Faculty of Mathematics and Computer Science

Master Thesis

Adjustable Syntactic Merge of Java Programs

Olaf Leßenich

February 2012

Advisors: Dr.-Ing. Sven Apel
Prof. Christian Lengauer Ph.D.

Abstract

Merging software artifacts while keeping the amount of conflicts at minimum is a difficult task. Several theoretical approaches addressing this problem exist, but due to the lack of implementations applicable in practice, the standard so far is performing line-based, textual merges.

The first part of this thesis introduces the topic of merging and one of its main application areas, version control systems. Afterwards, the focus is set on merging techniques, outlining weaknesses and strengths of the respective approaches.

In the second part, JDime is presented, an adjustable, syntactic merge tool for Java programs. The internal algorithms and the implementation of the tool are explained.

The last part contains an experimental study to evaluate performance and running time of JDime, compared to two other merge engines. Thereafter, the results are analyzed and discussed.

Contents

CHAPTER 1

Introduction

The demand for intelligent merging processes is present in a variety of application areas today. Among other data, we can already store our calendars, contacts, and bookmarks to a cloud and access it from multiple clients, which have to be kept in synchronization. With tablets, netbooks, and other mobile devices entering the market, and smartphones replacing conventional mobile phones, the amount of clients per user also increases. While the data is stored in databases on the server-side, clients often still use plain text files, largely XML. As it is obviously considered unacceptable to confront a smartphone user with a prompt to merge his XML-based calendar by hand, multiple approaches and tools for structured merging of XML-fragments have been developed [Lin03, Lin04, AKJK+02].

Although, other application areas are still behind, one of them being version control systems, which have become an essential tool in today's software engineering. Such systems enable developers spread around the world to work concurrently on a project. Instead of setting exclusive locks, merging is used in state-of-the-art revision control systems. In particular, many Open Source projects follow the principle of frequent integration, which leads to small but numerous merge operations. Also, the creation of many development branches or forks is very common in Open Source software engineering. Integrating branches or forks back into the mainline is a difficult, time-consuming, and often error-prone task. However, whereas developers nowadays are supported with several automated assistance features by their compilers and IDEs while programming or debugging, the assistance while performing merge operations is still poor. This is due

to the fact that only line-based, textual merge engines are used in version control so far. Those can process all files available in plain text very fast, but are rather weak at conflict resolution, as they use no structural information about the files being processed.

Previous work has shown that the use of structured information also improves the merging of source code, because differences can be identified with higher precision compared to textual techniques, which leads to fewer conflicts [Men02, Buf95, Wes91, ALL+]. Although, only few tools are available that merge source code of programs using structured information. Furthermore, the existing ones are mostly bound to specific IDEs. A main reason is that for each language, a specific tool has to be implemented. But even for popular programming languages, such as Java, C++, Perl, and Python, almost no tools applicable in practice can be found. Another reason is the rather high complexity of the algorithms needed to identify and merge structural changes, which results in runtimes being a lot higher than compared to textual merges. Chapters 2 and 3 provide background information about version control systems and merging approaches.

As part of the thesis, a structured merge tool for Java programs, JDime, has been developed. It identifies differences between programs by applying tree-matching algorithms to abstract syntax trees, and implements rules for a standard three-way merge. To make the tool applicable in practice, an optional auto-tuning approach that switches between structured and unstructured techniques is used to offer a trade-off between speed and precision. Chapter 4 explains the implementation of JDime and explains the underlying algorithms in detail.

An experimental study has been performed to compare JDime with other merge tools using eight real-world Java projects. The results and a respective discussion is presented in Chapter 5.

CHAPTER 2

Version Control

This chapter gives an overview of version control systems and their evolution. With them being one of the main application areas for merging software artifacts, the basic functionality and the main operations of such systems will be explained. Differences concerning architecture and design of different systems will be discussed, in particular regarding their impact on merging revisions. Scenarios requiring the execution of a merge will be shown, as well as how conflicts arise in this context.

2.1. Version Control Systems

In general, version control (also called revision control) is a way to maintain multiple versions of files. A naive approach to do version control by hand would be to save each change to a new file, whose filename includes a consecutive number [OB09]. A version control system (VCS) is a tool or a set of tools to automate this process and track the history of a file [OB09].

Indeed, early version control software, such as SCSS[1] (1972) and RCS[2] (1982) simply stored versions locally in the filesystem and were intended to be used by a single person

[1] `http://sccs.berlios.de/`
[2] `http://www.gnu.org/software/rcs/`

managing single files. There is a set of basic operations provided by such a system: `Check out` is used to copy a revision to a local directory, `check in` or `commit` propagates local changes to the VCS and `log` prints the history of a file.

A few years later, the need to work with multiple users on projects consisting of many files increased. To cope with the new requirements, in 1986 the Concurrent Versions System[3] (CVS) was developed, which is still one of the most used version control systems at this time. CVS is build on a central client-server architecture (see Section 2.2), which allows to store many files in a repository. The most important new operation was `update`, which synchronizes the local copy of a repository with the latest revision on the server and retrieves all newer files that have not been modified locally. Furthermore, it was now possible to `tag` a revision, e.g., to mark it as a release candidate or as a milestone. CVS also introduced the concept of branching, which allows to develop in parallel. A branch is an additional stream parallel to the mainline within a project, which can be developed separately. It is also possible, but in practice often difficult, to merge a branch again into another branch or the mainline. The concept of branching and tagging is shown in Figure 2.1.

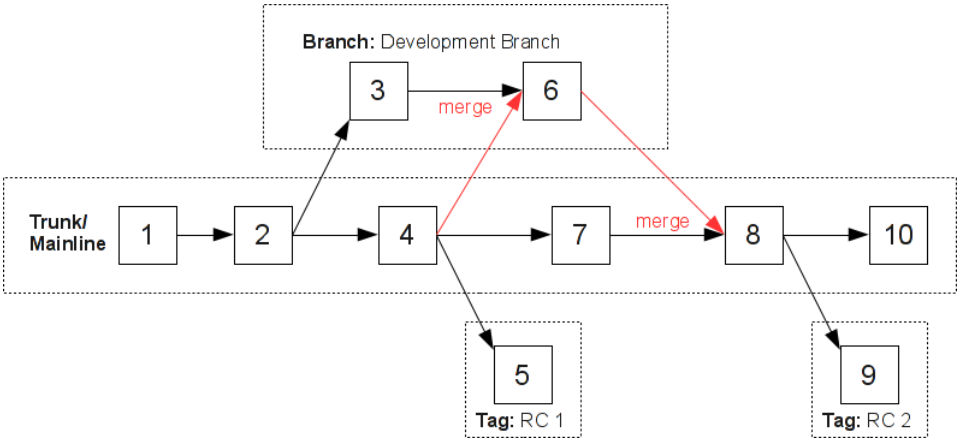


Figure 2.1.: Branching and Tagging in Version Control

Today, version control is an essential instrument for working on a project with several developers. Using a VCS, a user cannot only rollback to a former version, but also query the log to find out what change was introduced when and by whom.

Although the locally storing, single-file approach as implemented by RCS, is still used today by several applications, for example, to manage configuration files. Speaking of up-to-date version control software, we have to distinguish between centralized and distributed approaches.

[3] `http://savannah.nongnu.org/projects/cvs`

2.2. Centralized Version Control

A centralized VCS relies on a client-server architecture. The project data and its history are stored in a single repository on the server, but clients do not work directly on the server's data. If a client wants to change a document, it checks out the project and works on the local copy. When it has finished, it commits the version to the server again.

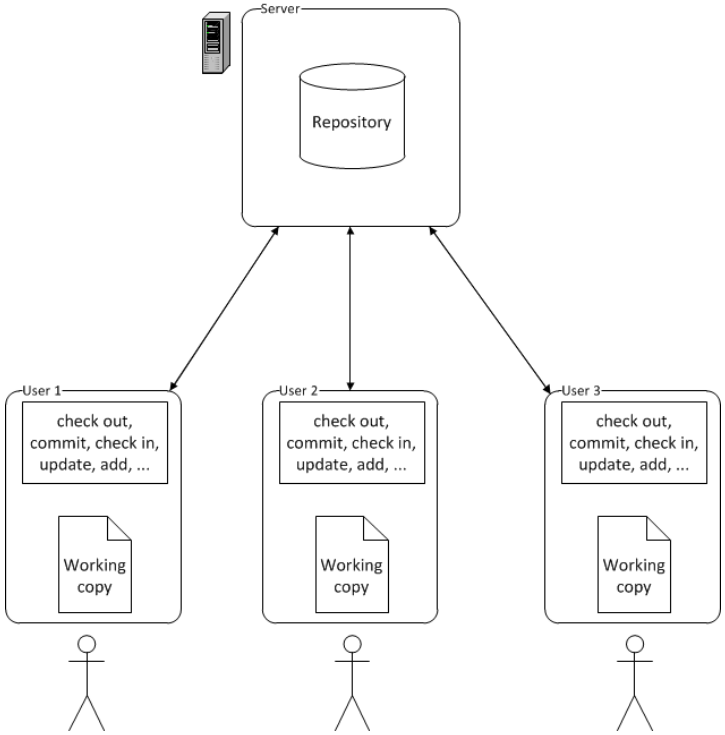


Figure 2.2.: Centralized Version Control System

An advantage of a centralized approach is that access control can be easily implemented, since every client has to connect to the server in order to request a file or commit a change. Furthermore, the amount of data stored at the client side stays minimal, because only one revision is kept in the local working copy. This can be significant when dealing with often changing, large binary data. However, each request concerning the history of a document requires a connection to the server, which is also the single point of failure because it is the only place where the project history is stored.

Popular centralized systems are CVS and Subversion[4], of which the latter was created in the year 2000 and designed to be the successor of CVS.

[4] `http://subversion.apache.org/`

2.3. Distributed Version Control

A distributed VCS relies on a peer-to-peer architecture. The main idea is, instead of fetching and committing data to a single server, each peer holds its own repository, including project data and history, and synchronizes on demand with repositories maintained by other peers.

The operations differ a little in distributed systems: To replicate a repository, the `clone` command is used. It retrieves a lot more information than the `check out` command in centralized systems does, since it also fetches the project history, all metadata and all deltas between the historic versions [OB09][Loe09]. After cloning a repository, a user is working on his own branch. To keep track of the original repository, the `pull` command is used, which again synchronizes the complete project including metadata. The equivalent for the `check in` operation is `push`, which pushes changes from the local branch to a remote branch. However, it is more common to send a pull request to the other peer, so that the remote peer can pull the changes and perform the merge.

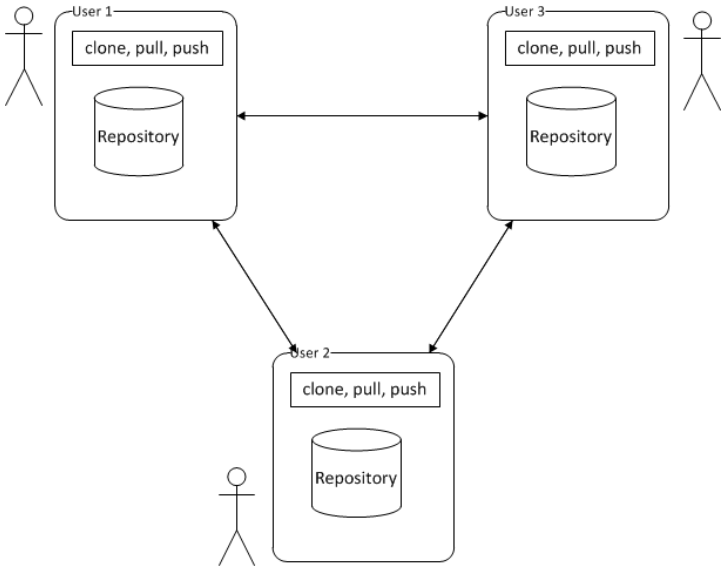


Figure 2.3.: Distributed Version Control System

The main advantage of distributed version control systems is that network connection is not necessary for querying the project history, committing changes or rolling back to older versions. Furthermore each peer has a full backup of both project data and history and can easily confirm the integrity of this data by comparing hash values with other peers. The downside is that the amount of data stored locally is rather large. Also, it is not trivial to implement access control for distributed systems, since there is nothing like a central place to define global access rules.

Popular distributed version control systems are GIT[5] and MERCURIAL[6], both developed in 2005.

2.4. Merge Scenarios and Conflicts

This section addresses the problems that can arise in version control when working with two or more users on the same resources. Several scenarios will be explained that lead to an inconsistent system or to a merge process that might result in a conflict.

Lost Update Problem

A main issue of parallel development is the lost update problem, also known as blind overwriting [CSFP10]. Figure 2.4 shows what happens if a system offers no mechanism to prevent the lost update problem. The changes in revision 2 committed by user A are lost when user B commits his changes, which become revision 3. If user A has also committed changes to other files that were not overwritten but interact with the overwritten changes, the system is in an inconsistent state. This problem is also known in the context of databases.

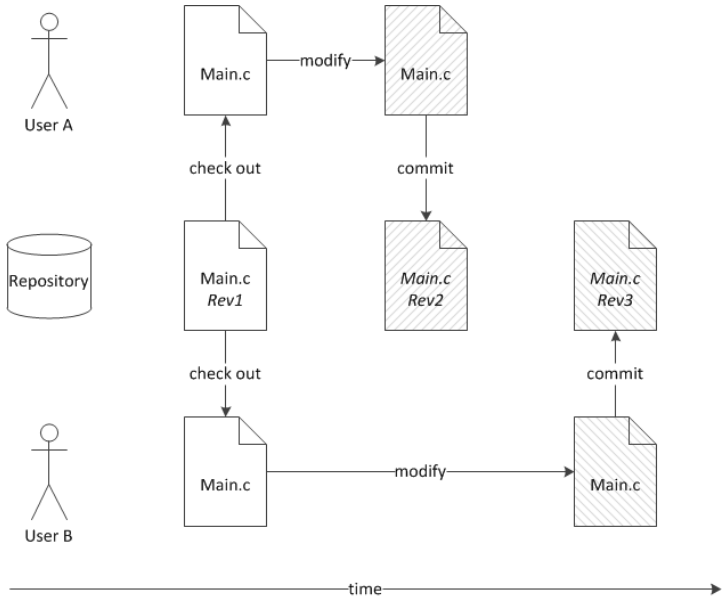


Figure 2.4.: Lost Update Problem/Blind Overwriting

[5] http://git-scm.com/
[6] http://mercurial.selenic.com/

The Lock-Modify-Write Solution

A safe way to prevent other users from checking in a newer version of a specific file while one is working on it, is locking the relevant file [CSFP10]. This mechanism ensures that only the person owning the lock can commit the file to the repository. In practice, this approach causes several problems, for example, developers forgetting to unlock a file or developers locking large parts of a project, both causing development to stall. Furthermore, using exclusive locking, it is not possible to work with two people on the same file, even if it is clear that no overlapping will occur. Imagine, for instance, a text document in which one person wants to edit the introduction and the other the conclusion. Only the person owning the lock on the file is able to work. Figure 2.5 demonstrates, how the locking mechanism solves the lost update problem.

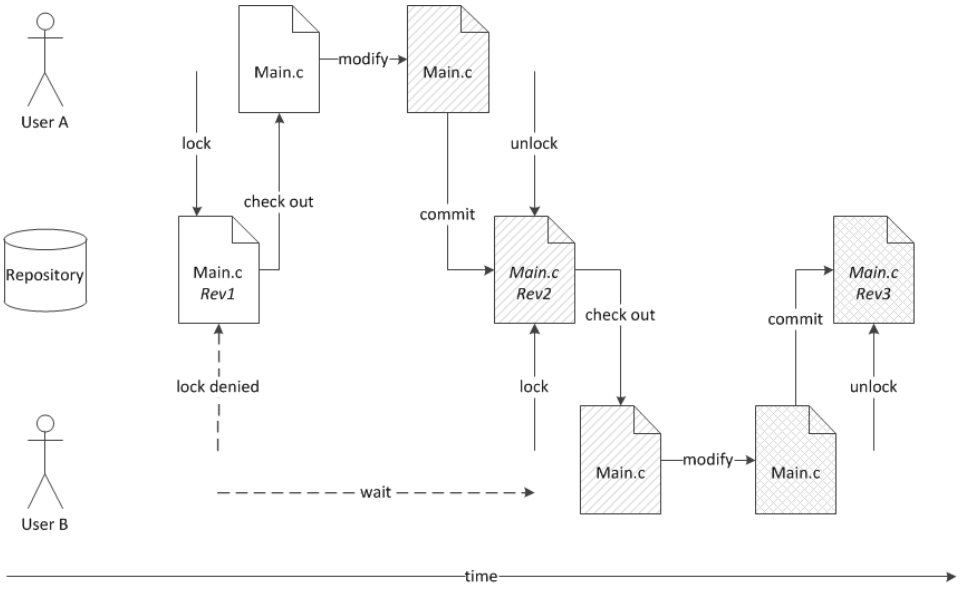


Figure 2.5.: Lock-Modify-Write Solution

The Copy-Modify-Merge Solution

To prevent unintended overwriting without enforcing the use of lock files, most modern version control systems accept a commit of a modified file to the repository only if its original local copy matches the latest revision available in the repository [CSFP10]. If somebody else has committed a newer version of the file to the repository in the meantime, one has to merge this newer file with the locally changed file before a commit. So, instead of overwriting or exclusive locking, the state-of-the-art procedure in version control is merging, although locking is supported as an additional instrument by most systems.

Figure 2.6 illustrates, how merging solves the lost update problem.

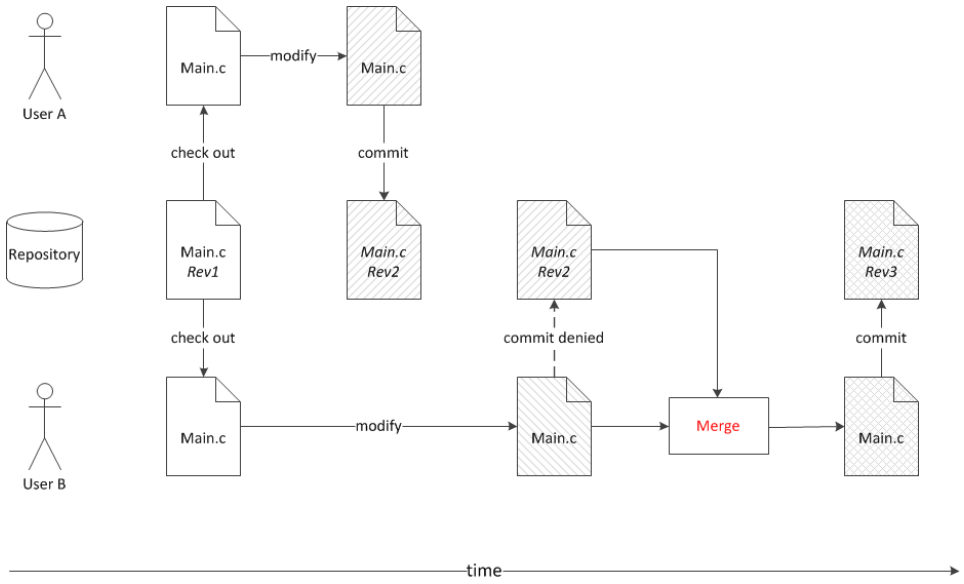


Figure 2.6.: Copy-Modify-Merge Solution

Note that this illustration is a little bit simplified, as the merge process shown takes two files as input and one file as output. As will be explained in Section 3.2, usually the common ancestor of both files (which would be revision 1 in the example) is also taken into account for the merge. Concerning the merge process, if there are non-overlapping changes performed by two users, all the changes can be integrated automatically. Otherwise, a conflict is generated. In version control, a conflict is an undesirable state, which occurs if two users changed the same part of a file and that requires manual resolution. In the documentation of many version control systems a phrase like the following can be found:

> "[The user will] be able to see both sets of conflicting changes and manually choose between them. Note that software can't automatically resolve conflicts; only humans are capable of understanding and making the necessary intelligent choices." [CSFP10]

Although the second sentence describes the current situation properly, the first sentence might give the impression that all the user has to do is choose between two alternatives presented to him by the client. Unfortunately, this is not true in many cases, where both alternatives fail. Then, the user has to merge both files manually to create a consistent version containing both changes.

Merging Two Branches

Another scenario requiring a merge occurs when two branches are being rejoined into a single branch or when changes of one branch are applied to the other. If both branches altered the same file, both files have to be merged in order to merge the branches. The more the development of the branches diverted, the more difficult a merge will be and the more conflicts will be created.

CHAPTER 3

Merging Software Artifacts

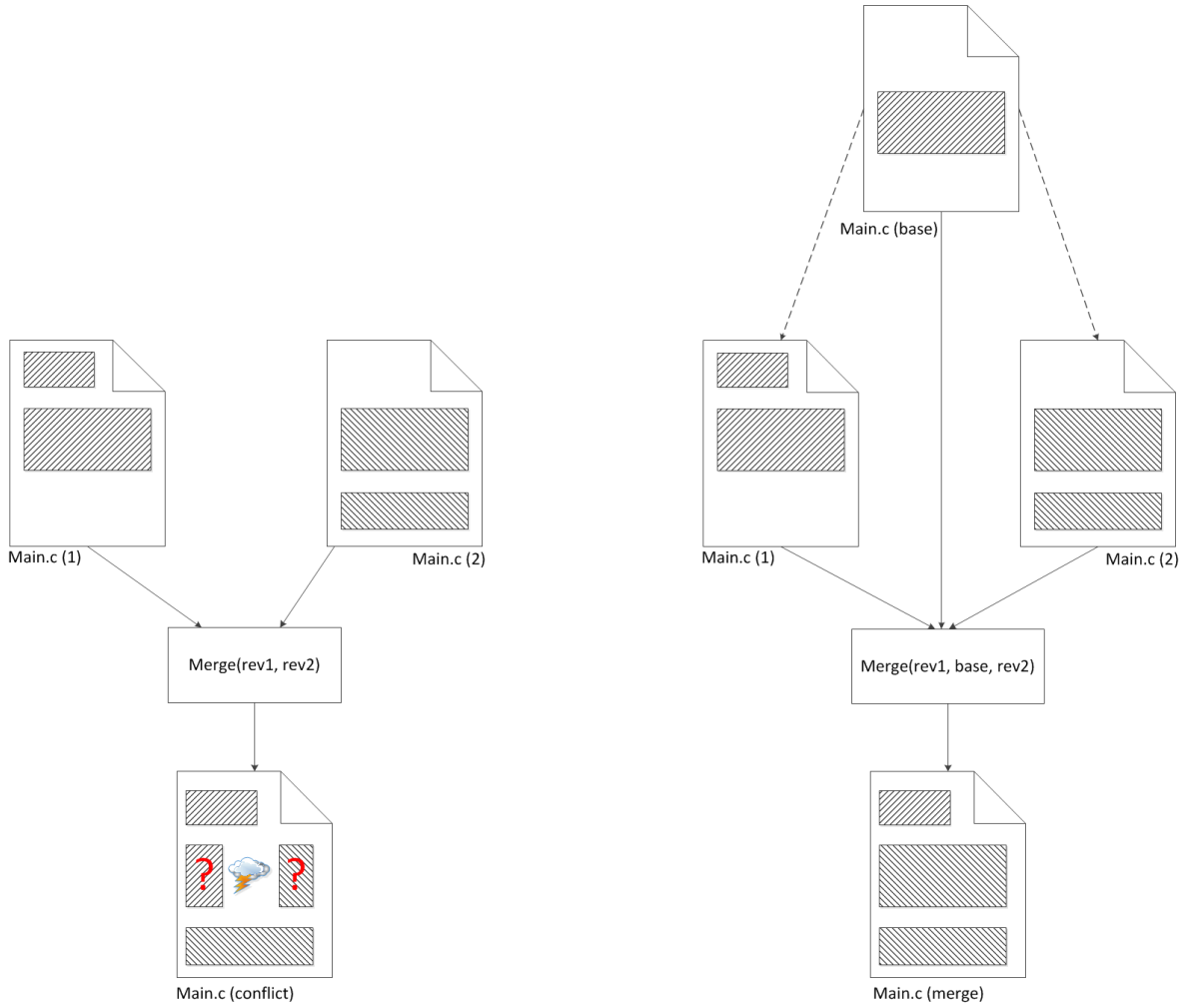
As shown in the previous chapter, there are several scenarios in version control that require merging. This chapter introduces, classifies and discusses the most important approaches for comparing and merging software artifacts.

3.1. Identifying Differences

In order to merge multiple artifacts of source code, it is essential to compare them and extract the differences. This separate process is crucial for the quality of the merge. If a program element is mistakenly detected as change by the compare process, whereas it is actually unchanged in both versions, this might lead to an unnecessary conflict while assembling the unified output in a later step of the merge. The implementation of the this process heavily depends on the underlying data structure, as we will see in the Sections 3.4 to 3.7.

3.2. Two-Way and Three-Way Merges

In the context of version control, a distinction has to be made between two-way merges and three-way merges. A two-way merge attempts to merge two revisions directly by comparing two files without using any other information from the VCS. Each difference between the two revisions leads to a conflict since it cannot be decided whether only one of the revisions introduced a change to the code or both. It also cannot be determined whether a certain program element has been created by one revision or has been deleted by the other one.



(a) Two-way merge

(b) Three-way merge

Figure 3.1.: Two-way and three-way merge

A three-way merge takes the common ancestor of both revisions into account and thus has more information at its hands to decide where a change came from and whether it creates a conflict or not [Men02]. The difference is illustrated in Figure 3.1.

In order to perform a three-way merge, the basic rules are given in Table 3.1 [Wes91]. Note, that due to the rules being symmetric, the table displays them in a shortened form.

