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.
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, Bu95, 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\[\text{OB09}\]. A version control system (VCS) is a tool or a set of tools to automate this process and track the history of a file \[\text{OB09}\].

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

\[\text{1} \text{http://sccs.berlios.de/}\]
\[\text{2} \text{http://www.gnu.org/software/rcs/}\]
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 (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](http://savannah.nongnu.org/projects/cvs)

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.
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.

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[^1], of which the latter was created in the year 2000 and designed to be the successor of CVS.

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.

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.
Adjustable Syntactic Merge of Java Programs

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 \[\text{CSFP10}\]. 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](http://git-scm.com/)

![Figure 2.4.: Lost Update Problem/Blind Overwriting](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.

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.
Adjustable Syntactic Merge of Java Programs

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.
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 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.

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.

<table>
<thead>
<tr>
<th>Type of i</th>
<th>Left Rev A1</th>
<th>Right Rev A2</th>
<th>Base-Rev B</th>
<th>Merge M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Element</td>
<td>$i_B$</td>
<td>$i_{A2}$</td>
<td>$i_B$</td>
<td>$i_{A2}$</td>
</tr>
<tr>
<td>2 Element</td>
<td>$i_{A1}$</td>
<td>$i_{A2}$</td>
<td>$i_B$</td>
<td>Conflict</td>
</tr>
<tr>
<td>3 List</td>
<td>$i_1 \in i_{A1}$</td>
<td>$i_1 \notin i_{A2}$</td>
<td>$i_1 \in i_B$</td>
<td>$i_1 \notin i_M$</td>
</tr>
<tr>
<td>4 List</td>
<td>$i_1 \in i_{A1}$</td>
<td>$i_1 \notin i_{A2}$</td>
<td>$i_1 \notin i_B$</td>
<td>$i_1 \in i_M \lor$ Conflict</td>
</tr>
</tbody>
</table>

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.

![Figure 3.2: GNU MERGE displaying a conflict](image)

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-
Adjustable Syntactic Merge of Java Programs

Figure 3.3.: Failed textual merge

public class Example {
    public static long fib(int n) {
        if (n <= 1) return n;
        else return fib(n-1) + fib(n-2);
    }
    public static long fac(int n) {
        long result = 1;
        for (int i = 1; i <= n; i++)
            result *= i;
        return result;
    }
    public static int square(int n) {
        return n*n;
    }
}

public class Example {
    public static long fac(int n) {
        long result = 1;
        if (n>1)
            for (int i = 2; i <= n; i++)
                result *= i;
        return result;
    }
    public static long fib(int n) {
        if (n <= 1) return n;
        else return fib(n-1) + fib(n-2);
    }


public class Example {
    public static long fac(int n) {
        long result = 1;
        if (n>1)
            for (int i = 2; i <= n; i++)
                result *= i;
        return result;
    }
    public static long fib(int n) {
        if (n <= 1) return n;
        else return fib(n-1) + fib(n-2);
    }
}

public class Example {
    public static long fac(int n) {
        long result = 1;
        if (n>=1)
            for (int i = 2; i <= n; i++)
                result *= i;
        return result;
    }
    public static int square(int n) {
        return n*n;
    }


 CHAPTER 3. MERGING SOFTWARE ARTIFACTS 19
Adjustable Syntactic Merge of Java Programs

"...companyed 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 [ALL87]. 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 reformatting 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 reformatting. 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 is an implementation of a

[http://fosd.de/SSMerge](http://fosd.de/SSMerge)
Adjustable Syntactic Merge of Java Programs

A semistructured merge engine build into FeatureHouse\textsuperscript{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 \([ALL^+\).

\[
\text{ClassDec: MailClient} \quad \oplus \quad \text{ClassDec: MailClient} = \text{ClassDec: MailClient}
\]

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 \(\text{compute()}\) and introduces an independent class member \(n\), which might be used elsewhere. The right revision wants to make sure, the argument of \(\text{compute()}\) is positive. After the syntactic merge, which did not produce a conflict, the new class member \(n\) is unintentionally changed within \(\text{compute()}\), 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,

\[\text{http://fosd.de/fh}\]
Adjustable Syntactic Merge of Java Programs

```java
public class Example2 {
    public static int compute(int n) {
        while(n != 1) {
            if(n % 2 == 0)
                n = n / 2;
            else
                n = 3*n + 1;
        }
        return n;
    }
}
```

```java
public class Example2 {
    public static int compute(int n) {
        if(n < 0) n = -1;
        while(n != 1) {
            if(n % 2 == 0)
                n = n / 2;
            else
                n = 3*n + 1;
        }
        return n;
    }
}
```

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.
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 JASTADD.\footnote{http://jastadd.org/web/jastaddj/}
4.2. JastAddJ

JastAddJ is an extensible Java compiler built from the metacompiler framework JastAddJ\(^2\). 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 \texttt{while}-loop has the attributes \texttt{getCondition()} and \texttt{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 \texttt{isOrdered()}, which returns \texttt{true} if the order of a language element is significant, and \texttt{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 ASTNodes, as will be described in detail in Section
4.6.

![Figure 4.1.: UML: Node](image)

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.
Adjustable Syntactic Merge of Java Programs

<<Interface>>

MergeInterface
+merge(output: StringWriter, left: File, right: File)
+merge3(output: StringWriter, left: File, base: File, right: File)

SemiStructuredMerge

SyntacticMerge
matcher: Matcher
-diff(left: Node, right: Node)
-diff3(left: Node, base: Node, baseCopy: Node, right: Node)

TextualMerge

Matcher
+match(a: Node, b: Node): Matching
+markMatching(m: Matching)

