesis we focus on the test suites provided by the project as-is.

2.2.1 *Flaky test cases*

A major problem with trusting test result is reproducibility. In some cases, the outcome of a test case depends on factors that cannot be sufficiently controlled by the test runner. This can result in false negatives, i.e. a test fails even though the code is bug-free. Luo et al. [13] identified multiple sources for non-determinism in test suites which can result in flaky tests, including concurrency, randomness, or floating-point operations. To combat these problems, some frameworks provide developers with additional tools to mark tests that they have identified as flaky. For example, Google provides Android developers with `@FlakyTest`[9]. Other sources of non-deterministic behavior might not be detectable for developers, like network or IO interactions.

MAVEN, A BUILD SYSTEM

To execute a Java program, the source code needs to be compiled by a Java compiler. The call to the compiler needs to include all relevant information for compilation, such as a list of all source files and their external dependencies. Often, additional parameters, depending on the type of build, the environment, or a feature configuration can be necessary. As a consequence, build commands can get convoluted even for small projects. Build tools provide a way to standardize the build command and therefore the configuration across multiple developers and systems.

One of the most popular build tools for Java is *Maven*[10]. Maven is a Java program, that reads the configuration for a project and then executes commands pertaining to the build process. In addition to the build execution it can also, among other things, help to manage external dependencies of the project (see Listing 2, line 9 ff.). Maven allows its user to configure the build process by specifying a project definition, relevant source files, dependencies, build profiles with compiler flags and other information through an XML file called `pom.xml` (Project Object Model) inside the project folder. In addition to the built-in functionalities, Maven can be extended by a multitude of plugins (see Listing 2, line 18 ff.). Those can be used to automate

9 https://developer.android.com/reference/android/support/test/filters/FlakyTest.html (visited on 2019-07-15).

10 Eugen Paraschiv. *The State of Java in 2018*. 2018. https://www.baeldung.com/java-in-2018 (visited on 2019-08-04).

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
       http://maven.apache.org/xsd/maven-4.0.0.xsd"
3           xmlns="http://maven.apache.org/POM/4.0.0"
4           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
5      <modelVersion>4.0.0</modelVersion>
6      <groupId>example</groupId>
7      <artifactId>hello-world</artifactId>
8      <version>1</version>
9      <dependencies>
10         <dependency>
11             <groupId>junit</groupId>
12             <artifactId>junit</artifactId>
13             <version>4.12</version>
14             <scope>test</scope>
15         </dependency>
16     </dependencies>
17     <build>
18         <plugins>
19             <plugin>
20                 <groupId>org.apache.maven.plugins</groupId>
21                 <artifactId>maven-surefire-plugin</artifactId>
22             </plugin>
23         </plugins>
24     </build>
25 </project>
```

Listing 2: An example pom.xml

processes in different stages of the project management, including testing, and their configuration is also part of the pom.xml file. An example configuration file is shown in Listing 2.

Detailed documentation on Maven can be found online[11].

Configuring and running test suites

One plugin that integrates test frameworks like JUnit or TestNG into the Maven infrastructure is the *Maven Surefire plugin*. Maven Surefire can integrate the existing test suite at the relevant points of the build process. Through configuration, test classes and methods can be selected and integrated into the build process. In addition to running the tests, the Maven Surefire plugin will generate reports about test execution status.

The plugin will use the configured test framework and its test runner to execute the specified tests. Then the results of all executed tests will be compiled into one report containing detailed results for

11 https://maven.apache.org/ (visited on 2019-05-01).

```xml
<?xml version="1.0" encoding="UTF-8" ?>
  <testsuite tests="3" failures="1" name="HelloWorldTest" time="
      0.017" errors="0" skipped="1">
  <properties>
  ...
  </properties>
  <testcase classname="HelloWorldTest" name="failTest" time="
      0.015">
    <failure type="java.lang.AssertionError">java.lang.
        AssertionError
    ...
    </failure>
  </testcase>
  <testcase classname="HelloWorldTest" name="passTest" time="0"/>
  <testcase classname="HelloWorldTest" name="skipTest" time="
      0.002">
    <skipped/>
  </testcase>
</testsuite>
```

Listing 3: Excerpt from a maven surefire report.

all tests. The machine-readable output in the form of an XML file for the example program in Listing 1 can be found in Listing 3

Like all Maven tasks, this is generally done by calling the Maven task for the intended action on the command-line interface of Maven, in the case of test suites this task is normally called `test`. To automate this process without the need to interact with Maven on its command-line interface, we employ the functionally of another Maven plugin called *Maven Invoke*. As the name suggests, this plugin enables programmatically invoking Maven tasks on a project from an external program.

Documentation on Maven Surefire[12] and Maven Invoker[13] can be found online.

2.3 JDIME

As described in Section 2.1.1, it is an essential part of the workflow when using version control, and git especially, to combine two or more revisions of the same code base often including two versions of the same source code file. When merging text-based content there are different aspects of the way the merge result is created. Mens [14] describes one way to differentiate merge tools is the classifica-

12 https://maven.apache.org/surefire/maven-surefire-report-plugin/index.html (visited on 2019-02-20).

13 http://maven.apache.org/shared/maven-invoker/index.html (visited on 2019-02-20).

tion into textual, syntactic, semantic, and structural merging. Additional, when presenting the conflict to the developer for resolution that method can be called cut-and-paste merging.

The merge algorithms git itself implements are all independent of the content within the repository and therefore cannot have any assumptions about the structure of the text data itself. This permits the use of git for a wide variety of projects but limits the merge algorithms to textual merges by looking at anchor lines to identify changed sections.

To improve on this generalization, one can limit the target projects and focus on a specific programming language and its structure and use this additional information to improve the results of the merge. One such tool is *JDime*, with its latest development presented by Leßenich et al. in their paper on performant merging of code containing renamings [11]. Its current release version can be found on GitHub[14]. JDime focuses on the Java programming language and allows merging of Java source code files with different merge strategies.

2.3.1 *JDime merge strategies*

Currently, JDime supports three different strategies (called *modes* in JDime) for merging Java source files. Firstly, by using `libgit2`[15], it provides access to the line based textual merge provided and used by git.

Further, JDime implements a syntactic merge (called *structured*). This included improved detection of moved code blocks or renames that would result in conflicts when using textual merging only. To achieve this, JDime parses the Java source code of all involved versions and uses the created abstract syntax tree (AST) to merge them before again generating Java source code and writing this merged version back. The basic algorithm used for merging the different versions of the AST first needs to find a matching of the AST nodes from both sides. Since in Java, there are parts of the AST that are ordered and others that can be unordered, an algorithm based on Yang's SimpleTreeMatching [19] for ordered matching as well as a linear programming approach for finding am ordering to compare unordered parts. When the matching is obtained, all unchanged or consistently changed nodes are copied to the new AST, conflicts are resolved as possible and conflicts that could not be resolved are inserted as dummy nodes to return both versions for later manual resolution.

Additionally, JDime provides a semi-structured merge strategy that is based on a tool called *FSTMerge* which was originally proposed by Apel et al. [3]. The standalone tool is an "extension to an existing

14 https://github.com/se-passau/jdime (visited on 2019-05-06).
15 https://libgit2.org/ (visited on 2019-08-04).

feature composition tool infrastructure, called FEATUREHOUSE" [3], [1]. This approach to merging software artifacts is similar to the previously described structured merge, but only uses the AST nodes and therefore structure information about the merge subject when merging certain parts of code. All other parts are handled as plain text and are merged as such. In the implementation for Java, method bodies, for example, are such a case. This trade-off was chosen to keep the necessary work for supporting additional programming languages smaller than what would be required for full structured merges while still providing additional information to resolve some conflicts.

Since the added complexity of those merge strategies comes with higher runtime costs, the authors of JDime implemented a fallback mode called *auto-tuning*. This mode provides a trade-off between the precision of allays calling an expensive strategy, but which potentially could detect conflicts on a higher level with the benefit of shorter runtime. In this mode, multiple strategies can be selected. In case the first strategy results in merge conflicts, the next strategy is tried, until either no conflict is produced, or all selected strategies have failed. This strategy has been successfully evaluated by Apel et al. [2].

2.3.2 *JDime integration*

JDime can be used in combination with git as an external merge driver. This allows the selective use of its advanced merge capabilities for existing Java projects and their git-based workflows. Due to its focus on the specific structure of Java, JDime is only capable of merging Java source files the moment. Other files in the repository that might change during a merge are merged by git and its default line-based algorithms.

This integration is done by defining a command for git to call in the case where both versions of a file have changed during a merge. The file suffix `.java` is used to identify Java source files, so JDime is only executed for those. Further, in the case of git's recursive merge strategy, it is configurable if only the actual merge should be performed by JDime.

3 AUTOMATED MERGE QUALITY ANALYSIS WITH TEST SUITES

To analyze merge scenarios, we implemented a test suite execution and analyzer tool. The tool was integrated into an existing framework called *MergeProfiler*. This chapter will describe the process of the merge analysis and its implementation in the MergeProfiler. An overview of the analysis process can be seen in Figure 3.

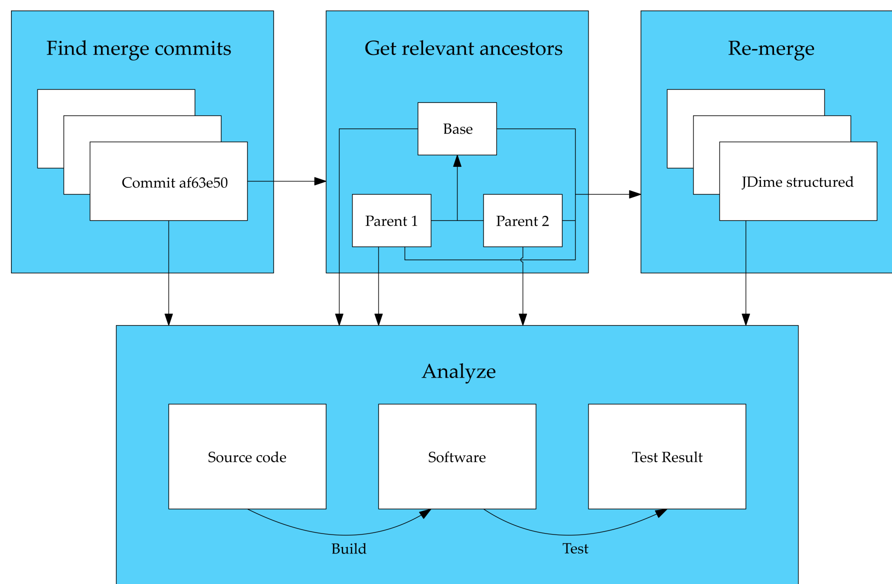


Figure 3: MergeProfiler analysis process

3.1 MERGEPROFILER

MergeProfiler was created to automate profiling and analysis tasks on merge scenarios for git repositories. This allows for the implementation of additional profilers or analysis tools, in this case, a class called `TestSuiteProfiler` was added to run test suites and gather information about their execution and results.

During the setup phase, the tool takes a list of git repositories to analyze and clones local copies to make code and history available. Since the main aspect of analysis for this tool are merge scenarios, all merge commits are gathered from each repository. Together with the configuration of the profilers, the hashes of these commits are used to generate individual scripts for each repository that can be executed in the analysis stage. These scrips are constricted in a manner that

makes the analysis of each commit independent, and therefore allows for parallelization of the next stage.

The profiling phase is initiated by running the generated scripts. Each configured profiler is run for each individual merge scenario and all generated results are accumulated into an XML report that can be used for further processing.

3.1.1 *Structure*

The MergeProfiler tool itself and the libraries used to get access to git and GitHub are written in the Java programming language and make uses of object orientation to describe data structures.

The heart of the tool itself is the abstract `Profiler` class. It provides an interface for all available profilers and analyzers as well as preparations that are necessary for each scenario. Each profiler, like TestSuiteProfiler used for our analysis the needs to implement a method called `profileMergeCommit` which takes the whole merge scenario consisting of the underlying repository, the actual merge commit, its two parent commits, and an optional base commit. This method should return all reportable information as an XML element in the resulting report.

3.1.1.1 *Git access*

The access to the data in the git repository is facilitated by a library called *GitWrapper* which is a wrapper around the command-line interface of git. Its central data structures are the `Repository` class and the `Reference` interface, in the form of `Commits` or `Branches`.

The Repository provides a general interface to the respective git repository, for example, to Commit objects by their hash. Stateful operations like getting the current status of the worktree are also possible.

To get additional access for data that is initially not directly present in the clone, we build an extension to this git wrapper called *GitHub-Wrapper*. By querying the web API of GitHub, the `GitHubRepository` subclass also has access to pull requests which implement the Reference interface and therefore can be used just like any other reference in git. The extension makes it possible to distinguish between a normal merge and a merge that is the result of a pull request and would even allow including of declined and pending pull requests into the analysis.

An overview of the combined class structure of both, the GitWrapper and the GitHubWrapper libraries can be seen in Figure 4.

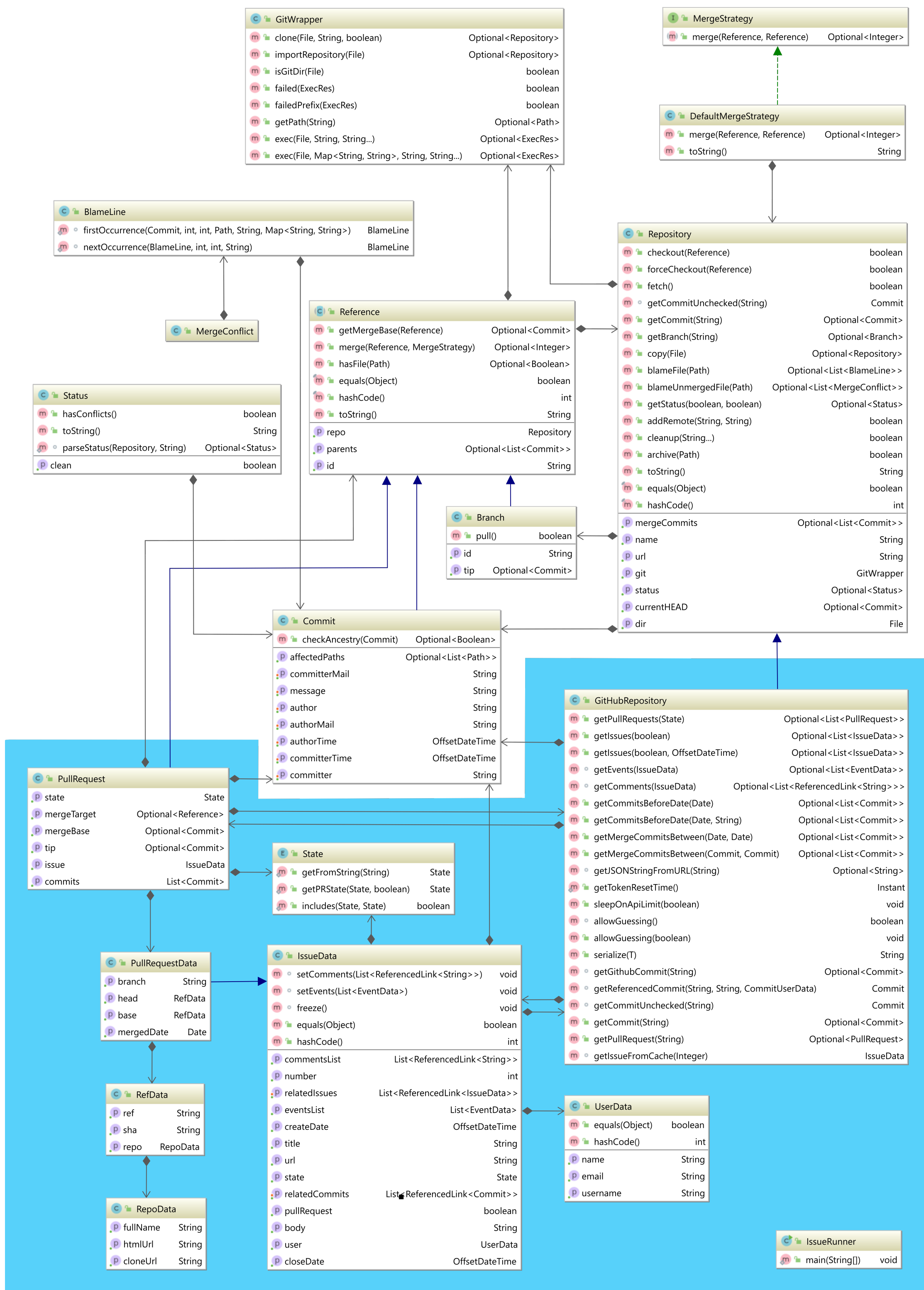


Figure 4: Simplified class diagram of the git / GitHub access library. The blue parts are the GitHub extension.

3.1.1.2 *Testsuite analysis*

As detailed above, the `TestSuiteProfiler` retrieves all relevant commits pertaining to the merge scenario. For each of these commits in

the scenario, the test suite is analyzed. First, this data is gathered for both of the parent commits. Then the merge commit is analyzed in the form it was committed to the repository. The same is then done for the base commit, which should be generally available in all supported scenarios.

Finally, the merge is performed again by the configured strategy and subsequently analyzed. As detailed in Section 2.1.1, git allows for using different strategies and—trough merge drivers—external tools. While initially configuring the tool, merge strategies are selected. In addition to the default strategy used by git, the tool will perform a re-merge with all of these selected strategies. This allows for the comparison of different strategies in the same scenario. Further analysis is skipped if git reports problems while merging. This also includes merge conflicts, which therefore always result in a classification as merge conflict for the whole merge.

The class diagram of the TestSuiteProfiler parts of the MergeProfiler tool can be found in Figure 5.

To gather the actual test data, TestSuiteProfiler uses an instance of the `BuildTool` interface to execute the test suite using the underlying build tool. For this thesis, the Maven build tool was chosen and the `MavenExecutor` class implements this interface to facilitate communication between the MergeProfiler and the Maven executable, but by using an interface the design allows for the inclusion of additional build tools as well. The `BuildTool` interface provides two actions, `build` and `test`.

During the analysis of each commit in a scenario, first, this commit is checked out in the worktree, so that the build tool has access to the current code base of the commit to analyze it. Then, in the case of Maven, a new Maven `InvocationRequest` is created and executed by an `Invoker`. These two classes are part of the Maven invoker plugin, which provides a programmatic interface to execute Maven tasks. A new instance is created for each commit that is checked out. This ensures, that even if the Maven configuration has changed as part of the project history, Maven is always executed with a configuration matching the current state of the project.

Using such InvocationRequests, first, the analyzed project is built with its default `compile` task to check if the code base is compilable. Besides merges that cannot be resolved automatically, which are reported as their own result type, this can happen if broken code was checked into the repository and this is classified as another type of test result, namely a *build failure*. If the build succeeds, the default `test` task is executed, additionally, any previous output is cleaned beforehand using the `clean` task. Cleaning the build directory ensures that there is no interference with previous versions like skipped tests due to up to date results. Secondly, this makes sure all tests are ex-

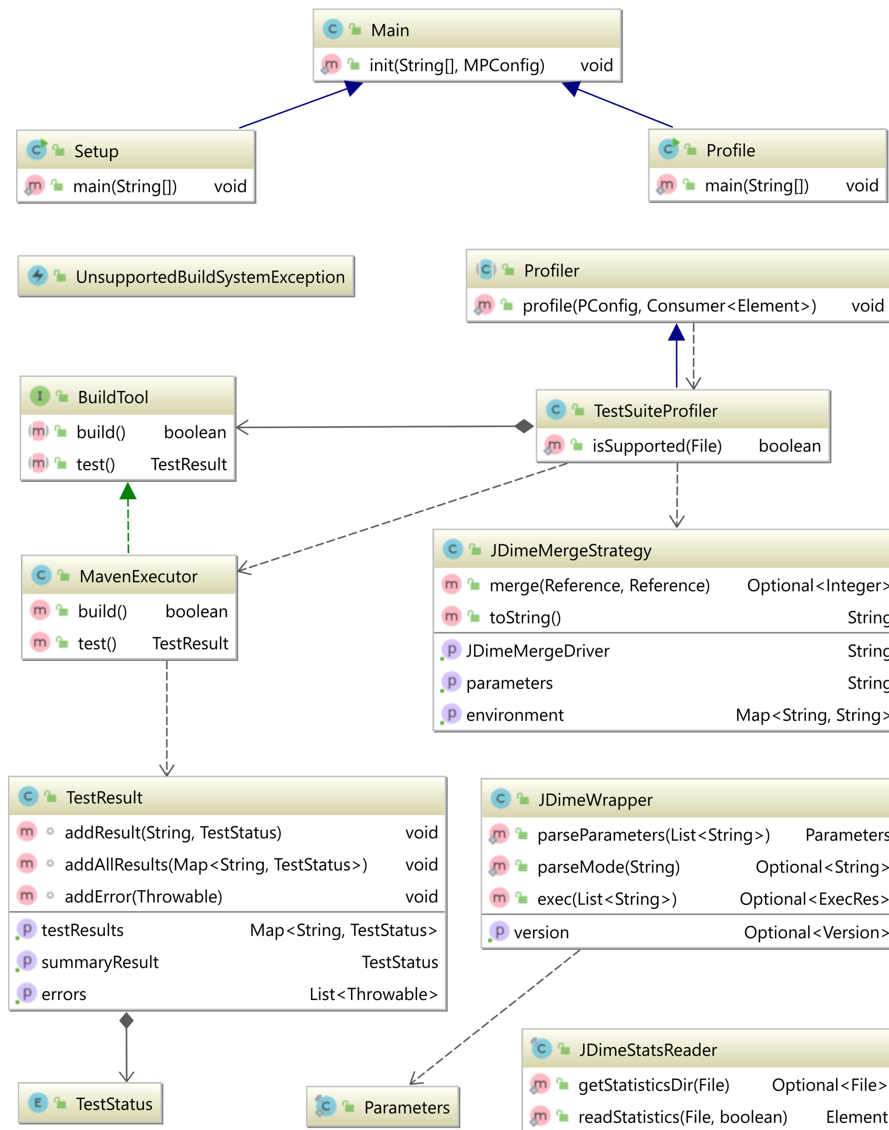


Figure 5: Simplified class diagram of the MergeProfiler tool, with focus on the TestSuiteProfiler.

ecuted like it would be the cases with a clean working directory including only the code from the repository.

As described in 2.2.1, the test results are compiled into a report. The tool reads the XML files generated by the Maven Surefire plugin back and builds a new XML structure for export. Each individual test result, which can either be `PASSED`, `FAILED`, or `SKIPPED`, is included together with the full name of the test. Additionally, general errors during the execution of the maven tasks are also regarded in the classification of the test suite result. An excerpt from the final XML file can be found in Listing 4.

Finally, the working directory is cleaned up so that any generated or changed files do not interfere with subsequent steps of the analysis.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <mergeCommitResults
3   mergeCommit="d6838d4ba7fbd86c24d20b26e733d1e811f3734e">
4    <leftCommit>a893a2de289bcfb167abc810ce3d0c68689131a2</leftCommit>
5    <baseCommit>a893a2de289bcfb167abc810ce3d0c68689131a2</baseCommit>
6    <rightCommit>498ec7ce80769e67e29c83f3b0b5ddcb12cb1c35</rightCommit>
7    <TestSuiteProfiler>
8      <parent_left
9       hash="a893a2de289bcfb167abc810ce3d0c68689131a2"
10      status="passed">
11       <test
12        name="com.yammer.dropwizard.jersey.tests.
             OptionalQueryParamInjectableProviderTest.
             injectsAPresentOptionalInsteadOfValue"
13        result="PASSED"/>
14       <test
15        name="com.yammer.dropwizard.assets.tests.AssetServletTest.
             supportsIfNoneMatchRequests"
16        result="FAILED"/>
17       <test
18       name="com.yammer.dropwizard.util.tests.DurationTest.
            hasAQuantity"
19       result="SKIPPED"/>
20       ...
21     <merge_strategy status="merge_conflicts" conflicts="4" strategy="
          JDimeMergeStrategy {--mode structured}">
22       <mergeResults>
23         <mergescenariostatistics status="OK">
24           <mergeScenario type="THREEWAY">
25             <artifact subclass="FileArtifact" type="FILE" id="left:.
                  merge_file_KNsOWD"/>
26             <artifact subclass="FileArtifact" type="FILE" id="base:.
                  merge_file_wOHaOh"/>
27             <artifact subclass="FileArtifact" type="FILE" id="right:.
                  merge_file_k0qs5Z"/>
28           </mergeScenario>
29           <mergeStatistics/>
30           <charStatistics total="5387" numAdded="0" numMerged="0"
                numDeleted="0" numOccurInConflict="20"/>
31           ...
32           <conflicts>2</conflicts>
33           <runtime label="merge" timeMS="25"/>
34         </mergescenariostatistics>
35         ...
36     <merge_strategy hash="6a9676c892767d0f9c03d94fc0bf72e18dc6c8c0"
          status="build_failure" strategy="JDimeMergeStrategy {--mode
          semistructured}"/>
37   </TestSuiteProfiler>
38 </mergeCommitResults>
```

