Structured Merge with Auto-Tuning: Balancing Precision and Performance

Sven Apel, Olaf Leßenich, and Christian Lengauer
University of Passau, Germany
{apel, lessenic, lengauer}@fim.uni-passau.de

ABSTRACT

Software-merging techniques face the challenge of finding a balance between precision and performance. In practice, developers use unstructured-merge (i.e., line-based) tools, which are fast but imprecise. In academia, many approaches incorporate information on the structure of the artifacts being merged. While this increases precision in conflict detection and resolution, it can induce severe performance penalties. Striving for a proper balance between precision and performance, we propose a *structured-merge* approach with *auto-tuning*. In a nutshell, we tune the merge process on-line by switching between unstructured and structured merge, depending on the presence of conflicts. We implemented a corresponding merge tool for Java, called JDime. Our experiments with 8 real-world Java projects, involving 72 merge scenarios with over 17 million lines of code, demonstrate that our approach indeed hits a sweet spot: While largely maintaining a precision that is superior to the one of unstructured merge, structured merge with auto-tuning is up to 12 times faster than purely structured merge, 5 times on average.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Version control*; D.2.9 [**Software Engineering**]: Management—*Software configuration management*

General Terms

Management, Measurement, Experimentation

Keywords

Version Control, Software Merging, Structured Merge, JDime

1. INTRODUCTION

Software-merging techniques are gaining momentum in the practice and theory of software engineering. They are important tools for programmers and software engineers not only in version control systems but also in product-line and model-driven engineering.

Contemporary software-merging techniques can be classified into (1) syntactic approaches and (2) semantic approaches. The former include (a) unstructured approaches that treat software artifacts as sequences of text lines and (b) structured approaches that are based on the artifacts' syntactic structure. In our attempt to push back the limits of practical software merging, we concentrate on syntactic approaches—semantic approaches are promising but still too immature to be used in real-world software projects.

The state of the art is that the most widely-used software-merging tools are unstructured; popular examples include the tools diff and merge of Unix, used in version-control systems such as CVS, Subversion, and Git. Unstructured merge is very simple and general: every software artifact that can be represented as text (i.e., as sequences of text lines) can be processed. So, a single tool that treats all software artifacts equally suffices. However, the downside is that unstructured merge is rather weak when it comes to expressing differences and handling merge conflicts: the basic unit is the line—all structure, all knowledge of the artifacts involved is lost [1, 16].

Previous work has shown that an exploitation of the syntactic structure of the artifacts involved improves the merge process in that differences between artifacts can be expressed in terms of their structure [1, 5, 16, 20], which also opens new opportunities for detecting and resolving merge conflicts [1]—one of the key problems in this field [16]. Unfortunately, no practical structured-merge tools for mainstream programming languages are available. Why?

A first problem is certainly that, when developing a structured tool, one must commit to a particular artifact language and, as a consequence, develop and use a different tool per language [1]. A second problem is that algorithms that take the structure of the artifacts involved into account are typically at least cubic, if not even $\mathcal{NP}$-complete—a major obstacle to their practical application [16]. While the first problem has been addressed, for example, by the technique of semistructured merge [1] (parts of the artifacts are treated as syntax trees and parts as plain text; see Sec. 5), we strive here for a solution to the second problem. Can we develop a merge approach that takes the structure of artifacts fully into account and that is efficient enough to be useful in real-world software projects?

We report on the development and application of a merge approach that is based on tree matching and amalgamation.

It is more precise in calculating differences and merges than an unstructured, lined-based approach, as it has more information about the artifacts at its disposal. To cope with the complexity of the tree-based merging operations involved, we use an *auto-tuning approach*. The basic idea is that the tool adjusts the precision of the merge operations (from unstructured, lined-based to structured, tree-based) guided by the conflicts detected in a software project. As long as no conflicts are detected, the tool uses unstructured merge, which is cheap in terms of performance. Once conflicts are detected, the tool switches to structured merge to increase the precision. So, the basic idea is simple: use the expensive technique only when necessary, which is in line with Mens' statement on the future of software-merge techniques:

> 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 [16].

While an auto-tuning approach is not as precise as a purely structured merge (the unstructured merge involved may miss conflicts or may not be able to resolve certain conflicts), it is likely faster and thus more practical in real-world software engineering, especially, if one believes Mens' conjecture that unstructured merge suffices in 90 % of all merge scenarios [16]. In fact, we strive for a solution that improves the state of practice, namely getting away from the exclusive use of unstructured-merge tools.

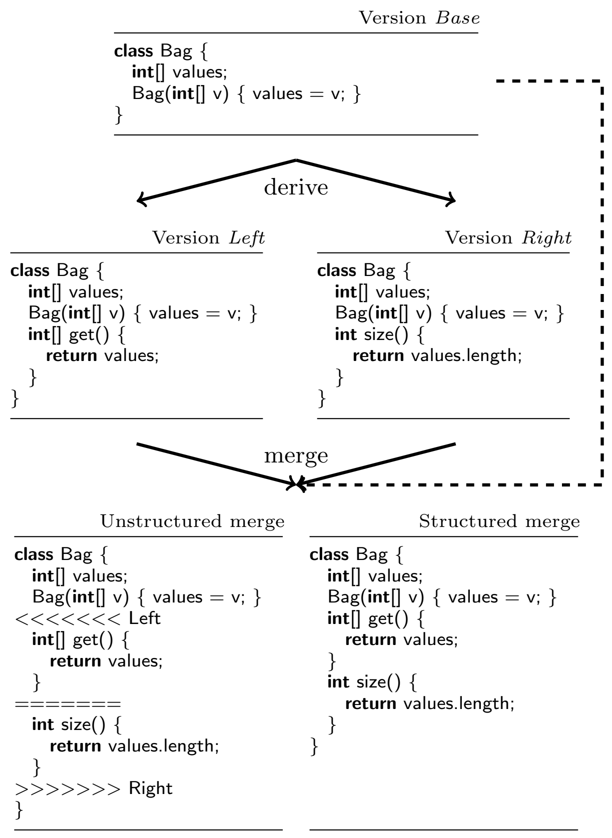
To demonstrate the practicality of our approach, we have implemented a tool for Java, called JDime, that performs structured merge (optionally) with auto-tuning. We used JDime in 72 merge scenarios of 8 software projects, involving over 17 million lines of code. Specifically, we compared the performance and the ability to resolve conflicts of unstructured and structured merge (with and without auto-tuning).

We found that purely structured merge is more precise than unstructured merge: It is able to resolve many more conflicts than unstructured merge but reveals also conflicts not noticed by unstructured merge. However, as expected, structured merge is slower by an order of magnitude, which is due to the more complex differencing and merge technique. Remarkably, the auto-tuning approach diverges only minimally from purely structured merge in terms of conflict detection, but it is up to 12 times faster than purely structured merge, 5 times on average.