	Type of i	Left Rev A1	Right Rev A2	Base-Rev B	Merge M
1	Element	i_B	i_{A2}	i_B	i_{A2}
2	Element	i_{A1}	i_{A2}	i_B	Conflict
3	List	$i_1 \in i_{A1}$	$i_1 \notin i_{A2}$	$i_1 \in i_B$	$i_1 \notin i_M$
4	List	$i_1 \in i_{A1}$	$i_1 \notin i_{A2}$	$i_1 \notin i_B$	$i_1 \in i_M \vee$ Conflict

Table 3.1.: Three-way merge rules

An element modified in exactly one of the opposing revisions is included into the merged program (row 1). In case it is modified in both revisions in a diverging way (row 2), a conflict is created, which has to be resolved manually. If an element contained in the base revision is missing in exactly one of the opposing revisions (row 3), it is considered as deleted and will therefore not be included in the merged output program. New items appearing in only one revision are inserted into the merged program (row 4). In the last case, the position at which the element has to be inserted is significant and has to be taken care of by the merge algorithm. If the insert position is ambiguous, a conflict has to be reported as well.

3.3. Cut-and-Paste Merging

The most simple and primitive way of merging is cut-and-paste merging. This kind of merge is performed without automatic assistance and requires the user to manually compare the source code and copy or delete relevant parts in order to resolve all conflicts and retrieve a merged file.

Today, this method is usually only necessary for single files which failed to be merged automatically. Since this thesis focuses on tool-assisted merges, this type of merge is not further discussed here.

3.4. Textual Merge

Chapter 2 has provided an overview of the history of version control systems. While these systems evolved over the years in several stages to cope with increasing demands, the tools that actually perform the merges have not evolved that much. When it comes to merges in version control, the state of the art is performing a textual, line-based

merge. A popular example for textual merge is GNU MERGE, which was developed to perform a three-way merge. GNU MERGE works in the same way as RCSMERGE, which was released as part of RCS in 1982 and is today part of the GNU project. Merge tools used by version control systems work and behave in a similar way as GNU MERGE does. In the following paragraphs, we will show, how the merge works, how it is used, and how its output presents conflicts to the user.

To compare two files, MERGE calls another well-known program: GNU DIFF. DIFF compares two files line by line and detects the smallest sets of differing lines. To compare three files, GNU MERGE calls DIFF3, which works in a similar way but applies the rules of a three-way merge. The syntax of GNU MERGE is `merge file1 file2 file3`, in which `file2` is the common ancestor of `file1` and `file3`.

The behavior of GNU MERGE is rather unusual compared to other Unix tools, since its output is written directly into `file1`, unless the command line switch `-p` is specified, which redirects the command's output to `stdout`. If no conflict occurred during the merge process, the output equals the successfully merged file. In the case a conflict was detected, it is displayed at the relevant place in the output as shown in Figure 3.2.

```
<<<<<<< file1
conflicting lines in file1
=======
conflicting files in file3
>>>>>>> file3
```

Figure 3.2.: GNU MERGE displaying a conflict

As mentioned in previous sections, it is possible that none of both alternatives leads to the desired solution. Such an example is shown in Figure 3.3. Choosing the first alternative would cause the output to be incompilable due to the duplicated `fac()`-method, choosing the second one would end up in losing the `square()`-method introduced by revision 1.

The benefits of a textual merge like the one implemented by GNU MERGE, are its generality and its performance. It can be applied to all non-binary files, even to very large ones, so there is only one tool needed regardless of which programming languages are used within a project. If the amount of changes is very small in comparison to the input files, or if there are no changes at all, this method is very effective.

But since this type of merge does not utilize knowledge about the structure of the input documents and the syntax of respective languages, it might miss conflicts, detect too many conflicts and is likely to produce syntactically incorrect output. In 1989, Horwitz [HPR89] presented several examples where textual merging fails and stated:

> "Integrating programs via textual comparison and merging operations is ac-

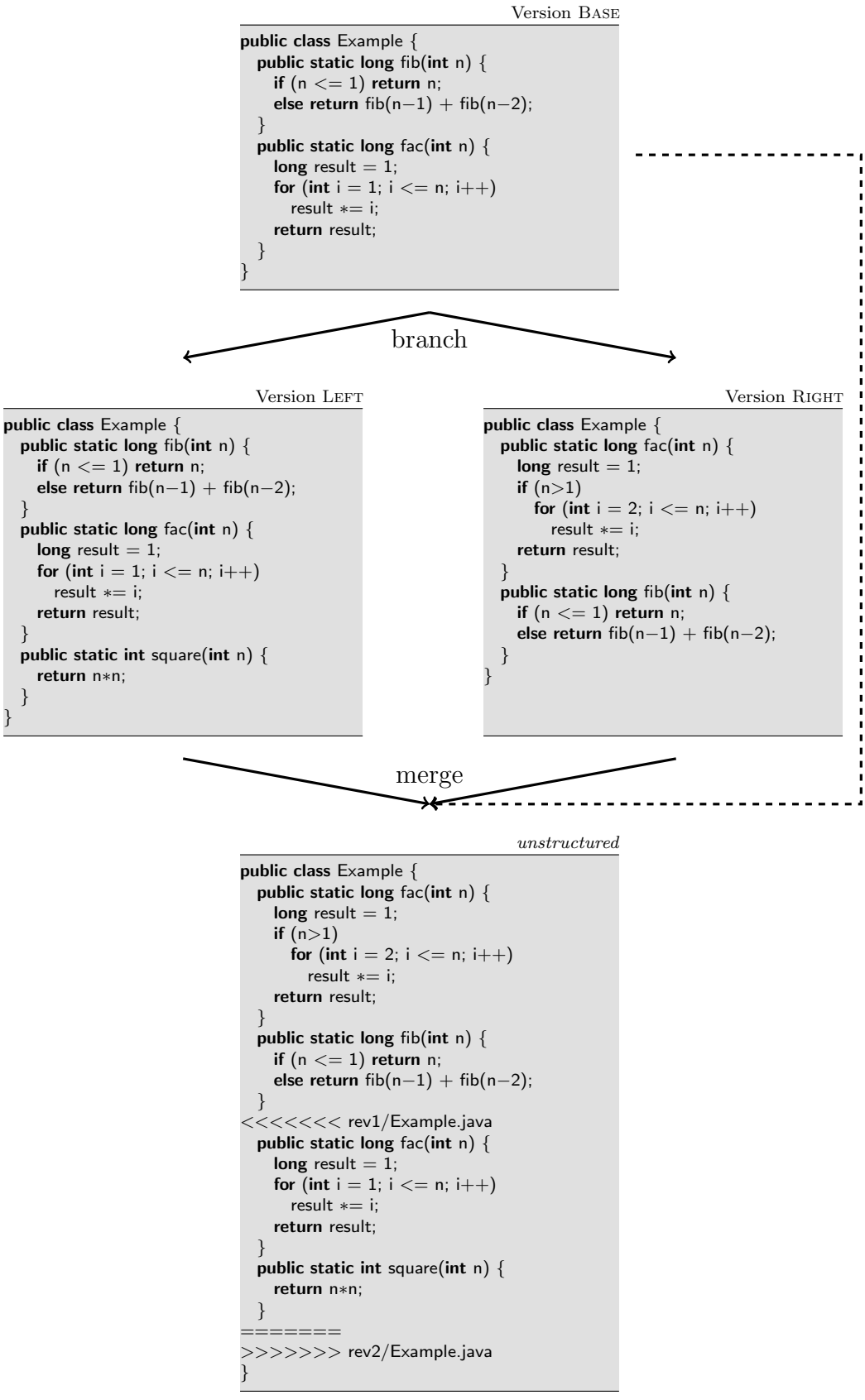


Figure 3.3.: Failed textual merge

companied by numerous hazards." [HPR89]

Additionally, as Buffenbarger [Buf95] shows, textual merges might produce uncompilable output from compilable input without detecting conflicts.

In practice, one of the main weaknesses of textual merges is their inability to detect reordered declarations [ALL+]. In Java, a change in the order of methods and fields has no semantic impact on the program behavior, but using a line-based algorithm to merge such changes will lead to a lot of conflicts which have to be manually resolved in order to complete the merge. As discussed later in the experimental study, reorderings occur very often in project.

Furthermore, (re)formatting of code produces conflicts when textual merge is used, because the position of brackets and the indentation style (e.g., tabs or spaces) might be different according to the settings of the developers' editor. Since reformattings also occur a lot in practice, many unnecessary conflicts are created by textual merges.

As mentioned before, the accuracy of the diff is essential for the merge process. GNU DIFF provides several switches to improve the matching, such as ignoring empty lines or trailing and leading whitespace characters, which would be of great aid when dealing with reformattings. Unfortunately, DIFF3 does not offer such fine-tuning capabilities and therefore they are not available for GNU MERGE either.

After examining the algorithm implemented by DIFF3 in 2007, Khanna et al. [KKP07] do not even consider it as stable, since two runs with similar input do not produce similar output in every case.

3.5. Syntactic Merge

Syntactic merges exploit language-specific knowledge and can therefore compare and merge software artifacts better than textual merges. The underlying data structure for syntactic merges are usually (abstract syntax) trees or graphs, which requires the merge tool to parse the programs in advance and generate the corresponding trees or graphs [Men02].

Comparing two programs for this type of merge means traversing both trees/graphs and finding the differing nodes. Since the merge is applied to the trees or graphs after parsing the program and comparing the trees/graphs, code formatting is no longer relevant. The previously mentioned ordering conflicts can also be detected as such, and in case of Java programs, simply be ignored. When the merge process has finished, the output document is generated by pretty-printing the abstract syntax tree (AST).

However, this approach has disadvantages as well: First of all, syntactic merges are much slower than textual merges, mostly due to the complexity of their compare algorithms, working on trees or graphs. Depending on how accurate the matching is supposed to be, this problem can happen to be $\mathcal{NP}$-complete [ZJ94]. A syntactic merge requires the input files to be syntactically correct, otherwise the parser is not able to build the AST. If an incorrect source file is committed to the repository, a syntactic merge tool is of no use. Furthermore, it is restricted to certain file types since it uses syntactic knowledge of the programming languages in which the programs to be merged are written. Supporting additional programming languages requires a lot of work: A parser must be written and several language-specific rules, like the relevance of the order of class members and miscellaneous other rules, have to be considered for the merge process. Changes in the specification of a language might break the correctness of a merge tool, which then must be subsequently adapted to the specification (e.g. Java 1.4 to 1.5). Another disadvantage is produced from using pretty-printers to generate the output: The original code formatting done by the developers is lost after the merge.

Westfechtel [Wes91] developed a language-independent, structured merge tool based on ASTs, but the price to pay for this generality is that all changes have to be performed within a special development environment. Both Yang [Yan91] and Buffenbarger [Buf95] created syntactic merge tools for C, though they did not address ordering conflicts in their respective implementations.

3.6. Semistructured Merge

Semistructured merge [ALL+] by Apel et al. is an approach to avoid many disadvantages of unstructured merges, while staying more general than syntactic merges. In particular, semistructured merge provides a solution for ordering conflicts, which arise in textual merging. This is achieved by working on simplified parse trees, which represent the program structure. Such parse trees, called *program structure trees* (PST), include some but not all structural information of a program. Concerning Java, for example, classes, methods and fields are contained in a PST, whereas statements and expressions are hidden in the leaves of a tree in form of plain text.

The idea is to use the structured information contained in the PSTs to merge revisions with less conflicts than a textual merge is able to, while not having to deal with the merging of separate statements on the syntax level inside of method bodies. This trade-off allows semistructured merge to support a larger set of programming languages than pure syntactic merges do, and to provide superior conflict resolution in most cases, compared to conventional textual merges. FSTMerge[1] is an implementation of a

[1] `http://fosd.de/SSMerge`

semistructured merge engine build into FeatureHouse[2] and currently supports C#, Java, and Python.

The merge algorithm itself is implemented via superimposition of the PSTs, working recursively and beginning at the root nodes. The principle of superimposition is illustrated in Figure 3.4. To handle the removal of constructs by revisions, the usage of superimposition was complemented to satisfy common three-way merging rules. This approach implies that method bodies only have to be merged if the signature of two methods is identical. In this case, a line-based merge using GNU merge is launched to merge the method bodies [ALL+].

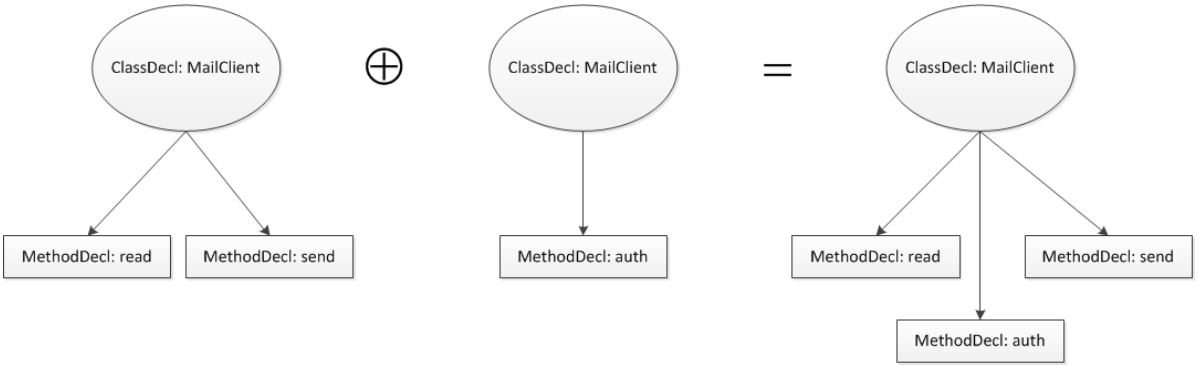


Figure 3.4.: Superimposition

3.7. Semantic Merge

Semantic merging addresses a set of conflicts that is not detectable by syntactic merge tools. An example for such conflicts is shown in Figure 3.5.

The left revision renamed `n` to `a` in the function `compute()` and introduces an independent class member `n`, which might be used elsewhere. The right revision wants to make sure, the argument of `compute()` is positive. After the syntactic merge, which did not produce a conflict, the new class member `n` is unintentionally changed within `compute()`, if it is negative.

The reason why the syntactic merge cannot detect this conflict is that it lacks a link between usage and declaration of variables. To handle these kind of conflicts, more information about the program is needed than is obtainable from its AST.

For his implementation of structured merging, Westfechtel [Wes91] enhanced his ASTs with context sensitive relations in order to provide the mentioned missing link. However,

[2] `http://fosd.de/fh`

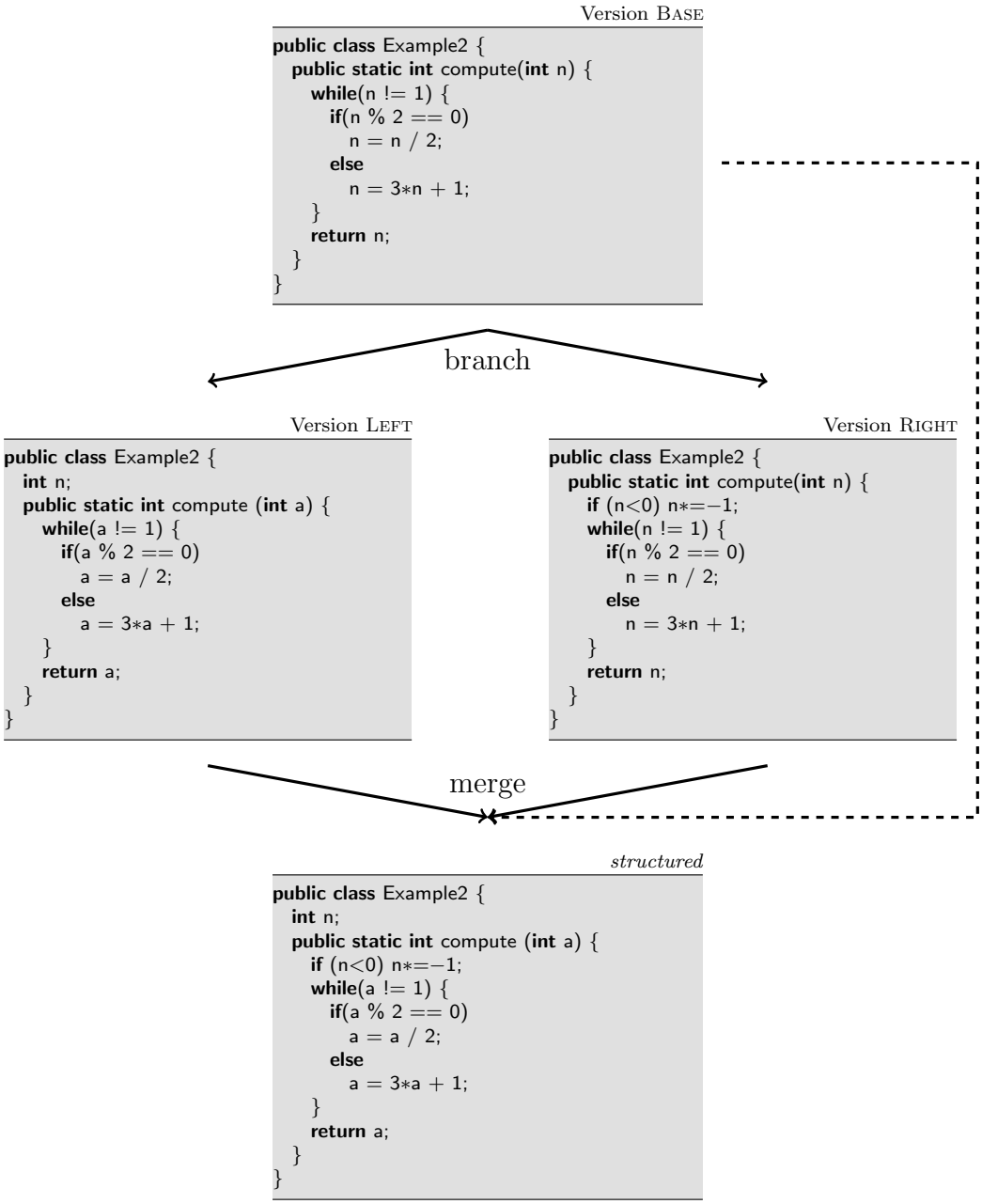


Figure 3.5.: Semantic Conflict

even using an approach like this, static information is not sufficient to accomplish the detection of all semantic conflicts [Men02]. Finding complex behavioral conflicts between programs is an extremely difficult task and the sophisticated algorithms required are very expensive concerning runtime.

3.8. Summary

Merging software artifacts is a difficult process, which has always been an issue within version control. And, despite the evolution of version control systems to facilitate parallel development, in the majority of systems available today, the tools to support users performing merges usually implement a textual three-way merge.

According to Mens [Men02], these tools can process 90% of all changed files automatically without detecting conflicts. With distributed version control systems gaining market share, the need to perform difficult merges of many branches is likely to increase. Replacing the textual merges completely would probably not be applicable in practice due to the heavily increased runtime of structured tools. But focusing on the remaining 10%, more powerful tools could be effectively used here to assist the user in a better way.

With the concept of semistructured merge and its implementation FSTMerge, ordering conflicts have been successfully tackled, and in most cases the merge output is of better quality than the one of conventional merges. However, the overhead brought in by FeatureHouse is quite a bit and noticeable concerning the runtime of a merge process.

In 2002, which was 3 years before popular distributed systems like GIT and Mercurial were available, Mens [Men02] came to the following conclusion:

> "An interesting avenue of research would be to find out how to combine the virtues of different merge techniques. For example, one could combine textual merging with more formal syntactic and semantic approaches in order to detect and resolve merge conflicts up to the level of detail required for each particular situation." [Men02]

One should assume, that the need for more adjustable tools since then has even grown, but still there is no usable implementation for popular programming languages.

CHAPTER 4

JDime: Structured Merge For Java

The goal within this thesis was to create a tool that performs a syntactic three-way merge on Java programs. Its main objectives are adjustability and extensibility. The former means being able to switch between implemented algorithms at runtime in order to speed up the merge process or minimize the number of conflicts, the way Mens [Men02] suggested it. The latter refers to the more or less easy integration of additional algorithms, which allow even more tuning options concerning granularity. Call syntax and visualization of conflicts are designed to be similar to GNU MERGE.

4.1. Basic Decisions

For the implementation of a syntactic merge tool, it seemed obvious to use either trees or graphs as underlying data structure.

There are already a lot of frameworks and tools providing parsers and pretty-printers that could be used for the merge tool. Because extensibility concerning the algorithms and language-specific logic was considered a very important property, the decision was made to use parts of the Java compiler JastAddJ[1].

[1] `http://jastadd.org/web/jastaddj/`

4.2. JastAddJ

JastAddJ is an extensible Java compiler built from the metacompiler framework JastAdd[2]. Its most interesting feature for building a syntactic merge tool on top of it, is its data structure: programs are represented as attributed ASTs, which are accessible via a Java class hierarchy. Each node within the AST is an instance of `ASTNode` or one of its subclasses, whereas its child nodes are themselves instances of `ASTNode` or a respective subclass [EH07].

There are about 370 subclasses corresponding to elements of the Java 1.5 programming language. Each node in this class hierarchy may have specific attributes. For example, a node representing a `while`-loop has the attributes `getCondition()` and `getBody()`, which link to nodes representing condition and body of the loop.

Using this data structure, the merge tool would have to compare and merge nodes of the attributed AST. A lot of information has to be considered in order to detect conflicts while merging, for instance, whether the order of two nodes is important or not for the semantic of the program. Extending the AST classes to provide this information results in a new attribute `isOrdered()`, which returns `true` if the order of a language element is significant, and `false` otherwise.

Adding attributes to the AST classes is fairly easy in JastAddJ: It supports an aspect language similar to AspectJ[3], so all that has to be done is creating an aspect and defining fields and methods that will be woven into the AST classes.

However, there is a downside to this design as well. It is not possible to extend the AST hierarchy in a sane way without using aspects. This practically means, every change done via the implementation of an aspect requires the whole class hierarchy to be regenerated and all aspects to be re-woven into the relevant destination files from scratch, to finally apply the change.

4.3. Design

A significant disadvantage of syntactic merge tools is the complexity of their underlying tree-based algorithms, which affects the runtime in a negative way. To merge two or three files, which might not even have changed at all, a syntactic merge tool has to parse the source code, compare all pairs of trees, and finally assemble and pretty-print the merged

[2] `http://jastadd.org`
[3] `http://www.eclipse.org/aspectj/`

tree. This way, merging two revisions of an entire project becomes a painfully time-consuming task. To overcome this problem, an auto-tuning approach is implemented for JDime: As long as no conflicts are detected, the tool uses an unstructured, textual merge relying on GNU merge, which is cheap in performance and does not require to build the ASTs. Only if a conflict is detected by the conventional merge, JDime switches to the more powerful syntactic merge, which has all of the structural information at its disposal and provides a better conflict resolution in most cases. Also, it is possible, to use the semistructured merge engine FSTMerge instead of the fully structured syntactic merge.

4.3.1. Data Structure

The main data structure used in JDime are trees, which are implemented using objects of the custom type `jdime.Node`. A `Node` object has child nodes, certain attributes, and encapsulates an `ASTNode`-instance provided by JastAddJ. Although, at first glance, this extra class seems to be redundant, it was absolutely necessary due to the behavior of JastAddJ while modifying its `ASTNode`s, as will be described in detail in Section 4.6.

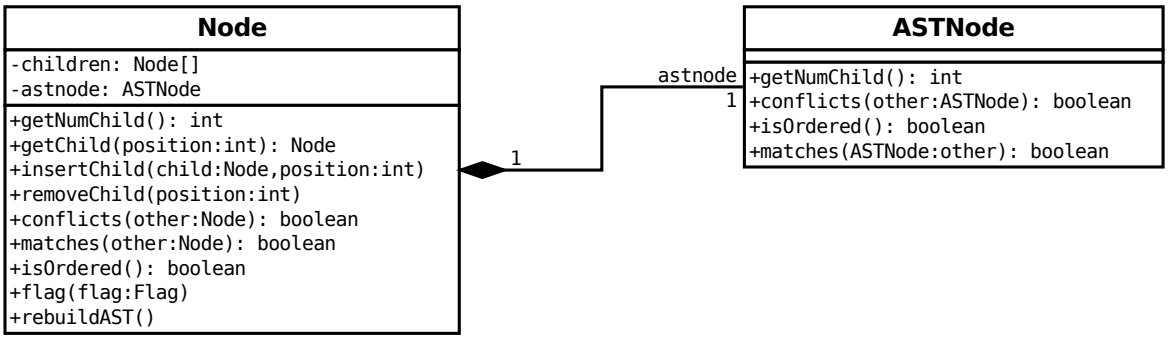


Figure 4.1.: UML: Node

4.3.2. Architecture

A merging algorithm only has to implement the `MergeInterface` in order to be used in JDime. Currently, algorithm classes for syntactic, textual, and semistructured merge exist in JDime, of which the last two are simple wrappers for external tools. Compare algorithms inherit from the `TreeMatcher`-Class, which already handles the flagging of found matchings. The implemented algorithms are explained in Section 4.4. Figure 4.2 illustrates how the algorithms work together.