Listing 4: Excerpt from a generated xml file.

3.1.1.3 *Performing automatic merges*

As stated in the last section, git, as well as our tool, supports different configurations. The default merge strategy is the one that git performs when `git merge` is executed. At the time of our analysis, the default strategy for git merge is *recursive*. This merge strategy tries to execute a three-way merge on the two parents and a virtual base version.

The other merge strategies used in the analysis are based on JDime. These are based on git merge drivers detailed in Section 2.1.1 and Section 2.3.2. Currently, if necessary, intermediate merges are performed with the default line-based merge by git itself.

3.1.1.4 *Detecting flaky test cases*

To detect the problem of non-deterministic test results described in Section 2.2.1, the tool provides an additional mode of operation. Since the support for marking test cases as flaky is neither consistent, universally support nor, as evident after reviewing selected projects, common practice, the easiest way of detecting problematic test cases is running the test suite multiple times. Considering, that doing this for all involved commits is very resource-intensive, as the tool follows the same steps of creating a clean environment, building the project and subsequently running the test suite, this is only done for the actual merge commit in the repository as this version is regarded as the baseline for our analysis.

Running all involved steps during this detection has the additional benefit of not only detecting flaky test cases but also detect non-determinism in the build. These results can be used in post-processing to eliminate the influence of wrongly reported test and build failures.

3.2 EVALUATING MERGE TEST SUITE DATA

In addition to the MergeProfiler tool, we created some scripts written in the Python programming language. These scripts are intended to process the XML reports generated by the MergeProfiler tool to visualize the results and extract statistics across the analyzed projects.

3.2.1 *Visualizing merge scenarios*

To visualize the merge scenario as a whole, the post-processed data can be presented by one of the Python scrips in the form of multi-colored stripes. This visualization makes it easy for humans to spot interesting individual test results over the whole scenario. An example of this visualization can be found in Figure 6. In this diagram, each test case is represented by a column of colored line. The test results of the individual commits in the scenario are arranged on top

of each other, sorted by their test name, so that the same test case is at the same position across all shown commits. Thereby, a change in color within one column indicates a different test result between the commits. A group of results for the same test over different versions of the project will be hereafter referred to as a test case scenario. In this representation, passed tests are colored in green, failed tests are colored in red. If the detection for flaky tests identified a test as problematic, a failure is orange instead of red. Tests that were not executed while the test suite itself did execute, which mostly happens when individual tests are marked to be skipped in the test suite, are indicated by a yellow line. To differentiate this result from cases where the whole test suite did not execute due to a failed build or an internal failure, those cases are represented by blue and cyan bars, respectively. If the merge itself was not successful, which also means that the test suite could not be executed, the version is colored in magenta. Finally, an uncolored white line indicates a test that is missing from this particular commit. Figure 6 shows this visualization for the project *ontop*.

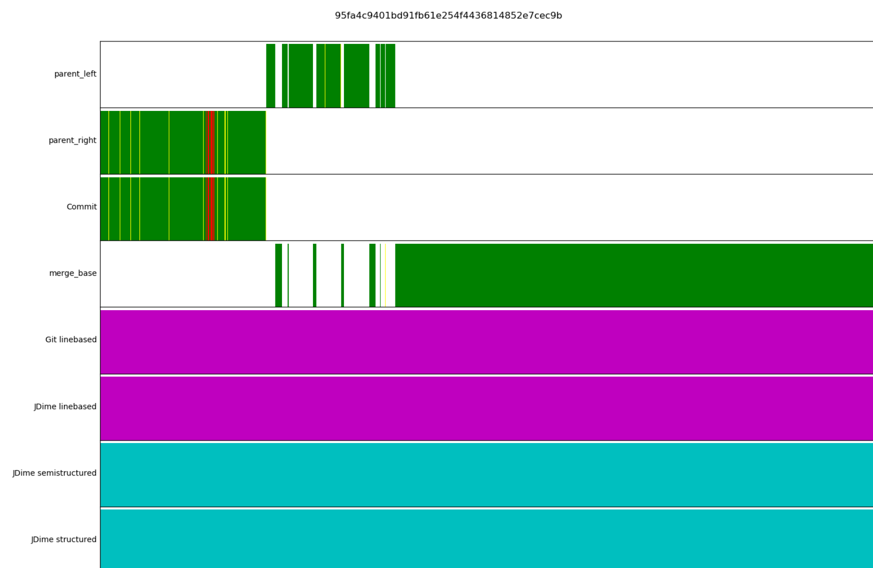


Figure 6: Visualization of one merge scenario from the *ontop* project. The green parts in the parents, base, and committed merge test cases show passed test, whereas failed tests can be seen as red lines in the right parent and the committed merge. The yellow lines in the first 4 indicated a skipped test while the white lines indicate tests not present in this commit. The magenta and cyan bars show merge conflicts and build failures, respectively.

3.2.2 *Visualizing results from the whole project*

Besides visualizing the individual merge scenarios, there is a script to visualize the breakdown into the different result categories the merge of a scenario can fall into. This script supports two different visualizations that were developed to highlight different aspects of the data.

The post-processing pipeline can use two different methods of determining the combined result of a merge scenario. Table 1 shows the full process of this process which we call filtering. The first method is to trust the build tool with its determination. The second method uses additional information besides the merge itself. A first filter that is applied uses the result of the flaky test case detection (see Section 2.2.1). If a test failed that was identified to be non-deterministic, this test case is excluded from the determination of the result. Similarly, if the build is non-deterministic, this merge scenario is counted as an error, since further classification is not possible without a stable build result.

Secondly, the parents are included in the determination of the result as well. We assume that a test that failed on both parents has no reasonable way of passing in a merge result. This may be a defect that is unrelated to the current development or a test for unimplemented functionality. Since this analysis only focuses on the merge results as a whole, we want to reduce the noise introduced by these unrelated failures. The same exclusion is performed when the test case is only present on one side of the merge, but it fails there. We assume that this is the addition of a new test case that is not implemented yet. These filter steps can be seen in Table 1, in lines 5–7 of the second section. A similar logic is used to exclude scenarios where the build fails in all instances since we assume this is related to syntax or configuration errors in the project that also cannot be fixed by the merge because otherwise one side of the merge would bring the fix and therefore the build would succeed. We count these cases with a passed result, because the failure is the expected result (see Table 1, second section, line 2 and 3).

The first visualization is a series of Sankey plots, one for each strategy. To allow for comparison across the strategies, the categories are scaled and aligned equally. The Sankey plot focuses on the distribution of the different result categories while still supporting comparability. An example of the Sankey plot for the project *dropwizard* can be found in Figure 7. The total number of analyzed scenarios is shown on the left of each Sankey plot, the number of scenarios remaining in each category is shown inside the graphs and the count for each category is shown on the bottom.

In the second visualization, only non-passing scenarios are included to improve the visibility of differences between the tested merge strate-

Table 1: The table shows the summery results that are assigned, based on the test/build results from the re-merge and both parents.

Parent 1	Parent 2	result	result after filter
filtering due to flaky detection			
any	any	flaky test failue	test ignored
any	any	flaky build failure	error
filtering due to parent data			
any	any	merge conflict	merge conflict
error	error	error	passed
build failure	build failure	build failure	passed
build failure	no build failure	build failure	build failure
test failure	test failure	test failure	test ignored
test failure	not present	test failure	test ignored
test failure	no test failure	test failure	test failure
any	any	passed	passed

gies. This visualization presents as a line chart displaying the number of merge scenarios that passed a certain stage in the merge-build-test process. Even though the results fall into discrete categories, values from the same strategy are connected. This is done to highlight the logical progression through the different stages. Connecting the stages shows the differences between logically consecutive categories and thereby the number of commits falling into that category. Since the different strategies are shown in the same frame of reference one can compare them by their absolute share of passing commits for each category. Showing the changes between categories as a slope additionally allows for quickly comparing the differences across multiple strategies. An example can be found in Figure 8, graphs for all projects can be found in Section A.2. Even though this plot is more complicated, it was preferred during this thesis because it provides more relevant information.

The example in Figure 8 shows the same project as the Sankey plot in Figure 7, but only showing the four strategies and the results from the actual merge. For JDime's structured strategy, around 1.6% of merge scenarios resulted in merge conflicts while 97% passed all stages. Besides the difference in the number of merge conflicts, all strategies performed similar in the build stage, while the test stage shows some differences.