In summary, we make the following contributions:

- We present a structured-merge approach that is based on tree matching and amalgamation, and that uses auto-tuning to improve performance while largely maintaining precision.
- We provide a practical implementation, called JDime, of our approach for Java.
- We apply our tool to a substantial set of merge scenarios and compare its performance and conflict-detection capability (with and without auto-tuning) to that of unstructured merge.

JDime as well as the sources of the merge scenarios and the collected data of all experiments are available at a supplementary Web site: `http://fosd.net/JDime`.



Version *Base*

```
class Bag {
  int[] values;
  Bag(int[] v) { values = v; }
}
```

Version *Left*

```
class Bag {
  int[] values;
  Bag(int[] v) { values = v; }
  int[] get() {
    return values;
  }
}
```

Version *Right*

```
class Bag {
  int[] values;
  Bag(int[] v) { values = v; }
  int size() {
    return values.length;
  }
}
```

Unstructured merge

```
class Bag {
  int[] values;
  Bag(int[] v) { values = v; }
<<<<<<< Left
  int[] get() {
    return values;
  }
=======
  int size() {
    return values.length;
  }
>>>>>>> Right
}
```

Structured merge

```
class Bag {
  int[] values;
  Bag(int[] v) { values = v; }
  int[] get() {
    return values;
  }
  int size() {
    return values.length;
  }
}
```

Figure 1: Conflict resolved with structured merge but not with unstructured merge

2. SOFTWARE MERGE

In his seminal survey, Mens provides a comprehensive overview of the field of software-merge techniques [16]. Here, we concentrate on the popular scenario of a three-way merge, which is used in every practical version control system. A three-way merge aims at joining two independently developed versions based on their common ancestor (e.g., the version from which both have been derived) by locating their differences and selecting and applying corresponding changes to the merged version. However, the merge may encounter conflicts when changes of the two versions are inconsistent (e.g., two versions apply mutually exclusive changes at the same position) [16]. A major goal is to empower merge tools to detect and resolve conflicts automatically.

As software projects grow, merge techniques have to scale. In the remaining section, we discuss the principal properties of unstructured and structure merge with regard to conflict detection and resolution as well as performance.

2.1 Unstructured Merge

For illustration, we use a simple example: an implementation of a bag data structure that can store integer values. In Figure 1 (top), we show the basic version, called *Base*, which contains a Java class with a field and a constructor.

Based on version *Base*, two versions have been derived independently (middle of Figure 1): Version *Left* adds a method size and version *Right* adds a method get. Merging *Left* and *Right*, based on their common ancestor *Base*, using unstructured merge results in a conflict, as shown in Figure 1 (bottom left). The conflict cannot be resolved automatically by any unstructured-merge tool and thus requires manual intervention. The reason is that an unstructured-merge tool is not able to recognize that the text is actually Java code and that the versions can be merged safely: The declarations of the methods get and size can be included in any order because method declarations can be permuted safely in Java, as illustrated in Figure 1 (bottom right).

In practice, most unstructured-merge tools compare and merge versions based on *largest common subsequences* [3] of text lines. This is not without benefits. The unstructured approach is applicable to a wide range of different software artifacts and it is fast: quadratic in the length of the artifacts involved. Mens conjectures that 90 % of all merge scenarios require only unstructured merge; the other 10 % require more sophisticated solutions such as structured merge, a fraction that is likely to grow with the popularity of decentralized version control systems [16].

2.2 Structured Merge

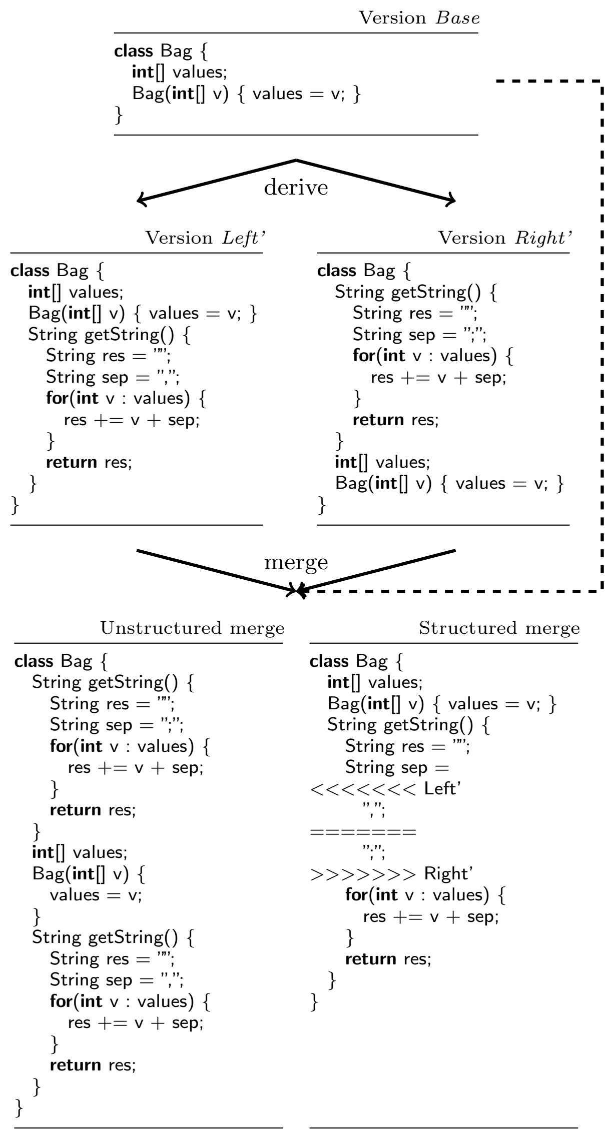
Structured merge aims at alleviating the problems of unstructured merge with regard to conflict detection and resolution by exploiting the artifacts' structure. Westfechtel and Buffenbarger pioneered this field by using structural information such as the context-free and context-sensitive syntax during the merge process [5, 20]. Subsequently, researchers proposed a wide variety of structural comparison and merge tools including tools for Java [2] and C++ [7] (see Sec. 5).

The idea underlying structured-merge tools is to represent the artifacts as trees (or graphs) and to merge them by tree (or graph) matching and amalgamation. Additionally, the merge process has all kinds of information on the language at its disposal, including information on which program elements can be permuted safely—which has been proved very useful in software merge [1]. This way, it is almost trivial to merge the two versions of Figure 1 (bottom right).