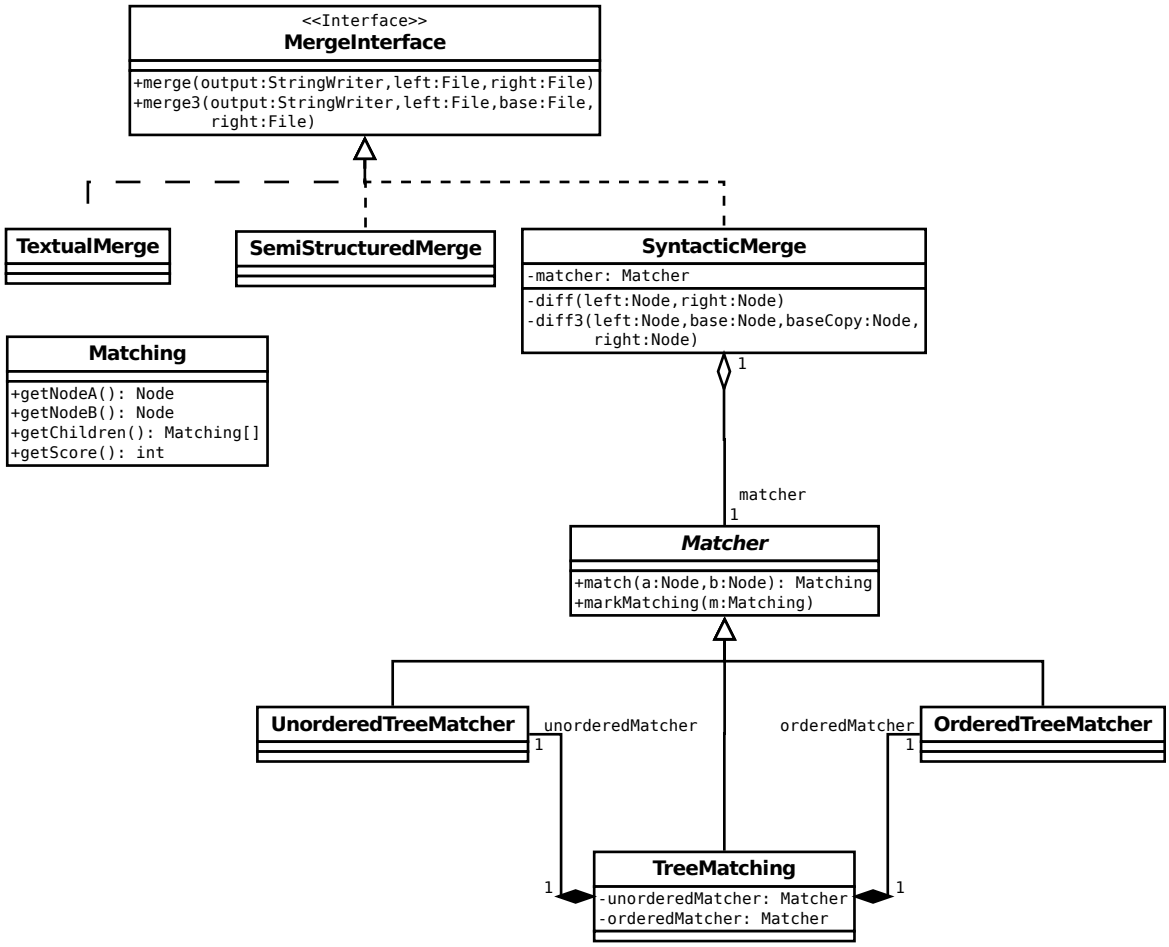


Figure 4.2.: UML: Algorithms

4.4. An Algorithm for Code Comparison

Prior to being able to merge two or three revisions, the actual changes between them have to be identified. Detecting the minimum set of changes is the same problem as detecting the maximum set of unchanged parts within the revisions. The quality of the matching result is crucial for the whole merge process. Missed links at this step will cause trouble while detecting conflicts and performing the actual merge.

When comparing trees, we have to distinguish between ordered and unordered trees. Furthermore, we have to decide, whether only nodes of the same height in the trees are compared, or also nodes being placed on different tree levels. While the former refers to the *Largest Common Subtree*-problem, the latter is known as *Largest Common* ***Embedded*** *Subtree*-problem. Finding a perfect matching in case of unordered trees, which means solving the more difficult problem of finding the largest common embedded subtree, is known to be $\mathcal{NP}$-hard. As Zhang and Jiang have shown, the problem is even hard to approximate for general cases [ZJ94]. Halldórsson and Tanaka provided a further reduction from 3-Set-Packing [HT96]. The *Largest Common Subtree*-problem is solvable in polynomial time for unordered trees. In case of ordered trees, both problems can be solved in $\mathcal{P}$.

To compare Java programs, algorithms to compare ordered and unordered trees are required. The algorithms implemented for JDime within this thesis are restricted to compare only nodes being placed on the same tree level, solving the *Largest Common Subtree*-problem, but therefore a perfect matching is computed for two levels. While this restriction limits the matching power of the tool in the way that renamings or code shifted between levels cannot be recognized, it allows an acceptable runtime without using heuristics. While Section 4.4.1 depicts, how the two algorithms work together, whereas Sections 4.4.2 and 4.4.3 explain them in detail.

4.4.1. Tree Matching

To decide which of the Algorithms 5 and 7 is used to compare two nodes, it has to be determined whether the order of their children is important or not. Algorithm 1 illustrates, how this is accomplished in JDime.

Algorithm 1 TREE MATCHING

```
function TreeMatching(Node A, Node B)
    m ← number of children of A
    n ← number of children of B
    Boolean ordered ← false
    for i ← 1, m do                          ▷ Can be determined using the attributed AST
        if order of A_i is significant then
            ordered ← true
            break
        end if
    end for
    if ordered = false then
        for j ← 1, n do
                                             ▷ Can be determined using the attributed AST
            if order of B_j is significant then
                ordered ← true
                break
            end if
        end for
    end if
    if ordered = true then
        return OrderedTreeMatching(A, B)
    else
        return UnorderedTreeMatching(A, B)
    end if
end function
```

4.4.2. Ordered Trees

Yang [Yan91] describes how to find a matching between ordered trees. He therefore introduces an algorithm based on a solution for sequence matching. As Yang states, the problem of finding the maximum common subtree for ordered trees is a generalization of the largest common subsequence problem. For the latter exists a dynamic programming approach, LCS (see Algorithm 2), which solves the problem in polynomial time [BHR00]. Yang extended the LCS algorithm to his SIMPLETREEMATCHING algorithm to find the maximum common subtree of two trees. This procedure is shown in Algorithm 4. For the implementation of JDIME, Yang's SIMPLETREEMATCHING algorithm was extended to Algorithm 5 and implemented in Java.

Largest Common Subsequence

In general, the largest common subsequence problem with n input sequences is known to be $\mathcal{NP}$-complete [Mai78]. However, for small n it is solvable in polynomial time [Hir75, BHR00]. Algorithm 2 finds the longest common subsequence of two input sequences by using a dynamic programming approach and returns its length.

Algorithm 2 Largest Common Subsequence (LCS)

```
function LCS(Sequence A, Sequence B)
    m ← length of A
    n ← length of B
    Matrix M ← (m + 1) × (n + 1) Matrix
    for i ← 0, m do
        M[i, 0] ← 0                                          ▷ Initialization
    end for
    for j ← 0, n do
        M[0, j] ← 0                                          ▷ Initialization
    end for
    for i ← 1, m do
        for j ← 1, n do
            if A_i = B_j then
                W[i, j] ← 1
            else
                W[i, j] ← 0
            end if
            M[i, j] ← max(M[i, j − 1], M[i − 1, j], M[i − 1, j − 1] + W[i, j])
        end for
    end for
    return M[m, n]
end function
```

To illustrate the idea, Table 4.1 shows the matrix produced by the algorithm. The value of 4 at the position (m, n) in the matrix indicates that the length of the longest common subsequence of $ABCBDAB$ and $BDCABA$ is 4. To identify the matching characters, backtracing is used: Starting at position (m, n), a path is followed until either the row or column index is zero. Finding a valid path via backtracing is shown in Algorithm 3. Whenever the matching decreases, two characters were part of the maximum matching, which is **BCBA** in the example from Table 4.1.

B	0	1	2	2	3	4	**4**
A	0	1	2	2	3	3	**4**
D	0	1	2	2	2	**3**	3
B	0	1	1	2	2	**3**	3
C	0	1	1	**2**	**2**	2	2
B	0	**1**	**1**	1	1	2	2
A	0	0	0	0	1	1	1
		0	0	0	0	0	0
		B	D	**C**	A	**B**	**A**

Table 4.1.: Matrix produced by LCS algorithm

Algorithm 3 BACKTRACING LCS

```
function BACKTRACELCS(Matrix M, Sequence A, Sequence B)
    m ← length of A
    n ← length of B
    i ← m
    while i > 1 do
        j ← n
        while j > 1 do
            if M(i − 1, j) = M(i, j) then
                i ← i − 1
            else if M(i, j − 1) = M(i, j) then
                j ← j − 1
            else
                if M(i − 1, j − 1) < M(i, j) then
                    Mark characters A(i) and B(j) as part of the maximum matching
                end if
                i ← i − 1
                j ← j − 1
            end if
        end while
    end while
end function
```

SimpleTreeMatching

The idea of SIMPLETREEMATCHING is the same as already known from LCS. A matching is computed for each level of the ASTs, the same way LCS does, just that nodes instead of characters are compared. Therefore, SIMPLETREEMATCHING works top down, recursively calling itself to compute the matching of the child nodes. The last matrix

filled is the one for the root nodes, which includes the maximum matching of the entire trees.

Algorithm 4 SimpleTreeMatching by Yang

```
function SimpleTreeMatching(Node A, Node B)
    if A ≠ B then
        return 0                                   ▷ root nodes do not match
    end if
    m ← number of children of A
    n ← number of children of B
    Matrix M ← (m + 1) × (n + 1) Matrix
    for i ← 0, m do
        M[i, 0] ← 0                                ▷ Initialization
    end for
    for j ← 0, n do
        M[0, j] ← 0                                ▷ Initialization
    end for
    for i ← 1, m do
        for j ← 1, n do
            W[i, j] ← SimpleTreeMatching(A_i, B_j)
            M[i, j] ← max(M[i, j − 1], M[i − 1, j], M[i − 1, j − 1] + W[i, j])
        end for
    end for
                    ▷ return sum of child matchings plus matching of the root rodes
    return M[m, n] + 1
end function
```

Ordered Tree Matching in JDime

This section presents the algorithm used in JDime to compare ordered trees. Pseudocode is provided in Algorithm 5. The first part is very similar to Yang's SimpleTreeMatching. To be able to reconstruct the maximum matching tree, the matchings for child nodes are stored. Furthermore, directions are stored to facilitate backtracing. When the algorithm has terminated, a tree of matchings is given due to the recursive calls. Each level of the matching-tree corresponds to a level of the input trees.

Algorithm 5 ORDERED TREE MATCHING - Part 1

```
function OrderedTreeMatching(Node A, Node B)
    if A ≠ B then
        return empty matching                          ▷ root nodes do not match
    end if
    m ← number of children of A
    n ← number of children of B
    Matrix M ← (m + 1) × (n + 1) Matrix
    for i ← 0, m do
        M[i, 0] ← 0                                    ▷ Initialization
    end for
    for j ← 0, n do
        M[0, j] ← 0                                    ▷ Initialization
    end for
    for i ← 1, m do
        for j ← 1, n do
            Matching w ← TreeMatching(A_i, B_j)
            if M[i, j − 1] > M[i − 1, j] then
                if M[i, j − 1] > M[i − 1, j − 1] + w then
                    M[i, j] ← M[i, j − 1]
                    M[i, j].Direction ← LEFT
                else
                    M[i, j] ← M[i − 1, j − 1] + w
                    M[i, j].Direction ← DIAG
                end if
            else
                if M[i − 1, j] > M[i − 1, j − 1] + w then
                    M[i, j] ← M[i − 1, j]
                    M[i, j].Direction ← DOWN
                else
                    M[i, j] ← M[i − 1, j − 1] + w
                    M[i, j].Direction ← DIAG
                end if
            end if
        end for
    end for
```

Algorithm 6 Ordered Tree Matching - Part 2

```
    List childrenMatchings ← empty List
    i ← m
    j ← n
    while i ≥ 1 ∧ j ≥ 1 do
        if M[i, j].Direction = DOWN then
            i ← i − 1
        else if M[i, j].Direction = LEFT then
            j ← j − 1
        else
            if M[i, j] > M[i − 1, j − 1] then
                add M[i, j] to childrenMatchings
            end if
        end if
        i ← i − 1
        j ← j − 1
    end while
                        ▷ return sum of child matchings plus matching of the root rodes
    Matching m ← M[m, n] + 1
    m.childrenMatchings ← childrenMatchings
    return m
end function
```

4.4.3. Unordered Trees

In order to compare Java programs, an algorithm for unordered trees is needed in addition to the one applicable for ordered trees.. Unordered nodes are encountered, for instance, when declarations of imports, methods and class members are compared. As already mentioned in Section 4.4, finding a maximum common embedded subtree for unordered trees is $\mathcal{APX}$-hard [ZJ94, HT96]. In JDime, the key to an acceptable runtime is the restriction to compare only nodes of the same tree level.

To compare unordered children of two nodes A and B, an $m \times n$ matrix M is created, whereas m and n are the amount of the respective children. As a first step, the matchings for all pairs A_iB_j of are computed, with $i = 1..m$ and $j = 1..n$. This is achieved by $n \cdot m$ calls of TreeMatching(A_i, B_j). Given this matching matrix, pairs A_iB_j providing the highest sum of matchings have to be found, whereas each node is part of at most one pair. Finding those pairs is equivalent to finding the maximum matching in a weighted bipartite graph, which is also known as assignment problem.

Per definition, the vertices of a bipartite graph can be divided into two disjoint sets (partitions) T and W, whereas every edge in the graph connects a vertex from T with a vertex from W. A common description for the general (unweighted) matching problem on bipartite graphs is the tasks-workers-example: Imagine, a company has a set of tasks (T) to finish and a set of workers (W) at its disposal, but not every worker can accomplish every task. If a task T_i can be done by a worker W_j, the graph contains an edge T_iW_j. Furthermore, every worker can only be assigned to a single task. The goal is to find a maximum matching between tasks and workers, to get as much done as possible.

If a weight k is added to the edges, a worker generates k units of profit for the company while accomplishing the task. Adding weights to the example means, among all workers that are able to fulfill a certain task, some of them are performing better than others. A weight of zero is equivalent to a non-existent edge. For weighted matching, the goal of the company is assigning workers to tasks in order to maximize profit.

Finding the best matching between unordered nodes of two levels within ASTs can be reduced to maximum weighted bipartite matching: The nodes of the left and right trees' levels embody the partitions, the matchings for all A_iB_j, which are stored in the matrix, represent the weights.

There exist several algorithms to solve the weighted bipartite matching problem in polynomial time [SL93]. A well-known approach is to reduce the problem to maximum flow: A source and a sink node is added to the graph, whereas the source is connected to all nodes of the left partition, and the sink to all nodes of the right partition. Algorithms computing maximum flow have been provided, for example, by Ford and Fulkerson or Edmonds and Karp [LR86, SL93]. A different approach is the Hungarian

method, which solves the weighted bipartite matching problem via matrix transformations [Fra05, SL93].

Also, it is possible to express the problem as linear program as follows[4]:

$$\max \sum_{t \in T} \sum_{w \in W} k_{tw} x_{tw}$$

subject to

$$\sum_{t \in T} x_{tw} = 1, \quad \text{for all } w \in W$$

$$\sum_{w \in W} x_{tw} = 1, \quad \text{for all } t \in T$$

$$x_{tw} \in \{0, 1\}, \quad \text{for all } (t, w) \in T \times W$$

where $x_{tw} = 1$ means, that task t is assigned to worker w. The first constraint set expresses that each worker must be assigned exactly one task, the second ensures that every task is carried out by exactly one worker.

The constraint matrix C of a linear program is a matrix with a row for each constraint and a column for each variable. An entry c_{ij} is set to 1, if the j-th variable is added in the i-th constraint, -1, if it is subtracted, and 0 if it does not appear at all.

For the weighted bipartite matching problem, C is equal to the unoriented incidence matrix of the underlying graph and therefore an optimal, integral solution can be computed in polynomial time, in contrast to general exponential time.

JDime uses the GNU linear programming kit[5] (GLPK) to solve maximum weighted bipartite matching and therefore retrieve the pairs providing the highest sum of matchings. During the experimental study performed within this thesis, which will be discussed in Chapter 5, GLPK provided the maximum matching in sufficient time. Nevertheless, as future work, for instance, the Hungarian method could be implemented in JDime, as it might lead to a further speed up while comparing unordered trees. Also, the use of superimposition, as introduced in Section 3.6, has to be considered in order to compare unordered nodes.

The procedure implemented in JDime is presented in Algorithm 7.

[4] `http://www.math.ucla.edu/~tom/LP.pdf`
[5] `http://www.gnu.org/software/glpk`

Algorithm 7 UNORDERED TREE MATCHING

```
function UnorderedTreeMatching(Node A, Node B)
    if A ≠ B then
        return 0                                   ▷ root nodes do not match
    end if
    m ← number of children of A
    n ← number of children of B
    Matrix M ← (m) × (n) Matrix
    for i ← 0, m do
        for j ← 0, n do
            W[i, j] ← TreeMatching(A_i, B_j)
        end for
    end for
                    ▷ return sum of child matchings plus matching of the root rodes
    Matching m ← M[m, n] + 1
    m.childrenMatchings ← SolveLP(M)
    return m
end function
```

4.4.4. Implementation

The Java implementation for JDIME offers two methods to compare ASTs: `diff()` and `diff3()`: `diff()` is used to compare two trees, and simply calls the TREEMATCHING algorithm, which was presented in Section 4.4.1, and retrieves a set of matching nodes. For each matching pair, it flags both nodes as equal and adds a reference to the corresponding node of the other tree. when processing a node in a later step, the corresponding node can be accessed in $\mathcal{O}(1)$. `diff3()` compares two trees by taking the tree of their common ancestor into account. It is implemented by three calls of TREEMATCHING, comparing all pairs of trees, followed by the appropriate flagging, as shown in Algorithm 8.

Algorithm 8 DIFF3

```
                                          ▷ base′ is a copy of base
function DIFF3(Tree left, Tree base, Tree base′, Tree right)
    matchings ← TREEMATCHING(base, left)
    for all Matching m : matchings do
        FLAG(m.A, m.B, EqualFlag)
    end for
    matchings ← TREEMATCHING(base′, right)
    for all Matching m : matchings do
        FLAG(m.A, m.B, EqualFlag)
    end for
    matchings ← TREEMATCHING(left, right)
    for all Matching m : matchings do
        FLAG(m.A, m.B, NewEqualFlag)
    end for
end function
```

To facilitate the process of merging, the base revision's tree is cloned before calling `diff3()`. That means, the tool internally works with `base` and `base'`. After `diff3()` has terminated, nodes of `base` have references to their corresponding nodes of `leftRevision`, nodes of `base'` have references to their corresponding matchings of `rightRevision`. As a future optimization, the first two steps could also be parallelized, as they do not depend on each other. With the matching process being one of the most time-consuming tasks during the merge, and dual-cores being lower standard for today's workstations, this optimization should fasten up the tool significantly in practice. Apart from such simple optimizations, the matching algorithms themselves could be improved to make use of concurrency. In case of the unordered matcher, this would be trivial at least for its first step, in which all nodes are compared to each other. For the last step of `diff3()`, which compares the left and right revision, a special flag is used to tag changes and additions only existent in those opposing revisions, but not in the common ancestor. The technical detail of using a supplementary clone of the base revision is omitted in the further discussion.

4.5. Merge Algorithm

After the ASTs of the three input programs have been compared to each other, the actual merge launches. Instead of modifying one of the existing trees, a new AST is created for the output program, which will be referred to in the following sections as merge-AST or merged tree. This decision was made in order to keep the three existing ASTs unmodified and allow the tool to query the original trees for additional information

even in an advanced step of the merge process. This property ensures, that the merge process of JDime is implemented in a non-destructive way, which could also be useful if additional examining algorithms are implemented in the future, for instance, to make use of type information. The main steps of the merge process is shown in Algorithm 9.

Algorithm 9 Merge - Part 1

function Merge(Node *left*, Node *base*, Node *right*)
    mergetree ← empty tree
    diff3(*left*, *base*, *right*)
    unchanged ← $\{n \mid n \in base \wedge n \in left \wedge n \in right\}$
    insert(*unchanged*, *merged*)
    consistent ← $\{n \mid n \notin base \wedge n \in left \wedge n \in right\}$
    insert(*consistent*, *merged*)
    lchanges ← $\{n \mid n \in left \wedge n \notin merged \wedge n \notin right \wedge n \notin base\}$
    rchanges ← $\{n \mid n \in right \wedge n \notin merged \wedge n \notin left \wedge n \notin base\}$
    DetectConflicts(*lchanges*, *rchanges*)
    MergeChanges(*merged*, *lchanges*)
    MergeChanges(*merged*, *rchanges*)
end function

4.5.1. Unchanged Nodes

The first step of the merge process is traversing the trees top-down, while copying all nodes into the merge-AST that are unchanged in all three input revisions. This is accomplished by traversing the base revision's tree top down, inserting all nodes into the merge tree that have corresponding nodes in both left and right revision. If a node is unchanged, but its parent node was modified, it will be skipped for now and be dealt with in a later step of the merge process.

4.5.2. Consistently Changed Nodes

The second step of merging addresses changes introduced in both left and right revision. In order to find those changes, the tree representing the left revision is traversed. If a node is reached that is not contained in the base revision, but has a corresponding node in the right revision, it will be inserted into the merge tree.

4.5.3. Separately Changed Nodes

The previous steps were rather simple, as all nodes added to the merge tree so far were originally contained in the trees of both left and right revision, and as such could not create any conflicts. Changes introduced in only one of the revisions are more difficult to merge, because they could conflict with changes of the opposing revision.

Thanks to the flagged nodes, finding the changes is easy. So before conflicts can be detected, a list of all changes introduced by the left revision is created. The same is performed for the right revision.

Detecting Conflicts

In order to detect conflicts, every change in a revision is checked against each change of the opposing revision. If two changes would be added to the same parent node in the merge tree, they might be conflicting. Now the algorithm has to distinguish between two scenarios: parent nodes being strict concerning the order of their children, and those whose children are unordered.

In the first case, implemented by extending the attribute system of JastAddJ's `ASTNode`-hierarchy with an appropriate method, a node can check by itself whether it conflicts with another node or not by calling its `conflicts(Node otherNode)` method. The implementation of this method depends on the language element represented by the node. For instance, a node representing a method declaration knows, it can conflict with another method declaration, if their names, their return values and the amount of their arguments equal. It also knows, it cannot conflict with an import declaration. If `conflicts(Node otherNode)` returns `false`, the change can be safely inserted into the merge tree. Otherwise, both nodes and their subtrees are marked as conflicting.

If the order of the parent node's children is significant, the insertion positions of the rivaling changes are decisive. Imagine, for example, a block of n statements with $n > 2$, where the left revision modified the first statement and the right revision the last while anything in between remained unchanged. Merging both changes in this example would be no problem, as they do not overlap. In case there is no way inserting both changes in an unambiguous way, they are marked as conflicting. This procedure is shown in Algorithm 10.

Algorithm 10 MERGE - Part 2

```
function detectConflicts(List leftChanges, List rightChanges)
    for all Node n : leftChanges do
        for all Node m : rightChanges do
            if parent of n in mergetree = parent of m in mergetree then
                if order of n and m is important then
                    if insert position is ambigious then
                        conflict(n, m)
                    end if
                else
                    if n.conflicts(m) then
                        conflict(n, m)
                    end if
                end if
            end if
        end for
    end for
end function
```

A node involved in a conflict has an appropriate conflict-flag, a list of conflicting changes (which belong to the opposing revision) and a list of related changes (belonging to its own revision). I node A is detected to be in conflict with node B, while B is already part of an existing conflict, the previously detected conflicting changes of B are added to A's list of related changes and vice versa. The procedure used to mark conflicts is shown in Algorithm 11.

Algorithm 11 MERGE - Part 3

```
function conflict(Node n, Node m)
    flag(n, m, ConflictFlag)
    add n to m.conflictingChanges
    for all Node r : n.relatedChanges do
        add r to m.conflictingChanges
    end for
    for all Node c : m.conflictingChanges do
        add c to n.relatedChanges
    end for
end function
```

Treating Conflicts

If all changes have been processed and no nodes are flagged as conflicting, the merge has finished. The remaining conflicts nevertheless have to be included into the merge tree, thus allowing the pretty-printer to find them. Therefore, a dummy node is created for each conflict, linking to the respective sets of conflicting changes. These sets have already been gathered by DETECTCONFLICTS, and are stored in the `relatedChanges` and `conflictingChanges` lists of a node. This dummy node is then inserted into the merge tree instead of the nodes being in conflict. In case of a certain node requiring a minimum amount of children in order to guarantee syntactical correctness, placeholder dummy nodes are inserted. Algorithm 12 illustrates this process. When the merge tree is transformed into source code, and the pretty-printer reaches a conflict dummy node, it prints two opposing sets of changes, visualizing them the same way GNU MERGE does.

Algorithm 12 MERGE - Part 4

```
function mergeChanges(Tree mergetree, List changes)
    for all Node n : changes do
        if n is flagged as conflict ∧n is not marked as processed then
            Node c ← new conflict dummy node
            c.leftAlternative ← n.relatedChanges
            c.rightAlternative ← n.conflictingChanges
            for all Node m : n.relatedChanges ∪ n.conflictingChanges do
                mark m as processed
            end for
        else
            insert n into mergetree
        end if
    end for
end function
```

4.6. Challenges

While working on the implementation in general, and with JASTADDJ in particular, a few difficulties surfaced. This section discusses the main issues and explains the solutions found most applicable for JDIME.