OrderedTreeMatcher
-orderedMatcher: Matcher

UnorderedTreeMatcher
unorderedMatcher: Matcher

TreeMatching
unorderedMatcher: Matcher

Figure 4.2.: UML: Algorithms

CHAPTER 4. JDIME: STRUCTURED MERGE FOR JAVA
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)
    \( m \leftarrow \text{number of children of } A \)
    \( n \leftarrow \text{number of children of } B \)
    Boolean ordered \( \leftarrow \text{false} \)
    for \( i \leftarrow 1, m \) do \( \triangleright \text{Can be determined using the attributed AST} \)
        if order of \( A_i \) is significant then
            ordered \( \leftarrow \text{true} \)
            break
        end if
    end for
    if ordered \( = \text{false} \) then
        for \( j \leftarrow 1, n \) do \( \triangleright \text{Can be determined using the attributed AST} \)
            if order of \( B_j \) is significant then
                ordered \( \leftarrow \text{true} \)
                break
            end if
        end for
    end if
    if ordered \( = \text{true} \) 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 NP-complete \cite{Mai78}. However, for small \( n \) it is solvable in polynomial time \cite{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)

\[
\begin{align*}
\text{function } & \text{ LCS(Sequence } A, \text{ Sequence } B) \\
m \leftarrow \text{ length of } A \\
n \leftarrow \text{ length of } B \\
\text{Matrix } M \leftarrow (m + 1) \times (n + 1) \text{ Matrix} \\
\text{for } i \leftarrow 0, m \text{ do} & \text{ Initialization} \\
& M[i, 0] \leftarrow 0 \\
\text{end for} \\
\text{for } j \leftarrow 0, n \text{ do} & \text{ Initialization} \\
& M[0, j] \leftarrow 0 \\
\text{end for} \\
\text{for } i \leftarrow 1, m \text{ do} & \\
& \text{for } j \leftarrow 1, n \text{ do} \\
& \text{if } A_i = B_j \text{ then} \\
& \quad W[i, j] \leftarrow 1 \\
& \text{else} \\
& \quad W[i, j] \leftarrow 0 \\
& \text{end if} \\
& M[i, j] \leftarrow \max(M[i, j - 1], M[i - 1, j], M[i - 1, j - 1] + W[i, j]) \\
& \text{end for} \\
\text{end for} \\
\text{return } M[m, n] \\
\end{align*}
\]

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 \textit{ABCBDAB} and \textit{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 \textit{BCBA} in the example from Table 4.1.
Algorithm 3 BACKTRACKING LCS

function BacktraceLCS(Matrix M, Sequence A, Sequence B)
    \[ m \leftarrow \text{length of } A \]
    \[ n \leftarrow \text{length of } B \]
    \[ i \leftarrow m \]
    while \( i > 1 \) do
        \[ j \leftarrow n \]
        while \( j > 1 \) do
            if \( M(i-1, j) = M(i, j) \) then
                \[ i \leftarrow i - 1 \]
            else if \( M(i, j-1) = M(i, j) \) then
                \[ j \leftarrow j - 1 \]
            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
                \[ i \leftarrow i - 1 \]
                \[ j \leftarrow j - 1 \]
            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**

```plaintext
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 nodes
    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 ORDERED_TREE_MATCHING(Node A, Node B)
    if $A \neq B$ then
        return empty matching  \Comment{root nodes do not match}
    end if
    $m \leftarrow$ number of children of $A$
    $n \leftarrow$ number of children of $B$
    Matrix $M \leftarrow (m + 1) \times (n + 1)$ Matrix
    for $i \leftarrow 0, m$ do
        $M[i, 0] \leftarrow 0$  \Comment{Initialization}
    end for
    for $j \leftarrow 0, n$ do
        $M[0, j] \leftarrow 0$  \Comment{Initialization}
    end for
    for $i \leftarrow 1, m$ do
        for $j \leftarrow 1, n$ do
            Matching $w \leftarrow TREE_MATCHING(A_i, B_j)$
            if $M[i, j - 1] > M[i - 1, j]$ then
                if $M[i, j - 1] > M[i - 1, j - 1] + w$ then
                    $M[i, j] \leftarrow M[i, j - 1]$
                    $M[i, j].Direction \leftarrow LEFT$
                else
                    $M[i, j] \leftarrow M[i - 1, j - 1] + w$
                    $M[i, j].Direction \leftarrow DIAG$
                end if
            else
                if $M[i - 1, j] > M[i - 1, j - 1] + w$ then
                    $M[i, j] \leftarrow M[i - 1, j]$
                    $M[i, j].Direction \leftarrow DOWN$
                else
                    $M[i, j] \leftarrow M[i - 1, j - 1] + w$
                    $M[i, j].Direction \leftarrow DIAG$
                end if
            end if
        end for
    end for
Algorithm 6 ORDERED TREE MATCHING - Part 2

List \( \text{childrenMatchings} \leftarrow \text{empty List} \)
\( i \leftarrow m \)
\( j \leftarrow n \)

while \( i \geq 1 \land j \geq 1 \) do
  if \( M[i,j].\text{Direction} = \text{DOWN} \) then
    \( i \leftarrow i - 1 \)
  else if \( M[i,j].\text{Direction} = \text{LEFT} \) then
    \( j \leftarrow j - 1 \)
  else
    if \( M[i,j] > M[i-1,j-1] \) then
      add \( M[i,j] \) to \( \text{childrenMatchings} \)
    end if
  end if
end while

\( \triangledown \) return sum of child matchings plus matching of the root rodes

Matching \( m \leftarrow M[m,n] + 1 \)
\( m.\text{childrenMatchings} \leftarrow \text{childrenMatchings} \)
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 \cite{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 \textsc{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 \cite{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 \cite{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
\[
\begin{align*}
\sum_{t \in T} x_{tw} &= 1, \quad \text{for all } w \in W \\
\sum_{w \in W} x_{tw} &= 1, \quad \text{for all } t \in T \\
x_{tw} &\in \{0, 1\}, \quad \text{for all } (t, w) \in T \times W
\end{align*}
\]

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 \[^7\].
Algorithm 7 Unordered Tree Matching

function UnorderedTreeMatching(Node A, Node B)
    if $A \neq B$ then
        return 0  \[\triangleright\text{ root nodes do not match}\]
    end if
    $m \leftarrow$ number of children of $A$
    $n \leftarrow$ number of children of $B$
    Matrix $M \leftarrow (m) \times (n)$ Matrix
    for $i \leftarrow 0, m$ do
        for $j \leftarrow 0, n$ do
            $W[i, j] \leftarrow$ TreeMatching($A_i, B_j$)
        end for
    end for  
    $\triangleright$ return sum of child matchings plus matching of the root nodes
    Matching $m \leftarrow M[m, n] + 1$
    $m.childrenMatchings \leftarrow$ 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 $O(1)$. `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.
To facilitate the process of merging, the base revision’s tree is cloned before calling \texttt{diff3()}. That means, the tool internally works with \texttt{base} and \texttt{base}'. After \texttt{diff3()} has terminated, nodes of \texttt{base} have references to their corresponding nodes of \texttt{leftRevision}, nodes of \texttt{base}' have references to their corresponding matchings of \texttt{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 \texttt{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 \land n \in left \land n \in right\}
    insert(unchanged, merged)
    consistent ← \{n \mid n \notin base \land n \in left \land n \in right\}
    insert(consistent, merged)
    lchanges ← \{n \mid n \in left \land n \notin merged \land n \notin right \land n \notin base\}
    rchanges ← \{n \mid n \in right \land n \notin merged \land n \notin left \land 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 ambiguous 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). If 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

```plaintext
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 syntactically correct input and produces a syntactically 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 ASTNodes 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.
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 \([\text{ALL}^+]\). 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 \([\text{ALL}^+]\). 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.