Structured merge is not only superior in that certain conflicts can be resolved automatically. There are situations in which unstructured merge misses conflicts that are detected by structured merge. In Figure 2, we show again the basic version of the bag example (top), but two other versions have been derived independently: *Left'* and *Right'*, both of which add a method getString (middle). Interestingly, unstructured merge (bottom left) does not report any conflict but results in a broken program that contains two methods getString: one before the declaration of array values and constructor Bag, as in version *Right'*, and one after, as in version *Left'*. In contrast, structured merge notices two versions of method getString and their difference in the initialization of the local variable sep, which results in a conflict reported to the user (bottom right). Note that conflicting code may be even well-typed and still misbehave.

On the downside, structured merge relies on information on the syntax of the artifacts to be merged. In practice, this means that one has to create one merge tool per artifact type or language. Although the creation of a merge tool can be automated to some extent, still manual effort is necessary to provide the specific information of the particular kind of artifact being processed [1]. Nevertheless, for languages that are widely used such as Java, it is certainly useful to spend the effort and to create and use a dedicated merge tool.

A more severe problem of structured merge —which we want to address here— is the run-time complexity of the



Version *Base*
```
class Bag {
  int[] values;
  Bag(int[] v) { values = v; }
}
```

Version *Left'*
```
class Bag {
  int[] values;
  Bag(int[] v) { values = v; }
  String getString() {
    String res = "";
    String sep = ",";
    for(int v : values) {
      res += v + sep;
    }
    return res;
  }
}
```

Version *Right'*
```
class Bag {
  String getString() {
    String res = "";
    String sep = ";";
    for(int v : values) {
      res += v + sep;
    }
    return res;
  }
  int[] values;
  Bag(int[] v) { values = v; }
}
```

Unstructured merge
```
class Bag {
  String getString() {
    String res = "";
    String sep = ";";
    for(int v : values) {
      res += v + sep;
    }
    return res;
  }
  int[] values;
  Bag(int[] v) {
    values = v;
  }
  String getString() {
    String res = "";
    String sep = ",";
    for(int v : values) {
      res += v + sep;
    }
    return res;
  }
}
```

Structured merge
```
class Bag {
  int[] values;
  Bag(int[] v) { values = v; }
  String getString() {
    String res = "";
    String sep =
<<<<<<< Left'
      ",";
=======
      ";";
>>>>>>> Right'
    for(int v : values) {
      res += v + sep;
    }
    return res;
  }
}
```

Figure 2: Conflict detected with structured merge but not with unstructured merge

internal merge algorithm. Typically, it relies on trees or graphs and corresponding matching and merging operations. Although there is the possibility of adjusting the complexity by considering only parts of the artifacts' structure (e.g., context-free syntax only) or by using a less precise matching, even these compromises result in at least cubic or even exponential time complexity. This inherent complexity seems to be a major obstacle to a practical application. In the next section, we present an approach based on tree matching and amalgamation paired with an auto-tuning approach to push back the limits of structured merge in this respect.

3. OUR APPROACH

Our approach has three ingredients:

- We represent artifacts as context-free syntax trees, including information on which program elements can be permuted safely.

- We use two tailored tree-matching algorithms, one for unordered and one for ordered child nodes (the former for program elements that can be permuted safely, the latter for those that must not be permuted); the rules for merge and conflict resolution are language-specific.
- We use the full power of structured merge only in situations in which unstructured merge reports conflicts.

3.1 Artifact Representation

We represent artifacts by terms of trees that reflect their context-free syntax. An alternative would be to model also the context-sensitive syntax, which would result in graphs rather than trees [20]. Of course, the matching and merging operations would be even more precise in this case, but also computationally even more complex.

The problems illustrated in Figure 1 (inability to resolve a conflict) and Figure 2 (inability to detect a conflict) arise from the fact that unstructured-merge tools have no information on which program elements can be permuted safely. We include this information in our structured-merge approach. For every type of program element (e.g., class declaration, method declaration, statement), the tool knows whether the corresponding elements can be permuted, and it uses this information during the matching and merge operations.

3.2 Algorithms

The overall merge process involves three phases: (1) calculating a matching between the trees of the input versions, (2) amalgamating the trees based on the calculated matching, and (3) resolving conflicts during the merge operation. Next, we discuss all three phases in detail.

Tree Matching. Tree matching takes two trees, computes the *largest common subtree*, and adds matching information to them. Matching of nodes depends on their syntactic category (e.g., two field declarations are considered equal if their types and names match). Tree matching distinguishes between ordered nodes (which must not be permuted) and unordered nodes (which can be permuted safely).

For ordered nodes, we use a variation of Yang's algorithm [21]: We compute for all pairs (A_i, B_j) recursively the number of matches (W) and the maximum matching (M), as shown in Algorithm 1. Note, the recursive call invokes TreeMatching, which calls OrderedTreeMatching or UnorderedTreeMatching (Algorithm 2), depending on whether the nodes at this level are ordered or unordered. The problem of finding the largest common subtree of ordered trees is *quadratic* in the number of nodes [21].

Algorithm 1 Ordered Tree Matching

```
function OrderedTreeMatching(Node $A$, Node $B$)
    if $A \neq B$ then return 0                              ▷ Nodes do not match
    end if
    $m \leftarrow$ number of children of $A$
    $n \leftarrow$ number of children of $B$
    Matrix $M \leftarrow (m+1) \times (n+1)$                 ▷ Initialize auxiliary matrix
    for $i \leftarrow 1..m$ do
        for $j \leftarrow 1..n$ do
            $W[i,j] \leftarrow$ TreeMatching$(A_i, B_j)$     ▷ Matching for children
            $M[i,j] \leftarrow \max(M[i,j-1], M[i-1,j], M[i-1,j-1] + W[i,j])$
        end for
    end for
    return $M[m,n] + 1$                                      ▷ Return maximum number of matches
end function
```

For unordered nodes, we solve the problem using a linear-programming approach, as shown in Algorithm 2. Again, we compute for all pairs (A_i, B_j) the number of matches, recursively. Finding the highest number of matches in the resulting matrix M is equivalent to computing the maximum number of matches in a weighted bipartite graph, which can be solved in *cubic* time [17]. We express the problem as a linear program and solve it using a linear-program solver, which proved to be fast in our experiments. In Algorithm 2, SolveLP creates the constraint matrix the input matrix, invokes the solver, and returns the maximum number of matches, of which we can compute the actual matching.