4.6.1. Altering Trees in JastAddJ

At the beginning of the implementation, the first approach was to operate directly on the abstract syntax trees provided by JastAddJ, which always assumes an AST to have a correct structure fulfilling the language-specific rules. While the merge algorithm requires syntactical correct input and produces a syntactical correct tree as final result, it temporarily works on fragmented and partially incorrect trees to accomplish this result. for example, an `if`-statement in Java has three children: a condition, an `if`-block and an optional `else`-block. If the `else`-block does not exist in the code, JastAddJ adds an empty block to the AST. Let's assume we are performing a three-way merge with a left revision, base revision, and right revision as input, and there is a difference between the conditions of the `if`-block in left and right revision. Detecting a potential conflict, the merge algorithm will not add the condition node to the merge tree in its first step, because in this step only all nodes being unchanged, and therefore contained in all three revisions, will be inserted into the merge tree. In a later step, it will decide whether the condition of one of the two revisions will be inserted, or whether a conflict exists and a dummy node has to be created. What seems stable and correct to the merge algorithm (namely not adding potentially conflicting nodes in its first steps), is not at all stable and correct for the AST generated by JastAddJ, since it requires the existence of a condition.

A concrete related problem with JastAddJ is, that whenever asked how many children a specific node has, it will always return the amount of children this node is supposed to have according to the language specification, and not how many children it actually has. The return value of the `getNumChild()`-method is hard-coded to a certain integer value for a lot of `ASTNode`'s subclasses. In case of an `if`-statement, JastAddJ will always return 3, when asked how many children the `if`-statement has, no matter if there is actually a condition node or not, since an `if`-construct is supposed to have three children, as specified in the language specification. Even after repeatedly calling the `removeChild()`-method of the node and deleting its first child node, the node representing the `if`-statement still claims to have all three children. The way adding and removing nodes is implemented in JastAddJ, which is shifting an array, complicates the situation even more. This behavior makes it very difficult to implement complex algorithms altering a tree, if not every single change results in a complete and correct AST.

After experimenting with a lot of potential workarounds like inserting dummy nodes, dealing with exceptions, and even trying to change the algorithms, the decision was made to let the merge algorithm work on a dedicated, much simpler tree structure, consisting of a custom node class: `jdime.Node`. Each of these custom nodes encapsulates a JastAddJ `ASTNode` internally, but its `getNumChild()`-method returns the actual amount of children the node has at the time the method is called. Furthermore, adding and removing child nodes is implemented in a safer way. Using these custom node

objects, working with fragmented trees is possible. When the merge algorithm has finished, a complete and correct AST containing of `ASTNode`s can be generated top-down, recursively from the custom root node.

An additional benefit of the new tree structure is that JDime could now theoretically be dismantled from JastAddJ in order to use another backend.

4.6.2. Inserting Nodes in the Right Order

Another difficult problem was to insert changes that occurred in only one revision into the merge tree in correct order. There are several, complex scenarios that have to be considered by the merge algorithm. Imagine, for instance, a block of statements, where the left revision includes the first two statements from the base revision without modification, while inserting an additional statement behind. If the right revision deleted the first and second statement of the originating revision, but introduced changes instead of them, how are both revisions going to be treated by the merge process? The correct solution is to create a conflict, since the insertion positions of the changes are ambiguous. Detecting such situations is not easy, but missing them is fatal to the merge algorithm's output. Therefore, the implementations of detectConflicts and mergeChanges have to take a large amount of special cases into account, which allows assembling a merge with few conflicts while not producing wrong insertions.

4.6.3. Representing Conflicts

After the merge process has finished, the merged AST has to fulfill the specification of the respective language. In order to produce source code from the AST, a pretty-printer traverses the tree and prints suitable code for each node. To print conflicts in an acceptable way, the pretty-printer has to identify them as such. This leads to the question how conflicts should be included into the AST without breaking its syntactical structure, which is not allowed in JastAddJ, as mentioned in Section 4.6.1. Inserting both alternative branches of a conflict into the tree would break the structure, so this is not an option.

In JDime, a specific dummy node is inserted in such cases. A conflict dummy node holds references to the rivaling branches, that can be followed by the pretty-printer. In case a removal of code conflicts with other changes, the removal is represented by an empty node.

4.6.4. Runtime

Another typical issue of syntactic merges is their rather high runtime, which is caused by the expensive algorithms and the overhead that is necessary to parse and pretty-print programs. The syntactic merge algorithm implemented yet is not even the most precise, one could think of. E.g., an algorithm could be added, which detects renamed methods or code shifted between different levels. The impact on the runtime would be even more noticeable in such cases. For unordered trees, a more precise matching would only be possible by implementing approximations.

To offer a trade-off between acceptable execution times and precision, an auto-tuning approach was implemented for JDime: Prior to launching an expensive syntactic merge, a cheap, conventional merge using GNU merge is performed. Only if the textual merge produces a conflict, the syntactic merge is started.

When there are more algorithms available for JDime in the future, this idea can be extended easily. Several configurations defining rankings of algorithms, from fast to precise, could be preset and provided to users or tools.

Of course the gain of speed has its price, in the form of a potential loss of precision: Conflicts missed by a cheap merge are not detected if the file being processed has no further conflicts that could be detected by the merge. Auto-tuning can be switched on and off via command line options.

4.7. Extensibility

Additional merge algorithms simply have to implement `MergeInterface` and have to be registered in the main method of JDime. `MergeInterface` requires the methods `merge()` and `merge3()` to be implemented. In order to add a different compare algorithm for ASTs, which can then be used by the existing syntactic merge algorithm, a class inheriting `TreeMatcher` can be created.

Of course, also integrating other, non-AST-based merging algorithms is possible via providing an appropriate wrapper class, as it was done for GNU merge and FSTMerge. Those algorithms can be as well used for further optimized auto-tuning capabilities.

CHAPTER 5

Empirical Study

In the previous chapter, JDime was presented, an adjustable, syntactic merge tool for Java programs. It uses a tree-based approach and was built within this thesis in order to produce merges of better quality than a standard textual merge does.

In this chapter, we will see how it actually performs compared to GNU merge, which still is a standard tool in merging software artifacts so far. GNU merge is a representative for line-based, textual merges, which have been described in Section 3.4. Also included in the comparison is FSTMerge, which was designed to be a trade-off between a textual and a syntactic merge. FSTMerge has been presented in Section 3.6.

Eight projects written in Java, with several merge scenarios given, have been chosen as subjects for the study. The values measured for the evaluation are number of conflicts, conflicting files, conflicting lines, and the runtime.

In Section 5.1, the sample projects and the environment for the evaluation are introduced. Afterwards, the results are presented in Section 5.3. In the last section, the most influencing factors leading to the results are discussed.

5.1. Setup and Choice of Projects

While working on semistructured merge, Apel et al. published an evaluation comparing the performance of FSTMerge with a textual merge [ALL+]. Therefore, they retrieved several projects varying in size and being written in Java, C#, or Python from SourceForge, examined the projects' subversion histories, and created a couple of merge scenarios. Those merge scenarios either have actually been performed during the evolution of a project and are documented in form of commit messages, or have been considered realistic in the context of the respective project [ALL+]. An example for the last case is merging a development branch back into trunk.

5.1.1. Sample Projects

A subset of appropriate projects is taken from the mentioned repertory for the evaluation performed within this thesis, as is shown in Table 5.1. The lines of code (LoC) stated in the table, are average values over all revisions that were checked out. The values for the separate revisions in order to build the average, have been computed using cloc[1]. Furthermore, the amount of merge scenarios (MS) is given for each project.

Project	**Domain**	**LoC**	**MS**
DrJava	development environment	77 K	9
FreeCol	turn-based strategy game	85 K	10
GenealogyJ	editor for genealogical data	55 K	10
iText	PDF library	85 K	8
jEdit	programmer's text editor	98 K	8
Jmol	viewer for chemical structures	128 K	7
PMD	bug finder	65 K	10
SQuirrelSQL	SQL GUI client	194 K	10

Table 5.1.: Sample projects

5.1.2. Test Environment

For each merge scenario, four merges are performed consecutively:

- textual merge (TM): GNU merge

[1] `http://cloc.sourceforge.net`

- syntactic merge (SM): JDime with disabled auto-tuning (always using syntactic merge)
- combined merge (CM): JDime with enabled auto-tuning (trying textual merge first)
- semistructured merge (SSM): FSTMerge

The framework for the evaluation is a Java program which already performs the basic three-way merge rules, as shown in 3.2, at directory level and executes the merge tools with files as appropriate input arguments. This decision was made in order to allow a more direct comparison between the merge tools. The framework program measures the time a tool needs to complete a merge, and analyzes the output produced. The runtime is retrieved by running each merge process 10 times, then computing the median.

All tests were carried out on an AMD Phenom II X6 1090T with 6 cores at 3.2 GHz with 16 GB of RAM. The machine runs Gentoo Linux with kernel 3.2.7 and uses an up-to-date version of the Oracle Java HotSpot 64-Bit Server VM (1.6.0_31).

5.2. Results

In this section, the overall results are presented. The complete statistics are contained in form of tables in Appendix A, and in form of diagrams in Appendix B.

Table 5.2 lists the mean values over all revisions for conflicting files, conflicts, and conflicting lines for all projects. Compared are textual merge (TM), syntactic merge (SM), combined merge (CM), and semistructured merge (SSM). As the syntactic approach was expected to be superior to textual merges concerning conflict detection and resolution, fields in the tables are highlighted if the opposite was the case. Table 5.3 includes the respective mean runtime values over all revisions in milliseconds. Figures 5.1 to 5.4 illustrate the collected data.

Another interesting representation of the results is given by Tables 5.4 and 5.5, which display percentage values for each merge using the textual merge as base line by setting it to 100 % (since textual merging is the de-facto standard today).

In most of the merge scenarios tested, the syntactic approach indeed was superior than the line-based and semistructured ones. FSTMerge tends to rank in between textual and syntactic merge, a result that is not surprising as well.

It becomes evident, that the results produced by JDime executing a pure syntactic

	conflicting files				conflicts				conflicting lines			
Project	TM	SM	CM	SSM	TM	SM	CM	SSM	TM	SM	CM	SSM
DrJava	7	2	2	7	2	4	4	17	285	61	57	203
FreeCol	63	48	47	58	47	200	187	212	4321	1745	1709	3254
GenealogyJ	8	5	5	5	5	6	6	6	150	55	55	105
iText	212	190	190	173	190	1847	1847	667	117234	6458	6443	35514
jEdit	3	2	2	2	2	2	2	4	40	16	16	51
Jmol	23	13	12	19	12	31	30	43	1117	234	225	609
PMD	10	7	7	10	7	10	10	15	691	357	357	471
SquirrelSQL	15	7	7	12	7	9	9	23	2492	214	214	563

Table 5.2.: mean results for all projects

	runtime in ms			
Project	TM	SM	CM	SSM
DrJava	5971	99901	33674	539996
FreeCol	5726	82539	27815	473963
GenealogyJ	4174	56378	6115	337669
iText	7163	338302	315969	427851
jEdit	5174	72536	6587	459207
Jmol	4937	313965	239141	556777
PMD	3194	33368	5222	190080
SquirrelSQL	18129	230279	23788	1101322

Table 5.3.: mean runtime values for all projects

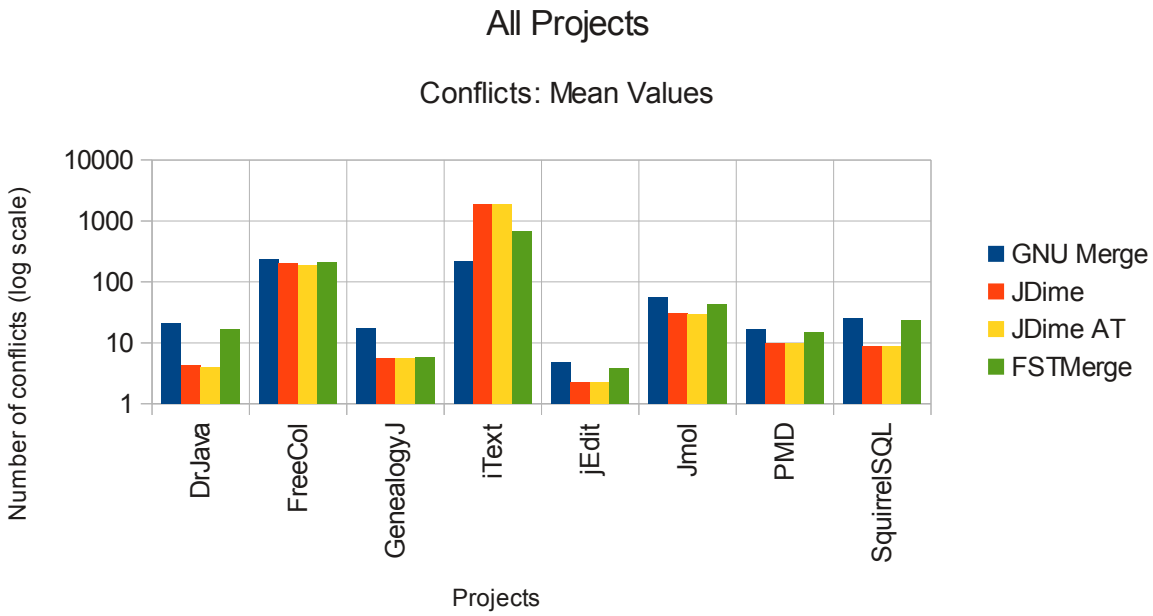


Figure 5.1.: All projects: mean values for conflicts

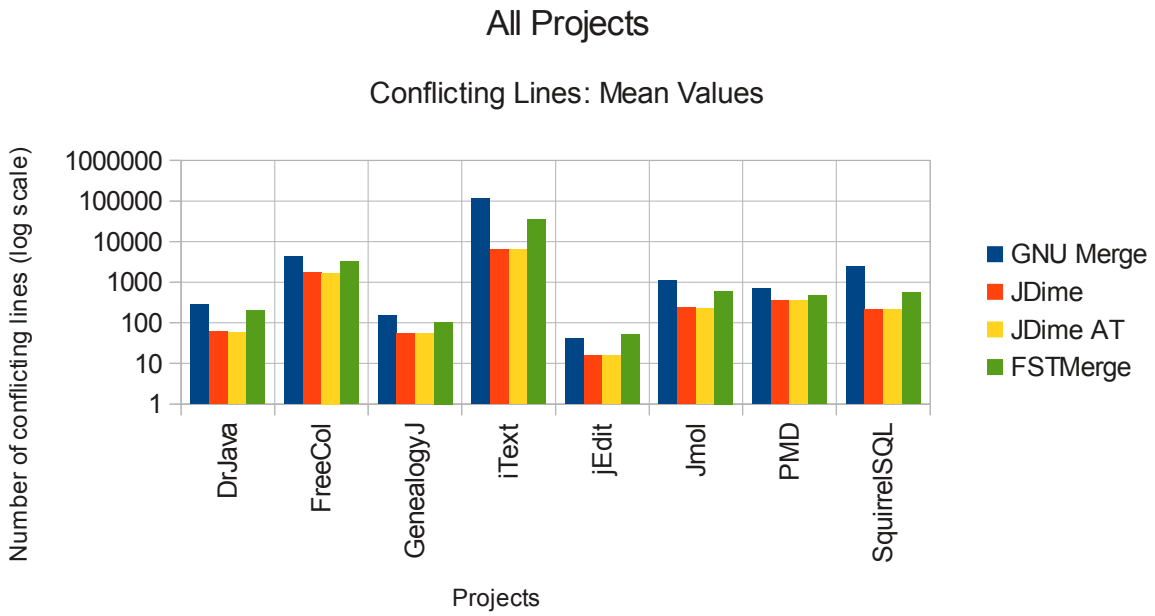


Figure 5.2.: All projects: mean values for conflicting lines

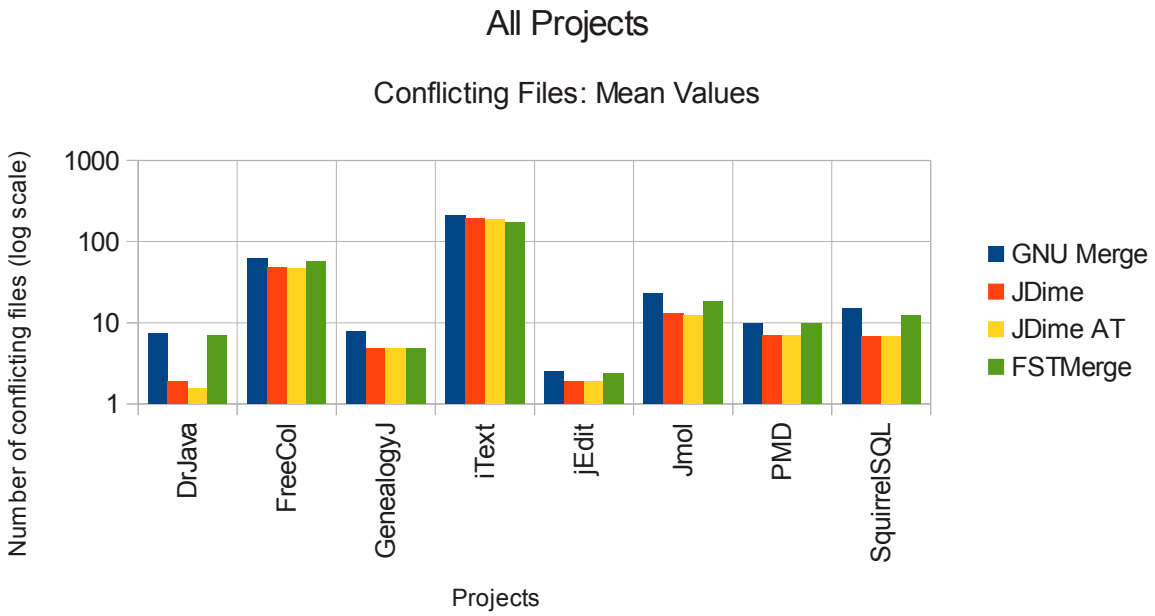


Figure 5.3.: All projects: mean values for conflicting files

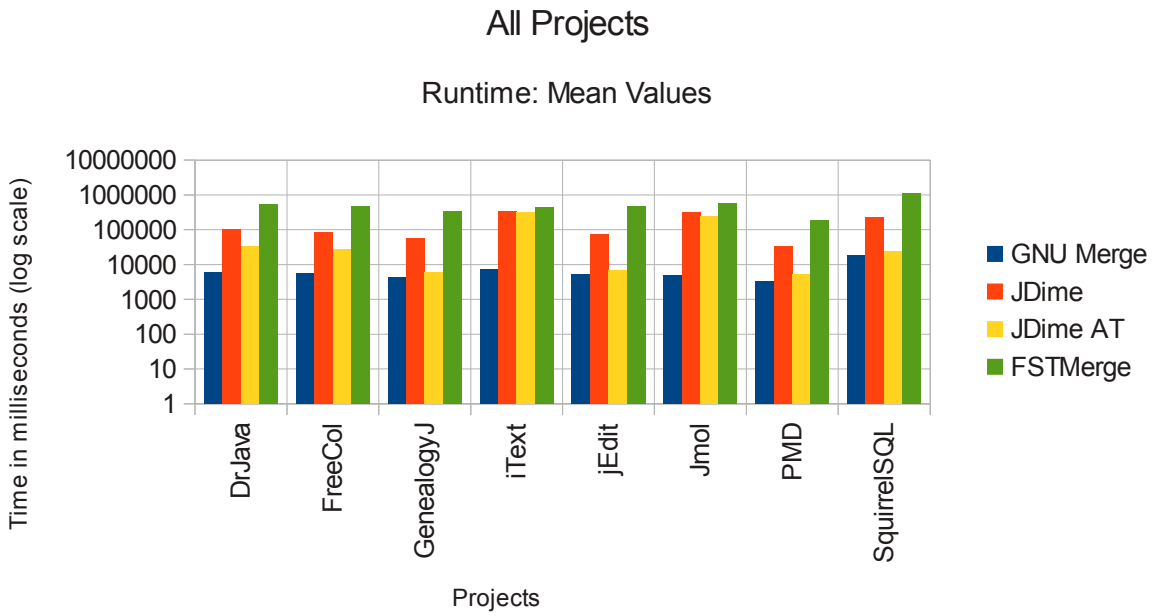


Figure 5.4.: All projects: mean values for runtime

	conflicting files				conflicts				conflicting lines			
Project	TM	SM	CM	SSM	TM	SM	CM	SSM	TM	SM	CM	SSM
DrJava	100	25.78	21.28	94	100	20.31	18.72	79.69	100	21.45	20.05	71.22
FreeCol	100	77	75.24	92.01	100	85.53	79.76	90.44	100	40.39	39.56	75.29
GenealogyJ	100	62.34	62.34	62.34	100	32.56	32.56	33.72	100	36.7	36.7	69.99
iText	100	89.63	89.45	81.56	100	855.27	854.98	308.62	100	5.51	5.5	30.29
jEdit	100	75.2	75.2	95.2	100	46.11	46.11	79.51	100	38.81	38.81	127.01
Jmol	100	56.18	52.46	80.25	100	54.73	53.19	76.73	100	20.95	20.19	54.56
PMD	100	70.71	70.71	98.99	100	58.33	58.33	89.88	100	51.66	51.66	68.12
SQuirrelSQL	100	45.03	45.03	82.12	100	34.66	34.66	92.03	100	8.6	8.6	22.57

Table 5.4.: results in percentage for all projects with textual merge as base level

	runtime			
Project	TM	SM	CM	SSM
DrJava	100	1673.08	563.96	9043.47
FreeCol	100	1441.5	485.78	8277.53
GenealogyJ	100	1350.73	146.51	8090
iText	100	4723.16	4411.36	5973.39
jEdit	100	1402.08	127.33	8876.13
Jmol	100	6359.42	4843.86	11277.64
PMD	100	1044.88	163.53	5952.08
SQuirrelSQL	100	1270.2	131.21	6074.79

Table 5.5.: runtime in percentage for all projects with textual merge as base level

merge are in the same range as the ones produced using the combined approach. In some specific cases, the combined version performed even better. The reasons for this behavior will be discussed in Section 5.3. Comparing the runtime of JDime with and without auto-tuning proves, that combining different approaches in order to achieve a trade-off between speed and precision is an effective way to get a tool more applicable in practice. The auto-tuning approach is up to 12 times faster than the pure structured merge, with 5 times being the average.

A few spikes are noticeable among the projects, e.g., iText with large amounts of conflicts but few conflicting lines for syntactic and semistructured merges. This observation will be examined further as well in the following section.

5.3. Discussion

As expected, the structured merge in form of JDime produces better results than textual or semistructured approaches in most of the merge scenarios, although some surprises can be observed for some projects. In this section, the most influencing factors are discussed, which will also depict the appearance of the spikes.

5.3.1. Insertions and Ordering of Members

As expected, and already been observed by Apel et al. [ALL+], ordering conflicts are a common problem for line-based merge tools, whereas they are more easily detected by semistructured or structured approaches. Ordering conflicts occur, if new elements are added between existing ones or if existing members are reordered. In particular, using IDE-features automatically sorting class members can be fatal for later merges, as long as line-based merging tools are used. The good results of FSTMerge, which was developed in order to address this issue, indicates how important in practice proper detection at this level is. Using a matching algorithm for unordered trees, which was presented in Section 4.4.3, JDime handles reorderings without creating conflicts as well.

A typical example, extracted from merge scenario `rev4007-4103` of SQuirrelSQL, is given in Figure 5.5. Both JDime and FSTMerge are capable of automatically merging such scenarios without conflict. Other typical elements in Java causing ordering conflicts are import declarations.

As long as the insert position is unambiguous, JDime is even able to merge insertions of elements whose order is significant, for example, inside of method bodies.

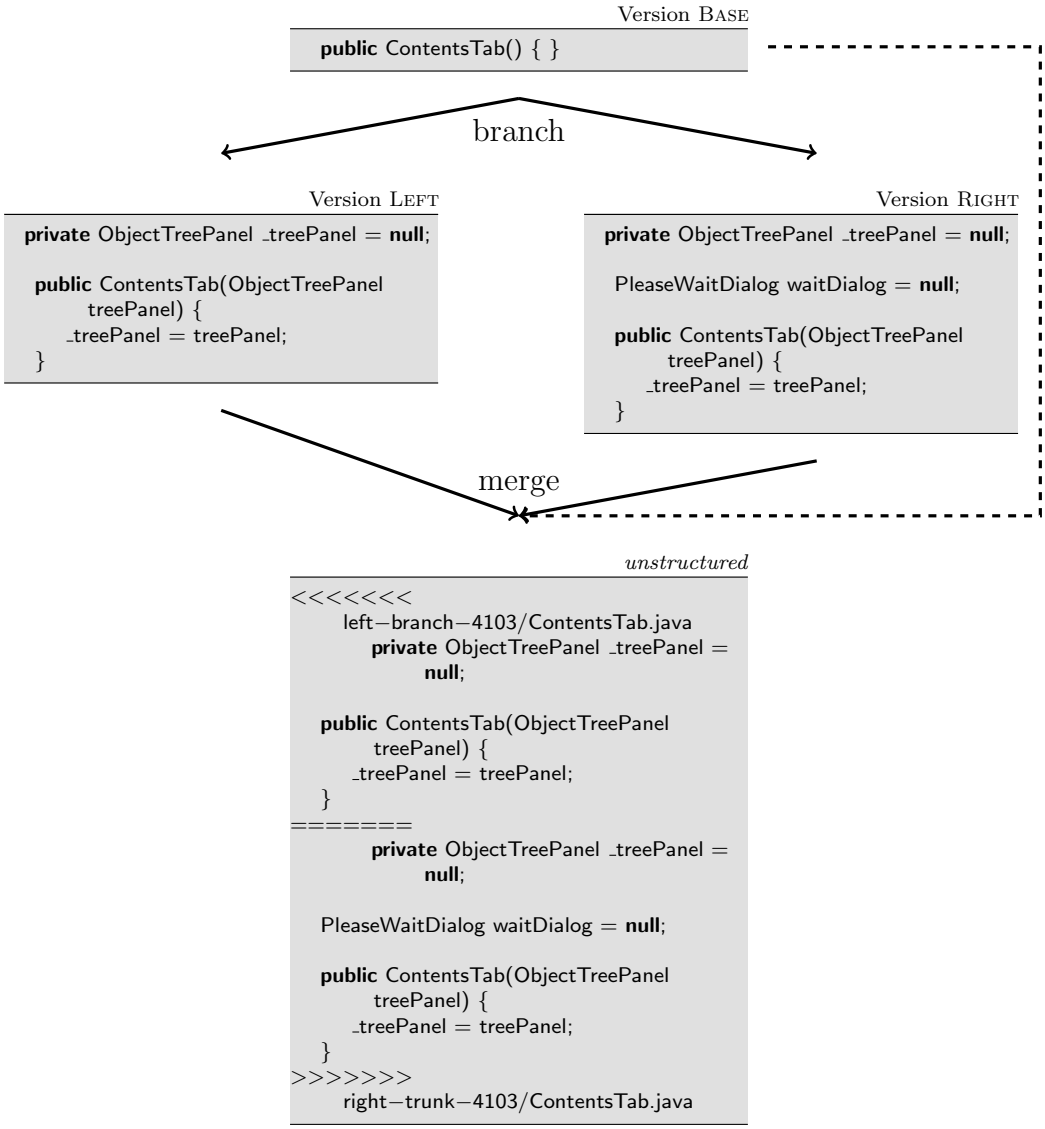


Figure 5.5.: Ordering conflict

5.3.2. Code Formatting

Code reformatting was another factor having a large impact on the results. If both rivaling branches changed indentation style, textual merges produce large conflicts with many conflicting lines. This issue also affects FSTMerge, which uses GNU merge in order to merge method bodies. As already mentioned in Section 3.4, it is possible to have GNU diff ignoring indentation changes. Unfortunately, this option is not provided in the GNU tools diff3 and merge.