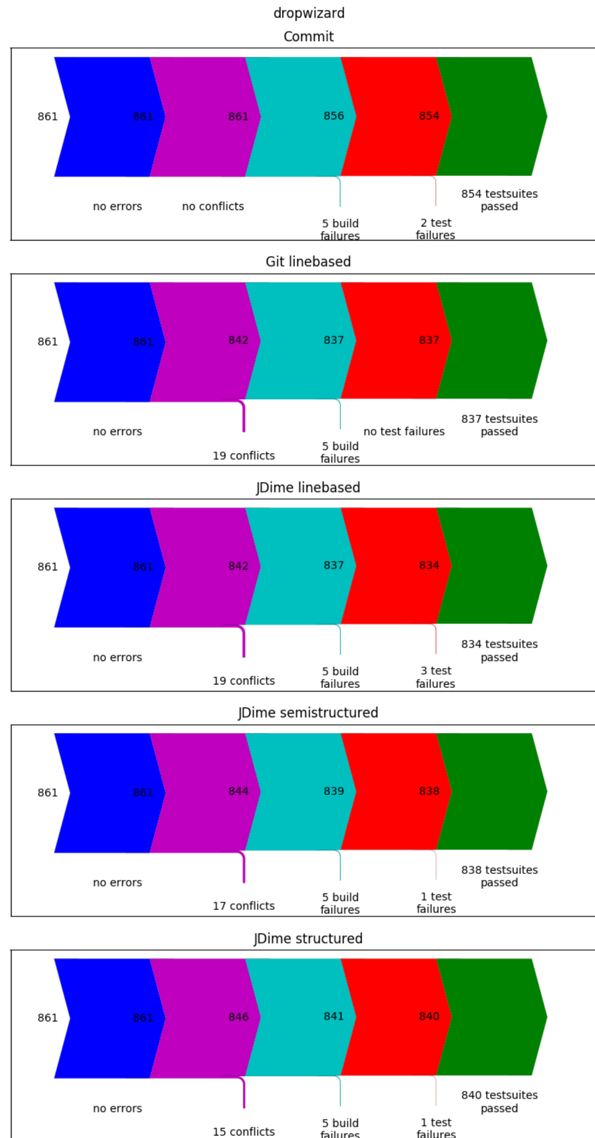


Figure 7: Sankey for project *dropwizard*.

3.2.3 *Statistics*

In addition to the visualizations, the Python scripts can generate statistics for each individual merge scenario as well as for all available scenarios of the analyzed project. Additionally, all analyzed projects are combined into global statistics. Those statistics include a summary of all different occurring test case scenarios. This means that all occurrences where, for example, a test fails in the committed merge but passes in all other analyzed states are grouped into one category. This allows for a quick overview of all different occurring test case scenarios. We call this group of results from one scenario a *pattern*, consisting of single letters for each relevant commit, where `P` stands for as passed test case (or the whole test suite), `T` for a test failure, `S` for

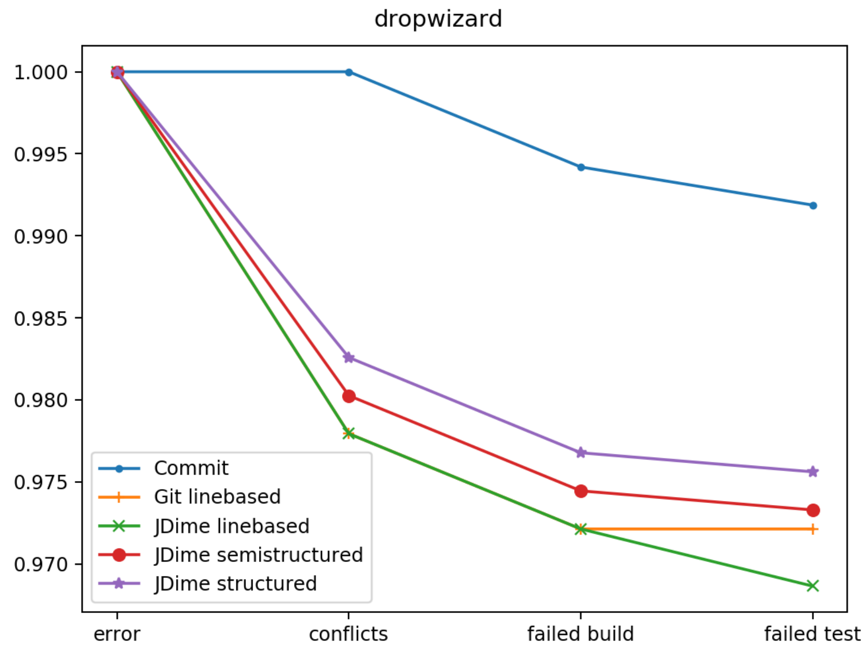


Figure 8: Merge result visualization for project *dropwizard*.

skipped tests, `B` for build failures, `M` for merge conflicts and `E` for any error. We also use `X` to stand in for an arbitrary result. The order of the revisions in the pattern is always the same, first the left and right parent followed by the original merge, the merge base and finally the examined strategies, in the order git, JDime linebased, JDime semistructured, and JDime structured. Further, these statistics can give a first indication, if a particular pattern of test failures might be present. This data is processed into different forms to map a pattern first to the scenario it occurred in and then, if applicable, to the test case.

Other statistics generated by the Python scripts include the data used to generate the visualizations shown in Section 3.2.2.

3.2.4 *JDime data*

In addition to the data generated by our tool, the Python scripts are also capable of analyzing the data about merge conflicts generated by JDime. From this data, statistics are gathered about the distribution, number, and size of conflicts.

4 EVALUATION

In this chapter, we will look into the analysis of real-world projects and what data we can gather from using our tool described in the previous chapter. To find suitable projects which meet the requirements for our analysis of being open-source, written in Java, using Maven as their build system, employed integrated unit test, and git as the version control system, we searched on GitHub. This allows for a pre-selection of projects with active development. Further, for projects from GitHub, our tool can include pull requests as a specific focus point.

4.1 RESEARCH QUESTIONS

To determine the usefulness of our analysis we formulated three questions we want to answer. In the ideal case, all test cases would pass for every commit in the repository. This is not always the case. In every project we analyzed we found some commits where some test cases did not pass. This can be caused by different situations. Besides the occurrence of a fault in the program, for example, the introduction of new test cases that check functionality that is not yet implemented can result in a failure of those test cases. Since we do not only look at the merge commit in isolation, but also other commits related to the merge, we hope to identify those different situations. Therefore, we focus on particular groups of patterns to analyze for each question.

4.1.1 *Finding bugs caused by a merge*

At first, we wanted to investigate, whether our analysis can even find bugs in the form of failed test cases that were introduced by a merge. For this, we focused on the committed merge and its changes relative to both parents. As stated above, a test case that passes in every considered commit is classified as bug-free. More specifically, we only looked at the parents and the merge as committed in the repository and a test case, that is present in both parents and passes in all three commits is regarded as a bug-free merge. When the same test case fails for either parent and the failure is still present in the merge, we cannot tell if something changed to affect this test case. In contrast, if the test case passed in both parents but fails in the merge there must be a change that occurred in the merge commit itself that broke the test case. Therefore, we classify this as a broken merge. We assume the same is true when the test case is missing in one of the

parents but passes in the other and fails in the merge since that would mean newly introduced or tested functionality that worked before the merge broke during integration with other changes.
Our first research question is therefore:

RQ1. *Can our analysis detect merge scenarios where a bug was introduced directly due to the merge?*

4.1.2 *Automatic merging with advanced tools does not introduce delayed defects*

The second question we tried to answer is in regard to automated tools. For this, we looked in detail at differences between the results of the manual merge committed by the developer, the automated merge performed by git itself, and the results of a merge performed by different JDime strategies. As previous findings by Leßenich et al. [12] have shown, JDime with its semi-structured and structured strategies can reduce the number of merge conflicts compared to a line-based approach. While trying to reproduce these findings, we examined the consequences of resolving more conflicts automatically. Since the objective of sophisticated merge strategies is to reduce the workload on developers, these strategies must resolve conflicts in a way that does not introduce new bugs to the program. The automated way of detecting bugs trough compilation and unit testing is used as a measure of detecting these defects and their delay into later development stages. The second research question, therefore, reads as follows:

RQ2. *Can JDime reduce the number of merge conflicts without increasing the number of build and test failures?*

4.1.3 *Differences between regular merges and pull requests*

The subject of our last question is pull requests. For this question, we examined the difference between merges that are the result of pull requests and merges performed by the developer directly. Since GitHub only allows merging of pull requests that can be merged without conflicts by git, merges marked as pull requests and therefore performed through the web interface should resolve cleanly without any conflicts. We can use this as a filter for commits that should be simple enough to be completely merged automatically or pre-resolved by a developer. In the second case, ideally, any defects introduced by a complex code merge should have been corrected by the developer requesting the merge. Thus, we hoped to find this selection of scenarios usable as a baseline.
The question we examined, is:

RQ3. *Are there significant differences between regular merges and pull requests regarding delayed defects?*

4.2 ANALYZED PROJECTS

Table 2: List of analyzed projects.

Project (GitHub URL)	lines of code[16] (thousand)	contributors	commits[17]	merge commits	(pull requests)	number of test[18]
commons-math (https://github.com/apache/commons-math)	12164	29	7428	109	(12)	5526
dropwizard (https://github.com/dropwizard/dropwizard)	2535	322	6788	1021	(745)	12
fastjson (https://github.com/alibaba/fastjson)	6740	84	3783	453	(239)	4646
ghprb-plugin (https://github.com/jenkinsci/ghprb-plugin)	403	112	1661	338	(19)	77
github-api (https://github.com/kohsuke/github-api)	558	94	1381	194	(125)	144
javaparser (https://github.com/javaparser/javaparser)	7625	93	6219	1664	(603)	1479
jedis (https://github.com/xetorthio/jedis)	1267	141	3214	366	(233)	420
okhttp (https://github.com/square/okhttp)	2927	172	4708	1829	(734)	2244
ontop (https://github.com/ontop/ontop)	6809	23	11176	1453	(18)	46
openmrs-core (https://github.com/openmrs/openmrs-core)	7431	321	9085	1152	(1045)	4022

Table 2 contains a list of all analyzed projects including some general statistics about each project. Information like the number of commits, the size of the code base in terms of lines of code or the number of contributors can help estimate the size of each project. Further, the number of merge commits and the number of individual unit tests can be used to gauge the value from the results of the analysis.

In order to have a wide range of real-world merge scenarios, we selected projects of different sizes—in regard to the number of commits as well as the size of the source code base—and type.

Besides the search functionality offered by GitHub, we used several lists of projects from earlier research to compile this list of projects. During preliminary examinations, several repositories needed to be

16 The line count is taken from the latest commit of the designated default branch, it might vary over the history. It includes only Java code as reported by GitHub
17 The number of commits is taken across all branches and forks.
18 The number of tests is taken from the latest merge commit that passed analysis, it might vary over the history.

excluded due to different problems. Some projects are configured in a way that building their history retroactively is not possible without advanced configuration that our tool currently cannot provide. Other projects included strict requirements in their build step, that JDime merge strategies were not able to adhere to, for example, due to whitespace and other formatting constraints imposed by the *Checkstyle* plugin[19]. Further, not included in this list are some small repositories where the different strategies resulted in no differences at all.

4.3 RESULTS

In the following section, the results of all the analyzed projects that were included in the final data set are examined in detail.

4.3.1 *Project commons-math*

Apache commons-math is a Java library of mathematical and statistical functions. Its goal is to provide additional functionality where the Java standard library's `math` package lacks specific functions while being as lightweight as possible. Like other Apache commons projects, commons-math has its origin at the Apaches Software Foundation.

4.3.1.1 *Overview*

Of 110 merges present at the time of the analysis, our tools could analyze 96, of which 12 are pull requests and 84 are regular merges found in the history. Of the 22 merges that failed in-depth analysis most had build errors due to missing dependencies. Some revisions, for example, rely on internal snapshot builds of libraries that are not available to the build tool any longer.

4.3.1.2 *Results and examples*

The results from this project show uniform numbers of conflicts and build failures across all strategies, while the number of test failures slightly increases for JDime. Further, this project shows the necessity for flaky test case detection. Within the test suite for the code for the commons-math library, a random number generator is used and even tested. This includes, at the current revision, at least one test case, in which the test cases documentation states:

19 https://maven.apache.org/plugins/maven-checkstyle-plugin/ (visited on 2019-05-01).

```
1  /**
2  * Generate 1000 random values and make sure they look OK.<br>
3  * Note that there is a non-zero (but very small) probability that
4  * these tests will fail even if the code is working as designed.
5  */
6  @Test
7  public void testNext() throws Exception {
8      tstGen(0.1);
9      tstDoubleGen(0.1);
10 }
```

Listing 5: Excerpt from the file `src/test/java/org/apache/commons/math4/distribution/EmpiricalDistributionTest.java` showing the only documentation for the possible test failure not due to an actual fault in the tested code nor the test itself.

During preliminary evaluations, this test failed multiple times due to the generated numbers not "looking OK". This prompted the extension of our tool to include detection for such cases. Since the test is not annotated in any way, the only way to detect this flakiness is to hope for a random failure during the detection run.

The merge scenarios from this project resulted in 19 different patterns. Most of those patterns occuedr only once or twice but would be interesting to the result of our analysis. Unfortunately, because of the problems detailed above, this was not alway the case.

4.3.2 *Project dropwizard*

Dropwizard is a Java framework for building web services adhering to the REST architectural style. It connects other libraries to provide a simple and fast framework. Dropwizard was created by Coda Hale at Yammer, Inc. and is maintained by the Dropwizard team.

Overview and results

There were 861 merges of the 1028 found scenarios that passed the analysis tool. These are comprised of 745 pull requests and 115 regular merges. Within these, 172 scenarios failed the build in all relevant versions, again mostly due to missing snapshot dependencies. Looking at the usable results shows a small reduction of merge conflicts for JDime semi-structured and a further reduction for JDime structured, while at the same time the number of build and test failures is constant.

4.3.3 *Project fastjson*

Fastjson is a Java library for parsing and generating JSON representations of Java objects. Is was created with high performance and wide

compatibility in mind. Fastjson is developed at Alibaba by the Fastjson Develop Team.

Overview

In this project 453 scenarios were found, of which 452 passed the analysis. These consist of 209 regular merges and 239 pull requests. A total of 16 scenarios failed to build due to various reasons, including, again, unavailable dependencies.

Results and examples

In this project, 19 different result patterns can be found but only three are common, all having uniform results across all merge tools. The rest show a reduction of merge conflicts of more than 50% for JDime structured compared to the line-based strategies and even though the number of test failures is higher, the total number of non-clean scenarios is lower.

Commit `e165825` shows a common problem with text-based merge tools, that JDime is able to correctly resolve. In the file `src/main/java/com/alibaba/fastjson/parser/deserializer/JavaBeanDeserializer.java`, a similar code block is added on both sides, on the one side, an `if`-statement is added, on the other side, the same statement is added, but with an additional statement in the body. Since both versions do not add the same changes but touch on the same lines, this results in a conflict. JDime's structured strategy is not bound by this limitation because the merge is done on an abstract representation of the code instead of plain text with context markers. Therefore, the merge succeeds. JDime's semi-structured strategy does not use the method of merging on AST representations for changes inside Java method bodies, thus, it is not able to resolve this conflict here.

4.3.4 *Project ghbrp-plugin*

The *GitHub Pull Request Builder Plugin* is a plugin for the build automation server Jenkins. It allows Jenkins to get access to pull requests on GitHub through the GitHub API. It is developed by the community behind the Jenkins project.

Overview and results

All of the 318 merge scenarios that were found in this project passed the analysis and none failed to build for all relevant versions. Of these, 299 are regular merges and 19 pull requests. The results show a similar trend as the *fastjson* project, a slight decrease in conflicts with an increase in build and test failures, while still maintaining an improvement.

4.3.5 *Project github-api*

GitHub API for Java is a library for accessing the GitHub API in an object orient way from Java. It tries to provide methods for interacting with most of GitHub's APIs. The project is maintained by Kohsuke Kawaguchi and developed by a community.

Overview

188 of the found 195 merges passed the analysis, 63 of which are regular merges and 125 are pull requests. Merely four scenarios returned only as build failures in all relevant revisions.

Results and examples

There were no relevant test failures in this project and the results across all strategies are almost uniform.

Merge scenario `240bcab` is an example of a merge conflict, that can be resolved by using structured merging. Here, on one side of the merge, a string constant is added in the file `src/main/java/org/kohsuke/github/Previews.java`, on the other side one is removed. Coincidentally, they are directly next to each other. This is of no relevance to JDime, since constants have no ordering in the AST. During the merge, JDime adds the constant from the one side and removes the other one from the result. Whereas text-based merge tools which, in the context of git, work by adding or removing lines and need to identify the corresponding locations by—among others—context markers, cannot find the anchor lines for the change and are therefore unable to resolve this conflict.

4.3.6 *Project javaparser*

The *JavaParser* project is a Java library providing tools to process and generate Java source code. It is used in many projects to parse and analyze Java code. The JavaParser project is maintained by Danny van Bruggen based on an earlier, now inactive project and it is developed by a community on GitHub.

Overview and results

Of 1679 scenarios present at the time of analysis, 1588 produced a result. These consist of 985 regular merged and 603 merged pull requests. Of these, 211 resulted in build failures for every relevant version, mostly due to errors while configuring the build system. For this project, JDime structured and semi-structured performed equally well and reduces the number of conflicts by more than 20%, while keeping the other results almost the same.

4.3.7 *Project jedis*

Jedis is a Java client for the key-value database server Redis. The goal for the Jedis project is to provide simple access to a Redis server for native Java use. It is maintained by Jonathan Leibiusky on GitHub.