Algorithm 2 Unordered Tree Matching

```
function UnorderedTreeMatching(Node $A$, Node $B$)
    if $A \neq B$ then return 0                              ▷ Nodes do not match
    end if
    $m \leftarrow$ number of children of $A$
    $n \leftarrow$ number of children of $B$
    Matrix $M \leftarrow (m) \times (n)$                     ▷ Initialize auxiliary matrix
    for $i \leftarrow 0..m$ do
        for $j \leftarrow 0..n$ do
            $M[i,j] \leftarrow$ TreeMatching$(A_i, B_j)$     ▷ Matching for children
        end for
    end for
    $sum \leftarrow$ SolveLP$(M)$                            ▷ Prepare and invoke LP solver
    return $sum + 1$                                         ▷ Return maximum number of matches
end function
```

Overall, we perform tree matching on each pair of trees: (left, base), (right, base), (left, right). As a result, the nodes of each tree are tagged with information on the nodes of the other versions that they match.

Note that tree matching based on computing the largest common subtree compares the input trees level-wise. Algorithms that compare trees across levels are more precise but also more complex, as we discuss in Section 6. We provide more details on the two matching algorithms elsewhere [12].

Tree Amalgamation. Tree amalgamation takes the three trees enriched with matching information (base, left, and right tree) and creates a merged tree as result. To this end, it distinguishes three kinds of nodes:

- Unchanged nodes that are contained in all three trees.
- Consistently changed or added nodes that are contained in the left *and* the right tree, but not in the base tree.
- Independently changed or added nodes that are contained *either* in the left *or* the right tree, and not in the base tree.

Based on this distinction, the algorithm fills the merged tree in three steps, as shown in Algorithm 3.

Algorithm 3 Tree Amalgamation (Merge)

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

Merging unchanged and consistently changed or added nodes is rather simple, because the left and the right version are not in conflict. (To perform two-way merges, the first step, the insertion of unchanged nodes, is omitted.) The challenge is to apply changes introduced by only one version. Finding such independent changes is easy, using matching information attached to the nodes. But, before we can apply the independent changes to the merge tree, we have to check whether they conflict with changes of the other respective version. To this end, a list of all independent changes of the left and right version is passed to the phase of conflict detection and resolution.

Conflict Detection and Resolution. To detect possible conflicts between two versions (left and right), each change of a version is checked against each of the other version. For brevity, we explain here only insertion conflicts; others, such as deletion-insertion conflicts, are handled similarly [12].

An insertion conflict occurs potentially when two nodes are inserted concurrently at the same parent node in the merge tree. Here again, the algorithm has to distinguish between ordered and unordered nodes. For ordered nodes, the insertion positions are decisive: if they overlap, the nodes are flagged as conflicting. For example, including a statement s_1 in a block at the first position does not conflict with another statement s_2 included later in the block; only if s_1 and s_2 are added to the same position, they are in conflict.

Whether unordered nodes are in conflict, depends on their type. Some nodes must be unique in the scope of their parent. So, two nodes added with the same name may be in conflict, even though added in different positions. For example, a class declaration must not contain multiple field declarations with the same name; two independently added fields with the same name raise a conflict. But a field does not conflict with a method. At this point, it becomes again apparent that structured merge is not language-independent. Beside information on which program elements can be permuted safely, conflict detection depends highly on the language specifics. Due to this fact and the sheer size, we cannot show Algorithm DetectConflicts. Its implementation is available on the supplementary Web site.

After conflict detection, the changes of both versions are inserted into the merge tree, as we show in Algorithm 4. Nodes not marked as conflicting are inserted straightforwardly. A node flagged as conflicting contains a list of conflicting changes that belong to the other version and a list of related changes that belong to its own version. Technically, we use dummy nodes to store conflicting nodes; the pretty printer displays conflicts according to this information.

Algorithm 4 Merge Changes

```
function MergeChanges(Tree $merged$, List $changes$)
  for all Node $n$ in $changes$ do
    if isconflict($n$) $\wedge$ $\neg$isprocessed($n$) then
      Node $c \leftarrow$ new dummy node
      $c.lvariant \leftarrow$ own($n$)                ▷ Store own changes
      $c.rvariant \leftarrow$ other($n$)              ▷ Store other changes
      insert($c, merged$)
      for all Node $m$ in own($n$) $\cup$ other($n$) do
        markprocessed($m$)
      end for
    else
      insert($n, merged$)
    end if
  end for
end function
```

3.3 Auto-Tuning

The algorithms presented in Section 3.2 are computationally complex. In particular, computing the largest common subtree is cubic in the number of (unordered) program elements [17]. This is a limitation of structured merge, compared to the quadratic-time algorithm of unstructured merge. But we do not want to abandon structured merge entirely. Instead, we strive for a balance between employing syntactic information to detect and resolve conflicts and attaining acceptable performance.

The idea is simple. We use unstructured merge as long as no conflicts are detected. The rationale is that, in software merge, usually only few parts of a program are changed and even fewer participate in conflicts, as postulated by Mens' 90/10 rule [16]. So, for most parts of a program, we can save the computation time for an expensive tree matching. However, this way, we may also miss conflicts due to the imprecision of unstructured merge. This is the price we pay for improving performance, but our experiments suggest that the price is acceptable, as we discuss in Section 4. Once unstructured merge detects conflicts, we use structured merge selectively on a per-file basis (i.e., for triples of file versions) instead. This way, we take advantage of the capabilities of structure merge to detect and resolve conflicts.

4. EVALUATION

To evaluate our approach, especially the balance between precision and performance that we strive for with auto-tuning, we implemented a prototype of a structured-merge tool, called JDime, and we conducted a series of experiments based on 8 real-world software projects. The tool as well as all merge scenarios and experimental data are available at the supplementary Web site.

4.1 Implementation

We have implemented JDime on top of the JastAddJ compiler framework.[1] The implementation was straightforward because JastAddJ provides excellent extension capabilities. The foundation for our artifact representation are the abstract-syntax trees generated by JastAddJ. For technical reasons, we had to build our own tree representation on top of it. We implemented the matching and merging algorithms straightforwardly by means of visitors and aspects. For unordered tree matching, we used the GLPK solver.[2] Information on which program elements can be reordered safely was included based on the Java language specification. We also took care of the fact that Java code usually comes with comments: They are extracted during parsing and put back in after the merge.

4.2 Hypotheses and Research Questions

To make our expectations precise, we pose four hypotheses and a research question:

- **H_1** The conflicts reported by unstructured merge differ from the ones reported by structured merge in terms of frequency, size, and kind.
- **H_2** Unstructured merge is substantially faster than structured merge.
- **H_3** Auto-tuning does not miss many conflicts detected by purely structured merge.

[1] `http://jastadd.org/web/jastaddj/`
[2] `http://www.gnu.org/software/glpk/`

Table 1: Overview of the sample projects (all from `http://sourceforge.net/`)

Project	Domain	Merge scenarios	Lines of code
DrJava	Development environment	9	89 K
FreeCol	Turn-based strategy game	10	86 K
GenealogyJ	Editor for genealogic data	10	56 K
iText	PDF library	8	71 K
jEdit	Programmer's text editor	8	107 K
Jmol	Molecule viewer	7	135 K
PMD	Bug finder	10	71 K
SQuirrelSQL	Graphical SQL client	10	218 K

H_4 Auto-tuning is substantially faster than purely structured merge.