<table>
<thead>
<tr>
<th>Project</th>
<th>Domain</th>
<th>LoC</th>
<th>MS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrJava</td>
<td>development environment</td>
<td>77 K</td>
<td>9</td>
</tr>
<tr>
<td>FreeCol</td>
<td>turn-based strategy game</td>
<td>85 K</td>
<td>10</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>editor for genealogical data</td>
<td>55 K</td>
<td>10</td>
</tr>
<tr>
<td>iText</td>
<td>PDF library</td>
<td>85 K</td>
<td>8</td>
</tr>
<tr>
<td>jEdit</td>
<td>programmer’s text editor</td>
<td>98 K</td>
<td>8</td>
</tr>
<tr>
<td>Jmol</td>
<td>viewer for chemical structures</td>
<td>128 K</td>
<td>7</td>
</tr>
<tr>
<td>PMD</td>
<td>bug finder</td>
<td>65 K</td>
<td>10</td>
</tr>
<tr>
<td>SquirrelSQL</td>
<td>SQL GUI client</td>
<td>194 K</td>
<td>10</td>
</tr>
</tbody>
</table>

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](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
### Table 5.2: mean results for all projects

<table>
<thead>
<tr>
<th>Project</th>
<th>TM</th>
<th>SM</th>
<th>CM</th>
<th>SSM</th>
<th>TM</th>
<th>SM</th>
<th>CM</th>
<th>SSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrJava</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>17</td>
</tr>
<tr>
<td>FreeCol</td>
<td>63</td>
<td>48</td>
<td>47</td>
<td>58</td>
<td>47</td>
<td>200</td>
<td>187</td>
<td>212</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>iText</td>
<td>212</td>
<td>190</td>
<td>190</td>
<td>173</td>
<td>190</td>
<td>1847</td>
<td>1847</td>
<td>667</td>
</tr>
<tr>
<td>jEdit</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Jmol</td>
<td>23</td>
<td>13</td>
<td>12</td>
<td>19</td>
<td>12</td>
<td>31</td>
<td>30</td>
<td>43</td>
</tr>
<tr>
<td>PMD</td>
<td>10</td>
<td>7</td>
<td>7</td>
<td>10</td>
<td>7</td>
<td>10</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>SquirrelSQL</td>
<td>15</td>
<td>7</td>
<td>7</td>
<td>12</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>23</td>
</tr>
</tbody>
</table>

### Table 5.3: mean runtime values for all projects

<table>
<thead>
<tr>
<th>Project</th>
<th>TM</th>
<th>SM</th>
<th>CM</th>
<th>SSM</th>
<th>TM</th>
<th>SM</th>
<th>CM</th>
<th>SSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrJava</td>
<td>5971</td>
<td>99901</td>
<td>33674</td>
<td>539996</td>
<td>5971</td>
<td>99901</td>
<td>33674</td>
<td>539996</td>
</tr>
<tr>
<td>FreeCol</td>
<td>5726</td>
<td>82539</td>
<td>27815</td>
<td>473963</td>
<td>5726</td>
<td>82539</td>
<td>27815</td>
<td>473963</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>4174</td>
<td>56378</td>
<td>6115</td>
<td>337669</td>
<td>4174</td>
<td>56378</td>
<td>6115</td>
<td>337669</td>
</tr>
<tr>
<td>iText</td>
<td>7163</td>
<td>338302</td>
<td>315969</td>
<td>427851</td>
<td>7163</td>
<td>338302</td>
<td>315969</td>
<td>427851</td>
</tr>
<tr>
<td>jEdit</td>
<td>5174</td>
<td>72536</td>
<td>6587</td>
<td>459207</td>
<td>5174</td>
<td>72536</td>
<td>6587</td>
<td>459207</td>
</tr>
<tr>
<td>Jmol</td>
<td>4937</td>
<td>313965</td>
<td>239141</td>
<td>556777</td>
<td>4937</td>
<td>313965</td>
<td>239141</td>
<td>556777</td>
</tr>
<tr>
<td>PMD</td>
<td>3194</td>
<td>33368</td>
<td>5222</td>
<td>190080</td>
<td>3194</td>
<td>33368</td>
<td>5222</td>
<td>190080</td>
</tr>
<tr>
<td>SquirrelSQL</td>
<td>18129</td>
<td>230279</td>
<td>23788</td>
<td>1101322</td>
<td>18129</td>
<td>230279</td>
<td>23788</td>
<td>1101322</td>
</tr>
</tbody>
</table>

Figure 5.1: All projects: mean values for conflicts
**CHAPTER 5. EMPIRICAL STUDY**