Overview

For this project, there were 366 merge scenarios available at the time, of which 312 merge scenarios passed through the analysis. Those 312 analyzable scenarios consist of 233 regular merges and 79 pull requests. Of the analyzed merge scenarios 163 had no compiling version, which leaves 149 for further analysis.

Results and examples

Looking at different test case scenarios that occurred within the result of this project we can find 15 different patterns of test suite results. More than half of the scenarios fall into the category of a passed test case in all analyzed revisions of the merge scenario. Besides uniformly failing tests, most other scenarios resulted in some combination of merge conflicts. Both advanced JDime strategies resulted in a significant reduction of merge conflicts while keeping the number of build conflicts low and stable. Further, even though the number of conflicts was halved (in the case of JDime structured), no relevant test failures were reported.

The commit `a0aa723` is a good example to show the relevance of both the first and second research questions. When using git to automatically merge the parents `2cfc07f` and `8f725f9` of revision `a0aa723` we get a merge conflict in the file `src/main/java/redis/clients/jedis/Protocol.java`. This merge conflict spans two lines of an enum definition, one of which added an element to the enum. Both advanced strategies of JDime resolve this conflict correctly which allows the project to build whereas the build after merging with line-based merge tools is faulty. The added member of the enum gets removed and results in a build failure due to a missing symbol in the commit merge `a0aa723`. Our tool identified the merge scenario as one where the committed merge introduces a new defect and we show that JDime is able to prevent the problem by not requiring the developer to manually resolve the conflict. Similar examples can be found in eight other commits.

4.3.8 *Project okhttp*

OkHttp is an HTTP and HTTP2 client library for Java and is also explicitly supporting Android apps. The aim is to provide an efficient

and stable connection to communicate with a web server. OkHttp is primarily developed by Square Inc.

Overview and results

Of 1869 available scenarios 1081 were analyzable, which includes 347 regular merges and 734 pull requests. Of those 1081 scenarios, 156 resulted in build failures for every analyzed version. The reasons include unresolvable dependencies and a misconfigured build system. For this project, JDime structured performed the worst out of all strategies, both, in regard to merge conflicts and test failures. However, the share of non-clean scenarios is only about 2%, even for JDime structured.

4.3.9 *Project ontop*

Ontop is a framework for interfacing between SPARQL, a graph-based query language for RDF data, and relational databases. Ontop is a project by the "Knowledge Representation meets Databases" research group at the Free University of Bozen Bolzano.

Overview

Of a total of 1480 merges present in the history, 1455 passed the analysis. These are comprised of 1437 regular merges and 18 pull requests. For 471 of the analyzed scenarios, no revision resulted in a successful build, which includes, like most previous projects, snapshot dependencies that could not be resolved.

Results and examples

For this project, the share of non-clean merges is relatively high, which is in part due to the mentioned build errors. Further, many merge conflicts are present across all strategies. While JDime is able to reduce this high number up to 15%, a few more build failures can be found. Nevertheless, as with the previous examples, the majority of scenarios resulted in a passed test suite, followed by uniform test failures and in this project many errors. In total, 53 different test suite result patterns can be found in the results of this repository. Other patterns occurred only a few times, some of which are interesting to our analysis.

With merge scenario `eae4d9f`, a counterexample to JDime's correctness can be seen. Here, even though JDime structured is the only merge tool that is seemingly able to resolve all merge conflicts, the following build fails. The java compiler reports a missing symbol, is this case `getPrimaryKeys()` inside the file `reformulation-core/src/`

`main/java/it/unibz/inf/ontop/executor/leftjoin/RedundantSelf-LeftJoinExecutor.java` is not present in the merged version. This is the case because on one side of the merge this method got renamed to `getUniqueConstraints()`, while the version on the other side of the merge contains a new implementation of the mentioned file, which relies on the old name. This is a clear case of a semantic conflict, which JDime is neither able to detect not resolve. A similar albeit simpler example has been used by Mens [14] to distinguish semantic from syntactic conflicts.

4.3.10 *Project openmrs-core*

OpenMRS is an open-source medical records system with a focus on customizability. The *openmrs-core* project is its base API and web application code. The program is developed by the OpenMRS Inc. non-profit.

Overview and results

Of 1153 available, 1139 merges passed the analysis. Those include 1045 pull requests and 108 regular merges. 351 scenarios had no meaningful result because of build failures due to a failure while setting up a specific dependency. For this project, there is a relatively high share of scenarios that result in test failures. Further, both JDime strategies have a slight increase in all failure categories.

4.4 DISCUSSION

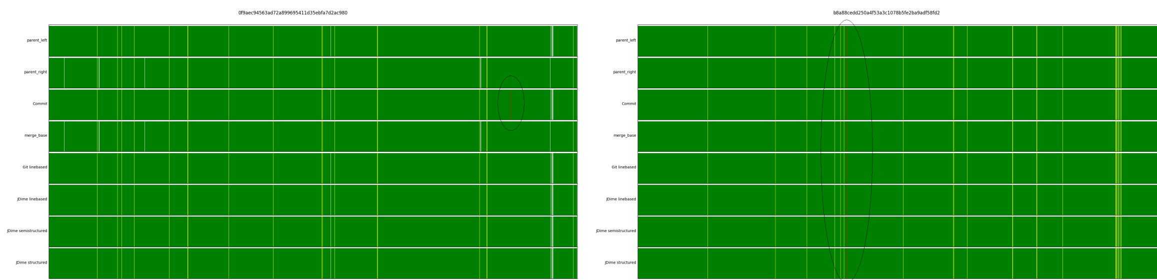
To answer our questions, we look at the detailed results of single repositories as well as the combination of all gathered data. As shown by the examples above, the analysis points out merge scenarios that, without introducing new changes by themselves, break the build or test cases.

Research question 1

Using the visualization provided by our tool, it is easy to identify merge scenarios where a test case previously passed but fails in the merge. A best practice that is followed in many projects is that at least revisions on the main branch should always compile. So, when developers merge their feature branches back into that main branch, they should always make sure that the changeset they are about to commit compiles. This gives the developer a chance to detect any faults introduced by the merge that would prevent clean compilation. Running the test suite on the changeset alone cannot always give the same information about the functionality. Considering the situation

where there is currently an unrelated test case failing on the target branch the test suite as a whole will fail after the merge. It might be hard for the developer to distinguish between such a case and a new failure introduced by the merge itself. This requires detailed information about test results from both parents. This additional information is clearly visible in the plots generated for each merge scenario. An example that visualizes the difference can be found in Figure 9.

During our analysis of the 7490 analyzed merge scenarios, we found 276 test cases across 65 scenarios that were broken by the merge commit, 63 of which (across 61 scenarios) at least one of the tested merge strategies was able to resolve. **We can, therefore, answer in regard to RQ1 that our tool is able to detect and even highlight merge scenarios that introduce bug by the merge itself.**



(a) A test case failure introduced by the merge commit.

(b) A failing test case unrelated to the merge.

Figure 9: The relevance of test suite information from the parents can be seen in these examples from the *openrms-core* project, Figure 9a shows a bug introduced by the merge while Figure 9b is an unrelated failing test case. In both cases, the test suite reports failed test cases.

Research question 2

The chart showing the results from all merge scenarios (Figure 10) suggests that our assumptions about the performance of JDime are confirmed. As shown by the data in Table 3, the results for the merge conflict category confirm that we were able to reproduce the results from Leßenich et al. [12]. While JDime's line-based strategy produced results similar to git internal implementation, the semi-structured strategy resulted in 57 fewer scenarios with conflicts while the structured approach produced the fewest merge conflicts (additional 36 scenarios fewer). Further, as shown in Figure 25, the conflict statistics provided by JDime itself, when comparing the number of conflicting sections, files, and lines within individual scenarios is also significantly reduced. This satisfies the prerequisite for answering the second question.

The results of categories *build failure* and *test failure* reveal that, across all analyzed projects, the ratio of commits failing at these cer-

tain stages to commits passing the immediately preceding stage does not change more than half a percentage point across different strategies. Since the number of commits reaching a certain stage differs between strategies, it is important to base all conclusions on this ratio of failing commits to previously passing commits instead of absolute figures. While the comparison of merge conflicts shows clear signs for the better performance of JDime, the share of build failures is constant across strategies. Whereas test suite analysis shows a low increase of failures for the more complex strategies.

When looking directly at individual test cases instead of combining all test case/build/merge results for a merge scenario, there is a similar pattern. Section A.3 shows that the most common pattern—aside from all versions passing all tests (with 5135042 occurrences)—is *merge conflicts in all merge tools*. Further down, but still with significant volume are cases where JDime structured is the only merge tool to merge successfully, and then cases, where both JDime structured and JDime semi-structured are able to resolve all conflicts without failure while line-based strategies are not. Grouping similar patterns, e.g. build failures in the merge base, the ranking is as follows:

1. `XXXXMMMM` (153747 occurrences)
2. `XXXXMMMP` (20798)
3. `XXXXMMPP` (9811)

Cases like the pattern `PPPPPPPT` occur only in a relatively small number of cases. These bould be problematic, as the `T` in the last column indicates a test failure that was introduces by the JDime structured merge (see Section 3.2.3 for further explanation on the pattern).

When focusing on singe projects, the results are not always showing the same outcome. Similar to the previous results from Leßenich et al. [12], in some instances, JDime is not able to reduce the number of merge conflicts. Further, for some projects like *commons-math*, using these advanced merge strategies will result in a slight increase in defects introduced by the merge. While the variations are not significant, it is important to note that JDime is not yet able to handle all codebases equally well. The same is true for the test suite results. While some projects show no increase or even a decrease of test failures, others show an increase.

Basing our reasoning on the combined results from all analyzed projects, we can answer **RQ2**. **The structured (and semi-structured) merge strategy provided by JDime is evidently able to reduce the number of conflicts while only introducing a small number delayed defects.**

One additional noteworthy observation can be made by looking at the occurring patterns. Patterns in the form where either parent has a build failure and JDime structured or semi-structured strategies result in merge failures might be explained due to the fact that JDime

must be able to parse the source code into an AST to be able to merge. A common reason for build failures is code that cannot be compiled due to syntax errors. These errors might, in turn, prevent JDime from working correctly and thus resulting in a failed merge.

	error	% (of)	merge conflict	% (of)	build failure	% (of)	test failure	% (of)	passed	% (of)
Commit	43	0.0057 (7490)	0	0 (7447)	169	0.023 (7447)	70	0.0096 (7278)	7208	1 (7208)
Git linebased	264	0.035 (7490)	459	0.064 (7226)	106	0.016 (6767)	64	0.0096 (6661)	6597	1 (6597)
JDime linebased	264	0.035 (7490)	449	0.062 (7226)	111	0.016 (6777)	58	0.0087 (6666)	6608	1 (6608)
JDime semistructured	269	0.036 (7490)	401	0.056 (7221)	122	0.018 (6820)	65	0.0097 (6698)	6633	1 (6633)
JDime structured	267	0.036 (7490)	365	0.051 (7223)	124	0.018 (6858)	73	0.011 (6734)	6661	1 (6661)

Table 3: Combined results from all merge scenarios.

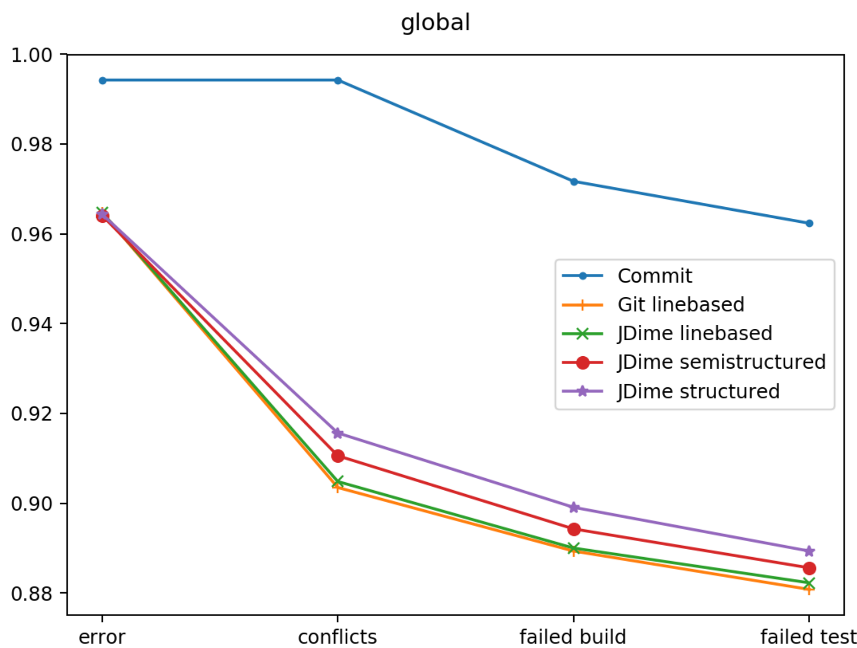


Figure 10: The data from Table 3 visualized.

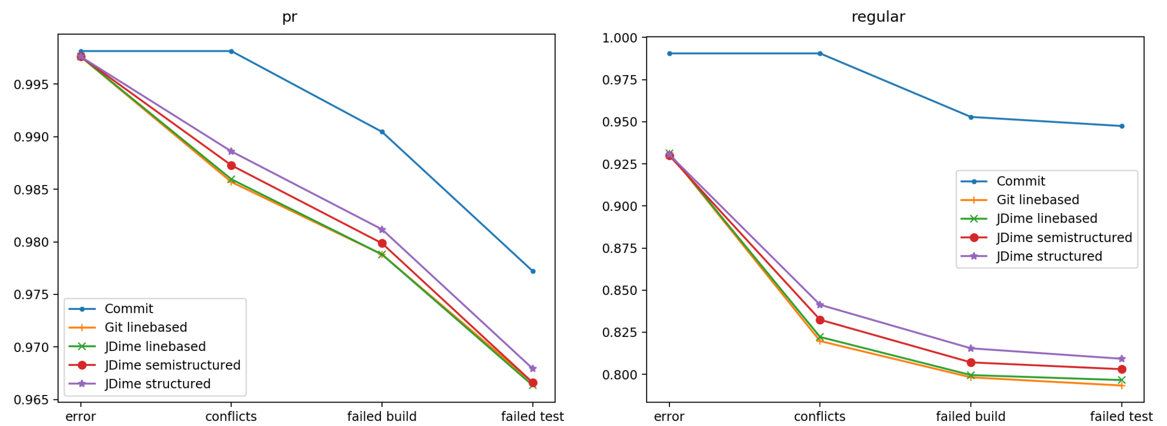
Research question 3

In Figure 11, a comparison between 3773 merged pull requests and 3712 regular merges can be seen. While the pull request group still shows some conflicting merges, with approximately one percent of all scenarios falling into this category our assumptions are confirmed. Similarly, build failures are also reduced by almost a factor of 5. In contrast, the test failure rate has more than doubled. This might be a result of survivorship bias because the higher rate of merge conflicts in the regular merges group might filter out more scenarios early on.

This is supported by the fact that the merged version that was found in the history shows change at the same scale.

To answer RQ3, as expected, the share of merge conflicts for pull requests is greatly reduced, while the share of test failures is significantly increased.

This can also help us interpret the findings from the previous question. The data shows a slight increase of test failures for both advanced JDime strategies. Using the findings for pull requests, this could also be attributed to the survivorship bias, but a conclusive argument cannot be made based on the available data.



(a) An test case failure introduced by the merge commit.

(b) A failing test case unrelated to the merge.

Figure 11: Comparison between merged pull requests and regular merges

5 CONCLUSION

5.1 SUMMARY

In this thesis, we have verified that using JDime can help to reduce the number of merge conflicts and their size in real-world applications. An improvement can be seen with the semi-structured strategy, a further reduction is evident with the structured strategy. Additionally, we have shown that this reduction does not introduce substantially more bugs in the merge result that would lead to build or test failures than standard tools do. We have therefore shown, that while JDime provides improvements, the possible drawback in regard to the quality of the results is almost neglectable.

5.2 RELATED WORK

The approach of using the results of build tool and unit test suite execution is the basis for other tools with the goal of helping developers during the development of features by detecting potential merge conflicts with other branches. For this purpose, Brun et al. [6] proposed a tool called *Crystal* that detects and classifies conflicts into similar categories as our MergeProfiler. Guimarães et al. [9] expand on the concept to provide similar information not only isolated for one developer but all developers working on the code base and integrate the results into the developer's IDE. In both cases, the goal was to aid the developer during the development process by showing potential conflicts that could occur later on.

Besides JDime, there are other tools that incorporate structure information into the merge process. Pioneer work was done by Westfechtel [17] and Buffenbarger [7], and a range of merge and differencing tools have been proposed [4], while other tools even try to use semantic information to detect and resolve additional conflicts [10].