The impact of reformattings can be seen most clearly by examining merge scenario `rev4989-5044` of the DrJava project. JDime could merge almost all files without a conflict, whereas GNU merge and FSTMerge produced more than 20 conflicting files. With auto-tuning enabled, even the last file could be merged without a conflict by JDime. Examining the relevant file manually proved that the textual merge did not miss a conflict in this case. A lot of changes in code formatting were introduced in several revisions of DrJava, changing not even trailing and leading spaces and tabs, but also spaces between arithmetic arguments, for instance, `(x+1)` was changed to `(x + 1)`, which cannot (and probably should not, as it would be a semantic difference in case of strings) be detected by textual merges at all. These changes explain the results for syntactic merge for DrJava, which are much better than the average improvement.

An example merge from DrJava is shown in Figure 5.6. JDime does not distinguish between the code used in left and right revision.

5.3.3. Two-Way Merging

The project iText is an example where the syntactic merge produces a very high amount of conflicts compared to the textual merge (850 %), therefore having a lot fewer conflicting lines (5%). FSTMerge has a similar tendency for these scenarios, but with less extreme dimensions (300 % conflicts, 30 % conflicting lines). Those results differ a lot from the other projects, so there had to be something special about iText. Investigating the merge scenarios for iText closer, shows that a large amount of two-way merges is performed due to renamings in the project directory, resulting in a missing common ancestor. After re-adjusting the benchmark framework in order to gain more statistical information about which merges were performed, the situation was a lot clearer: To be precise, 78 % of all merges executed while processing the merge scenarios of iText were two-way merges. This is much more than the average percentage of two-way merges for the other projects, which tends to be below 1 %. Indeed, manually inspecting the input files of the merges as well as the output generated by the competing tools, confirms the suspicion.

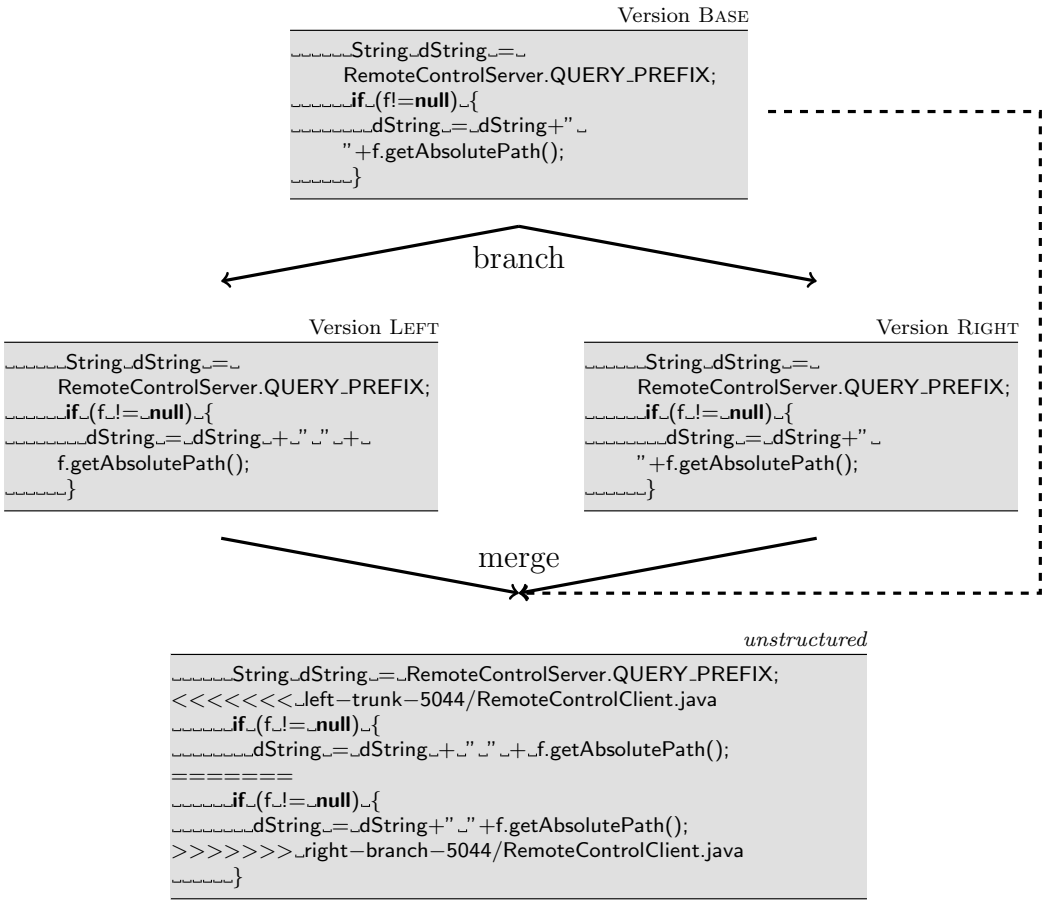


Figure 5.6.: Code reformatting

Two-way merges are handled by GNU MERGE by creating a conflict and adding all lines of the files as alternatives, which results in few conflicts with very many conflicting lines. The semistructured approach is able to merge some of the structural changes that do not overlap concerning their declarations, but has to rely on the textual merge in order to process the remaining methods' bodies. This results in more conflicts, but fewer affected lines due to the finer granularity. Finally, the syntactic merge produces a very high number of close-grained conflicts which affect a comparatively low amount of lines.

5.3.4. Renamings and Changed Signatures

As already discovered in the evaluation for semistructured merge, renamings are problematic for structural merging under some circumstances [ALL+]. Since the matching algorithms of JDIME are currently not able to detect renamed methods or classes, scenarios might occur in which a textual merge produces less conflicts than the structured merge. The same can happen if the signature of a method was changed in one revision, but at the same time, changes were introduced by the other revision to the old method.

An example for the last case can be found in merge scenario `rev5082-5155` of SQuirrelSQL. In the file `UpdateControllerImpl.java`, the right revision changed the method signature of `pullDownUpdateFiles()`, whereas the left revision changed statements in the body of the method. While merging this file, the syntactic merge produced more conflicting lines than the textual merge.

To match the children of a renamed element to their corresponding nodes of the opposing revision, the matching algorithm has to solve the maximum common **embedded** subtree problem, which is known to be $\mathcal{APX}$-hard [ZJ94].

A simpler approach to improve the performance when encountering renamings while auto-tuning is enabled, would be to use the result produced by the textual merge if it contains less conflicting lines. So far, the result of the textual merge is only used if it created no conflicts at all. A simple comparison after both runs in case of detected conflicts would be a simple and convincing way to improve the output of the merge tool. This solution has also been proposed in the evaluation of semistructured merge [ALL+], and could be easily integrated into JDIME.

5.3.5. Runtime

One of the main benefits of textual merges are their speed, which was confirmed by the measures for GNU MERGE during the evaluation. The other approaches were a lot slower compared to the line-based merge.

The syntactic merge was expected to be the slowest in the field of competitors, but surprisingly this was not the case. For all projects benchmarked during the study, FSTMERGE needed longer to complete its merges. While JDIME is tailored for Java, FSTMERGE, as a plugin of FEATUREHOUSE, is more general concerning the support of programming languages. The price for this generality might be increased runtime, but this would have to be inspected with more detail in the case of FSTMERGE. Using a profile tool measuring the amount of time spent in single methods, might be a promising attempt to find possible bottlenecks.

The combination of textual and syntactic merge provides an acceptable trade-off for many of the tested projects. In particular, large files are problematic for the syntactic merge, since they lead to ASTs with many nodes per level, which increases the time for each compare process significantly. Especially the algorithm used to match unordered nodes affects the runtime in a negative way in those cases.

CHAPTER 6

Conclusion

This thesis gave an overview of version control software and depicted situations in which merging of software artifacts is required. Different approaches and techniques in order to merge programs have been explained in the second chapter. The de-facto standard in merging today is a line-based, textual method. Its benefits are speed and generality, making it applicable to all plain text files. But using no information about the structure of a document, such tools tend to produce a large number of conflicts, which are in some cases hard to resolve manually. Structured algorithms provide more power to match elements and detect conflicts in order to assist users while merging files, but they are slower and restricted to specific program languages A semistructured approach was explained as well, which uses structured information to a certain degree and relies on textual merges for lower levels, thus allowing it to provide better resolution than conventional merges while staying more general than structured ones.

Within this thesis, an AST-based tool performing a syntactic merge for Java source code, called JDime was developed. In addition to the implementation of a pure syntactic merge, the tool was designed to combine different algorithms at runtime, in order to use cheaper algorithms, like line-based ones, as long as no conflicts occur. This functionality can be switched on and off. The chapter about JDime presents idea and architecture of the tool, as well as the algorithms implemented. It was expected, that the tool would be superior in practice to both textual and semistructured merge concerning the detection and resolution of conflicts.

In an empirical study, described in Chapter 5, this assumption was confirmed for most cases, although scenarios have been found where the conventional merge produced less conflicts. Those cases have been examined in Section 5.3. Furthermore, the results obtained in the evaluation show, that the semistructured approach indeed ranks in the middle concerning conflict resolution.

As to future research, even more detailed algorithms, detecting the renaming of elements, could be integrated into JDime. It would also be interesting to use type information and call-graphs in order to detect semantic conflicts. To reduce the runtime, the auto-tuning mechanism could be modified at some points to provide even better results in practice, as mentioned in Section 5.3.4. Furthermore, several independent procedures could be parallelized to benefit from the capabilities of todays' desktop computers.

Bibliography

[AKJK+02] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava, and Yuqing Wu. Structural joins: a primitive for efficient xml query pattern matching. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 141 –152, 2002.

[ALL+] Sven Apel, Jörg Liebig, Christian Lengauer, Christian Kästner, and William R. Cook. Semistructured merge in revision control systems.

[BHR00] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 39 –48, 2000.

[Buf95] Jim Buffenbarger. Syntactic software merging. In Jacky Estublier, editor, *Software Configuration Management*, volume 1005 of *Lecture Notes in Computer Science*, pages 153–172. Springer Berlin / Heidelberg, 1995. 10.1007/3-540-60578-9_14.

[CSFP10] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control With Subversion for Subversion 1.6: The Official Guide And Reference Manual*. CreateSpace, Paramount, CA, 2010.

[EH07] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-*

oriented programming systems and applications, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[Fra05] András Frank. On kuhn's hungarian method—a tribute from hungary. *Naval Research Logistics (NRL)*, 52(1):2–5, 2005.

[Hir75] D. Hirschberg. A Linear Space Algorithm for Computing Maximal Common Subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[HPR89] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11:345–387, 1989.

[HT96] Magnús Halldórsson and Keisuke Tanaka. Approximation and special cases of common subtrees and editing distance. In Tetsuo Asano, Yoshihide Igarashi, Hiroshi Nagamochi, Satoru Miyano, and Subhash Suri, editors, *Algorithms and Computation*, volume 1178 of *Lecture Notes in Computer Science*, pages 75–84. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0009483.

[KKP07] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. A formal investigation of diff3. In *Proceedings of the 27th international conference on Foundations of software technology and theoretical computer science*, FSTTCS'07, pages 485–496, Berlin, Heidelberg, 2007. Springer-Verlag.

[Lin03] Tancred Lindholm. Xml three-way merge as a reconciliation engine for mobile data. In *Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, MobiDe '03, pages 93–97, New York, NY, USA, 2003. ACM.

[Lin04] Tancred Lindholm. A three-way merge for xml documents. In *Proceedings of the 2004 ACM symposium on Document engineering*, DocEng '04, pages 1–10, New York, NY, USA, 2004. ACM.

[Loe09] Jon Loeliger. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. O'Reilly Media, Inc., 1st edition, 2009.

[LR86] J. Van Leeuwen and I Representation. Graph algorithms, 1986.

[Mai78] David Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25:322–336, April 1978.

[Men02] T. Mens. A state-of-the-art survey on software merging. *Software Engineering, IEEE Transactions on*, 28(5):449 –462, may 2002.

[OB09] Bryan O'Sullivan and O'Sullivan Bryan. *Mercurial: The Definitive Guide*. O'Reilly Media, Inc., 2009.

[SL93] Herbert Alexander Baier Saip and Claudio Leonardo Lucchesi. Matching algorithms for bipartite graphs matching algorithms for bipartite graphs. *Relatorio Tecnico*, 700(DCC-03/93):21, 1993.

[Wes91] Bernhard Westfechtel. Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd international workshop on Software configuration management*, SCM '91, pages 68–79, New York, NY, USA, 1991. ACM.

[Yan91] Wuu Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 21:739–755, 1991.

[ZJ94] Kaizhong Zhang and Tao Jiang. Some max snp-hard results concerning unordered labeled trees. *Inf. Process. Lett.*, 49:249–254, March 1994.

List of Figures

List of Tables

List of Algorithms

APPENDIX A

Results: Tables

A.1. Conflicts

		conflicting files				conflicts				conflicting lines			
Project	Revision	TM	SM	CM	SSM	TM	SM	CM	SSM	TM	SM	CM	SSM
DrJava	rev3734-3786	7	3	3	5	3	6	6	6	99	163	163	49
DrJava	rev3734-3788	7	3	3	5	3	6	6	8	131	163	163	59
DrJava	rev3734-3807	8	2	2	6	2	4	4	10	876	73	73	328
DrJava	rev4989-5004	2	3	2	2	2	12	11	9	499	125	95	459
DrJava	rev4989-5019	11	1	1	11	1	1	1	25	160	2	2	130
DrJava	rev4989-5044	23	1	0	26	0	1	0	51	265	2	0	367
DrJava	rev4989-5058	5	1	0	4	0	1	0	13	133	4	0	131
DrJava	rev5319-5330	2	2	2	2	2	4	4	15	236	10	10	167
DrJava	rev5319-5332	1	1	1	1	1	3	3	12	165	8	8	136
FreeCol	rev5884-5962	5	3	2	4	2	7	6	4	363	17	15	155
FreeCol	rev5884-6055	9	4	4	8	4	8	8	12	1094	328	328	492
FreeCol	rev5884-6110	8	2	2	6	2	9	9	16	1121	621	621	724
FreeCol	rev5884-6265	12	4	3	12	3	15	14	28	1766	684	680	1257
FreeCol	rev5884-6362	20	8	6	19	6	45	24	43	2630	863	806	1494
FreeCol	rev5884-6440	72	56	55	66	55	243	223	245	5784	2383	2330	4319
FreeCol	rev5884-6616	119	97	96	110	96	403	383	436	6358	2788	2735	5610
FreeCol	rev5884-6672	123	99	98	115	98	404	384	442	6884	3044	2991	5848
FreeCol	rev5884-6742	124	103	101	116	101	431	405	440	7985	3230	3162	6118
FreeCol	rev5884-6843	134	106	104	120	104	438	412	452	9229	3494	3426	6519
GenealogyJ	rev5531-5561	1	0	0	0	0	0	0	0	3	0	0	0
GenealogyJ	rev5531-5725	3	3	3	3	3	5	5	5	89	75	75	89
GenealogyJ	rev5537-5610	28	20	20	12	20	23	23	16	556	188	188	134
GenealogyJ	rev5537-5673	28	20	20	12	20	23	23	16	556	188	188	134
GenealogyJ	rev5676-6013	1	0	0	1	0	0	0	1	3	0	0	27

Continued on next page

		conflicting files				conflicts				conflicting lines			
Project	Revision	TM	SM	CM	SSM	TM	SM	CM	SSM	TM	SM	CM	SSM
GenealogyJ	rev5676-6125	1	0	0	1	0	0	0	1	3	0	0	27
GenealogyJ	rev6127-6244	2	1	1	3	1	1	1	3	30	16	16	95
GenealogyJ	rev6127-6310	4	2	2	5	2	2	2	5	56	38	38	151
GenealogyJ	rev6127-6410	4	1	1	5	1	1	1	5	94	22	22	189
GenealogyJ	rev6127-6531	5	1	1	6	1	1	1	6	106	22	22	201
iText	rev2818-3036	24	20	17	23	17	65	60	51	2897	569	451	1613
iText	rev2818-3191	178	155	155	157	155	1834	1834	652	115577	6284	6284	36374
iText	rev2818-3306	234	204	204	204	204	2097	2097	787	133913	6943	6943	41102
iText	rev2818-3392	249	230	230	204	230	2263	2263	802	138768	7791	7791	42509
iText	rev2818-3560	261	226	226	208	226	2098	2098	796	142685	7391	7391	45099
iText	rev2818-3625	264	243	243	200	243	2358	2358	746	136078	7825	7825	42231
iText	rev2818-3988	252	225	225	201	225	2059	2059	774	135368	7794	7794	39204
iText	rev2818-4022	235	218	218	187	218	2005	2005	725	132585	7067	7067	35976
jEdit	rev4676-4998	2	0	0	1	0	0	0	1	28	0	0	2
jEdit	rev16588-16755	2	2	2	2	2	2	2	2	9	7	7	9
jEdit	rev16588-16883	2	1	1	2	1	1	1	2	9	2	2	9
jEdit	rev16588-17060	2	2	2	2	2	2	2	3	24	4	4	24
jEdit	rev16588-17316	2	2	2	2	2	2	2	3	20	4	4	20
jEdit	rev16588-17492	2	2	2	2	2	2	2	3	20	4	4	20
jEdit	rev16588-17551	2	2	2	2	2	2	2	3	20	4	4	20
jEdit	rev16883-16964	6	4	4	6	4	7	7	14	192	100	100	305
Jmol	rev11338-11438	4	3	2	3	2	3	2	7	350	20	10	131
Jmol	rev11338-11538	9	6	5	7	5	12	11	21	521	67	57	269
Jmol	rev11338-11638	15	8	7	10	7	17	16	25	667	78	68	344
Jmol	rev11338-11738	23	13	12	18	12	35	34	43	1070	203	193	493
Jmol	rev11338-11838	34	19	18	27	18	46	45	58	1637	333	323	915
Jmol	rev11338-11938	35	20	19	29	19	48	47	67	1703	344	334	998
Jmol	rev11338-12038	42	22	22	36	22	53	53	79	1869	593	593	1115
PMD	rev5929-6010	1	0	0	1	0	0	0	1	5	0	0	5
PMD	rev5929-6135	3	2	2	3	2	2	2	3	22	17	17	24
PMD	rev5929-6198	7	3	3	7	3	4	4	13	784	342	342	359
PMD	rev5929-6296	9	5	5	10	5	6	6	16	734	349	349	450
PMD	rev5929-6425	9	5	5	9	5	6	6	15	727	349	349	439
PMD	rev5929-6595	12	9	9	12	9	12	12	19	670	462	462	478
PMD	rev5929-6700	13	11	11	13	11	15	15	20	1003	539	539	767
PMD	rev5929-6835	12	10	10	12	10	16	16	19	939	506	506	721
PMD	rev5929-7018	16	12	12	15	12	18	18	22	1053	510	510	737
PMD	rev5929-7073	17	13	13	16	13	19	19	23	977	498	498	730
SQuirrelSQL	rev4007-4051	1	0	0	0	0	0	0	0	4	0	0	0
SQuirrelSQL	rev4007-4103	3	1	1	2	1	1	1	3	299	7	7	33
SQuirrelSQL	rev4007-4212	11	6	6	9	6	8	8	19	1384	108	108	303
SQuirrelSQL	rev4007-4321	17	7	7	14	7	9	9	27	3862	164	164	546
SQuirrelSQL	rev4007-4394	21	10	10	18	10	12	12	32	4218	357	357	824
SQuirrelSQL	rev4007-4516	25	12	12	21	12	15	15	35	4258	414	414	871
SQuirrelSQL	rev4007-4908	33	13	13	27	13	18	18	53	6111	420	420	1468
SQuirrelSQL	rev4007-5081	34	15	15	28	15	20	20	55	4574	615	615	1482
SQuirrelSQL	rev5082-5155	2	1	1	2	1	1	1	3	17	50	50	37
SQuirrelSQL	rev5082-5351	4	3	3	3	3	3	3	4	194	9	9	61

Table A.1.: Conflicts of all projects

A.2. Runtime

		runtime			
Project	Revision	TM	SM	CM	SSM
DrJava	rev3734-3786	5556	95582	39398	470021
DrJava	rev3734-3788	5557	100648	44350	491475
DrJava	rev3734-3807	5663	69549	11692	517842
DrJava	rev4989-5004	6029	95760	22869	606357
DrJava	rev4989-5019	6186	83276	13749	559721
DrJava	rev4989-5044	6278	81310	16619	532562
DrJava	rev4989-5058	6163	81592	14812	523317
DrJava	rev5319-5330	6158	145662	68712	598992
DrJava	rev5319-5332	6150	145734	70869	559674
Freecol	rev5884-5962	5160	74649	7307	446411
Freecol	rev5884-6055	5285	75085	9579	449462
Freecol	rev5884-6110	5276	75268	9600	449972
Freecol	rev5884-6265	5355	75738	10803	458225
Freecol	rev5884-6362	5458	78238	13333	455063
Freecol	rev5884-6440	5924	87671	34643	449400
Freecol	rev5884-6616	6157	89137	47010	446345
Freecol	rev5884-6672	6205	90245	48095	452306
Freecol	rev5884-6742	6220	91477	48572	567253
Freecol	rev5884-6843	6219	87880	49211	565195
GenealogyJ	rev5531-5561	4017	54710	4513	333400
GenealogyJ	rev5531-5725	3950	52353	4728	322092
GenealogyJ	rev5537-5610	4108	54049	9364	326318
GenealogyJ	rev5537-5673	4032	52881	9402	323886
GenealogyJ	rev5676-6013	3929	52942	4310	319795
GenealogyJ	rev5676-6125	3848	52995	4323	320054
GenealogyJ	rev6127-6244	4445	61138	5955	363692
GenealogyJ	rev6127-6310	4455	61860	6218	361185
GenealogyJ	rev6127-6410	4504	61223	5794	351949
GenealogyJ	rev6127-6531	4451	59632	6546	354315
iText	rev2818-3036	4501	105082	18275	426837
iText	rev2818-3191	5949	135344	130619	334554
iText	rev2818-3306	7206	366823	345364	401390
iText	rev2818-3392	7851	406303	389471	436913
iText	rev2818-3560	7984	430900	426416	447347
iText	rev2818-3625	7929	411803	404812	450984
iText	rev2818-3988	7790	387069	404926	461890
iText	rev2818-4022	8091	463090	407869	462895
jEdit	rev4676-4998	3584	49316	4033	324090
jEdit	rev16588-16755	5405	76507	6840	478027
jEdit	rev16588-16883	5411	75910	6811	476061
jEdit	rev16588-17060	5420	76301	6921	476359
jEdit	rev16588-17316	5377	75085	6926	483460
jEdit	rev16588-17492	5375	76129	6876	481671
jEdit	rev16588-17551	5381	77070	6932	479139
jEdit	rev16883-16964	5435	73973	7359	474847
Jmol	rev11338-11438	4721	299620	211114	551740
Jmol	rev11338-11538	4786	297964	213165	553854
Jmol	rev11338-11638	4893	288328	208357	558043
Jmol	rev11338-11738	4970	281843	203392	554943
Jmol	rev11338-11838	5068	329436	270839	560149
Jmol	rev11338-11938	5080	344811	277888	559467
Jmol	rev11338-12038	5041	355751	289234	559244
PMD	rev5929-6010	4155	42865	4371	245628
PMD	rev5929-6135	4136	43667	4875	241109
PMD	rev5929-6198	3407	34931	4635	194273
PMD	rev5929-6296	3360	33896	4833	194605
PMD	rev5929-6425	2842	29597	4640	173048
PMD	rev5929-6595	2814	29879	5373	170519
PMD	rev5929-6700	2836	30182	5507	170600