R_1 What fraction of a merge scenario (in terms of files) can be handled by unstructured merge in that no conflicts are reported? In other words, can we confirm Mens' postulate that 90 % of merge scenarios can be handled properly with unstructured merge?

4.3 Sample Projects

We selected 8 open-source Java projects that we have used in the past to assess and compare merge approaches [1].[3] The projects are of reasonable but varying sizes, from different domains, and have a substantial version history. For each project, there are multiple merge scenarios that give rise to conflicts [1]. Technically, a merge scenario is a triple consisting of a base, a left, and a right version, whereby the base version is the common ancestor of the other two.

In Table 1, we list information on the sample projects including name, domain, number of merge scenarios, and number of lines of code. Each of the 8 projects comes with 7 to 10 merge scenarios. All 72 merge scenarios together consist of more than 17 million lines of Java code. They are available and documented at the supplementary Web site.

4.4 Methodology

Our method of evaluation was twofold. First, we compared unstructured and structured merge (with and without auto-tuning) with regard to conflict detection and performance. Second, we analyzed a subset of conflicts manually in order to learn more about the capabilities of unstructured and structured merge. Overall, we applied our merge tool to each of the 72 merge scenarios thrice: using unstructured merge, purely structured merge, and structured merge with auto-tuning. For each merge pass, we measured the execution time 10 times and computed the median, and we counted the number of reported conflicts and conflicting lines of code. We conducted all measurements on a desktop machine (AMD Phenom II X6 1090T, with 6 cores @3.2 GHz, and 16 GB RAM) with Gentoo Linux (Kernel 3.2.7) and Oracle Java HotSpot 64-Bit Server VM 1.6.0_31.

4.5 Results

In Figure 3, we depict the average number of conflicts for each project as reported by unstructured merge, purely structured merge, and structured merge with auto-tuning.

[3]We had to exclude 2 projects and 8 merge scenarios due to technical problems with the JastAddJ implementation; the problems are not related to our approach.

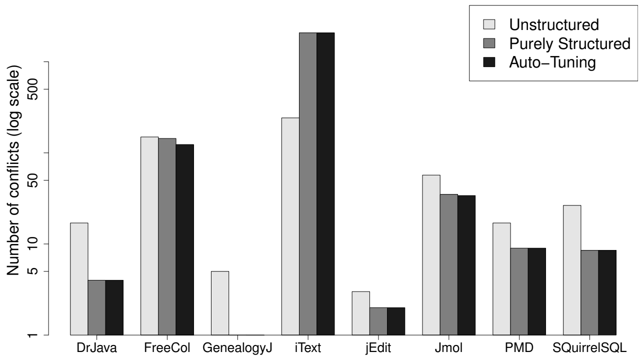


Figure 3: Number of reported conflicts

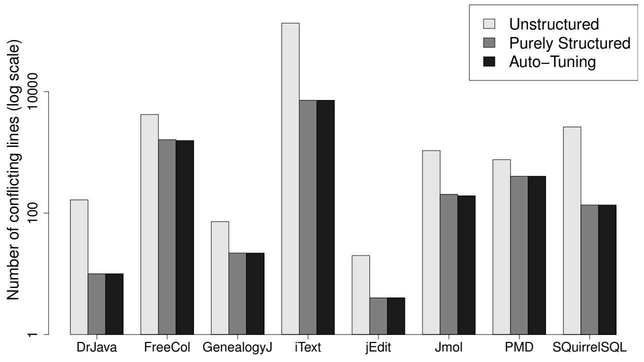


Figure 4: Number of conflicting lines of code

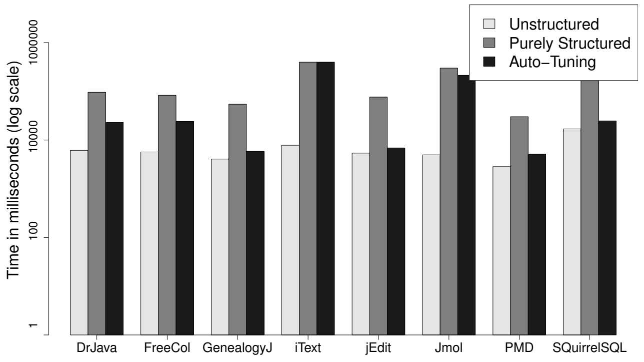


Figure 5: Merging time in seconds

Similarly, in Figure 4, we depict the respective numbers of lines of code involved in conflicts. Finally, in Figure 5, we depict the times consumed by three merge approaches. Table 2 shows the experimental data for all merge scenarios. All raw data, on a per-file-scenario basis, are available at the supplementary Web site.

At a glance, the numbers and sizes of conflicts reported by unstructured merge differ significantly from the ones reported by structured merge. In almost all projects, structured merge reports fewer and smaller conflicts. Interestingly, purely structured merge and structured merge with auto-tuning report almost similar numbers of conflicts, which means that only few conflicts are missed due to the selective use of unstructured merge (apart from the fact that the reported sets of conflicts are equal). With regard to performance, structured merge is substantially slower than unstructured merge: unstructured merge is up to 71 times faster, 15 times on average. But structured merge with auto-tuning is up to 12 times faster than purely structured merge, 5 times on average.

4.6 Discussion

Hypotheses & research questions. Based on the results, we can confirm hypothesis $\mathbf{H}_1$: The conflicts reported by unstructured merge differ significantly in terms of number, size, and kind from the ones reported by structured merge. In all projects but iText, structured merge is able to resolve more merge conflicts than unstructured merge. On average, structured merge reports 39 % of the number of conflicts and 24 % of the number of conflicting lines of unstructured merge. Analyzing a random subset of conflicts, we found that these numbers are mainly due to ordering conflicts that cannot be handled properly in unstructured merge. Project iText is an outlier in that structured merge reports significantly more conflicts. A manual inspection revealed that this is due to a renaming in the directory structure in the project. This leads the three-way merge to miss a version in the triple, which results in one conflict per file with unstructured merge and many conflicts at the syntactic level with structured merge. A similar problem occurs in FreeCol, although less serious.

Interestingly, our experiments support hypothesis $\mathbf{H}_3$: The conflicts reported by purely structured merge are largely the same as the ones reported by using the auto-tuning approach. That is, the strategy to use unstructured merge as long as no conflicts are detected, and to switch upon detection of a conflict to structured merge, seems to suffice. Not many conflicts are being missed: up to 26 (in FreeCol) and 2 on average. Considering the performance gains and the state of the art, this seems acceptable.