5.3 FUTURE WORK

The current version of the MergeProfiler is restricted in some aspects, for example only Maven is supported as build system. However, the structure of the MergeProfiler tool, especially the `BuildTool` interface, allows us to extend the possibilities for analyzing projects to other build tools in the Java world as well. Implementations for Gradle and Apache Ant, both popular build management tools, are being currently worked on. The `TestSuiteProfiler` can check for a supported

build system and select the one present in the repository. With the possibility of analyzing projects built with the three most popular build tool for Java programs and libraries, the techniques shown in this thesis can be used for a wide range of candidates. This would also allow the analysis of projects where the build tool was switched during the development. Also, as shown in Section 4.3, in some projects a high rate of merge commits failed the analysis. We are working on improving the share of scenarios that can be analyzed.

As mentioned in Section 2.3.2, JDime is currently only enabled for the actual merge. In the future, the analysis can be expanded to execute JDime on all necessary merges, including intermediate ones.

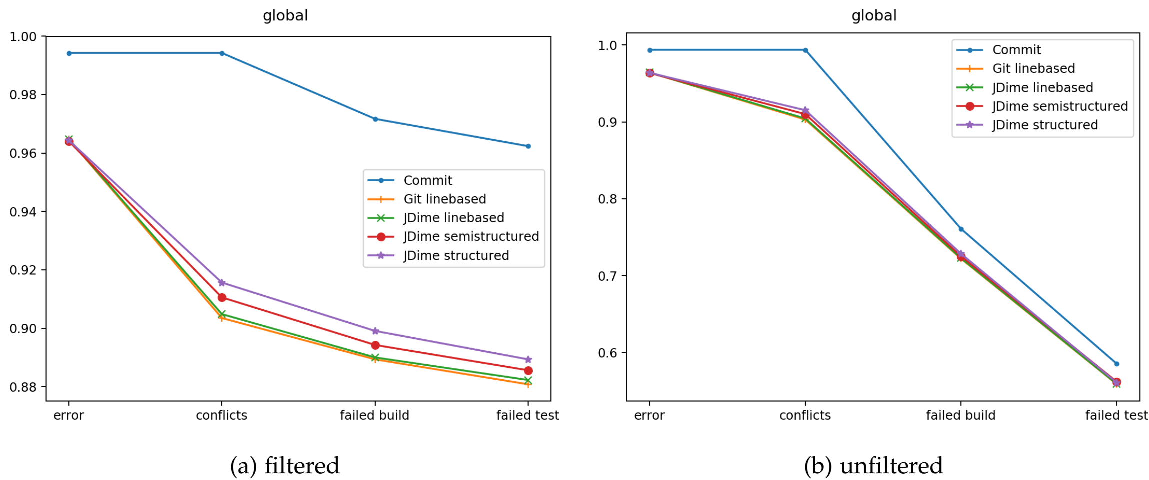
The integration of additional data from GitHub would allow the tool to analyze more scenarios like unmerged or rejected pull requests. As stated in **RQ**3, GitHub only allows pull requests to merge if they resolve cleanly. It might be interesting to check if this contributes to the decision of rejecting pull requests as their merge would be to complex.

A

APPENDIX

A.1 GLOBAL MERGE SCENARIO RESULTS

In this section we show all data generated by our tools during the analysis of the 10 projects.



(a) filtered

(b) unfiltered

Figure 12: Comparison of merge scenario results filtered as detailed in Section 3.2.2 and the raw results provided by the merge/build tools.

Table 4: Results from all merge scenarios that could be identified as pull requests.

	error	% (of)	merge conflict	% (of)	build failure	% (of)	test failure	% (of)	passed	% (of)
Commit	7	0.0019 (3773)	0	0 (3766)	29	0.0077 (3766)	50	0.013 (3737)	3687	1 (3687)
Git linebased	9	0.0024 (3773)	45	0.012 (3764)	26	0.007 (3719)	46	0.012 (3693)	3647	1 (3647)
JDime linebased	9	0.0024 (3773)	44	0.012 (3764)	27	0.0073 (3720)	47	0.013 (3693)	3646	1 (3646)
JDime semistructured	9	0.0024 (3773)	39	0.01 (3764)	28	0.0075 (3725)	50	0.014 (3697)	3647	1 (3647)
JDime structured	9	0.0024 (3773)	34	0.009 (3764)	28	0.0075 (3730)	50	0.014 (3702)	3652	1 (3652)

Table 5: Combined results from all remaining merge scenarios.

	error	% (of)	merge conflict	% (of)	build failure	% (of)	test failure	% (of)	passed	% (of)
Commit	35	0.0094 (3712)	0	0 (3677)	140	0.038 (3677)	20	0.0057 (3537)	3517	1 (3517)
Git linebased	255	0.069 (3712)	414	0.12 (3457)	80	0.026 (3043)	18	0.0061 (2963)	2945	1 (2945)
JDime linebased	255	0.069 (3712)	405	0.12 (3457)	84	0.028 (3052)	11	0.0037 (2968)	2957	1 (2957)
JDime semistructured	260	0.07 (3712)	362	0.1 (3452)	94	0.03 (3090)	15	0.005 (2996)	2981	1 (2981)
JDime structured	258	0.07 (3712)	331	0.096 (3454)	96	0.031 (3123)	23	0.0076 (3027)	3004	1 (3004)

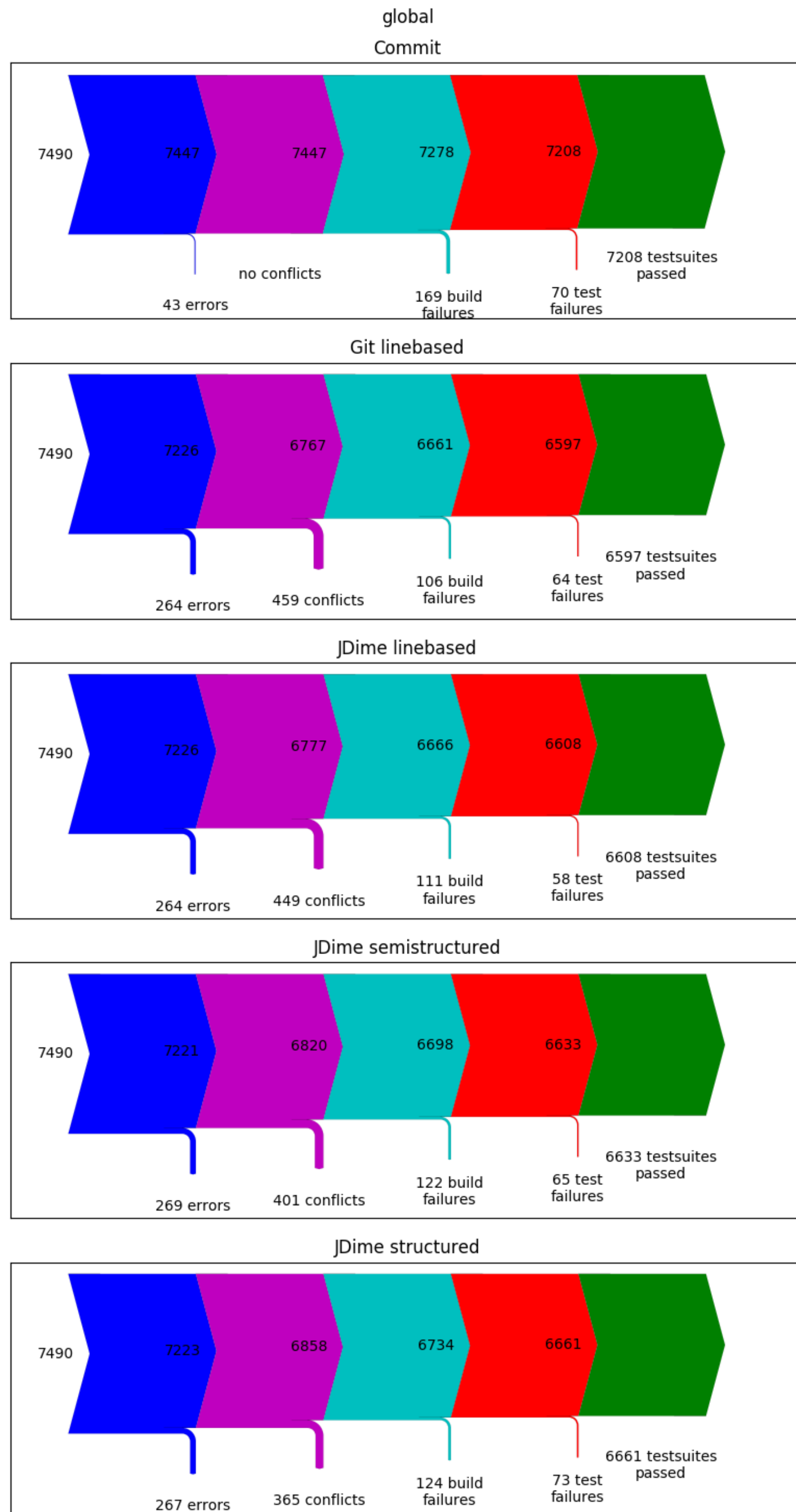


Figure 13: The data for JDime results, same as in Figure 12a, visualized as aSankey plot.

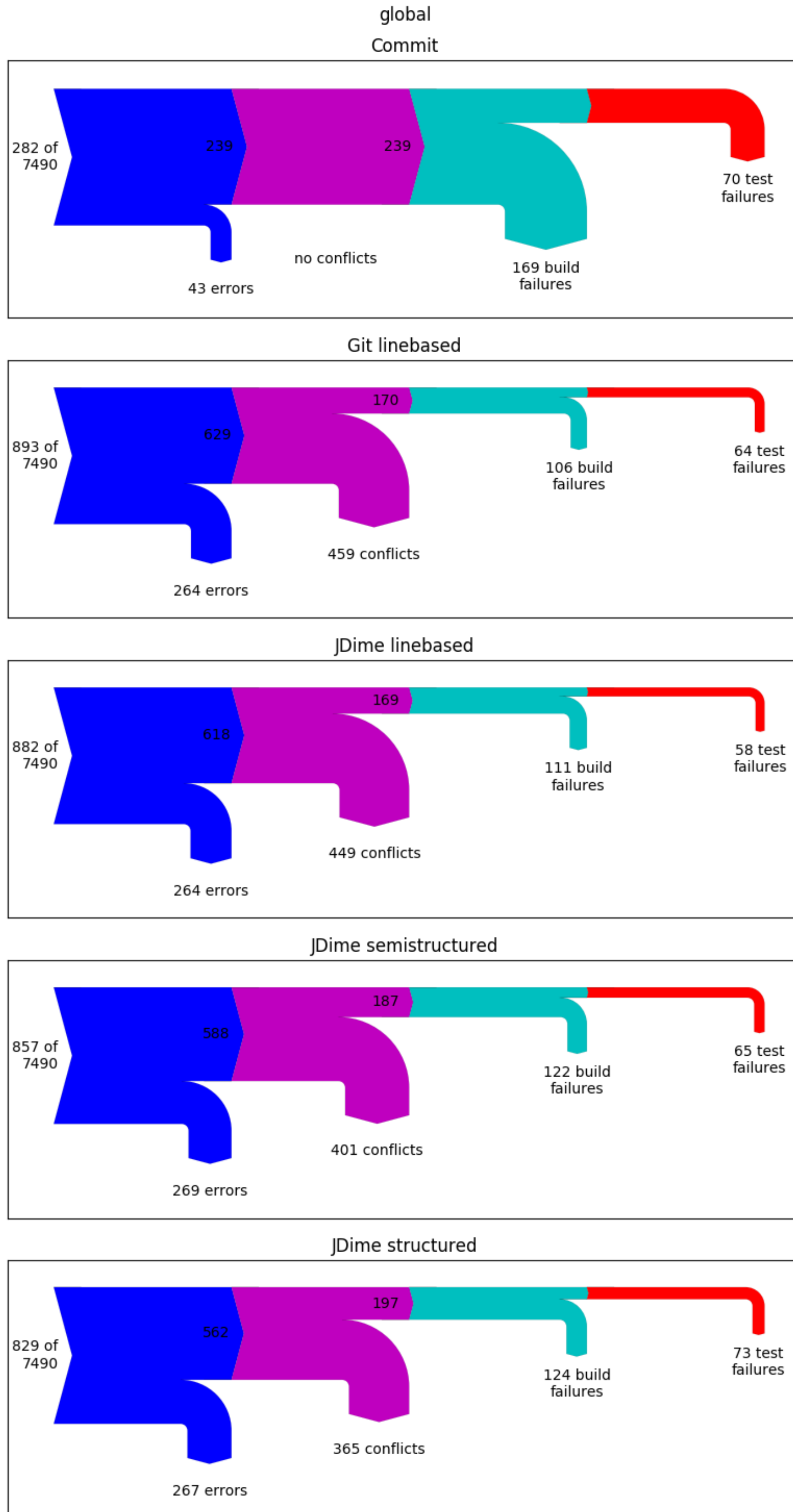


Figure 14: Similar to Figure 13, but with all passing merge scenarios removed, to improve visibility.

A.2 MERGE SCENARIO RESULTS PER PROJECTS

In this section, the line plots for each project are shown.

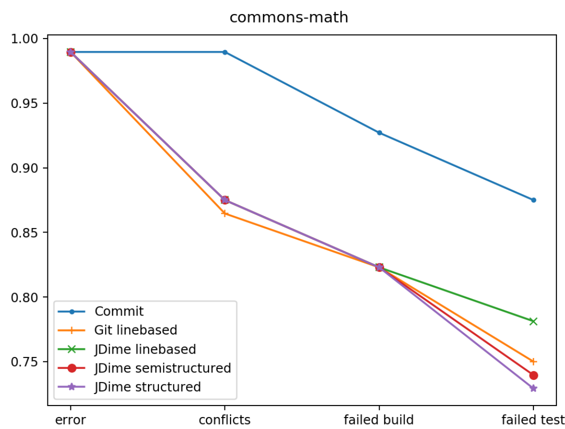


Figure 15: The result category plot for project commons-math.

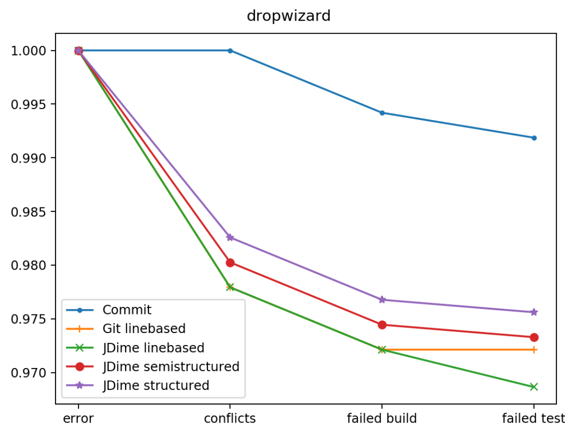


Figure 16: The result category plot for project dropwizard.

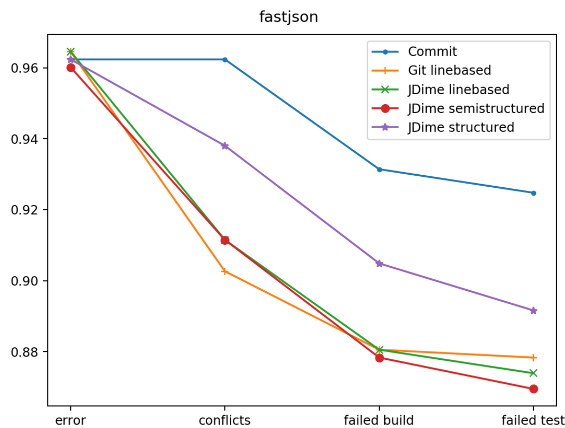


Figure 17: The result category plot for project fastjson.

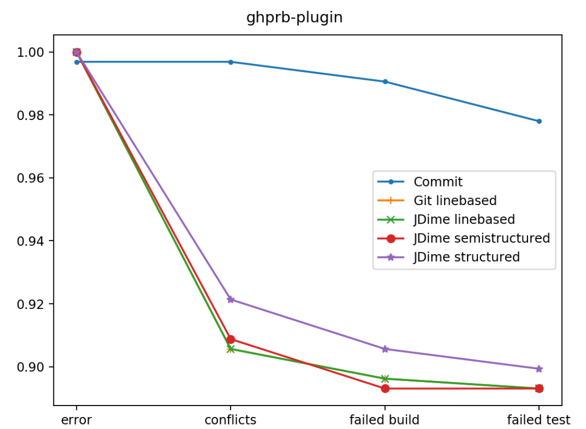


Figure 18: The result category plot for project ghprb-plugin.

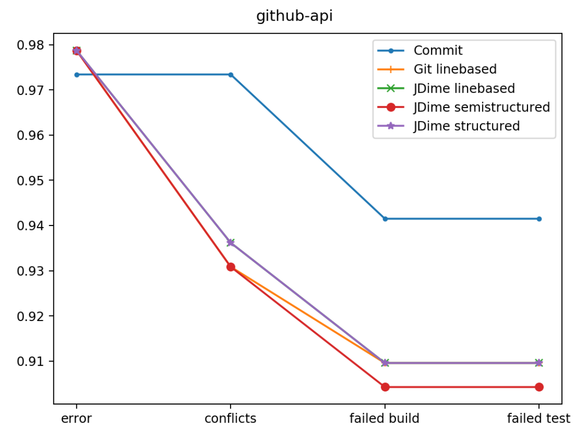


Figure 19: The result category plot for project github-api.