Continued on next page

		runtime			
Project	Revision	TM	SM	CM	SSM
PMD	rev5929-6835	2775	30094	5419	169000
PMD	rev5929-7018	2793	28456	6169	170862
PMD	rev5929-7073	2817	30117	6400	171152
SQuirrelSQL	rev4007-4051	16938	215394	18358	1048074
SQuirrelSQL	rev4007-4103	16968	222290	19273	1047812
SQuirrelSQL	rev4007-4212	16831	213916	21380	1025056
SQuirrelSQL	rev4007-4321	16906	216931	23217	1022810
SQuirrelSQL	rev4007-4394	16902	212638	23996	1022106
SQuirrelSQL	rev4007-4516	16955	224109	25483	1024647
SQuirrelSQL	rev4007-4908	16644	208173	26564	1008498
SQuirrelSQL	rev4007-5081	16524	209951	26406	989299
SQuirrelSQL	rev5082-5155	23351	286927	25940	1416288
SQuirrelSQL	rev5082-5351	23275	292464	27264	1408631

Table A.2.: Runtime in ms of all projects

APPENDIX B

Results: Graphs

B.1. DrJava

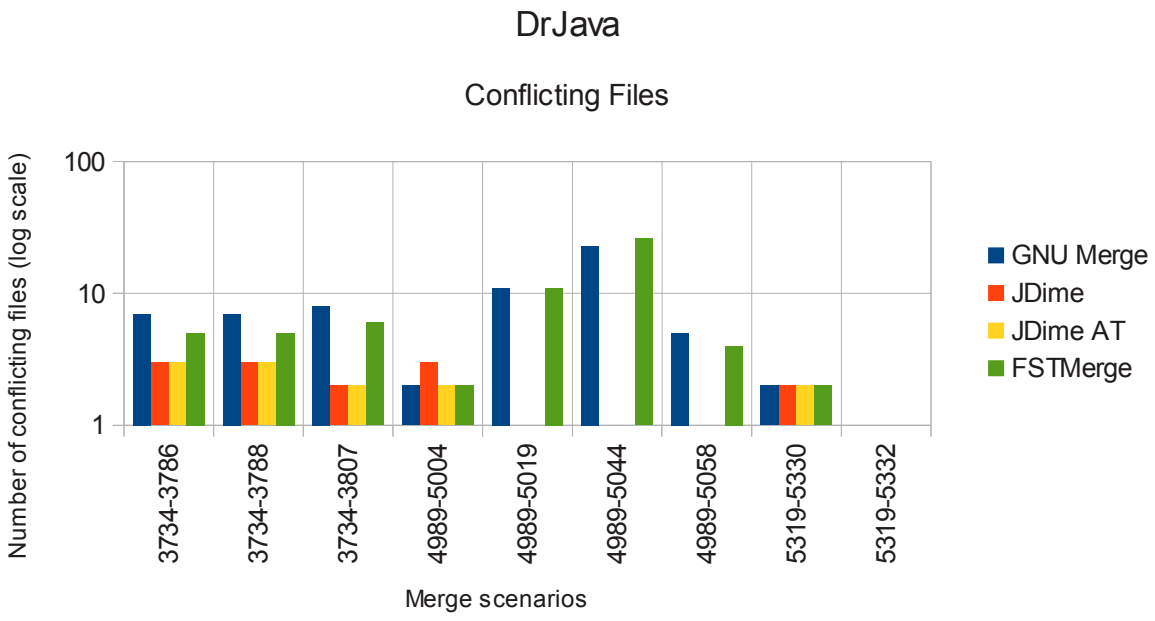


Figure B.1.: DrJava: conflicting files

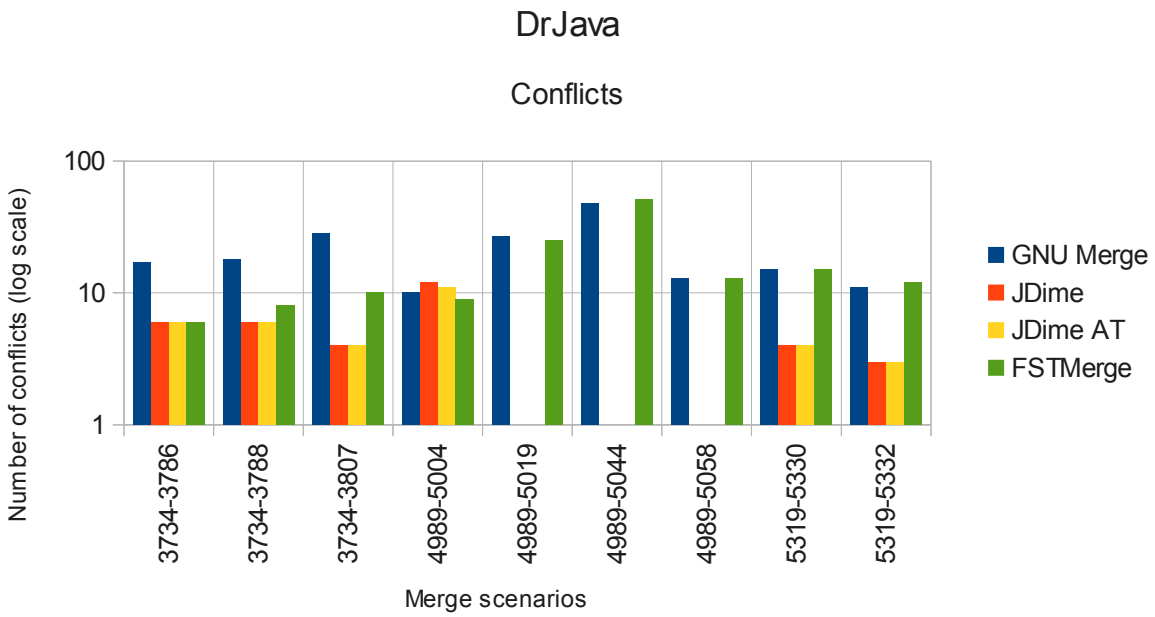


Figure B.2.: DrJava: conflicts

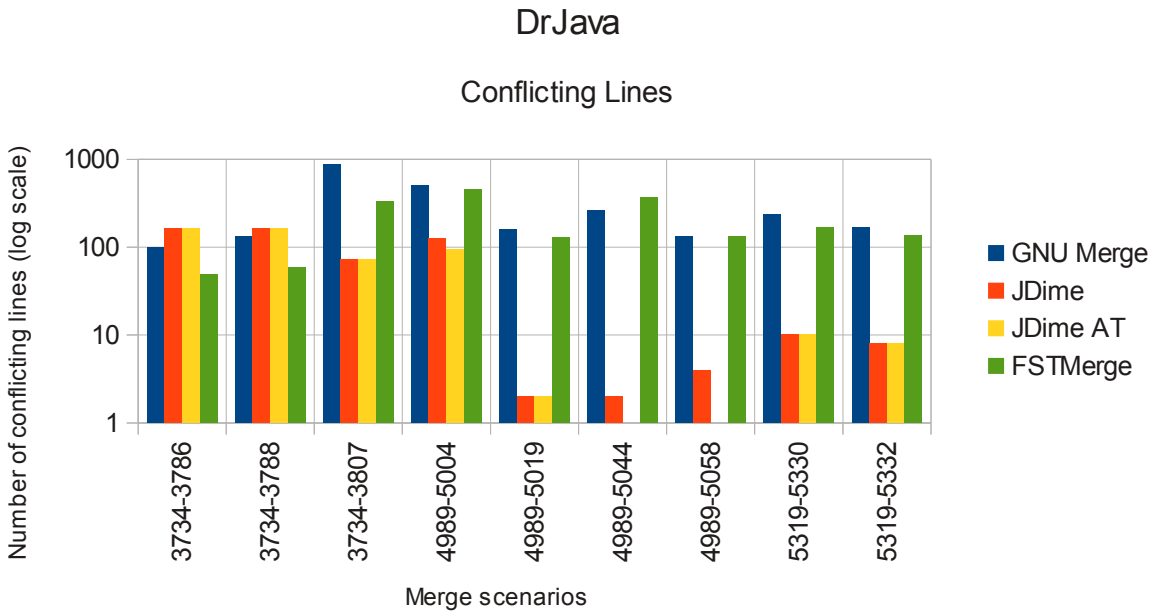


Figure B.3.: DrJava: conflicting lines

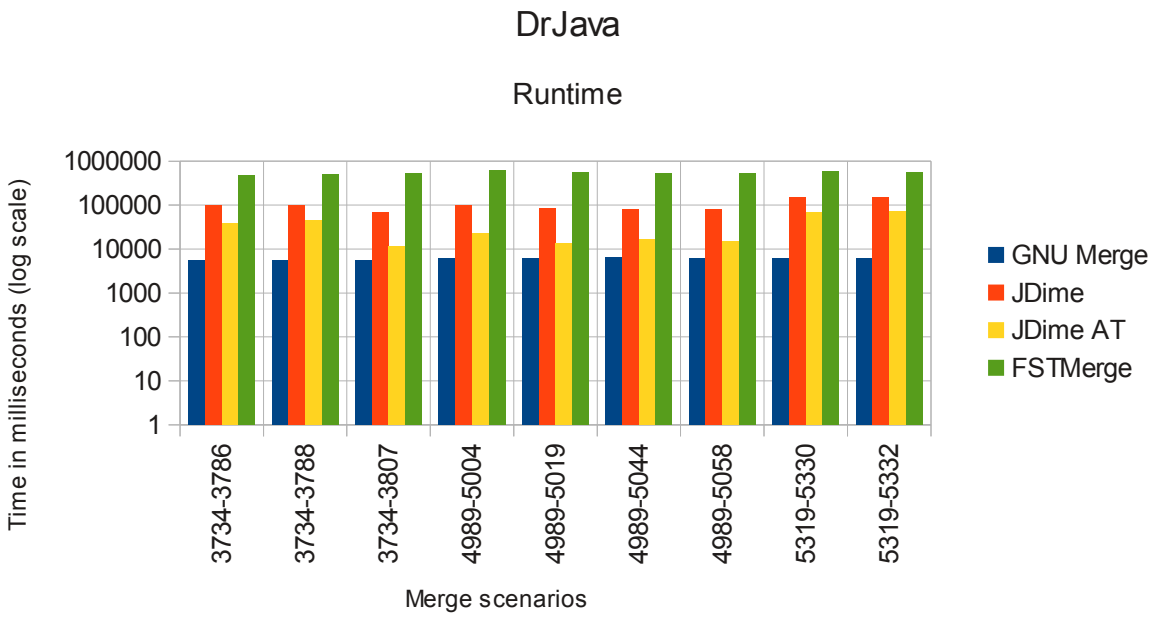


Figure B.4.: DrJava: runtime

B.2. FreeCol

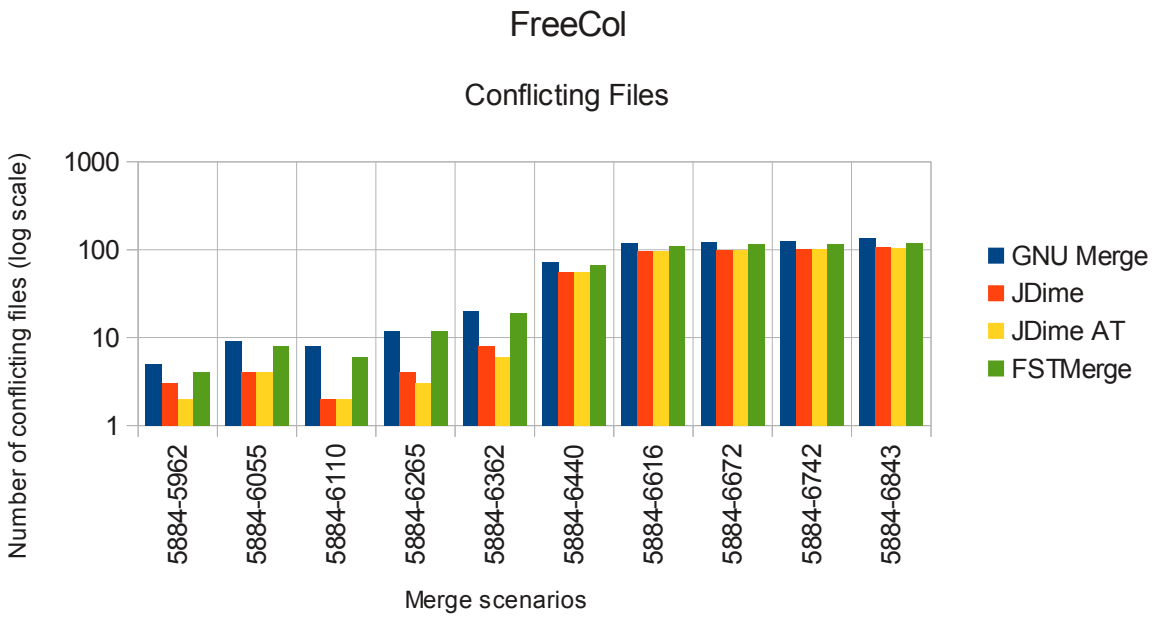


Figure B.5.: FreeCol: conflicting files