Furthermore, our experiments confirm hypothesis $\mathbf{H}_2$: In all projects, structured merge is substantially slower than unstructured merge. Unstructured merge is up to 71 times faster than structured merge, 15 times on average. This result does not need much interpretation. Although we could optimize JDime further, we cannot escape the complexity of the algorithms involved in the structured merge process.

Also, we can confirm hypothesis $\mathbf{H}_4$: structured merge with auto-tuning is faster than purely structured merge in almost all projects: up to 12 times and 5 times on average. In the projects GenealogyJ, jEdit, PMD, and SQuirrelSQL, it is even in the order of the time of using unstructured merge. So, auto-tuning seems to be, at least, promising to hit a sweet spot between precision and performance. In Jmol and iText, very large files diminish the relative benefit of auto-tuning (up to 36 000 nodes per file and 1 000 nodes per level).

Finally, as for research question $\mathbf{R}_1$, we found that 21 % of the changed files cannot be merged with unstructured merge—twice as many files as predicted by Mens. With structured merge, this fraction can be decreased to 15 %. So, we cannot confirm that 90 % of merge scenarios can be handled properly with unstructured merge.

Runtime complexity. In Figure 6, we compare the file size in terms of lines of code (mean number of lines of code of the file versions involved in a merge) and the time needed for the merge of the file versions in question using unstructured and purely structured merge. Apart from two groups

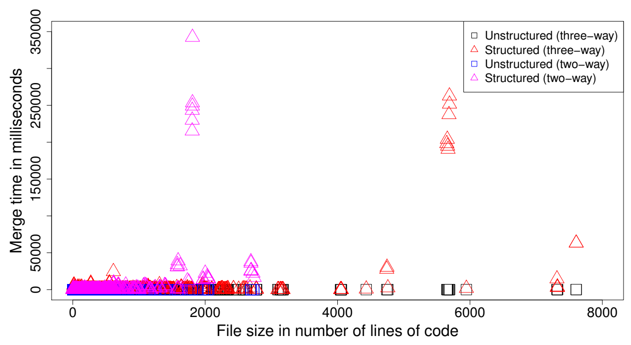


Figure 6: File size (in number of lines of code) versus merge time (in milliseconds)

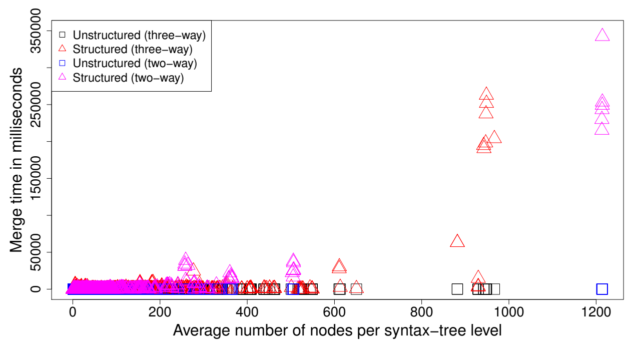


Figure 7: Average number of nodes per syntax-tree level versus merge time (in milliseconds)

of outliers at files sizes of 2 000 and of 6 000 lines of code, the merge time grows smoothly with the file size. While the group at 6 000 can be explained with the cubic runtime complexity of the algorithms involved, we were curious as to why there are so many outliers around file sizes of 2 000 lines of code. An analysis revealed that these outliers stem from merges that are actually two-way, mainly due to a renaming in project iText (see above). So, two-way merges are substantially more expensive than three-way merges, and the corresponding merge time grows more quickly than for three-way merges. Note that, in the color version of the paper, data points related to two-way merges are displayed with alternative colors.

In our quest of understanding the merits of structured merge, we computed a number of further statistics such as the average number of nodes, depths, and widths of the syntax trees involved in a merge. We found that the number of nodes per syntax-tree level is a more accurate measure than lines of code: The merge time grows smoothly, polynomially with the number of nodes per syntax-tree level, as displayed in Figure 7. Notice that the outlier group of Figure 6 moved to the right, which demonstrates that small files may be more complex to merge than large files (when there are many nodes per level in the syntax trees).

Furthermore, we were interested in the role of conflicts for the time needed for merging. Our analysis revealed that almost all outliers of merge times (> 50 000 milliseconds) stem from structured merges that reported conflicts. So, the insight is that structured merge is able to resolve many conflicts that unstructured merge is not able to handle, and that structured merge is able to do so in acceptable time. But, if structured merge encounters conflicts it cannot resolve, its time consumption increases substantially.

Further observations. To learn more about the capabilities of structure merge, we inspected a subset of the conflicts manually. Next, we report the most interesting observations.

Tree matching is at the heart of structured merge. Its precision is much higher than that of unstructured merge, but it is not perfect. We found situations in which structured merge was not able to resolve a conflict, even though it could be resolved manually. The reason is that algorithms based on computing largest common subtrees consider only corresponding tree levels. To establish a matching across different levels (e.g., to detect shifted code), one can use algorithms that compute *largest common embedded subtrees*, but they are generally $\mathcal{APX}$-hard [22].[4] It will be an interesting avenue of further research to incorporate even such complex algorithms during auto-tuning, including approximations.

Furthermore, we found that most conflicts raised by unstructured merge are related to the order of program elements. These conflicts can be handled by structured merge—be it in terms of automatic conflict resolution, as in Figure 1, or in terms of uncovering hidden conflicts, as in Figure 2. We also found that conflicts reported by structured merge are typically fine-grained and align with the syntactic program structure. With unstructured merge, the conflicts are typically larger and often crosscut the syntactic program structure, which makes them harder to track and understand. Project iText is an extreme case: due to a renaming in the project's directory structure, often one component of the merge triple is missing. In such cases, the three-way merge becomes actually a two-way merge. For unstructured merge, we get a large conflict per file; for structured merge, we get many small conflicts. So, two-way merges challenge structured merge, which shall be an avenue of further research.

4.7 Threats to Validity

In empirical research, a threat to internal validity is that the data gathered may be influenced by (hidden) variables—in our case, variables other than the kind of merge. Due to the simplicity of our setting, we can largely rule out such confounding variables. We applied unstructured and structured merge to the same set of merge scenarios and counted the number of conflicts and lines of conflicting code that occurred in the merged code. We performed all performance measurements repeatedly in order to minimize measurement bias. Furthermore, we used a comparatively large sample to rule out confounding variables such as programming experience and style.

A common issue is whether we can generalize our conclusions to other projects, of other domains and written in other languages. To increase external validity, we collected a substantial number of projects and merge scenarios. We argue that the simplicity of our setting as well as the randomized sample allow us to draw conclusions beyond the projects we looked at. Our findings should even apply to languages that are similar to Java (e.g., C#).