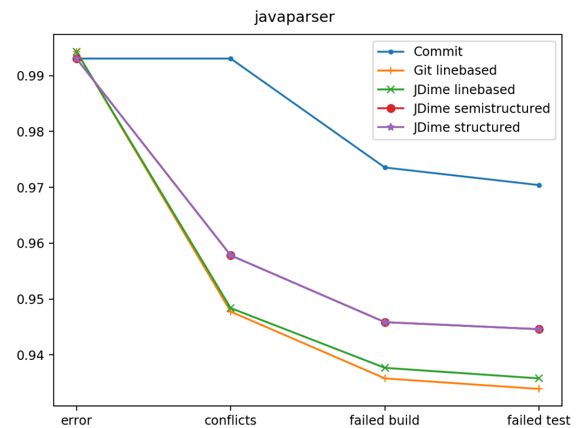


Figure 20: The result category plot for project javaparser.

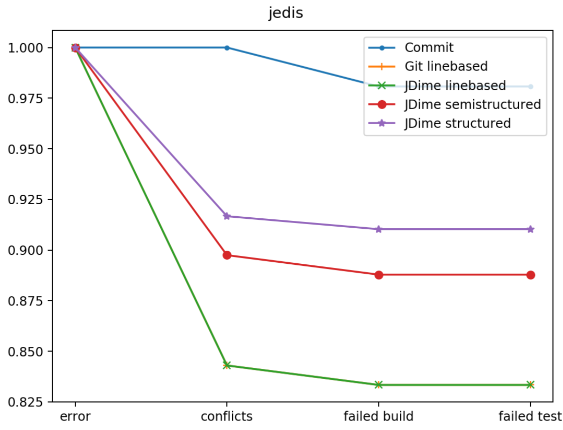


Figure 21: The result category plot for project jedis.

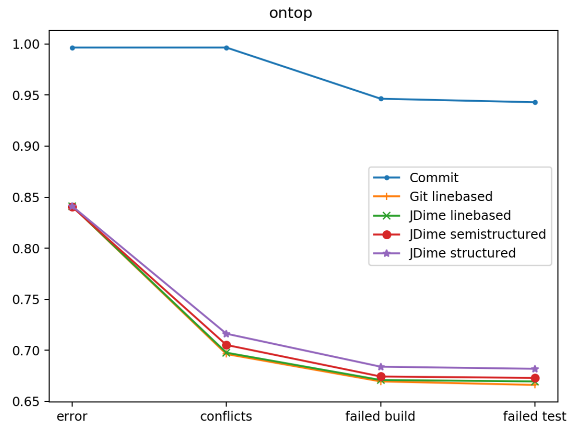


Figure 23: The result category plot for project ontop.

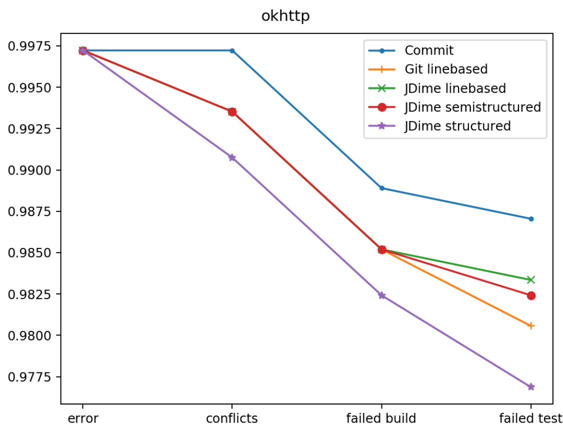


Figure 22: The result category plot for project okhttp.

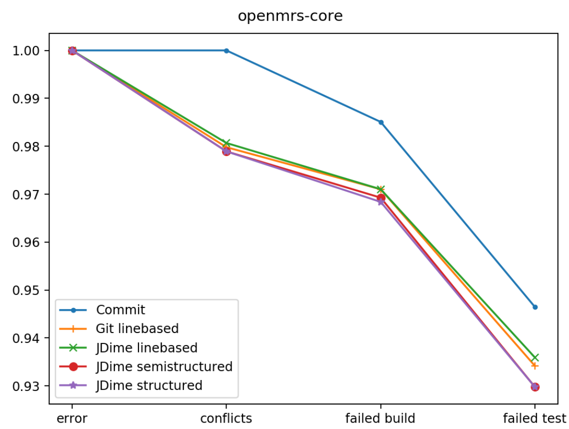


Figure 24: The result category plot for project openmrs-core.

The data on which the plots are based is provided in the following tables.

Table 6: The result category data for project commons-math.

	error	% (of)	merge conflict	% (of)	build failure	% (of)	test failure	% (of)	passed	% (of)
Commit	1	0.01 (96)	0	0 (95)	6	0.063 (95)	5	0.056 (89)	84	1 (84)
Git linebased	1	0.01 (96)	12	0.13 (95)	4	0.048 (83)	7	0.089 (79)	72	1 (72)
JDime linebased	1	0.01 (96)	11	0.12 (95)	5	0.06 (84)	4	0.051 (79)	75	1 (75)
JDime semistructured	1	0.01 (96)	11	0.12 (95)	5	0.06 (84)	8	0.1 (79)	71	1 (71)
JDime structured	1	0.01 (96)	11	0.12 (95)	5	0.06 (84)	9	0.11 (79)	70	1 (70)

Table 7: The result category data for project dropwizard.

	error	% (of)	merge conflict	% (of)	build failure	% (of)	test failure	% (of)	passed	% (of)
Commit	0	0 (861)	0	0 (861)	5	0.0058 (861)	2	0.0023 (856)	854	1 (854)
Git linebased	0	0 (861)	19	0.022 (861)	5	0.0059 (842)	0	0 (837)	837	1 (837)
JDime linebased	0	0 (861)	19	0.022 (861)	5	0.0059 (842)	3	0.0036 (837)	834	1 (834)
JDime semistructured	0	0 (861)	17	0.02 (861)	5	0.0059 (844)	1	0.0012 (839)	838	1 (838)
JDime structured	0	0 (861)	15	0.017 (861)	5	0.0059 (846)	1	0.0012 (841)	840	1 (840)

Table 8: The result category data for project fastjson.

	error	% (of)	merge conflict	% (of)	build failure	% (of)	test failure	% (of)	passed	% (of)
Commit	17	0.038 (452)	0	0 (435)	14	0.032 (435)	3	0.0071 (421)	418	1 (418)
Git linebased	16	0.035 (452)	28	0.064 (436)	10	0.025 (408)	1	0.0025 (398)	397	1 (397)
JDime linebased	16	0.035 (452)	24	0.055 (436)	14	0.034 (412)	3	0.0075 (398)	395	1 (395)
JDime semistructured	18	0.04 (452)	22	0.051 (434)	15	0.036 (412)	4	0.01 (397)	393	1 (393)
JDime structured	17	0.038 (452)	11	0.025 (435)	15	0.035 (424)	6	0.015 (409)	403	1 (403)

Table 9: The result category data for project ghprb-plugin.

	error	% (of)	merge conflict	% (of)	build failure	% (of)	test failure	% (of)	passed	% (of)
Commit	1	0.0031 (318)	0	0 (317)	2	0.0063 (317)	4	0.013 (315)	311	1 (311)
Git linebased	0	0 (318)	30	0.094 (318)	3	0.01 (288)	1	0.0035 (285)	284	1 (284)
JDime linebased	0	0 (318)	30	0.094 (318)	3	0.01 (288)	1	0.0035 (285)	284	1 (284)
JDime semistructured	0	0 (318)	29	0.091 (318)	5	0.017 (289)	0	0 (284)	284	1 (284)
JDime structured	0	0 (318)	25	0.079 (318)	5	0.017 (293)	2	0.0069 (288)	286	1 (286)

Table 10: The result category data for project github-api.

	error	% (of)	merge conflict	% (of)	build failure	% (of)	test failure	% (of)	passed	% (of)
Commit	5	0.027 (188)	0	0 (183)	6	0.033 (183)	0	0 (177)	177	1 (177)
Git linebased	4	0.021 (188)	9	0.049 (184)	4	0.023 (175)	0	0 (171)	171	1 (171)
JDime linebased	4	0.021 (188)	8	0.043 (184)	5	0.028 (176)	0	0 (171)	171	1 (171)
JDime semistructured	4	0.021 (188)	9	0.049 (184)	5	0.029 (175)	0	0 (170)	170	1 (170)
JDime structured	4	0.021 (188)	8	0.043 (184)	5	0.028 (176)	0	0 (171)	171	1 (171)

Table 11: The result category data for project javaparser.

	error	% (of)	merge conflict	% (of)	build failure	% (of)	test failure	% (of)	passed	% (of)
Commit	11	0.0069 (1588)	0	0 (1577)	31	0.02 (1577)	5	0.0032 (1546)	1541	1 (1541)
Git linebased	9	0.0057 (1588)	74	0.047 (1579)	19	0.013 (1505)	3	0.002 (1486)	1483	1 (1483)
JDime linebased	9	0.0057 (1588)	73	0.046 (1579)	17	0.011 (1506)	3	0.002 (1489)	1486	1 (1486)
JDime semistructured	11	0.0069 (1588)	56	0.036 (1577)	19	0.012 (1521)	2	0.0013 (1502)	1500	1 (1500)
JDime structured	11	0.0069 (1588)	56	0.036 (1577)	19	0.012 (1521)	2	0.0013 (1502)	1500	1 (1500)

Table 12: The result category data for project jedis.

	error	% (of)	merge conflict	% (of)	build failure	% (of)	test failure	% (of)	passed	% (of)
Commit	0	0 (312)	0	0 (312)	6	0.019 (312)	0	0 (306)	306	1 (306)
Git linebased	0	0 (312)	49	0.16 (312)	3	0.011 (263)	0	0 (260)	260	1 (260)
JDime linebased	0	0 (312)	49	0.16 (312)	3	0.011 (263)	0	0 (260)	260	1 (260)
JDime semistructured	0	0 (312)	32	0.1 (312)	3	0.011 (280)	0	0 (277)	277	1 (277)
JDime structured	0	0 (312)	26	0.083 (312)	2	0.007 (286)	0	0 (284)	284	1 (284)

Table 13: The result category data for project okhttp.

	error	% (of)	merge conflict	% (of)	build failure	% (of)	test failure	% (of)	passed	% (of)
Commit	3	0.0028 (1081)	0	0 (1078)	9	0.0083 (1078)	2	0.0019 (1069)	1067	1 (1067)
Git linebased	3	0.0028 (1081)	4	0.0037 (1078)	9	0.0084 (1074)	5	0.0047 (1065)	1060	1 (1060)
JDime linebased	3	0.0028 (1081)	4	0.0037 (1078)	9	0.0084 (1074)	2	0.0019 (1065)	1063	1 (1063)
JDime semistructured	3	0.0028 (1081)	4	0.0037 (1078)	9	0.0084 (1074)	3	0.0028 (1065)	1062	1 (1062)
JDime structured	3	0.0028 (1081)	7	0.0065 (1078)	9	0.0084 (1071)	6	0.0056 (1062)	1056	1 (1056)

Table 14: The result category data for project ontop.

	error	% (of)	merge conflict	% (of)	build failure	% (of)	test failure	% (of)	passed	% (of)
Commit	5	0.0034 (1455)	0	0 (1450)	73	0.05 (1450)	5	0.0036 (1377)	1372	1 (1372)
Git linebased	231	0.16 (1455)	211	0.17 (1224)	39	0.038 (1013)	5	0.0051 (974)	969	1 (969)
JDime linebased	231	0.16 (1455)	209	0.17 (1224)	39	0.038 (1015)	2	0.002 (976)	974	1 (974)
JDime semistructured	232	0.16 (1455)	197	0.16 (1223)	45	0.044 (1026)	2	0.002 (981)	979	1 (979)
JDime structured	231	0.16 (1455)	182	0.15 (1224)	47	0.045 (1042)	3	0.003 (995)	992	1 (992)

Table 15: The result category data for project openmrs-core.

	error	% (of)	merge conflict	% (of)	build failure	% (of)	test failure	% (of)	passed	% (of)
Commit	0	0 (1139)	0	0 (1139)	17	0.015 (1139)	44	0.039 (1122)	1078	1 (1078)
Git linebased	0	0 (1139)	23	0.02 (1139)	10	0.009 (1116)	42	0.038 (1106)	1064	1 (1064)
JDime linebased	0	0 (1139)	22	0.019 (1139)	11	0.0098 (1117)	40	0.036 (1106)	1066	1 (1066)
JDime semistructured	0	0 (1139)	24	0.021 (1139)	11	0.0099 (1115)	45	0.041 (1104)	1059	1 (1059)
JDime structured	0	0 (1139)	24	0.021 (1139)	12	0.011 (1115)	44	0.04 (1103)	1059	1 (1059)

A.3 TEST CASE RESULTS

The following table contains the results of all test case scenarios where the test case is present in all related commits with all scenarios ignored due to flaky tests filtered out. `M` indicates a merge conflict, `B` indicates a build failure with the merge result, `T` indicates a failed test case, `P` a passed one.

Table 16: Results of all test case scenarios.

Scenario								Number of occurrences
left parent	right parent	committed merge	merge base	git merge	JDime linebased	JDime semi-structured	JDime structured	
P	P	P	P	P	P	P	P	5135042
P	P	P	P	M	M	M	M	101824
P	B	P	B	B	B	B	B	59091
P	P	B	B	B	B	B	B	36594
B	P	B	P	P	P	P	P	22183
B	B	P	P	P	P	P	P	21104
P	P	P	P	M	M	M	P	20262
P	B	P	B	M	M	M	M	19726
P	P	P	B	M	M	M	M	15392
P	P	P	P	M	M	P	P	9811
P	P	P	P	M	B	B	B	9208
P	B	P	B	M	B	B	B	8416
B	B	P	P	M	M	M	M	7154
P	P	P	B	B	B	B	B	5334
P	P	P	B	M	M	B	B	5011
P	B	B	B	M	M	M	M	2979
P	B	P	B	M	M	M	B	2955
P	P	P	P	P	P	M	M	2928
P	P	P	P	P	P	P	M	2595
B	B	P	B	M	M	M	M	2584
P	P	P	P	P	P	M	P	1608
P	P	B	P	M	M	M	M	1577
P	P	P	B	M	M	P	P	1368
P	P	P	P	M	M	B	M	1304
P	P	P	B	M	M	M	B	1193
P	P	P	B	M	M	P	M	1127
P	P	B	B	M	M	M	M	1019

Continued

Scenario								Number of occurrences
left parent	right parent	committed merge	merge base	git merge	JDime linebased	JDime semi-structured	JDime structured	
P	P	P	P	M	M	M	B	920
P	P	P	P	M	M	B	B	591
P	P	P	B	M	M	M	P	534
B	P	P	P	M	M	M	M	530
B	P	P	B	B	B	B	B	521
P	P	P	P	B	P	P	P	517
B	P	B	B	M	M	M	M	513
P	P	P	B	B	B	B	M	411
P	B	P	P	P	P	P	P	393
T	T	B	B	B	B	B	B	323
B	P	P	P	P	P	P	P	276
P	P	P	P	M	M	P	M	275
P	P	P	P	P	P	B	B	263
P	P	P	B	B	P	P	P	259
P	P	P	P	M	M	B	P	258
P	P	P	P	M	P	M	P	246
P	P	P	P	M	P	P	M	246
T	T	P	P	P	P	P	P	208
P	P	B	P	P	P	P	P	200
P	P	T	P	M	M	M	M	149
T	T	T	T	M	M	M	M	145
P	P	P	P	P	P	P	T	127
P	P	P	P	M	B	P	P	119
B	B	T	T	T	T	T	T	110
B	T	B	T	T	T	T	T	93
P	P	T	T	T	T	T	T	84
T	T	T	T	M	B	B	B	76
B	B	B	B	M	M	M	M	62
P	T	P	T	T	T	T	T	61
T	B	T	B	B	B	B	B	60
T	T	T	B	M	M	M	M	60
P	T	P	P	P	P	P	P	51
P	P	P	T	M	M	M	M	49

Continued

left parent	right parent	committed merge	merge base	git merge	JDime linebased	JDime semi-structured	JDime structured	Number of occurrences
P	P	P	P	P	P	T	P	34
T	P	P	P	P	P	P	P	32
P	P	P	T	P	P	P	P	31
P	P	T	P	P	P	P	P	31
T	P	T	P	P	P	P	P	28
P	P	P	P	T	P	P	P	26
T	T	B	T	M	M	M	M	25
P	B	P	B	B	B	M	M	24
P	P	P	B	M	M	B	M	24
P	P	P	P	P	T	P	P	21
P	P	P	T	T	T	T	T	17
P	P	B	P	B	B	B	B	12
T	B	P	B	B	B	B	B	11
P	P	T	B	B	B	B	B	9
P	B	T	B	B	B	B	B	9
P	P	P	P	T	P	T	P	6
P	T	P	P	P	T	P	P	5
P	P	P	P	P	T	T	P	4
T	P	P	P	P	T	P	P	4
P	T	P	P	T	P	P	P	4
B	B	T	T	M	M	M	M	4
P	T	P	T	M	M	M	M	4
P	P	P	P	T	P	P	T	3
P	P	P	P	T	T	P	P	3
T	P	P	P	P	P	T	P	3
T	P	P	P	T	P	T	T	3
T	P	P	P	T	T	P	P	3
P	P	P	T	P	P	T	T	3
P	P	P	T	T	P	P	P	3
T	P	P	T	P	P	P	P	3
T	P	P	T	P	P	P	T	3
P	P	T	P	P	P	P	T	3
P	P	T	P	P	P	T	P	3