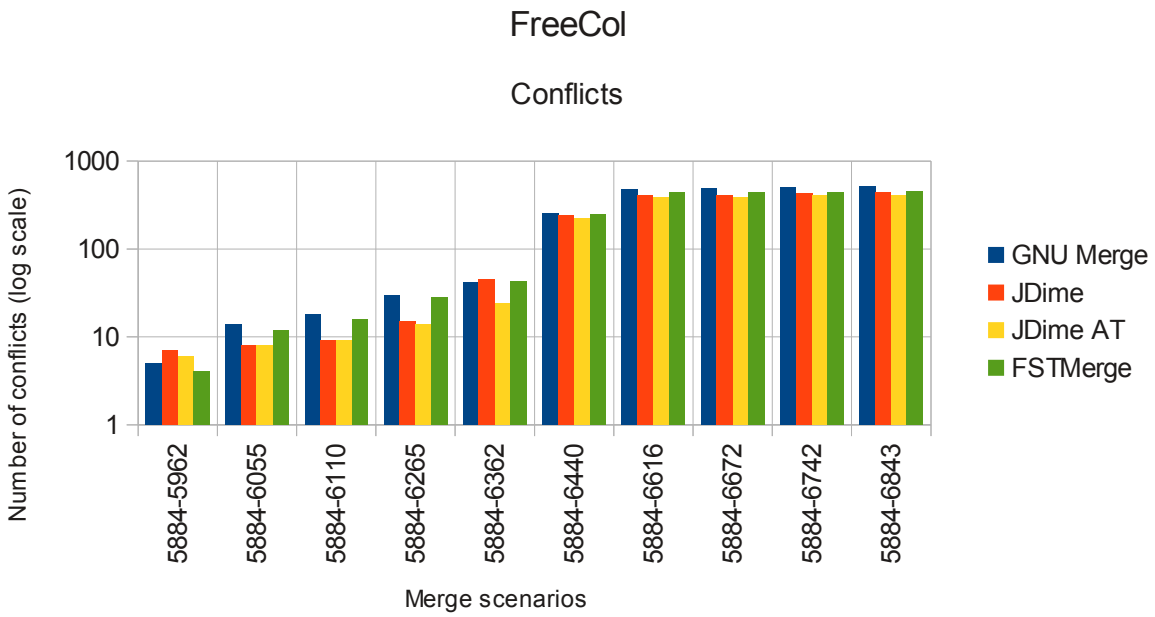


Figure B.6.: FreeCol: conflicts

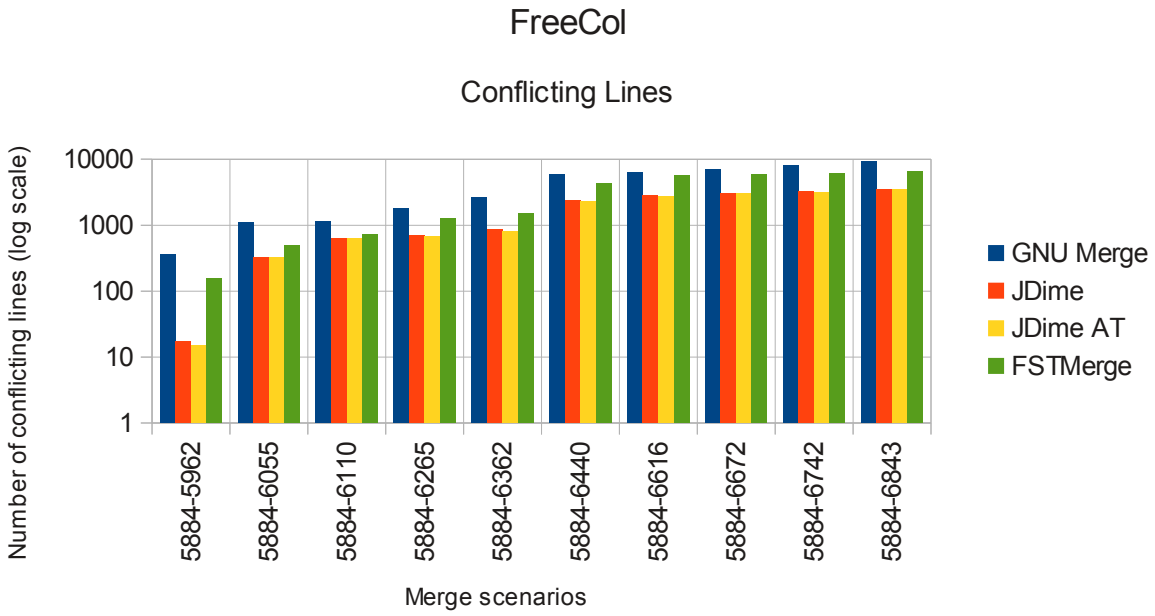


Figure B.7.: FreeCol: conflicting lines

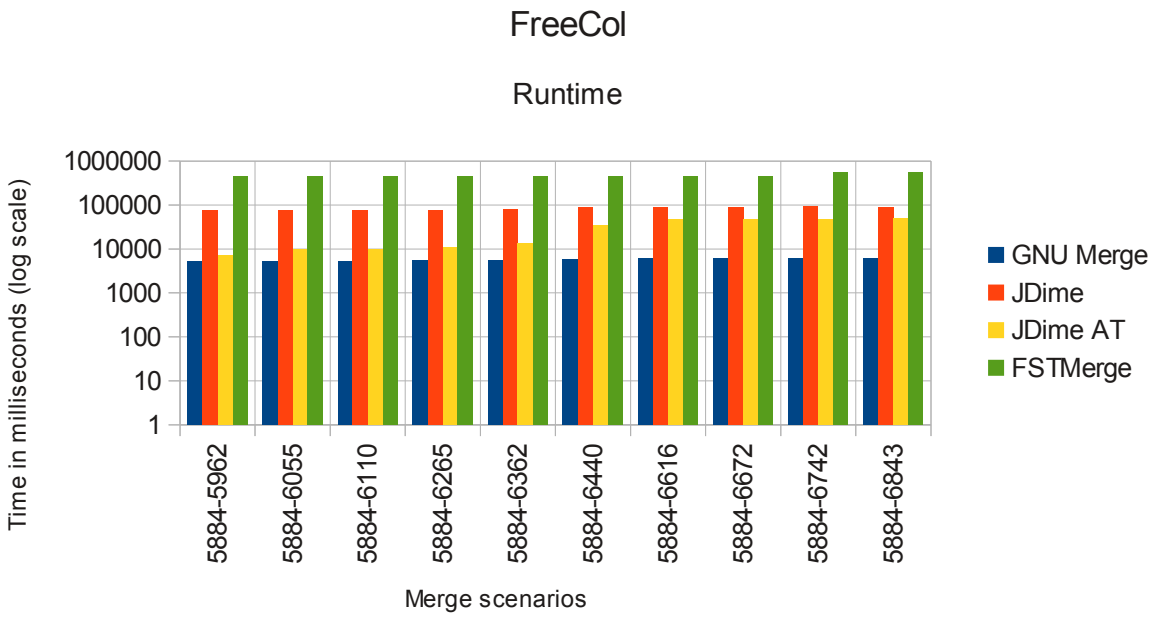


Figure B.8.: FreeCol: runtime

B.3. GenealogyJ

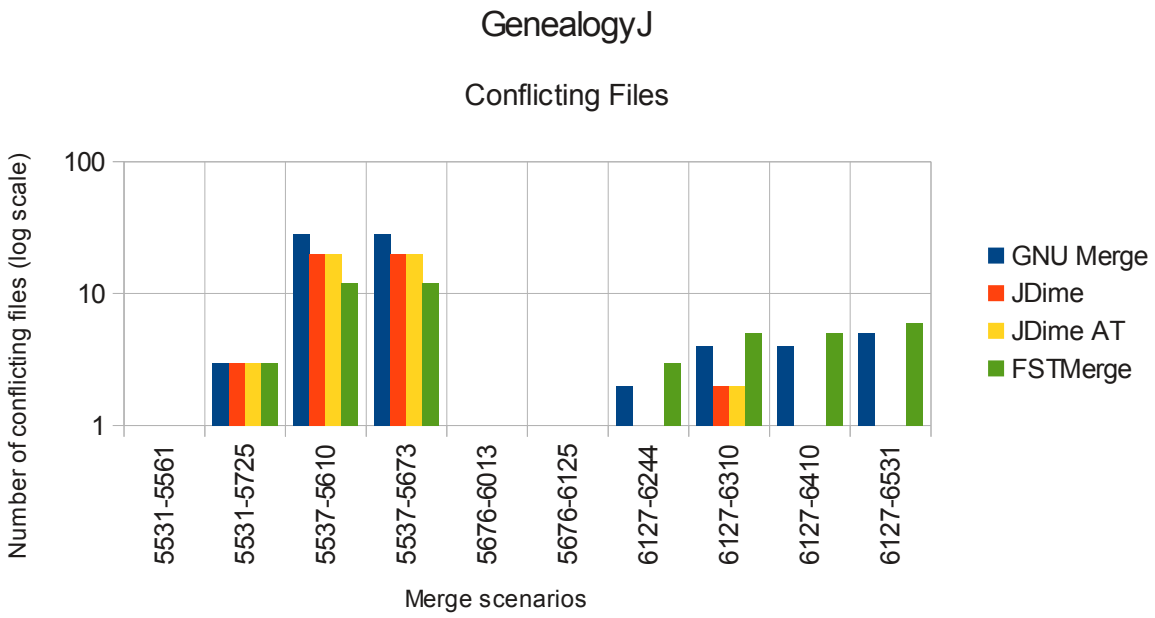


Figure B.9.: GenealogyJ: conflicting files

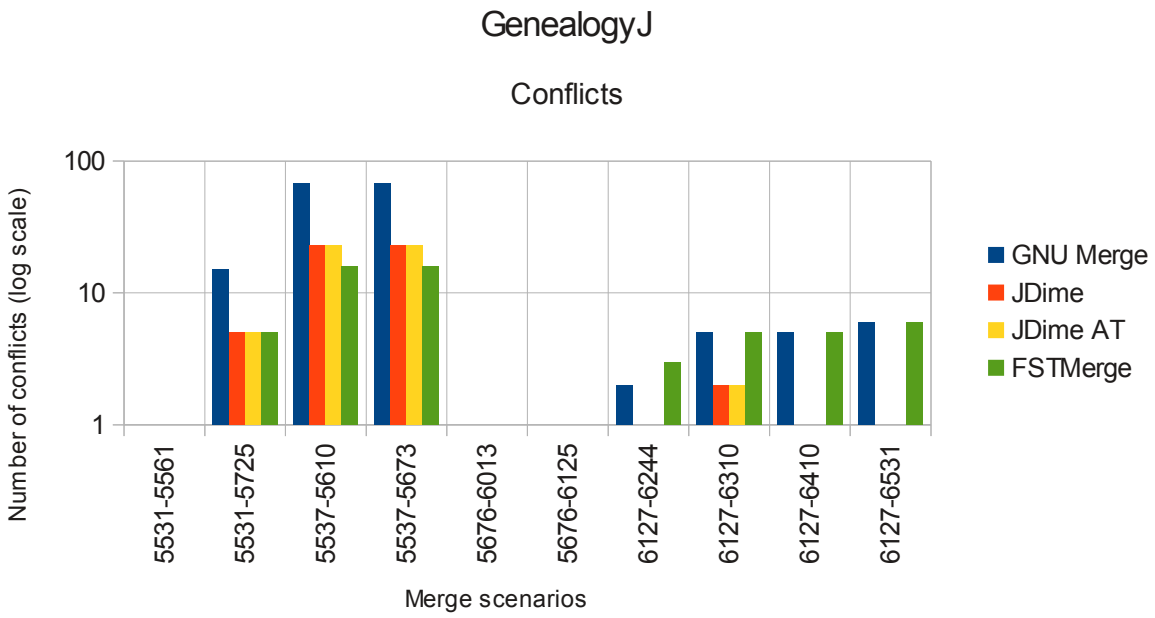


Figure B.10.: GenealogyJ: conflicts

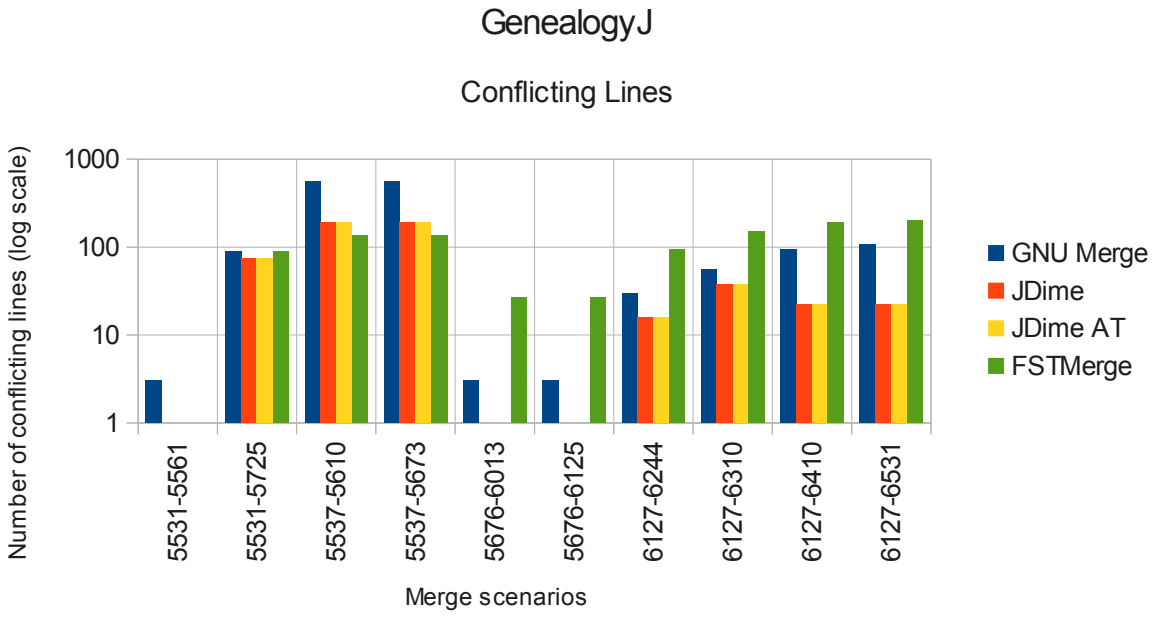


Figure B.11.: GenealogyJ: conflicting lines

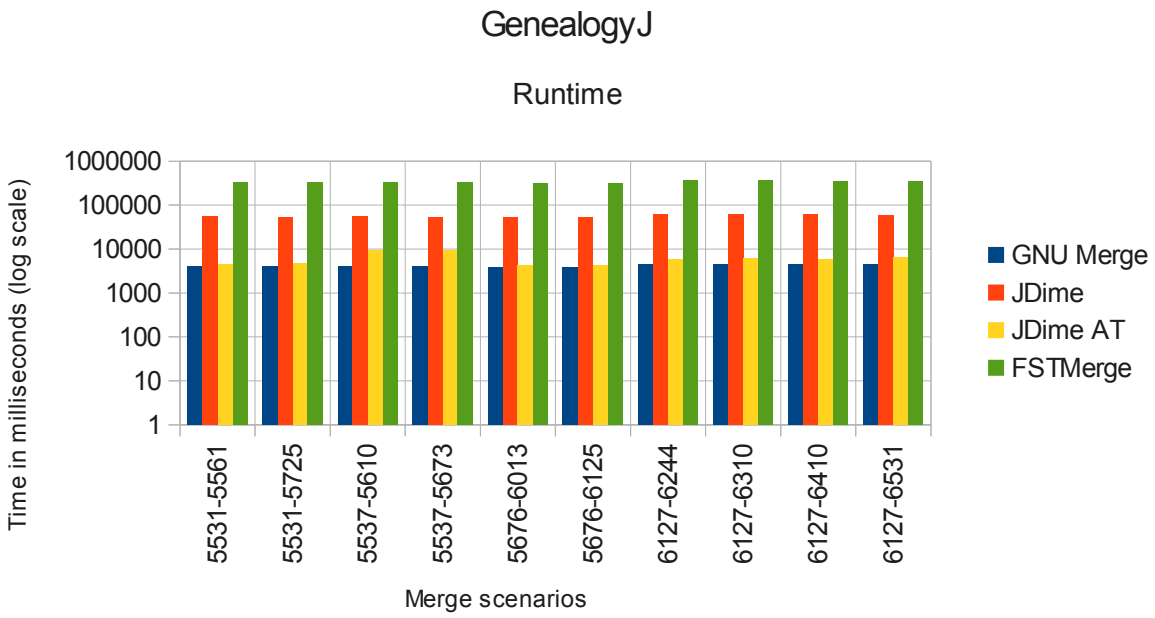


Figure B.12.: GenealogyJ: runtime

B.4. iText

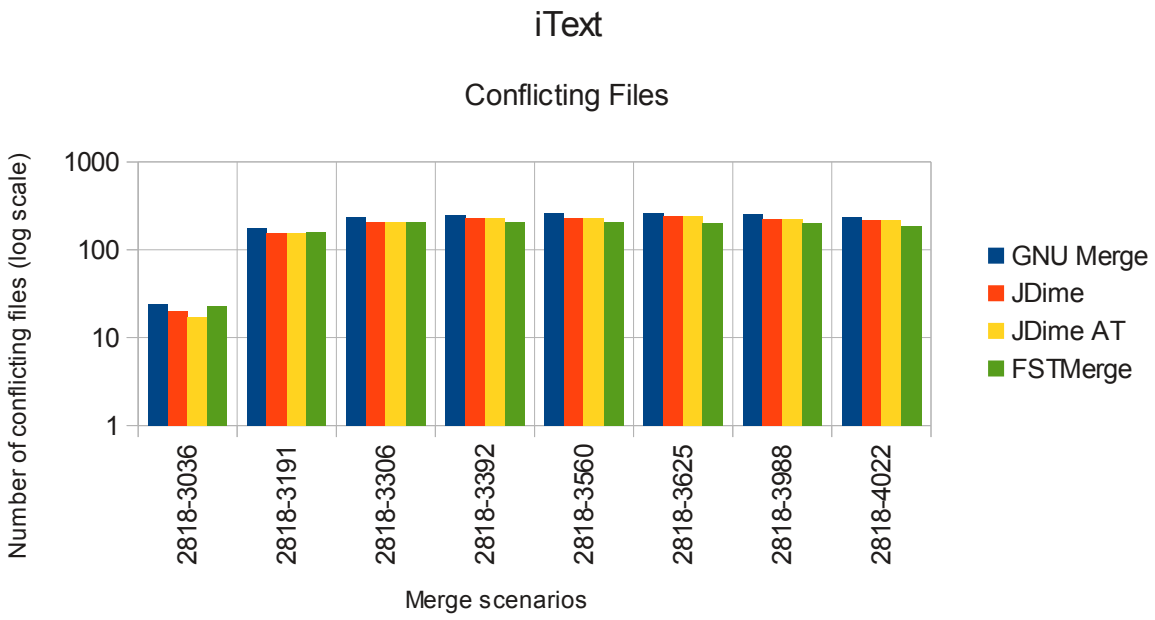


Figure B.13.: iText: conflicting files

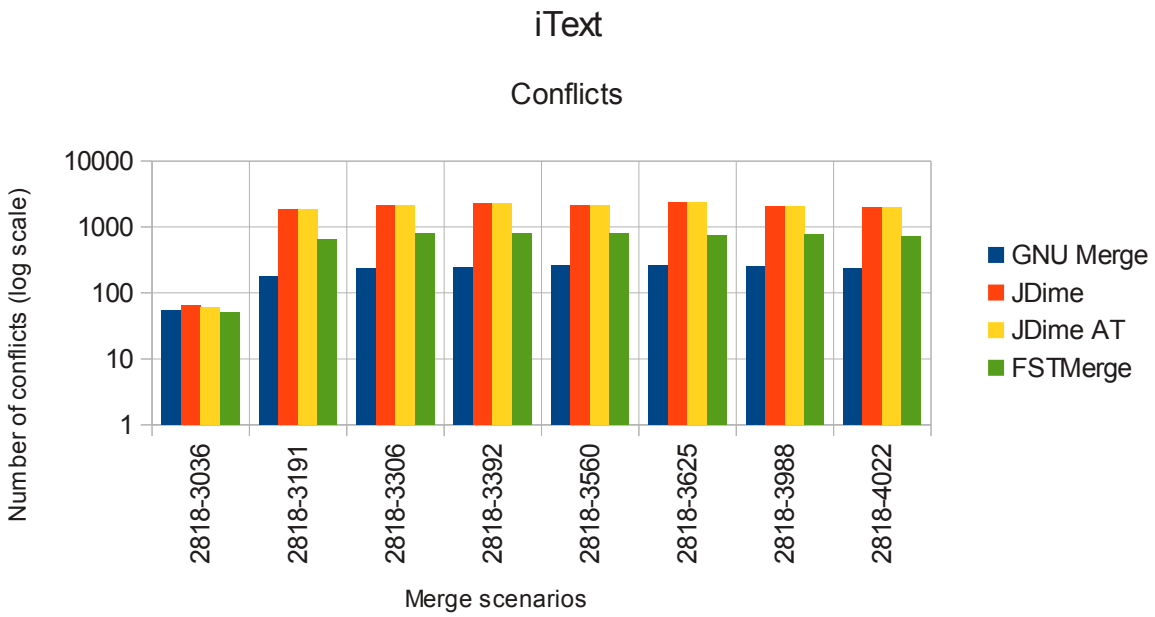


Figure B.14.: iText: conflicts

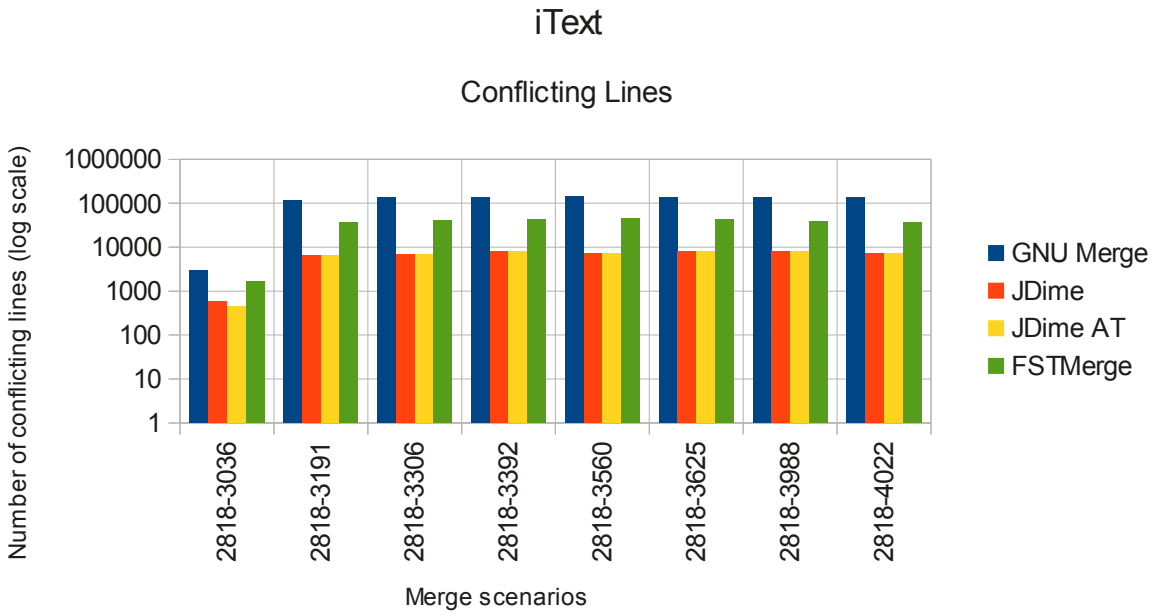


Figure B.15.: iText: conflicting lines

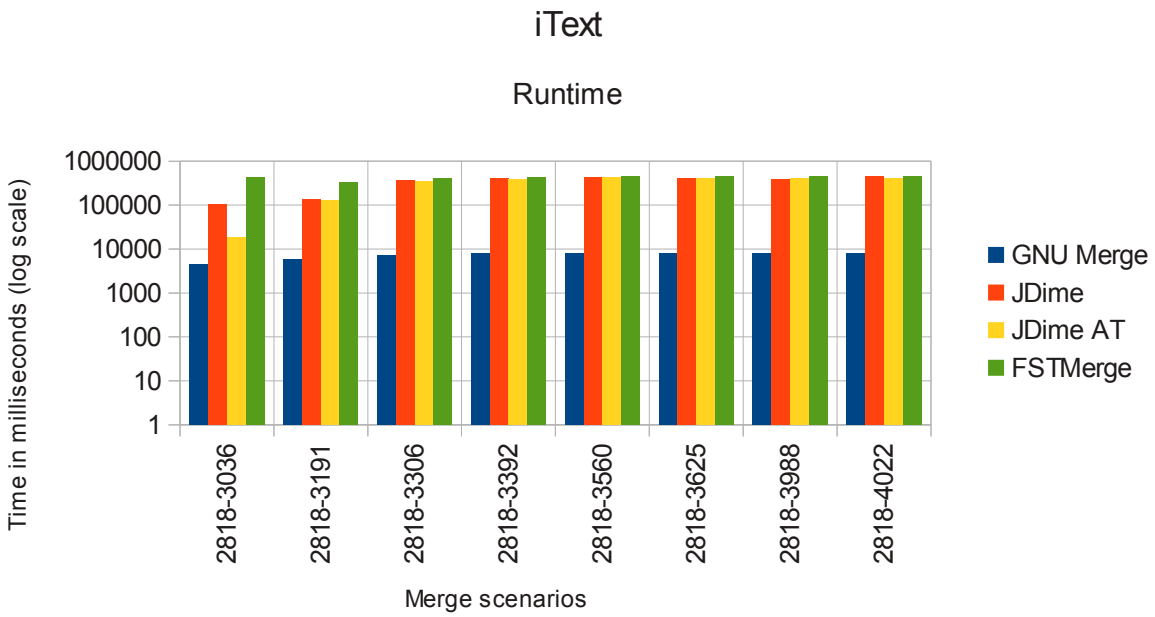


Figure B.16.: iText: runtime

B.5. jEdit

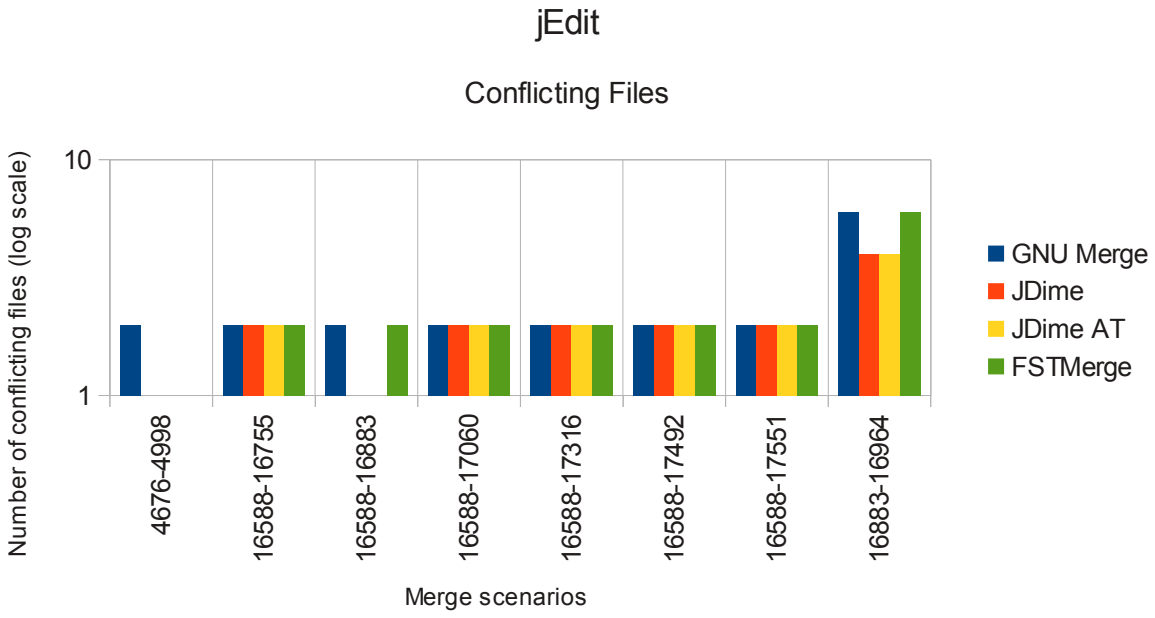


Figure B.17.: jEdit: conflicting files

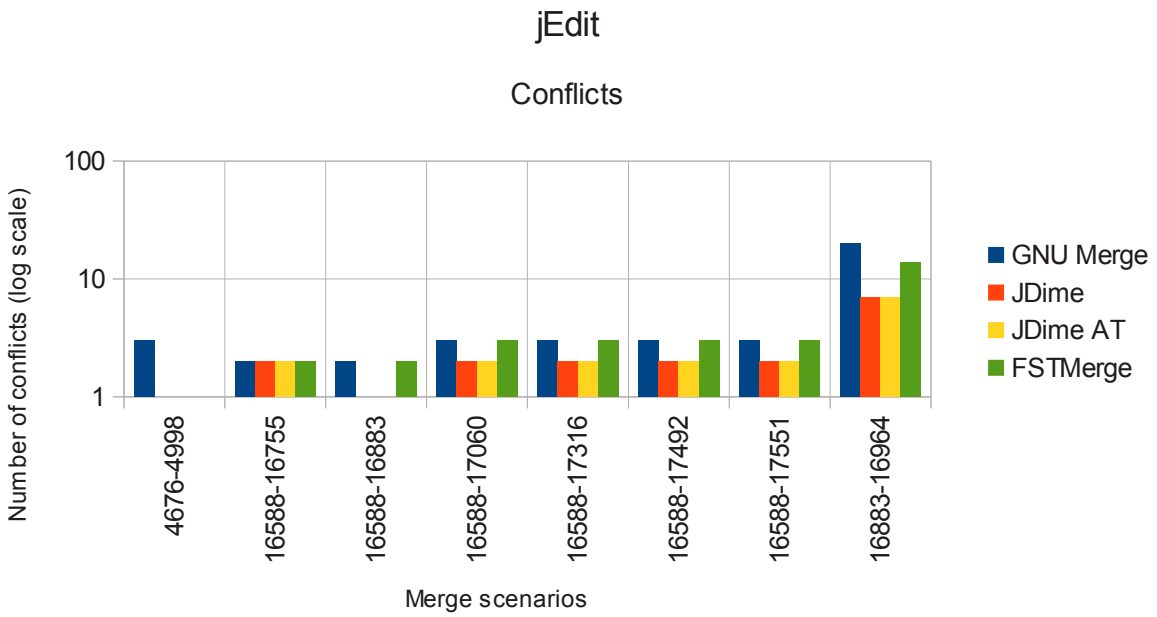


Figure B.18.: jEdit: conflicts

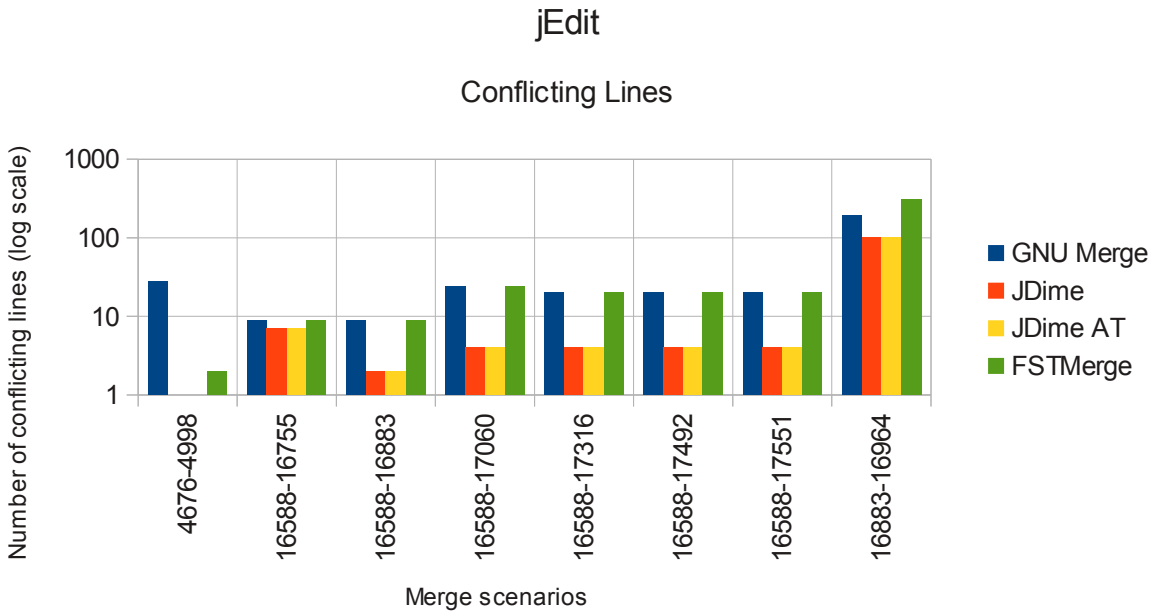


Figure B.19.: jEdit: conflicting lines

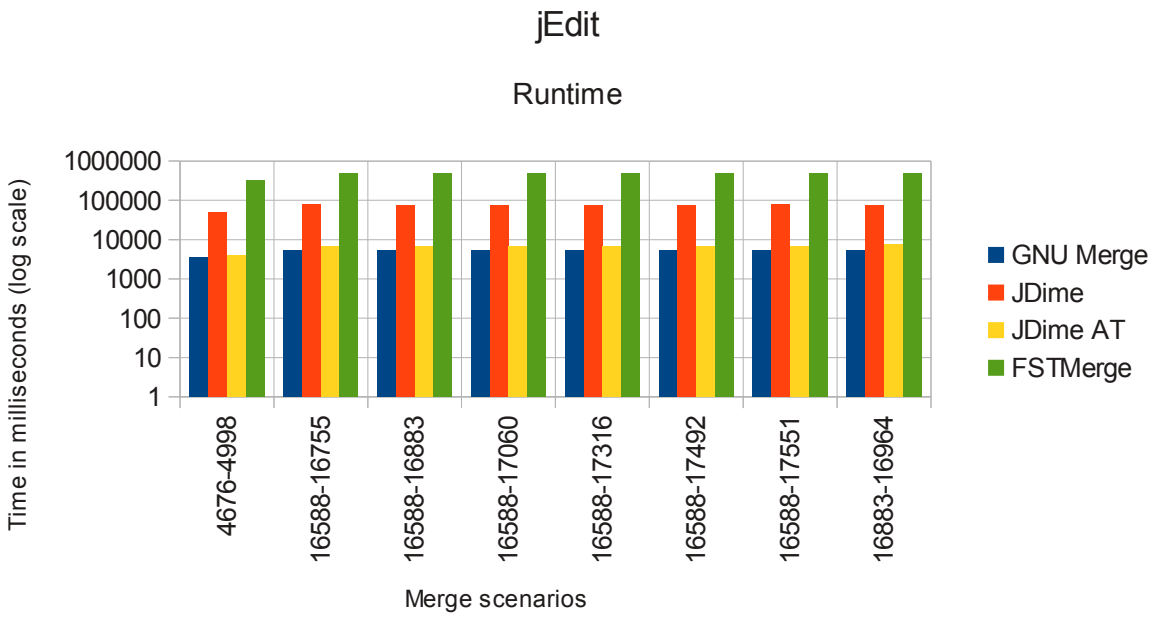


Figure B.20.: jEdit: runtime