Adjustable Syntactic Merge of Java Programs

---

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

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

Runtime: Mean Values

Figure 5.4.: All projects: mean values for runtime

<table>
<thead>
<tr>
<th>Project</th>
<th>TM</th>
<th>SM</th>
<th>CM</th>
<th>SSM</th>
<th>TM</th>
<th>SM</th>
<th>CM</th>
<th>SSM</th>
<th>TM</th>
<th>SM</th>
<th>CM</th>
<th>SSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrJava</td>
<td>100</td>
<td>25.78</td>
<td>21.28</td>
<td>9463</td>
<td>100</td>
<td>20.31</td>
<td>18.72</td>
<td>79.69</td>
<td>100</td>
<td>21.45</td>
<td>20.05</td>
<td>71.22</td>
</tr>
<tr>
<td>FreeCol</td>
<td>100</td>
<td>77.2</td>
<td>75.24</td>
<td>92.01</td>
<td>100</td>
<td>85.53</td>
<td>79.76</td>
<td>90.44</td>
<td>100</td>
<td>40.39</td>
<td>39.56</td>
<td>75.29</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>100</td>
<td>62.34</td>
<td>62.34</td>
<td>62.34</td>
<td>100</td>
<td>32.56</td>
<td>32.56</td>
<td>33.72</td>
<td>100</td>
<td>36.7</td>
<td>36.7</td>
<td>69.99</td>
</tr>
<tr>
<td>iText</td>
<td>100</td>
<td>89.63</td>
<td>89.45</td>
<td>81.56</td>
<td>100</td>
<td><strong>855.27</strong></td>
<td>854.98</td>
<td>308.62</td>
<td>100</td>
<td>5.51</td>
<td>5.5</td>
<td>30.29</td>
</tr>
<tr>
<td>jEdit</td>
<td>100</td>
<td>75.2</td>
<td>75.2</td>
<td>95.2</td>
<td>100</td>
<td>46.11</td>
<td>46.11</td>
<td>79.51</td>
<td>100</td>
<td>38.81</td>
<td>38.81</td>
<td>127.01</td>
</tr>
<tr>
<td>Jmol</td>
<td>100</td>
<td>56.18</td>
<td>52.46</td>
<td>80.25</td>
<td>100</td>
<td>54.73</td>
<td>53.19</td>
<td>76.73</td>
<td>100</td>
<td>20.95</td>
<td>20.19</td>
<td>54.56</td>
</tr>
<tr>
<td>PMD</td>
<td>100</td>
<td>70.71</td>
<td>70.71</td>
<td>98.99</td>
<td>100</td>
<td>58.33</td>
<td>58.33</td>
<td>89.88</td>
<td>100</td>
<td>51.66</td>
<td>51.66</td>
<td>68.12</td>
</tr>
<tr>
<td>SQuirrelSQL</td>
<td>100</td>
<td>45.03</td>
<td>45.03</td>
<td>82.12</td>
<td>100</td>
<td>34.66</td>
<td>34.66</td>
<td>92.03</td>
<td>100</td>
<td>8.6</td>
<td>8.6</td>
<td>22.57</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Project</th>
<th>TM</th>
<th>SM</th>
<th>CM</th>
<th>SSM</th>
<th>TM</th>
<th>SM</th>
<th>CM</th>
<th>SSM</th>
<th>TM</th>
<th>SM</th>
<th>CM</th>
<th>SSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrJava</td>
<td>100</td>
<td>1673.08</td>
<td>563.96</td>
<td>9043.47</td>
<td>100</td>
<td>1411.5</td>
<td>485.78</td>
<td>8277.53</td>
<td>100</td>
<td>1350.73</td>
<td>146.51</td>
<td>8090</td>
</tr>
<tr>
<td>FreeCol</td>
<td>100</td>
<td>1441.5</td>
<td>485.78</td>
<td>8277.53</td>
<td>100</td>
<td>1350.73</td>
<td>146.51</td>
<td>8090</td>
<td>100</td>
<td>1402.08</td>
<td>127.33</td>
<td>8876.13</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>100</td>
<td>4723.16</td>
<td>4411.36</td>
<td>5973.39</td>
<td>100</td>
<td>4723.16</td>
<td>4411.36</td>
<td>5973.39</td>
<td>100</td>
<td>6359.42</td>
<td>4843.86</td>
<td>11277.64</td>
</tr>
<tr>
<td>iText</td>
<td>100</td>
<td>1044.88</td>
<td>163.53</td>
<td>5952.08</td>
<td>100</td>
<td>1044.88</td>
<td>163.53</td>
<td>5952.08</td>
<td>100</td>
<td>1270.2</td>
<td>131.21</td>
<td>6074.79</td>
</tr>
</tbody>
</table>

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.
Adjustable Syntactic Merge of Java Programs

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 reformatting 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.
Adjustable Syntactic Merge of Java Programs

Version Base

```java
String dString = RemoteControlServer.QUERY_PREFIX;
if (f!=null)
    dString += f.getAbsolutePath();
```

Version Left

```java
String dString = RemoteControlServer.QUERY_PREFIX;
if (f!=null)
    dString += f.getAbsolutePath();
```

Version Right

```java
String dString = RemoteControlServer.QUERY_PREFIX;
if (f!=null)
    dString += f.getAbsolutePath();
```

unstructured

```java
String dString = RemoteControlServer.QUERY_PREFIX;
if (f!=null)
    dString += f.getAbsolutePath();

==
String dString = RemoteControlServer.QUERY_PREFIX;
if (f!=null)
    dString += f.getAbsolutePath();

right - branch - 5044/RemoteControlClient.java
```

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 SQuirrel-SQL. 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 \( \mathcal{AP\chi} \)-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


[EH07] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In Proceedings of the 22nd annual ACM SIGPLAN conference on Object-
expanded on how these systems and applications are oriented programming systems and applications, OOPSLA ’07, pages 1–18, New York, NY, USA, 2007. ACM.


List of Figures

2.1. Branching and Tagging in Version Control . . . . . . . . . . . . . . . . . . 8
2.2. Centralized Version Control System . . . . . . . . . . . . . . . . . . . . . 9
2.3. Distributed Version Control System . . . . . . . . . . . . . . . . . . . . . 10
2.4. Lost Update Problem/Blind Overwriting . . . . . . . . . . . . . . . . . . . 11
2.5. Lock-Modify-Write Solution . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.6. Copy-Modify-Merge Solution . . . . . . . . . . . . . . . . . . . . . . . . . . 13

3.1. Two-way and three-way merge . . . . . . . . . . . . . . . . . . . . . . . . 16
3.2. GNU merge displaying a conflict . . . . . . . . . . . . . . . . . . . . . . . . 18
3.3. Failed textual merge . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.4. Superimposition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.5. Semantic Conflict . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

4.1. UML: Node . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.2. UML: Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

5.1. All projects: mean conflicts . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.2. All projects: mean conflicting lines . . . . . . . . . . . . . . . . . . . . . . 51
5.3. All projects: mean conflicting files . . . . . . . . . . . . . . . . . . . . . . 51
5.4. All projects: mean runtime . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
5.5. Ordering conflict . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
5.6. Code reformatting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