Continued

left parent	right parent	committed merge	merge base	git merge	JDime linebased	JDime semi-structured	JDime structured	Number of occurrences
P	P	T	P	P	T	P	P	3
P	P	T	P	T	P	P	P	3
P	P	T	P	T	P	T	P	3
P	T	P	P	P	P	T	P	3
P	T	P	P	T	P	T	P	3
P	T	P	P	T	T	P	P	3
T	T	P	P	P	P	P	T	3
P	T	P	T	P	P	P	P	3
T	T	B	B	M	M	M	M	3
T	T	T	B	B	B	B	B	3
B	B	T	B	M	M	M	M	2
P	P	P	P	P	P	T	T	2
P	P	P	P	P	T	T	T	2
P	P	P	P	T	T	P	T	2
T	P	P	P	P	T	T	T	2
P	P	P	T	P	T	P	P	2
P	P	P	T	P	T	P	T	2
P	P	P	T	P	T	T	P	2
T	P	P	T	T	P	P	P	2
T	P	P	T	T	P	T	P	2
P	P	T	T	P	P	T	T	2
P	T	P	P	P	P	P	T	2
P	T	P	T	P	P	T	P	2
T	T	P	T	P	P	P	P	2
P	T	T	P	P	P	P	P	2
T	P	P	T	T	T	T	T	2
P	T	P	T	M	M	T	T	2
T	B	T	T	T	T	T	T	2
B	T	B	B	M	M	M	M	2
B	T	T	T	M	M	M	M	2
P	T	B	B	B	B	B	B	2
B	B	T	P	P	P	P	P	1
P	P	P	P	T	T	T	P	1

Continued

Scenario								Number of occurrences
left parent	right parent	committed merge	merge base	git merge	JDime linebased	JDime semi-structured	JDime structured	
T	P	P	P	P	P	P	T	1
T	P	P	P	P	P	T	T	1
T	P	P	P	P	T	P	T	1
T	P	P	P	T	P	P	P	1
T	P	P	P	T	T	T	P	1
T	P	P	P	T	T	T	T	1
P	P	P	T	P	P	P	T	1
P	P	P	T	T	T	T	P	1
T	P	P	T	P	P	T	T	1
T	P	P	T	P	T	P	P	1
T	P	P	T	P	T	P	T	1
T	P	P	T	P	T	T	P	1
T	P	P	T	T	P	T	T	1
P	P	T	P	P	P	T	T	1
P	P	T	P	P	T	T	P	1
P	P	T	P	T	P	P	T	1
P	P	T	P	T	P	T	T	1
P	P	T	P	T	T	P	T	1
P	P	T	P	T	T	T	T	1
T	P	T	P	M	M	M	M	1
T	P	T	P	P	P	T	P	1
T	P	T	P	T	P	P	P	1
T	P	T	P	T	T	P	P	1
P	P	T	T	P	T	T	P	1
P	P	T	T	T	P	T	T	1
P	P	T	T	T	T	P	T	1
T	P	T	T	P	P	P	P	1
T	P	T	T	T	P	P	P	1
T	P	T	T	T	P	T	P	1
T	P	T	T	T	P	T	T	1
T	P	T	T	T	T	T	P	1
T	P	T	T	T	T	T	T	1
P	T	P	P	P	T	T	P	1

Continued

Scenario								Number of occurrences
left parent	right parent	committed merge	merge base	git merge	JDime linebased	JDime semi-structured	JDime structured	
P	T	P	P	P	T	T	T	1
P	T	P	P	T	P	P	T	1
P	T	P	P	T	T	T	T	1
T	T	P	P	P	P	T	T	1
T	T	P	P	P	T	P	P	1
T	T	P	P	P	T	T	P	1
T	T	P	P	T	T	P	T	1
T	T	P	P	T	T	T	P	1
P	T	P	T	P	T	P	P	1
P	T	P	T	P	T	P	T	1
P	T	P	T	T	P	P	P	1
T	T	P	T	P	P	T	T	1
P	T	T	T	P	P	P	P	1
T	B	T	B	M	M	M	M	1
P	P	T	T	P	P	P	P	1
B	B	P	T	P	P	P	P	1
P	P	P	P	T	T	T	T	1
P	T	P	P	P	T	P	T	1
T	T	P	T	P	P	P	T	1
T	P	T	P	M	M	M	P	1
T	T	T	T	M	M	B	B	1
T	T	T	T	M	M	M	B	1
T	B	P	B	M	B	B	B	1
T	P	B	B	B	B	B	B	1
P	P	P	T	T	T	P	P	1
P	T	P	T	P	P	P	T	1
T	P	P	P	M	M	M	P	1
T	T	P	P	M	B	B	B	1

A.4 GLOBAL JDIME MERGE STATISTICS

Hereafter follow statistics and box plots generated from data recorded by JDime during its runs. First an overview of all projects combined is given. For better visibility, there are two versions, on with all data points and on without displaying outliers. The p-Values below are obtained by running a Wilcoxon signed-rank test ([18]) on the paired samples. If a strategy does not produce a conflict, it is recorded with a `0`. Values in red indicate a small sample set with limited significance.

Table 17: Global JDime merge conflict statistics.

type	sum	mean	median	standard deviation	min	max
			JDime linebased			
conflicts	4041	9.78	2	38.27	0	647
files	1709	4.14	1	12.32	0	198
lines	76570	185.40	22	733.60	0	8129
			JDime semistructured			
conflicts	2922	7.12	1	27.87	0	454
files	1275	3.08	1	10.24	0	157
lines	199854	483.91	11	2418.40	0	29551
			JDime structured			
conflicts	2752	6.66	1	29.95	0	514
files	1164	2.82	1	10.06	0	127
lines	274677	665.08	4	3389.57	0	38655

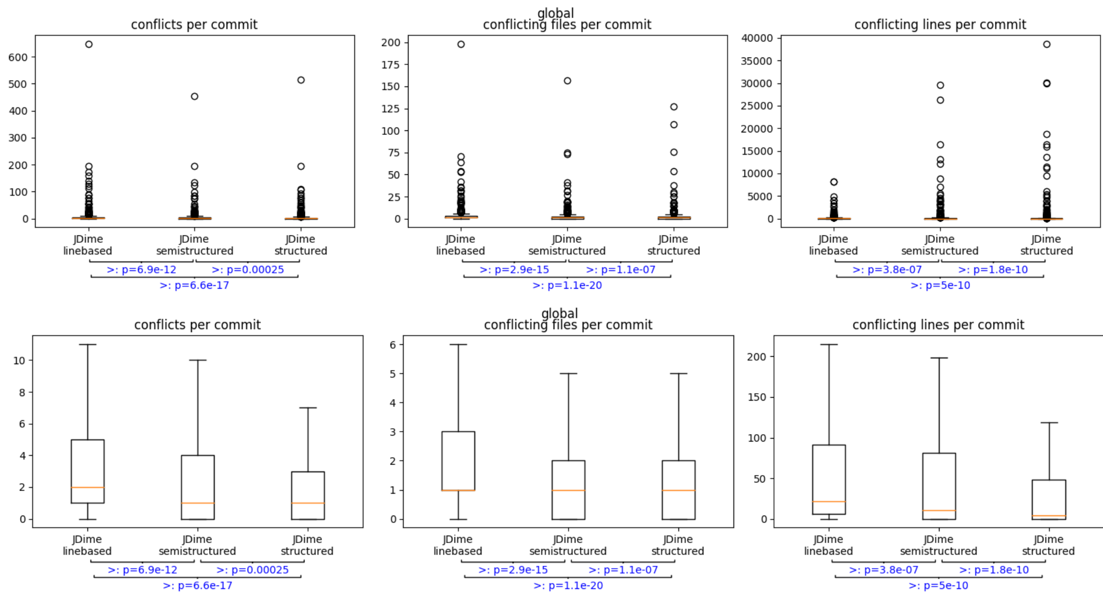


Figure 25: The global JDime merge statistics.

A.5 JDIME MERGE STATISTICS PER PROJECT

The JDime conflict statistic box plots for each project are presented in this section.

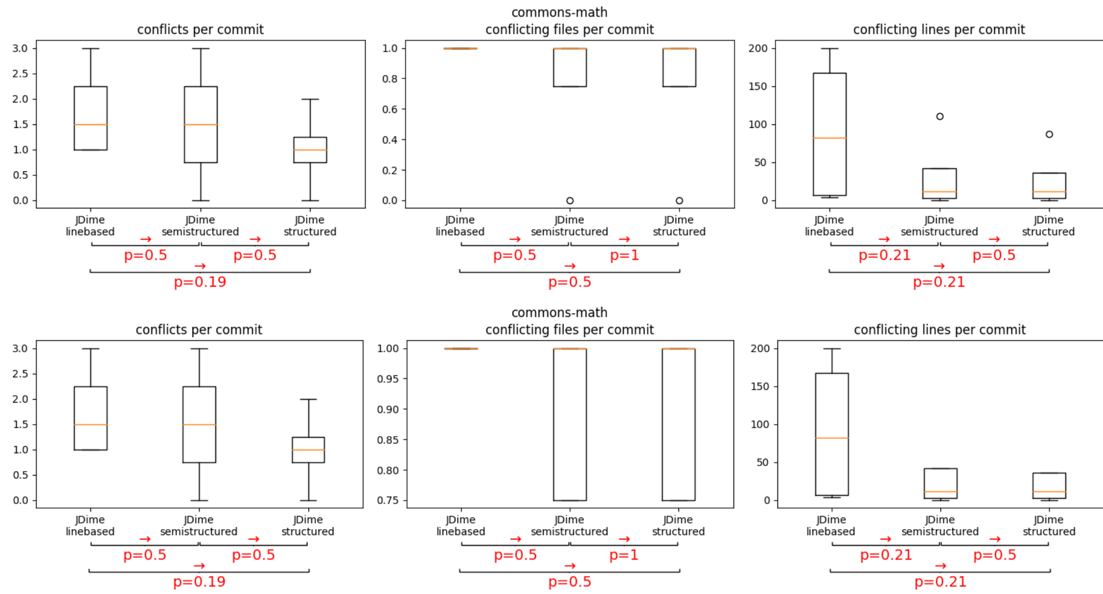


Figure 26: The JDime merge statistics for project commons-math.

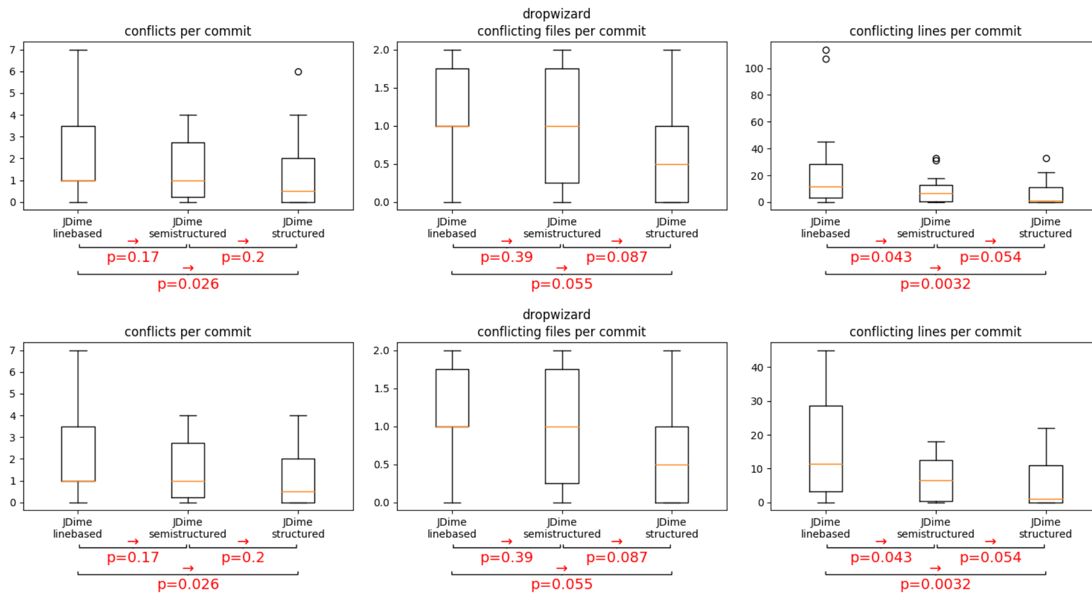


Figure 27: The JDime merge statistics for project dropwizard.

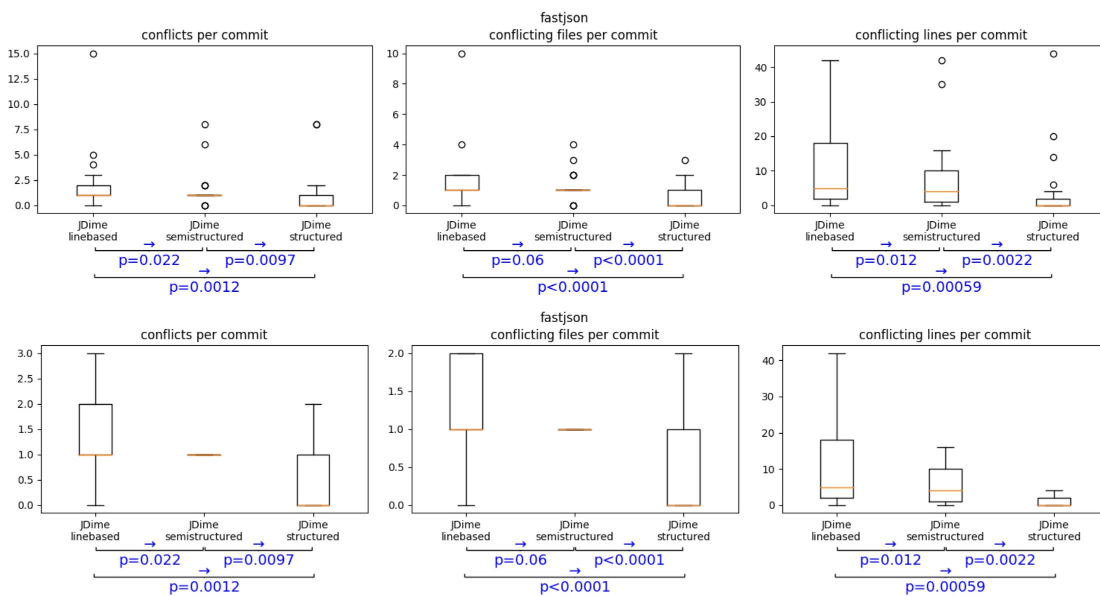


Figure 28: The JDime merge statistics for project fastjson.

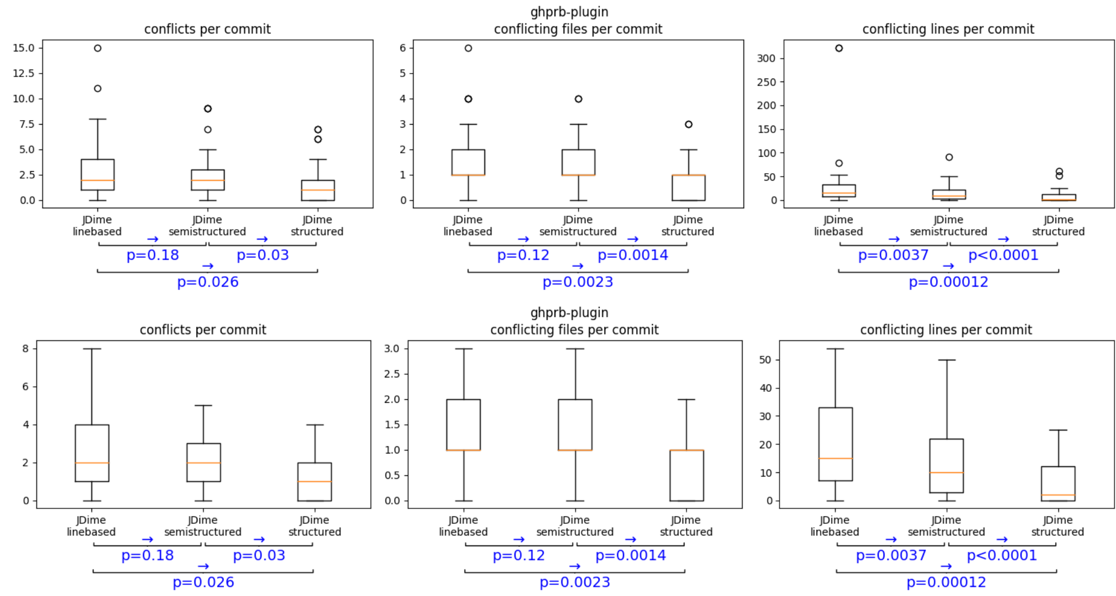


Figure 29: The JDime merge statistics for project ghprb-plugin.