B.6. Jmol

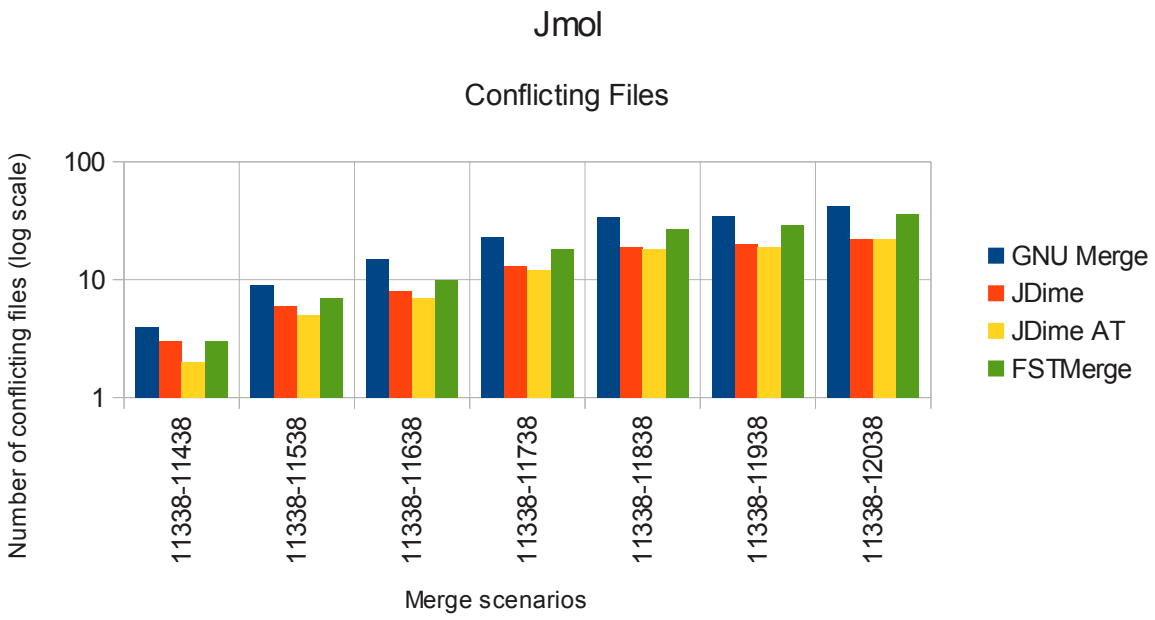


Figure B.21.: Jmol: conflicting files

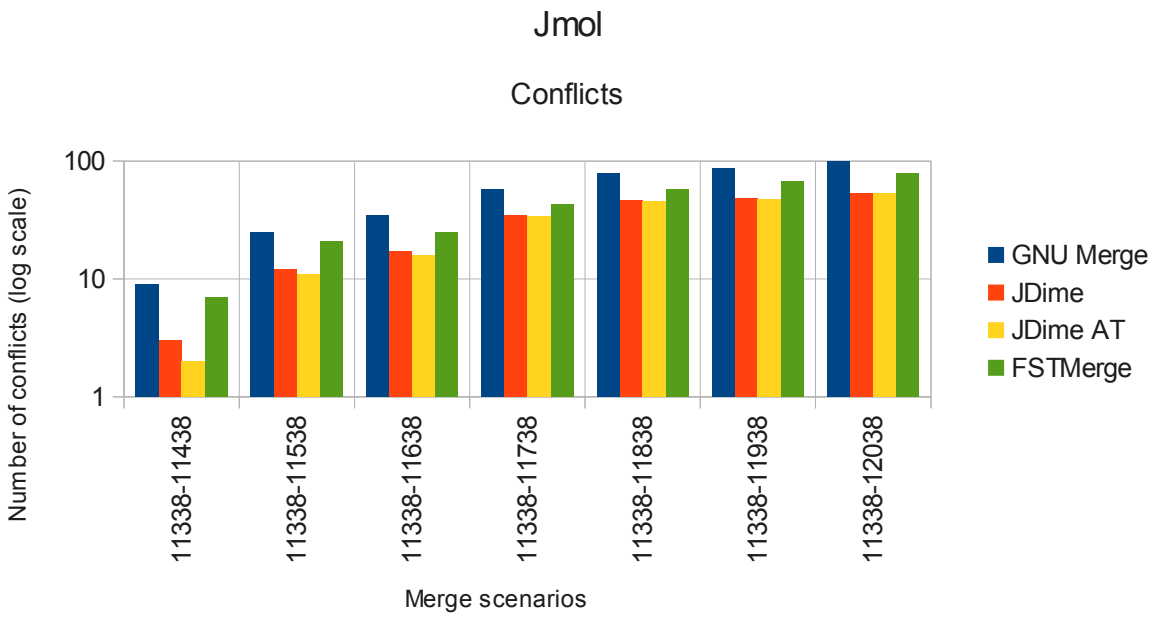


Figure B.22.: Jmol: conflicts

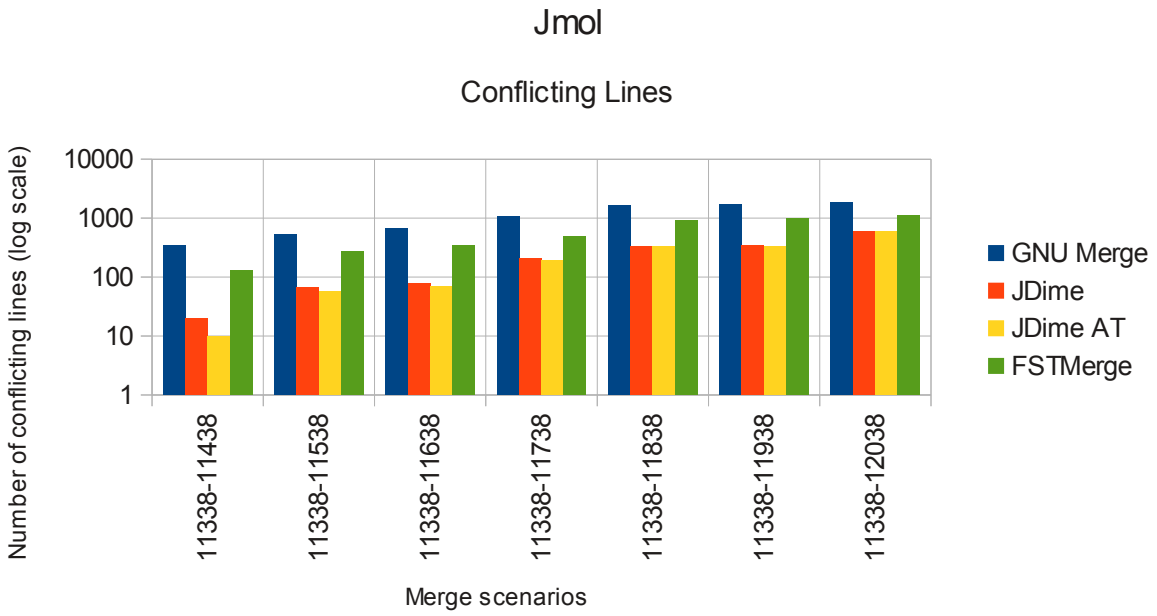


Figure B.23.: Jmol: conflicting lines

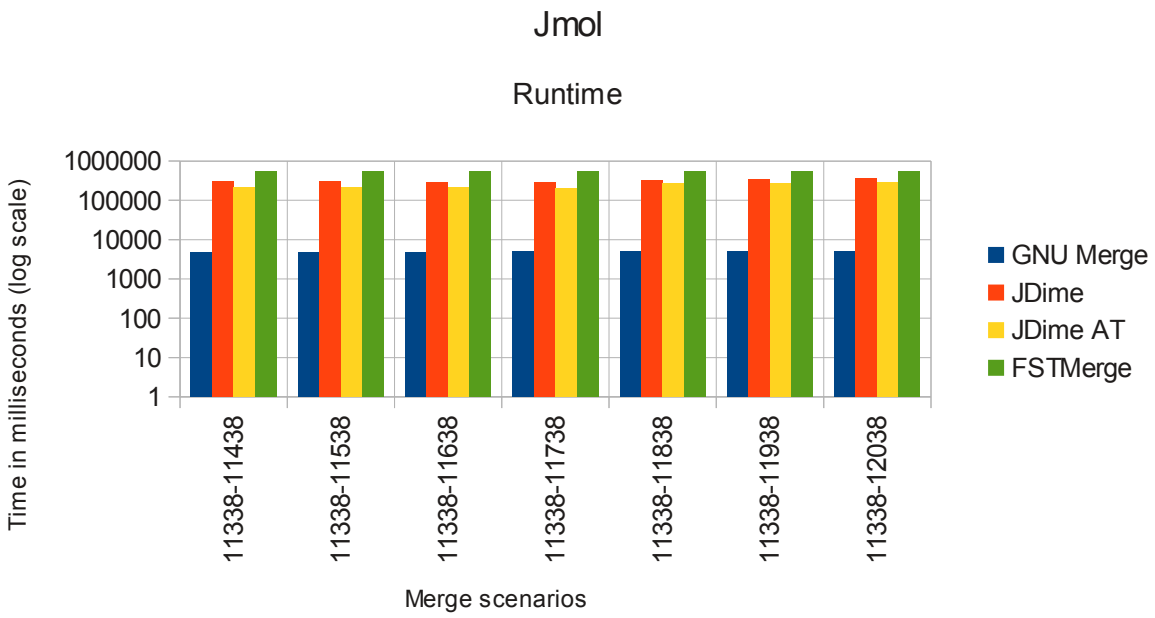


Figure B.24.: Jmol: runtime

B.7. PMD

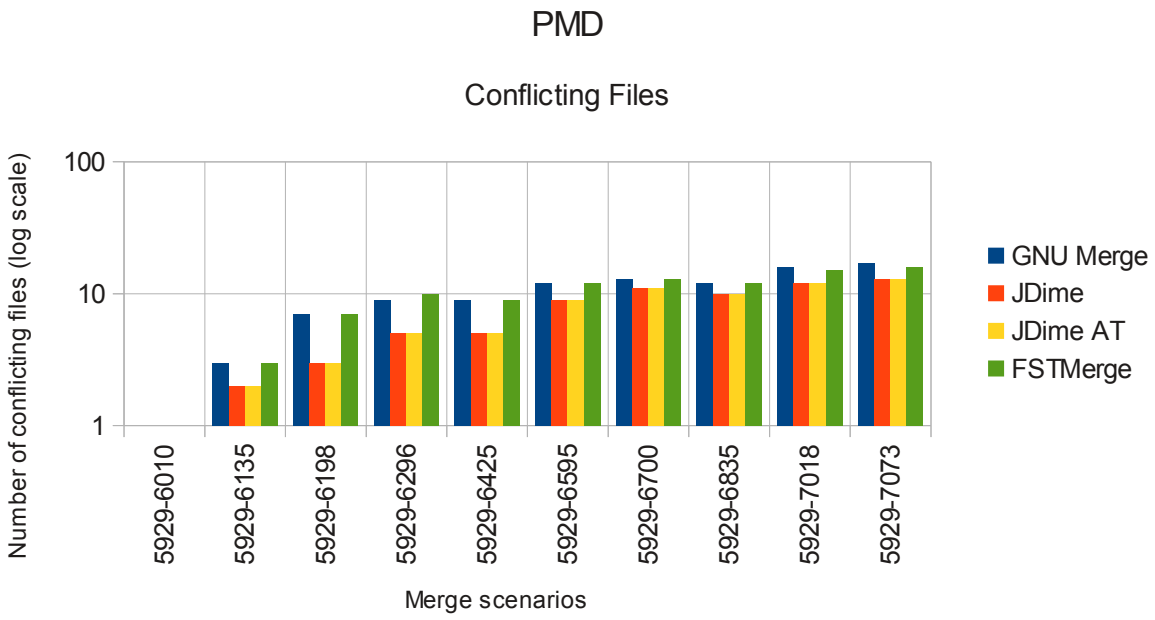


Figure B.25.: PMD: conflicting files

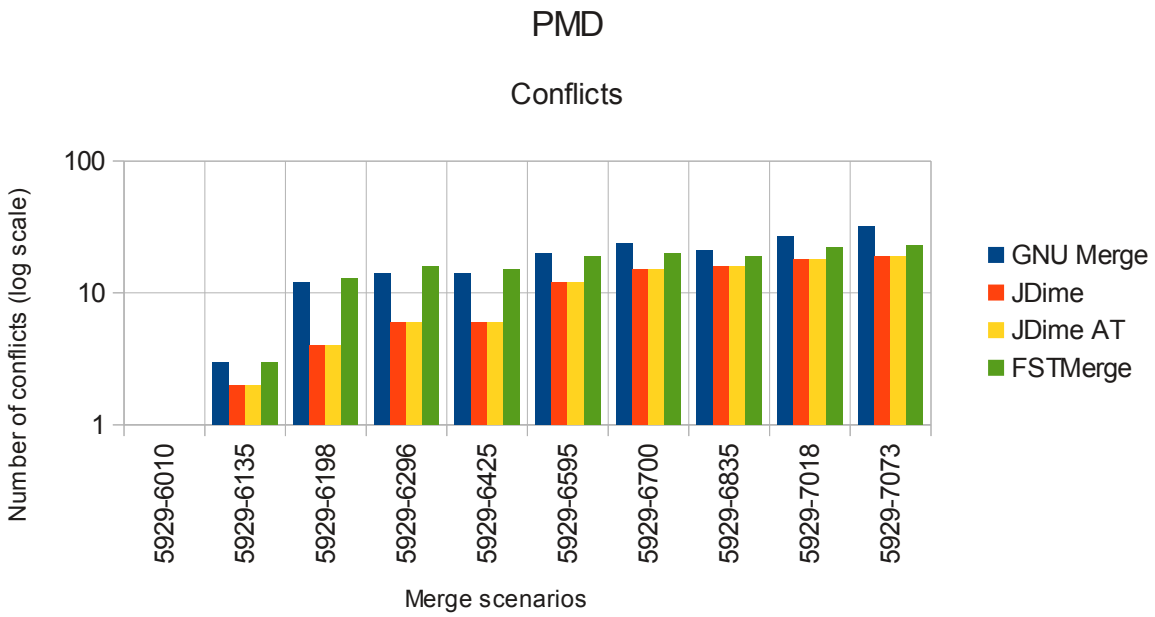


Figure B.26.: PMD: conflicts

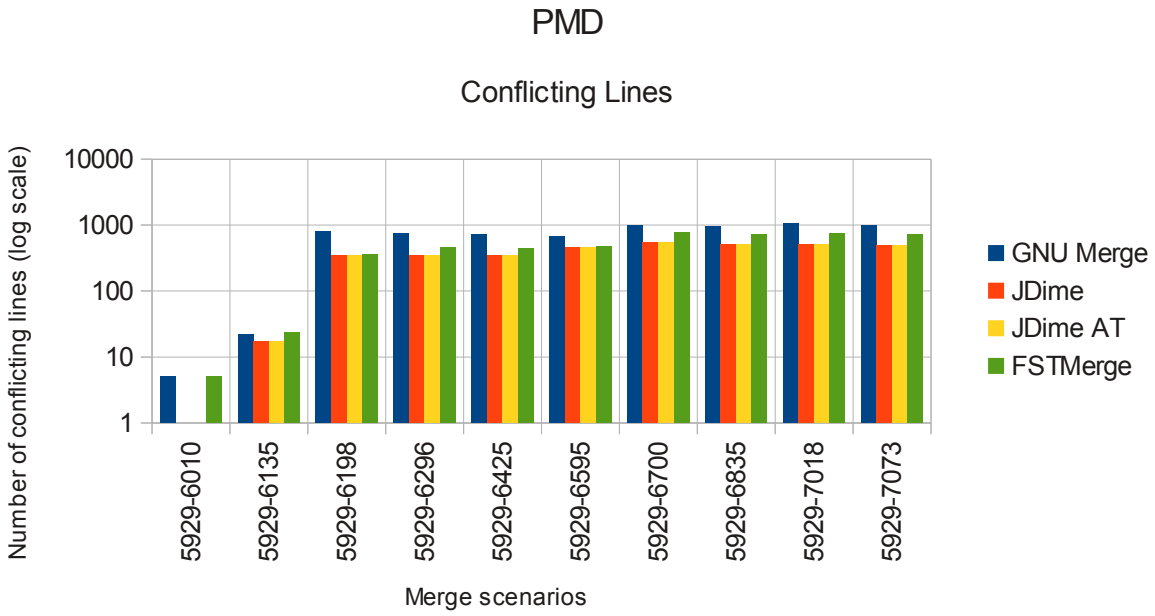


Figure B.27.: PMD: conflicting lines

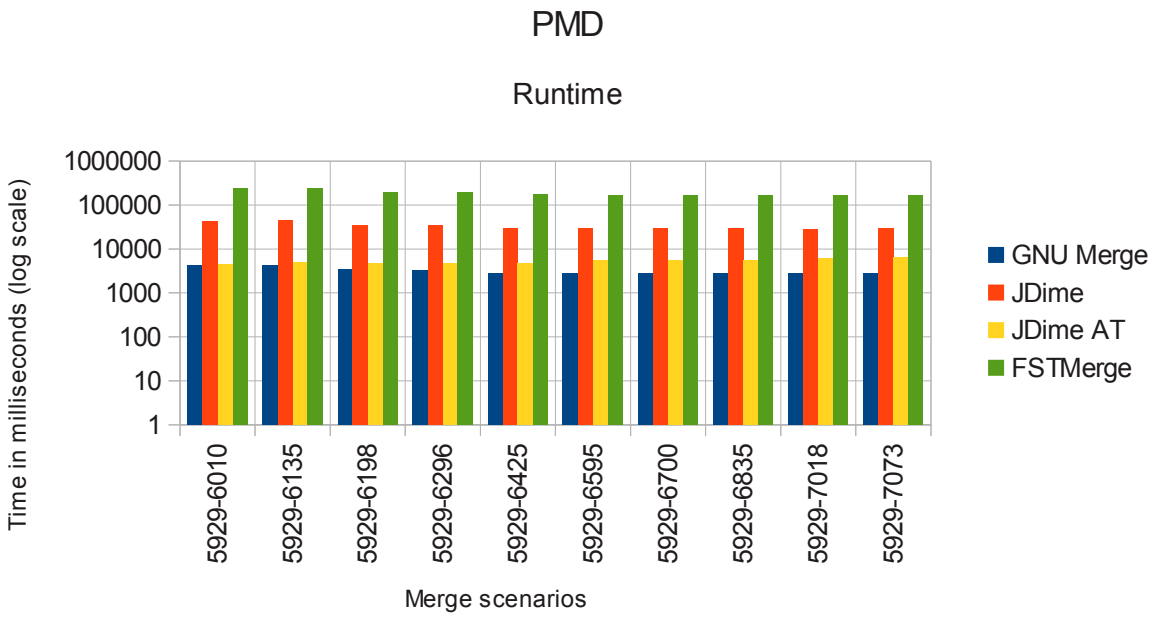


Figure B.28.: PMD: runtime

B.8. SQuirrelSQL

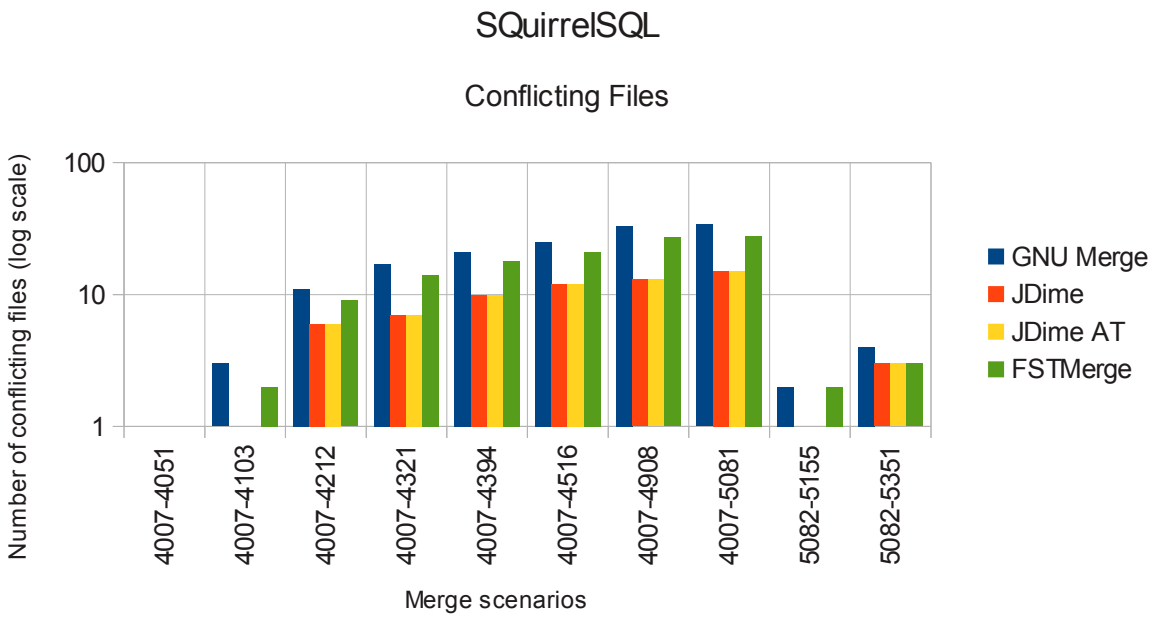


Figure B.29.: SQuirrelSQL: conflicting files

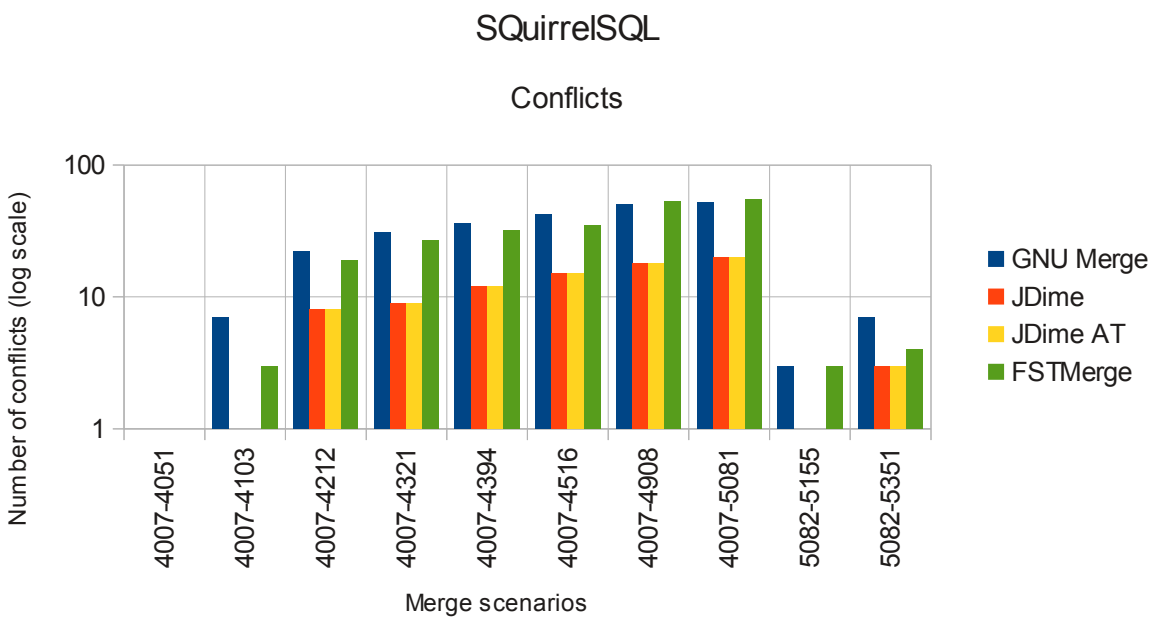


Figure B.30.: SQuirrelSQL: conflicts

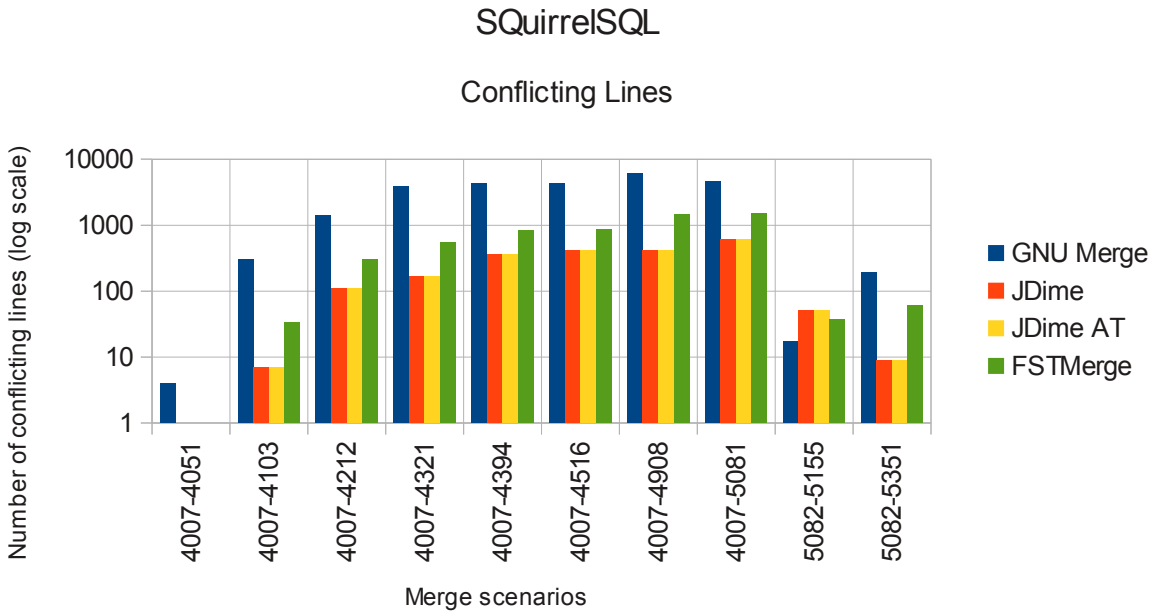


Figure B.31.: SQuirrelSQL: conflicting lines

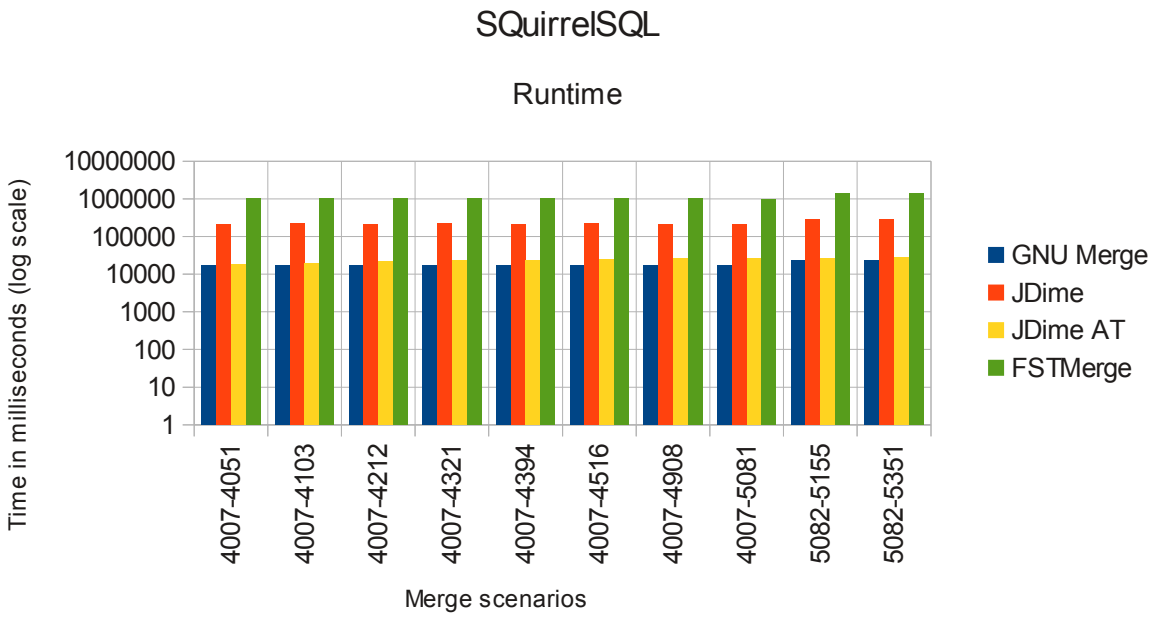


Figure B.32.: SQuirrelSQL: runtime

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich habe die Arbeit nicht in gleicher oder ähnlicher Form bei einer anderen Prüfungsbehörde vorgelegt.

Passau, den 27. Februar 2012

. . . . . .
(Olaf Leßenich)