B.1. DrJava: conflicting files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
B.2. DrJava: conflicts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
B.3. DrJava: conflicting lines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.4</td>
<td>DrJava: runtime</td>
<td>74</td>
</tr>
<tr>
<td>B.5</td>
<td>FreeCol: conflicting files</td>
<td>74</td>
</tr>
<tr>
<td>B.6</td>
<td>FreeCol: conflicts</td>
<td>75</td>
</tr>
<tr>
<td>B.7</td>
<td>FreeCol: conflicting lines</td>
<td>75</td>
</tr>
<tr>
<td>B.8</td>
<td>FreeCol: runtime</td>
<td>76</td>
</tr>
<tr>
<td>B.9</td>
<td>GenealogyJ: conflicting files</td>
<td>76</td>
</tr>
<tr>
<td>B.10</td>
<td>GenealogyJ: conflicts</td>
<td>77</td>
</tr>
<tr>
<td>B.11</td>
<td>GenealogyJ: conflicting lines</td>
<td>77</td>
</tr>
<tr>
<td>B.12</td>
<td>GenealogyJ: runtime</td>
<td>78</td>
</tr>
<tr>
<td>B.13</td>
<td>iTex: conflicting files</td>
<td>78</td>
</tr>
<tr>
<td>B.14</td>
<td>iTex: conflicts</td>
<td>79</td>
</tr>
<tr>
<td>B.15</td>
<td>iTex: conflicting lines</td>
<td>79</td>
</tr>
<tr>
<td>B.16</td>
<td>iTex: runtime</td>
<td>80</td>
</tr>
<tr>
<td>B.17</td>
<td>jEdit: conflicting files</td>
<td>80</td>
</tr>
<tr>
<td>B.18</td>
<td>jEdit: conflicts</td>
<td>81</td>
</tr>
<tr>
<td>B.19</td>
<td>jEdit: conflicting lines</td>
<td>81</td>
</tr>
<tr>
<td>B.20</td>
<td>jEdit: runtime</td>
<td>82</td>
</tr>
<tr>
<td>B.21</td>
<td>Jmol: conflicting files</td>
<td>82</td>
</tr>
<tr>
<td>B.22</td>
<td>Jmol: conflicts</td>
<td>83</td>
</tr>
<tr>
<td>B.23</td>
<td>Jmol: conflicting lines</td>
<td>83</td>
</tr>
<tr>
<td>B.24</td>
<td>Jmol: runtime</td>
<td>84</td>
</tr>
<tr>
<td>B.25</td>
<td>PMD: conflicting files</td>
<td>84</td>
</tr>
<tr>
<td>B.26</td>
<td>PMD: conflicts</td>
<td>85</td>
</tr>
<tr>
<td>B.27</td>
<td>PMD: conflicting lines</td>
<td>85</td>
</tr>
<tr>
<td>B.28</td>
<td>PMD: runtime</td>
<td>86</td>
</tr>
<tr>
<td>B.29</td>
<td>SQuirrelSQL: conflicting files</td>
<td>86</td>
</tr>
<tr>
<td>B.30</td>
<td>SQuirrelSQL: conflicts</td>
<td>87</td>
</tr>
<tr>
<td>B.31</td>
<td>SQuirrelSQL: conflicting lines</td>
<td>87</td>
</tr>
<tr>
<td>B.32</td>
<td>SQuirrelSQL: runtime</td>
<td>88</td>
</tr>
</tbody>
</table>
List of Tables

3.1. Three-way merge rules ............................................. 17
4.1. Matrix produced by LCS algorithm ............................... 32
5.1. Sample projects .................................................... 48
5.2. All projects: mean results ........................................ 50
5.3. All projects: mean runtime ...................................... 50
5.4. All projects: results in percentage ............................. 52
5.5. All projects: runtime in percentage ............................ 52
A.1. Conflicts of all projects ......................................... 69
A.2. Runtime in ms of all projects ................................. 71
<table>
<thead>
<tr>
<th>No.</th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Tree Matching</td>
<td>30</td>
</tr>
<tr>
<td>2.</td>
<td>Largest Common Subsequence (LCS)</td>
<td>31</td>
</tr>
<tr>
<td>3.</td>
<td>Backtracing LCS</td>
<td>32</td>
</tr>
<tr>
<td>4.</td>
<td>SimpleTreeMatching by Yang</td>
<td>33</td>
</tr>
<tr>
<td>5.</td>
<td>Ordered Tree Matching - Part 1</td>
<td>34</td>
</tr>
<tr>
<td>6.</td>
<td>Ordered Tree Matching - Part 2</td>
<td>35</td>
</tr>
<tr>
<td>7.</td>
<td>Unordered Tree Matching</td>
<td>38</td>
</tr>
<tr>
<td>8.</td>
<td>DIFF3</td>
<td>39</td>
</tr>
<tr>
<td>9.</td>
<td>Merge - Part 1</td>
<td>40</td>
</tr>
<tr>
<td>10.</td>
<td>Merge - Part 2</td>
<td>42</td>
</tr>
<tr>
<td>11.</td>
<td>Merge - Part 3</td>
<td>42</td>
</tr>
<tr>
<td>12.</td>
<td>Merge - Part 4</td>
<td>43</td>
</tr>
</tbody>
</table>
# APPENDIX A

## Results: Tables

### A.1. Conflicts

<table>
<thead>
<tr>
<th>Project</th>
<th>Revision</th>
<th>conflicting files</th>
<th>conflicts</th>
<th>conflicting lines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>TM</td>
<td>SM</td>
<td>CM</td>
</tr>
<tr>
<td>DrJava</td>
<td>rev3734-3786</td>
<td>7</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>DrJava</td>
<td>rev3734-3788</td>
<td>7</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>DrJava</td>
<td>rev3734-3807</td>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>DrJava</td>
<td>rev4989-5004</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>DrJava</td>
<td>rev4989-5019</td>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DrJava</td>
<td>rev4989-5044</td>
<td>23</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>DrJava</td>
<td>rev4989-5058</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>DrJava</td>
<td>rev5319-5330</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>DrJava</td>
<td>rev5319-5332</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FreeCol</td>
<td>rev5884-5962</td>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>FreeCol</td>
<td>rev5884-6055</td>
<td>9</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>FreeCol</td>
<td>rev5884-6110</td>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>FreeCol</td>
<td>rev5884-6265</td>
<td>12</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>FreeCol</td>
<td>rev5884-6362</td>
<td>20</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>FreeCol</td>
<td>rev5884-6440</td>
<td>72</td>
<td>56</td>
<td>55</td>
</tr>
<tr>
<td>FreeCol</td>
<td>rev5884-6616</td>
<td>119</td>
<td>97</td>
<td>96</td>
</tr>
<tr>
<td>FreeCol</td>
<td>rev5884-6672</td>
<td>123</td>
<td>99</td>
<td>98</td>
</tr>
<tr>
<td>FreeCol</td>
<td>rev5884-6742</td>
<td>124</td>
<td>103</td>
<td>101</td>
</tr>
<tr>
<td>FreeCol</td>
<td>rev5884-6843</td>
<td>134</td>
<td>106</td>
<td>104</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev5531-5563</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev5531-5725</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev5537-5610</td>
<td>28</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev5537-5673</td>
<td>28</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev5676-6013</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