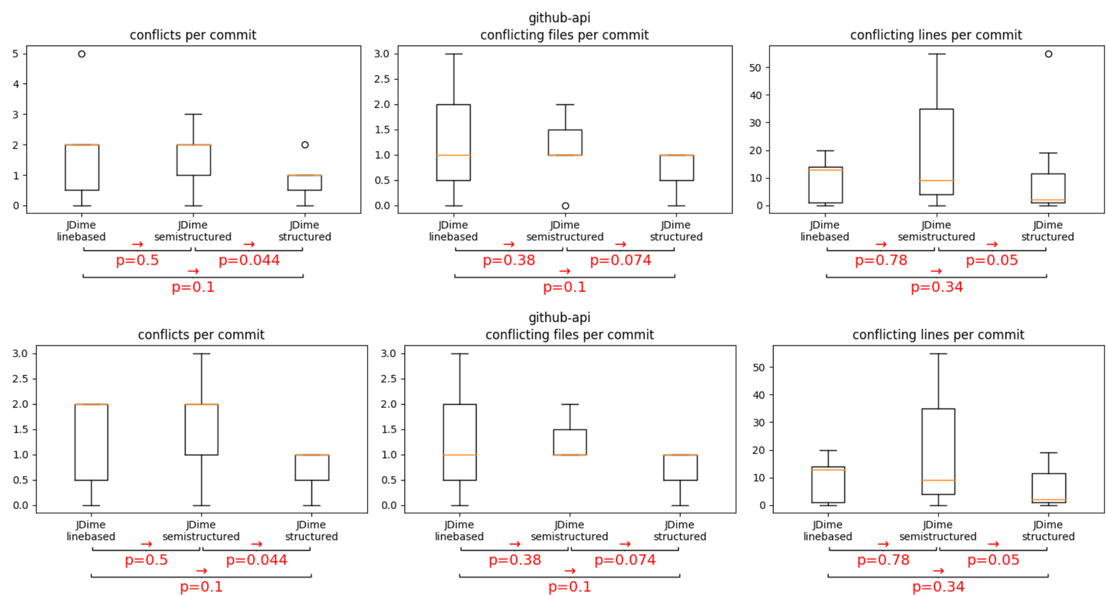


Figure 30: The JDime merge statistics for project github-api.

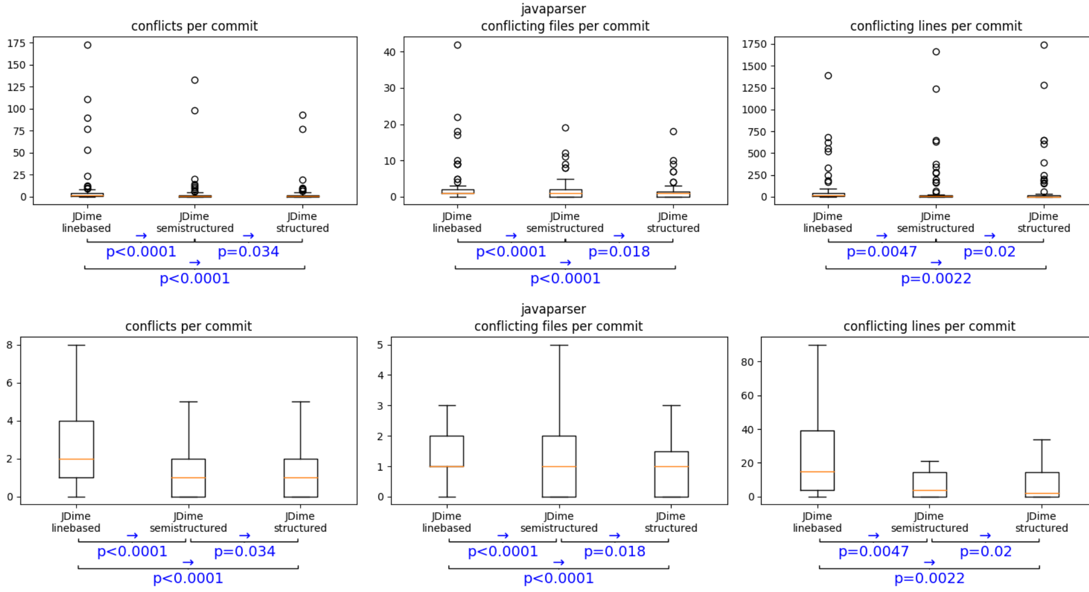


Figure 31: The JDime merge statistics for project javaparser.

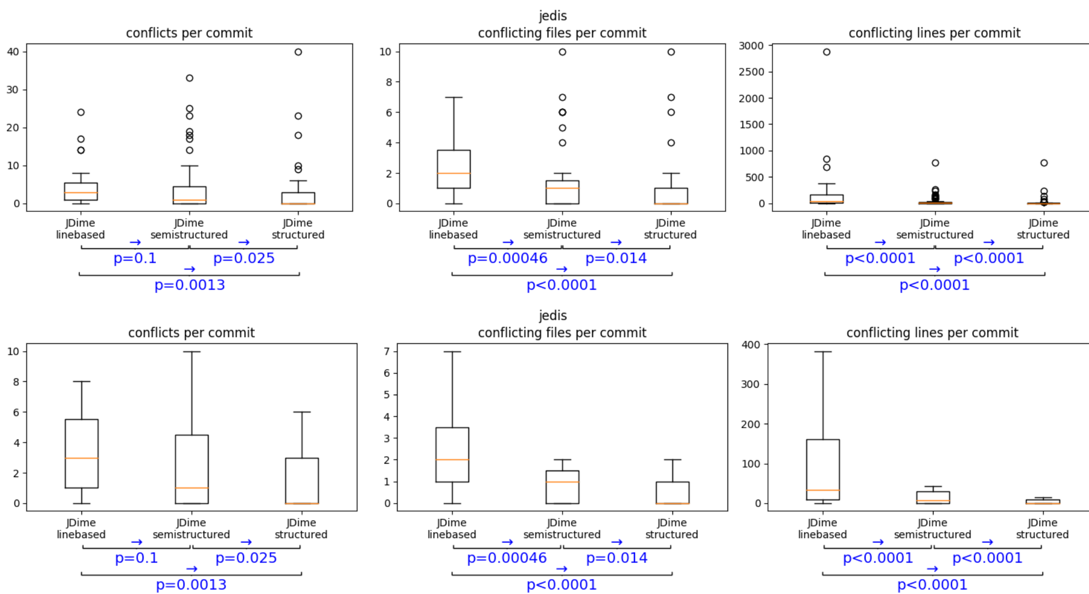


Figure 32: The JDime merge statistics for project jedis.

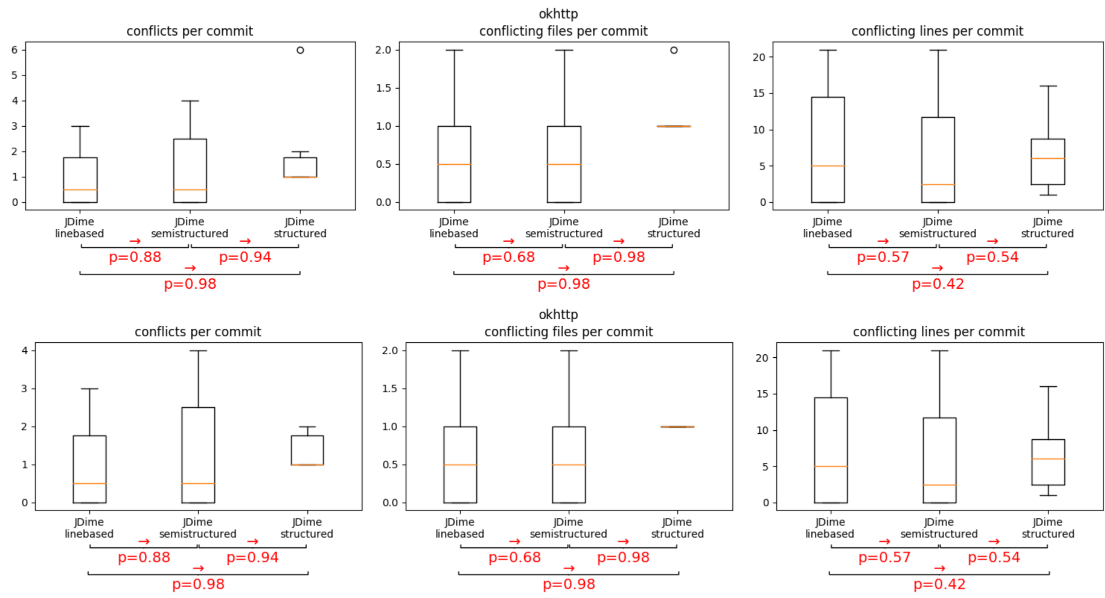


Figure 33: The JDime merge statistics for project okhttp.

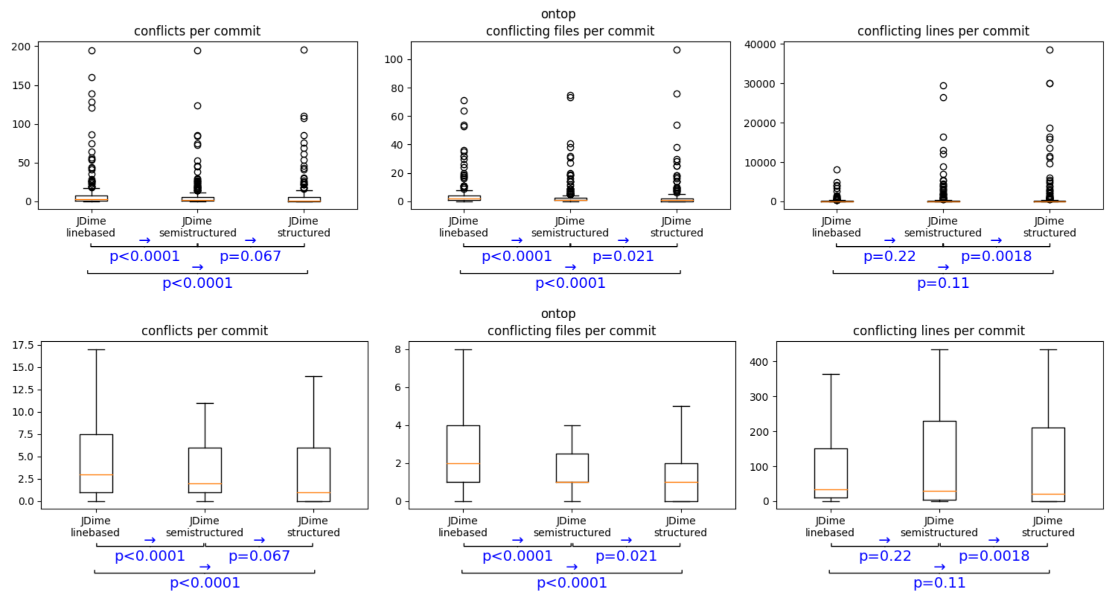


Figure 34: The JDime merge statistics for project ontop.

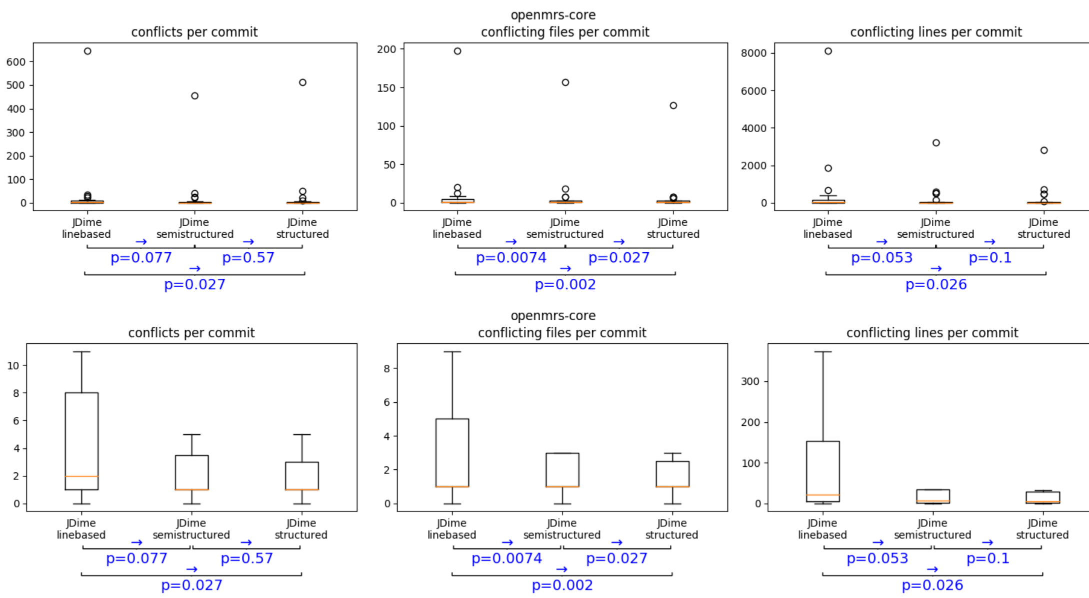


Figure 35: The JDime merge statistics for project openmrs-core.

BIBLIOGRAPHY

[1] Sven Apel, Christian Kästner, and Christian Lengauer. "FeatureHouse: Language-independent, automated software composition." In: *2009 IEEE 31st International Conference on Software Engineering*. 2009, pp. 221–231. DOI: 10.1109/ICSE.2009.5070523.

[2] Sven Apel, Olaf Leßenich, and Christian Lengauer. "Structured merge with auto-tuning: balancing precision and performance." In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 2012, pp. 120–129. DOI: 10.1145/2351676.2351694.

[3] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kastner. "Semistructured Merge: Rethinking Merge in Revision Control Systems." In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. 2011, pp. 190–200. DOI: 10.1145/2025113.2025141.

[4] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. "A differencing algorithm for object-oriented programs." In: *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*. 2004, pp. 2–13. DOI: 10.1109/ASE.2004.1342719.

[5] Boris Beizer. *Software Testing Techniques (2Nd Ed.)* Van Nostrand Reinhold Co., 1990. ISBN: 0-442-20672-0.

[6] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. "Proactive detection of collaboration conflicts." In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. 2011, pp. 168–178. DOI: 10.1145/2025113.2025139.

[7] Jim Buffenbarger. "Syntactic software merging." In: *Selected Papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management*. 1995, pp. 153–172. DOI: 10.1007/3-540-60578-9_14.

[8] Gordon Fraser and Andrea Arcuri. "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software." In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. 2011, pp. 416–419. DOI: 10.1145/2025113.2025179.

[9] Mário Luís Guimarães and António Rito Silva. "Improving early detection of software merge conflicts." In: *Proceedings of the 34th International Conference on Software Engineering*. 2012, pp. 342–352. DOI: 10.1109/ICSE.2012.6227180.

[10] Daniel Jackson and Davida Ladd. "Semantic Diff : A Tool for Summarizing the Effects of Modifications." In: *Proceedings 1994 International Conference on Software Maintenance*. 1994, pp. 243–252. DOI: 10.1109/ICSM.1994.336770.

[11] Olaf Lesenich, Sven Apel, Christian Kastner, Georg Seibt, and Janet Siegmund. "Renaming and shifted code in structured merging: Looking ahead for precision and performance." In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. 2017, pp. 543–553. DOI: 10.1109/ASE.2017.8115665.

[12] Olaf Leßenich, Sven Apel, and Christian Lengauer. "Balancing precision and performance in structured merge." In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 2015, pp. 367–397. DOI: 10.1007/s10515-014-0151-5.

[13] Qingzhou Luo, Lamyaa Eloussi, Farah Hariri, Darko Marinov, and A Motivation. "Can We Trust Test Outcomes?" 2014.

[14] Tom Mens. "A state-of-the-art survey on software merging." In: *IEEE Transactions on Software Engineering* 28.5 (2002), pp. 449–462. DOI: 10.1109/TSE.2002.1000449.

[15] Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, 2006. ISBN: 0131495054.

[16] Bryan O'Sullivan. "Making Sense of Revision-control Systems." In: *Communications of the ACM* 52.9 (2009), pp. 56–62. DOI: 10.1145/1594204.1595636.

[17] Bernhard Westfechtel. "Structure-oriented merging of revisions of software documents." In: *Proceedings of the 3rd International Workshop on Software Configuration Management*. 2004, pp. 68–79. DOI: 10.1145/111062.111071.

[18] Frank Wilcoxon. "Individual comparisons of grouped data by ranking methods." In: *Journal of economic entomology* 39.6 (1946), pp. 269–270. DOI: 10.1093/jee/39.2.269.

[19] Wuu Yang. "Identifying syntactic differences between two programs." In: *Software: Practice and Experience* 21.7 (1991), pp. 739–755. DOI: 10.1002/spe.4380210706.

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und nicht veröffentlicht.

Passau, 2019-09-11

Florian Heck