[4]The set of $\mathcal{APX}$ problems is a subset of $\mathcal{NP}$ optimization problems for which polynomial-time approximation algorithms can be found.

5. RELATED WORK

Structured Merge. After the seminal work of Westfechtel [20] and Buffenbarger [5], many proposals of structured-merge techniques have been made. On the one hand, there are proposals for structured-merge tools that are specific to mainstream programming languages such as Java [2] and C++ [7]. On the other hand, there are many proposals of structured two-way and three-way merge techniques for modeling artifacts [11,15,19]—a comprehensive bibliography is available on the Web.[5] The approaches are mostly based on graphs, which allow precise merging but harm scalability. So, it is unclear how they perform on projects of the size of our case studies. Although aiming at modeling, the different representations and merge algorithms are promising input for our auto-tuning approach. Additionally, renaming analysis could be integrated to further improve the precision of tree matching [8].

Semistructured Merge. Semistructured merge aims at another sweet spot: one between precision in conflict handling and generality in the sense that many artifact types can be processed [1]. Much like structured merge, semistructured merge represents artifacts as trees. But an artifact is only partly exposed in the tree, the rest is treated as plain text—that is why it is called 'semistructured'. This way, a certain degree of language independence can be achieved by a generic merge algorithm that merges artifacts by tree superimposition and that concentrates on ordering conflicts. Language-specific information is fed into the merge engine via a plugin mechanism. Semistructured merge is less precise than structured merge because only parts of an artifact are treated structurally. For example, bodies of Java methods are treated as plain text and merged using a line-based approach. Our experiments with an existing implementation of semistructured merge[6] confirm it to perform between structured and unstructured merge in terms of conflict handling (it is able to resolve only 50 % of conflicts, compared to structured merge), but even significantly slower than structured merge (on average, 5 times slower in our sample merge scenarios). At first sight, the latter finding is surprising, but semistructured merge was not designed with performance in mind and, possibly, the language independence attained by the plugin mechanism has to be paid for in terms of performance penalties. Finally, auto-tuning was not considered in semistructured merge, but could be combined with it.

Other Approaches. A trace of which change operations gave rise to the different versions to be merged can help in the detection and resolution of conflicts [6, 10, 13, 18]. However, such an operation-based approach is not applicable when tracing information is not available, which is common in practice. Other approaches require that the documents to be merged come with a formal semantics [4, 9], which is rarely feasible in practice; even for mainstream languages such as Java, there is no formal semantics available. Finally, approaches that rely on model finders and checkers for semantic merge have serious limitations with regard to scalability [14].

[5]`http://pi.informatik.uni-siegen.de/CVSM/`
[6]`http://http://fosd.net/SSMerge`

6. CONCLUSION

We offer an approach to software merging that aims at a proper balance between precision and performance. First, it represents software artifacts as trees and merges them using tree matching and amalgamation. Second, it relies on auto-tuning to improve the performance. The idea is to use unstructured merge as long as no conflicts occur, and to switch to the more precise structured merge when conflicts are detected. This way, expensive differencing and merge operations are used only in relevant situations. The auto-tuning approach may miss critical conflicts due to the imprecision of the unstructured merge involved, but experiments suggest that this happens only to an acceptable degree.

We developed the tool JDime, which implements our approach for Java, and applied it in a series of experiments on 8 real-world projects, including 72 merge scenarios with over 17 million lines of code. We found that, in almost all projects, structured merge reports fewer and smaller conflicts than unstructured merge, and structured merge is substantially slower. Interestingly, purely structured merge and structured merge with auto-tuning report almost similar sets of conflicts, but the auto-tuning approach is up to 12 times faster, 5 times on average.

Our results give us confidence that auto-tuning is a viable approach to balancing precision and performance in software merging. However, we explored only a subset of possible options. For example, we base our approach on syntax trees and algorithms that compute largest common subtrees. But other representations and algorithms are possible. Several approaches —especially, in the modeling community [11, 19]— represent software artifacts as graphs, for example, incorporating the context-sensitive syntax [20]. While graph algorithms are computationally more complex than corresponding tree algorithms, they are also more precise. An interesting issue is how we can exploit the wealth of representations and algorithms during the merge process. We believe that, with the auto-tuning approach, we have made a step in the right direction. Future approaches of auto-tuning shall be more flexible in adjusting precision. The choice of the amount of information used as well as of the algorithms involved could be guided by knowledge collected on-line during the merge process. For example, it is promising to selectively use graph-based algorithms to resolve conflicts that tree-based algorithms cannot resolve, or to try to detect abnormal situations such as the many two-way merges in iText and to react on-the-fly by selecting a more suitable algorithm.

7. ACKNOWLEDGMENTS

We thank Christian Kästner for feedback on the potential of this work, and for the design we used in the Figures 1 and 2. This work has been supported by the German Research Foundation (DFG—AP 206/2, AP 206/4, and LE 912/13).

8. REFERENCES

[1] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner. Semistructured Merge: Rethinking Merge in Revision Control Systems. In *Proc. ESEC/FSE*, pages 190–200. ACM, 2011.

[2] T. Apiwattanapong, A. Orso, and M. Harrold. JDiff: A Differencing Technique and Tool for Object-Oriented Programs. *Automated Software Engineering*, 14(1):3–36, 2007.

[3] L. Bergroth, H. Hakonen, and T. Raita. A Survey of Longest Common Subsequence Algorithms. In *Proc. SPIRE*, pages 39–48. IEEE, 2000.

[4] V. Berzins. Software Merge: Semantics of Combining Changes to Programs. *ACM TOPLAS*, 16(6):1875–1903, 1994.

[5] J. Buffenbarger. Syntactic Software Merging. In *Selected Papers from SCM-4 and SCM-5*, LNCS 1005, pages 153–172. Springer, 1995.

[6] D. Dig, K. Manzoor, R. Johnson, and T. Nguyen. Refactoring-Aware Configuration Management for Object-Oriented Programs. In *Proc. ICSE*, pages 427–436. IEEE, 2007.

[7] J. Grass. Cdiff: A Syntax Directed Differencer for C++ Programs. In *Proc. USENIX C++ Conference*, pages 181–193. USENIX Association, 1992.

[8] J. Hunt and W. Tichy. Extensible Language-Aware Merging. In *Proc. ICSM*, pages 511–520. IEEE, 2002.

[9] D. Jackson and D. Ladd. Semantic Diff: A Tool for Summarizing the Effects of Modifications. In *Proc. ICSM*, pages 243–252. IEEE, 1994.

[10] M. Koegel, J. Helming, and S. Seyboth. Operation-Based Conflict Detection and Resolution. In *Proc. CVSM*, pages 43–48. IEEE, 2009.