*Continued on next page*
<table>
<thead>
<tr>
<th>Project</th>
<th>Revision</th>
<th>conflicts files</th>
<th>conflicts</th>
<th>conflicting lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenealogyJ</td>
<td>rev5676-6125</td>
<td>1 0 0 1</td>
<td>0 0 0 1</td>
<td>3 0 0 27</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev6127-6244</td>
<td>2 1 1 3</td>
<td>1 1 1 3</td>
<td>30 16 16 95</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev6127-6310</td>
<td>4 2 2 5</td>
<td>2 2 2 5</td>
<td>56 38 38 151</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev6127-6410</td>
<td>4 1 1 5</td>
<td>1 1 1 5</td>
<td>94 22 22 189</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev6127-6531</td>
<td>5 1 1 6</td>
<td>1 1 1 6</td>
<td>106 22 22 201</td>
</tr>
<tr>
<td>iTex</td>
<td>rev2818-3036</td>
<td>24 20 17 23</td>
<td>17 65 60 51</td>
<td>2897 569 451 1613</td>
</tr>
<tr>
<td>iTex</td>
<td>rev2818-3191</td>
<td>178 155 155 157</td>
<td>155 1834 1834 652</td>
<td>115577 6284 6284 36374</td>
</tr>
<tr>
<td>iTex</td>
<td>rev2818-3306</td>
<td>234 204 204 204</td>
<td>204 2097 2097 787</td>
<td>133913 6943 6943 41102</td>
</tr>
<tr>
<td>iTex</td>
<td>rev2818-3392</td>
<td>249 230 230 204</td>
<td>230 2263 2263 802</td>
<td>138768 7791 7791 42509</td>
</tr>
<tr>
<td>iTex</td>
<td>rev2818-3560</td>
<td>261 226 226 208</td>
<td>226 2098 2098 796</td>
<td>142685 7391 7391 45099</td>
</tr>
<tr>
<td>iTex</td>
<td>rev2818-3625</td>
<td>264 243 243 200</td>
<td>243 2358 2358 746</td>
<td>136078 7825 7825 42231</td>
</tr>
<tr>
<td>iTex</td>
<td>rev2818-3988</td>
<td>252 225 225 201</td>
<td>225 2059 2059 774</td>
<td>135368 7794 7794 39204</td>
</tr>
<tr>
<td>iTex</td>
<td>rev2818-4022</td>
<td>235 218 218 187</td>
<td>218 2005 2005 725</td>
<td>132585 7067 7067 35976</td>
</tr>
<tr>
<td>jEdit</td>
<td>rev4676-4998</td>
<td>2 0 0 1</td>
<td>0 0 0 1</td>
<td>28 0 0 2</td>
</tr>
<tr>
<td>jEdit</td>
<td>rev16588-16755</td>
<td>2 2 2 2</td>
<td>2 2 2 2</td>
<td>9 7 7 9</td>
</tr>
<tr>
<td>jEdit</td>
<td>rev16588-16883</td>
<td>2 1 1 2</td>
<td>1 1 1 2</td>
<td>9 2 2 9</td>
</tr>
<tr>
<td>jEdit</td>
<td>rev16588-17060</td>
<td>2 2 2 2</td>
<td>2 2 2 2</td>
<td>24 4 4 24</td>
</tr>
<tr>
<td>jEdit</td>
<td>rev16588-17316</td>
<td>2 2 2 2</td>
<td>2 2 2 2</td>
<td>20 4 4 20</td>
</tr>
<tr>
<td>jEdit</td>
<td>rev16588-17492</td>
<td>2 2 2 2</td>
<td>2 2 2 2</td>
<td>20 4 4 20</td>
</tr>
<tr>
<td>jEdit</td>
<td>rev16588-17551</td>
<td>2 2 2 2</td>
<td>2 2 2 2</td>
<td>20 4 4 20</td>
</tr>
<tr>
<td>jEdit</td>
<td>rev16883-16964</td>
<td>6 4 4 6</td>
<td>4 7 7 14</td>
<td>192 100 100 305</td>
</tr>
<tr>
<td>Jmol</td>
<td>rev11338-11438</td>
<td>4 3 2 3</td>
<td>2 3 2 7</td>
<td>350 20 10 131</td>
</tr>
<tr>
<td>Jmol</td>
<td>rev11338-11538</td>
<td>9 6 5 7</td>
<td>5 12 11 21</td>
<td>521 67 57 269</td>
</tr>
<tr>
<td>Jmol</td>
<td>rev11338-11638</td>
<td>15 8 7 10</td>
<td>7 17 16 25</td>
<td>667 78 68 344</td>
</tr>
<tr>
<td>Jmol</td>
<td>rev11338-11738</td>
<td>23 13 12 18</td>
<td>12 35 34 43</td>
<td>1070 203 193 493</td>
</tr>
<tr>
<td>Jmol</td>
<td>rev11338-11838</td>
<td>34 19 18 27</td>
<td>18 46 45 58</td>
<td>1637 333 323 915</td>
</tr>
<tr>
<td>Jmol</td>
<td>rev11338-11938</td>
<td>35 20 19 29</td>
<td>19 48 47 67</td>
<td>1703 344 334 998</td>
</tr>
<tr>
<td>Jmol</td>
<td>rev11338-12038</td>
<td>42 22 22 36</td>
<td>22 53 53 79</td>
<td>1869 593 593 1115</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-6010</td>
<td>1 0 0 1</td>
<td>0 0 0 1</td>
<td>5 0 0 5</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-6135</td>
<td>3 2 2 3</td>
<td>2 2 2 3</td>
<td>22 17 17 24</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-6198</td>
<td>7 3 3 7</td>
<td>3 4 4 13</td>
<td>784 342 342 359</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-6296</td>
<td>9 5 5 10</td>
<td>5 6 6 16</td>
<td>734 349 349 450</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-6425</td>
<td>2 2 2 2</td>
<td>2 2 2 2</td>
<td>20 4 4 20</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-6565</td>
<td>12 9 9 12</td>
<td>9 12 12 19</td>
<td>670 462 462 478</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-6700</td>
<td>13 11 11 13</td>
<td>11 15 15 20</td>
<td>1003 539 539 767</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-6835</td>
<td>12 10 10 12</td>
<td>10 16 16 19</td>
<td>939 506 506 721</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-7018</td>
<td>16 12 12 15</td>
<td>12 18 18 22</td>
<td>1053 510 510 737</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-7073</td>
<td>17 13 13 16</td>
<td>13 19 19 23</td>
<td>977 498 498 730</td>
</tr>
<tr>
<td>SQuirrelSQL</td>
<td>rev4007-4051</td>
<td>1 0 0 0</td>
<td>0 0 0 0</td>
<td>4 0 0 0</td>
</tr>
<tr>
<td>SQuirrelSQL</td>
<td>rev4007-4103</td>
<td>3 1 1 2</td>
<td>1 1 1 3</td>
<td>299 7 7 33</td>
</tr>
<tr>
<td>SQuirrelSQL</td>
<td>rev4007-4212</td>
<td>11 6 6 9</td>
<td>6 8 8 19</td>
<td>1384 108 108 303</td>
</tr>
<tr>
<td>SQuirrelSQL</td>
<td>rev4007-4321</td>
<td>17 7 7 14</td>
<td>7 9 9 27</td>
<td>3862 164 164 546</td>
</tr>
<tr>
<td>SQuirrelSQL</td>
<td>rev4007-4394</td>
<td>21 10 10 18</td>
<td>10 12 12 32</td>
<td>4218 357 357 824</td>
</tr>
<tr>
<td>SQuirrelSQL</td>
<td>rev4007-4516</td>
<td>25 12 12 21</td>
<td>12 15 15 35</td>
<td>4258 414 414 871</td>
</tr>
<tr>
<td>SQuirrelSQL</td>
<td>rev4007-4908</td>
<td>33 13 13 27</td>
<td>13 18 18 53</td>
<td>6111 420 420 1468</td>
</tr>
<tr>
<td>SQuirrelSQL</td>
<td>rev4007-5081</td>
<td>34 15 15 28</td>
<td>15 20 20 55</td>
<td>4574 615 615 1482</td>
</tr>
<tr>
<td>SQuirrelSQL</td>
<td>rev5082-5155</td>
<td>2 1 1 2</td>
<td>1 1 1 3</td>
<td>17 50 50 37</td>
</tr>
<tr>
<td>SQuirrelSQL</td>
<td>rev5082-5351</td>
<td>4 3 3 3</td>
<td>3 3 3 4</td>
<td>194 9 9 61</td>
</tr>
</tbody>
</table>

Table A.1.: Conflicts of all projects
### A.2. Runtime

<table>
<thead>
<tr>
<th>Project</th>
<th>Revision</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrJava</td>
<td>rev3734-3786</td>
<td>5556</td>
</tr>
<tr>
<td>DrJava</td>
<td>rev3734-3788</td>
<td>5557</td>
</tr>
<tr>
<td>DrJava</td>
<td>rev3734-3807</td>
<td>5663</td>
</tr>
<tr>
<td>DrJava</td>
<td>rev4989-5004</td>
<td>6029</td>
</tr>
<tr>
<td>DrJava</td>
<td>rev4989-5019</td>
<td>6186</td>
</tr>
<tr>
<td>DrJava</td>
<td>rev4989-5044</td>
<td>6278</td>
</tr>
<tr>
<td>DrJava</td>
<td>rev4989-5058</td>
<td>6163</td>
</tr>
<tr>
<td>DrJava</td>
<td>rev5319-5330</td>
<td>6158</td>
</tr>
<tr>
<td>DrJava</td>
<td>rev5319-5332</td>
<td>6150</td>
</tr>
<tr>
<td>Freecol</td>
<td>rev5884-5962</td>
<td>5285</td>
</tr>
<tr>
<td>Freecol</td>
<td>rev5884-6055</td>
<td>5276</td>
</tr>
<tr>
<td>Freecol</td>
<td>rev5884-6110</td>
<td>5355</td>
</tr>
<tr>
<td>Freecol</td>
<td>rev5884-6265</td>
<td>5458</td>
</tr>
<tr>
<td>Freecol</td>
<td>rev5884-6362</td>
<td>5924</td>
</tr>
<tr>
<td>Freecol</td>
<td>rev5884-6410</td>
<td>6157</td>
</tr>
<tr>
<td>Freecol</td>
<td>rev5884-6616</td>
<td>6205</td>
</tr>
<tr>
<td>Freecol</td>
<td>rev5884-6672</td>
<td>6220</td>
</tr>
<tr>
<td>Freecol</td>
<td>rev5884-6742</td>
<td>6219</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev5531-5561</td>
<td>4017</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev5531-5725</td>
<td>3950</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev5537-5610</td>
<td>4108</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev5537-5673</td>
<td>4032</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev5676-6013</td>
<td>3929</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev5676-6125</td>
<td>3848</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev6127-6244</td>
<td>4445</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev6127-6310</td>
<td>4455</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev6127-6410</td>
<td>4504</td>
</tr>
<tr>
<td>GenealogyJ</td>
<td>rev6127-6531</td>
<td>4451</td>
</tr>
<tr>
<td>iText</td>
<td>rev2818-3036</td>
<td>4501</td>
</tr>
<tr>
<td>iText</td>
<td>rev2818-3191</td>
<td>5949</td>
</tr>
<tr>
<td>iText</td>
<td>rev2818-3306</td>
<td>7206</td>
</tr>
<tr>
<td>iText</td>
<td>rev2818-3392</td>
<td>7851</td>
</tr>
<tr>
<td>iText</td>
<td>rev2818-3560</td>
<td>7984</td>
</tr>
<tr>
<td>iText</td>
<td>rev2818-3625</td>
<td>7929</td>
</tr>
<tr>
<td>iText</td>
<td>rev2818-3988</td>
<td>7790</td>
</tr>
<tr>
<td>iText</td>
<td>rev2818-4022</td>
<td>8091</td>
</tr>
<tr>
<td>jEdit</td>
<td>rev4676-4998</td>
<td>3584</td>
</tr>
<tr>
<td>jEdit</td>
<td>rev16588-16755</td>
<td>5405</td>
</tr>
<tr>
<td>jEdit</td>
<td>rev16588-16883</td>
<td>5411</td>
</tr>
<tr>
<td>jEdit</td>
<td>rev16588-17060</td>
<td>5420</td>
</tr>
<tr>
<td>jEdit</td>
<td>rev16588-17316</td>
<td>5377</td>
</tr>
<tr>
<td>jEdit</td>
<td>rev16588-17492</td>
<td>5375</td>
</tr>
<tr>
<td>jEdit</td>
<td>rev16588-17551</td>
<td>5381</td>
</tr>
<tr>
<td>jEdit</td>
<td>rev16883-16964</td>
<td>5435</td>
</tr>
<tr>
<td>Jmol</td>
<td>rev11338-11438</td>
<td>4721</td>
</tr>
<tr>
<td>Jmol</td>
<td>rev11338-11538</td>
<td>4786</td>
</tr>
<tr>
<td>Jmol</td>
<td>rev11338-11638</td>
<td>4893</td>
</tr>
<tr>
<td>Jmol</td>
<td>rev11338-11738</td>
<td>4970</td>
</tr>
<tr>
<td>Jmol</td>
<td>rev11338-11838</td>
<td>5068</td>
</tr>
<tr>
<td>Jmol</td>
<td>rev11338-11938</td>
<td>5080</td>
</tr>
<tr>
<td>Jmol</td>
<td>rev11338-12038</td>
<td>5041</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-6010</td>
<td>4155</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-6135</td>
<td>4136</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-6198</td>
<td>3407</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-6296</td>
<td>3360</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-6425</td>
<td>2842</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-6595</td>
<td>2814</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-6700</td>
<td>2836</td>
</tr>
</tbody>
</table>

**Continued on next page**
Table A.2.: Runtime in ms of all projects

<table>
<thead>
<tr>
<th>Project</th>
<th>Revision</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>TM</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-6835</td>
<td>2775</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-7018</td>
<td>2793</td>
</tr>
<tr>
<td>PMD</td>
<td>rev5929-7073</td>
<td>2817</td>
</tr>
<tr>
<td>SquirrelSQL</td>
<td>rev4007-4051</td>
<td>16938</td>
</tr>
<tr>
<td>SquirrelSQL</td>
<td>rev4007-4103</td>
<td>16968</td>
</tr>
<tr>
<td>SquirrelSQL</td>
<td>rev4007-4212</td>
<td>16831</td>
</tr>
<tr>
<td>SquirrelSQL</td>
<td>rev4007-4321</td>
<td>16906</td>
</tr>
<tr>
<td>SquirrelSQL</td>
<td>rev4007-4394</td>
<td>16902</td>
</tr>
<tr>
<td>SquirrelSQL</td>
<td>rev4007-4516</td>
<td>16955</td>
</tr>
<tr>
<td>SquirrelSQL</td>
<td>rev4007-4908</td>
<td>16644</td>
</tr>
<tr>
<td>SquirrelSQL</td>
<td>rev4007-5081</td>
<td>16524</td>
</tr>
<tr>
<td>SquirrelSQL</td>
<td>rev5082-5155</td>
<td>23351</td>
</tr>
<tr>
<td>SquirrelSQL</td>
<td>rev5082-5351</td>
<td>23275</td>
</tr>
</tbody>
</table>
APPENDIX B

Results: Graphs

B.1. DrJava

Figure B.1.: DrJava: conflicting files
Figure B.2.: DrJava: conflicts

Figure B.3.: DrJava: conflicting lines
**B.2. FreeCol**

![FreeCol: conflicting files](image)

**Figure B.5:** FreeCol: conflicting files
Adjustable Syntactic Merge of Java Programs

FreeCol

Conflicts

Number of conflicts (log scale)

Merge scenarios

FreeCol

Conflicting Lines

Number of conflicting lines (log scale)

Merge scenarios

Figure B.6.: FreeCol: conflicts

Figure B.7.: FreeCol: conflicting lines
B.3. GenealogyJ

Figure B.8.: FreeCol: runtime

Figure B.9.: GenealogyJ: conflicting files
Figure B.10.: GenealogyJ: conflicts

Figure B.11.: GenealogyJ: conflicting lines
Adjustable Syntactic Merge of Java Programs

GenealogyJ

Runtime

Time in milliseconds (log scale)

Merge scenarios

Figure B.12.: GenealogyJ: runtime

B.4. iText

iText

Conflicting Files

Number of conflicting files (log scale)

Merge scenarios

Figure B.13.: iText: conflicting files

APPENDIX B. RESULTS: GRAPHS
Figure B.14.: iText: conflicts

Figure B.15.: iText: conflicting lines
B.5. jEdit

Figure B.17.: jEdit: conflicting files
Figure B.18.: jEdit: conflicts

Figure B.19.: jEdit: conflicting lines
B.6. Jmol

Figure B.21.: Jmol: conflicting files
Figure B.22.: Jmol: conflicts

Figure B.23.: Jmol: conflicting lines
B.7. PMD

Figure B.24.: Jmol: runtime

Figure B.25.: PMD: conflicting files
APPENDIX B. RESULTS: GRAPHS

Figure B.26.: PMD: conflicts

Figure B.27.: PMD: conflicting lines
B.8. SQuirrelSQL

Figure B.28.: PMD: runtime

Figure B.29.: SQuirrelSQL: conflicting files
Adjustable Syntactic Merge of Java Programs

SQuirrelSQL

Conflicts

Number of conflicts (log scale)

Merge scenarios

4007-4051 4007-4103 4007-4212 4007-4321 4007-4394 4007-4516 4007-4908 4007-5081 5082-5155 5082-5351

GNU Merge  JDime  JDime AT  FSTMerge

Figure B.30.: SQuirrelSQL: conflicts

SQuirrelSQL

Conflicting Lines

Number of conflicting lines (log scale)

Merge scenarios

4007-4051 4007-4103 4007-4212 4007-4321 4007-4394 4007-4516 4007-4908 4007-5081 5082-5155 5082-5351

GNU Merge  JDime  JDime AT  FSTMerge

Figure B.31.: SQuirrelSQL: conflicting lines
Figure B.32.: SQuirrelSQL: runtime
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)