[11] D. Kolovos, R. Paige, and F. Polack. Merging Models with the Epsilon Merging Language (EML). In *Proc. MODELS*, LNCS 4199, pages 215–229. Springer, 2006.

[12] O. Leßenich. Adjustable Syntactic Merge Tool For Java. Master's thesis, University of Passau, 2012.

[13] E. Lippe and N. van Oosterom. Operation-Based Merging. In *Proc. SDE*, pages 78–87. ACM, 1992.

[14] S. Maoz, J. Ringert, and B. Rumpe. CDDiff: Semantic Differencing for Class Diagrams. In *Proc. ECOOP*, LNCS 6813, pages 230–254. Springer, 2011.

[15] A. Mehra, J. Grundy, and J. Hosking. A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. In *Proc. ASE*, pages 204–213. ACM, 2005.

[16] T. Mens. A State-of-the-Art Survey on Software Merging. *IEEE TSE*, 28(5):449–462, 2002.

[17] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency.* Springer, 2002.

[18] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer. Conflict Detection for Model Versioning Based on Graph Modifications. In *Proc. ICGT*, LNCS 6372, pages 171–186. Springer, 2010.

[19] C. Treude, S. Berlik, S. Wenzel, and U. Kelter. Difference Computation of Large Models. In *Proc. ESEC/FSE*, pages 295–304. ACM, 2007.

[20] B. Westfechtel. Structure-Oriented Merging of Revisions of Software Documents. In *Proc. SCM*, pages 68–79. ACM, 1991.

[21] W. Yang. Identifying Syntactic Differences Between Two Programs. *Software: Practice and Experience*, 21(7):739–755, 1991.

[22] K. Zhang and T. Jiang. Some MAX SNP-hard Results Concerning Unordered Labeled Trees. *Information Processing Letters*, 49(5):249–254, 1994.

Table 2: Experimental data (LOC: lines of code; UM: unstructured merge; SM: purely structured merge; AT: structured merge with auto-tuning)

Project	Revision	# LOC	# Files	# Conflicts			# Conflicting lines			Merge time in milliseconds		
				UM	SM	AT	UM	SM	AT	UM	SM	AT
DrJava	rev3734-3786	104652	676	17	6	6	99	163	163	5556	95582	39398
DrJava	rev3734-3788	106196	680	18	6	6	131	163	163	5557	100648	44350
DrJava	rev3734-3807	104218	671	28	4	4	876	73	73	5663	69549	11692
DrJava	rev4989-5004	134137	697	10	12	11	499	125	95	6029	95760	22869
DrJava	rev4989-5019	147840	727	27	1	1	160	2	2	6186	83276	13749
DrJava	rev4989-5044	169262	798	48	1	0	265	2	0	6278	81310	16619
DrJava	rev4989-5058	152126	748	13	1	0	133	4	0	6163	81592	14812
DrJava	rev5319-5330	111496	614	15	4	4	236	10	10	6158	145662	68712
DrJava	rev5319-5332	111336	613	11	3	3	165	8	8	6150	145734	70869
FreeCol	rev5884-5962	133201	587	5	7	6	363	17	15	5160	74649	7307
FreeCol	rev5884-6055	150074	637	14	8	8	1094	328	328	5285	75085	9579
FreeCol	rev5884-6110	151883	645	18	9	9	1121	621	621	5276	75268	9600
FreeCol	rev5884-6265	162731	682	30	15	14	1766	684	680	5355	75738	10803
FreeCol	rev5884-6362	169143	701	42	45	24	2630	863	806	5458	78238	13333
FreeCol	rev5884-6440	195346	833	257	243	223	5784	2383	2330	5924	87671	34643
FreeCol	rev5884-6616	205104	897	480	403	383	6358	2788	2735	6157	89137	47010
FreeCol	rev5884-6672	208381	925	488	404	384	6884	3044	2991	6205	90245	48095
FreeCol	rev5884-6742	211313	943	497	431	405	7985	3230	3162	6220	91477	48572
FreeCol	rev5884-6843	214783	990	511	438	412	9229	3494	3426	6219	87880	49211
GenealogyJ	rev5531-5561	80131	654	1	0	0	3	0	0	4017	54710	4513
GenealogyJ	rev5531-5725	95691	752	15	5	5	89	75	75	3950	52353	4728
GenealogyJ	rev5537-5610	84207	640	68	23	23	556	188	188	4108	54049	9364
GenealogyJ	rev5537-5673	84207	640	68	23	23	556	188	188	4032	52881	9402
GenealogyJ	rev5676-6013	88515	688	1	0	0	3	0	0	3929	52942	4310
GenealogyJ	rev5676-6125	92715	704	1	0	0	3	0	0	3848	52995	4323
GenealogyJ	rev6127-6244	75394	555	2	1	1	30	16	16	4445	61138	5955
GenealogyJ	rev6127-6310	79542	578	5	2	2	56	38	38	4455	61860	6218
GenealogyJ	rev6127-6410	81868	592	5	1	1	94	22	22	4504	61223	5794
GenealogyJ	rev6127-6531	85887	620	6	1	1	106	22	22	4451	59632	6546
iText	rev2818-3036	178300	687	55	65	60	2897	569	451	4501	105082	18275
iText	rev2818-3191	192288	841	178	1834	1834	115577	6284	6284	5949	135344	130619
iText	rev2818-3306	204662	929	234	2097	2097	133913	6943	6943	7206	366823	345364
iText	rev2818-3392	210521	966	249	2263	2263	138768	7791	7791	7851	406303	389471
iText	rev2818-3560	220093	1007	261	2098	2098	142685	7391	7391	7984	430900	426416
iText	rev2818-3625	219242	1033	264	2358	2358	136078	7825	7825	7929	411803	404812
iText	rev2818-3988	222996	1067	252	2059	2059	135368	7794	7794	7790	387069	404926
iText	rev2818-4022	222640	1055	235	2005	2005	132585	7067	7067	8091	463090	407869
jEdit	rev4676-4998	147473	614	3	0	0	28	0	0	3584	49316	4033
jEdit	rev16588-16755	136345	579	2	2	2	9	7	7	5405	76507	6840
jEdit	rev16588-16883	141584	585	2	1	1	9	2	2	5411	75910	6811
jEdit	rev16588-17060	144900	596	3	2	2	24	4	4	5420	76301	6921
jEdit	rev16588-17316	148189	613	3	2	2	20	4	4	5377	75085	6926
jEdit	rev16588-17492	151264	628	3	2	2	20	4	4	5375	76129	6876
jEdit	rev16588-17551	155592	644	3	2	2	20	4	4	5381	77070	6932
jEdit	rev16883-16964	116297	545	20	7	7	192	100	100	5435	73973	7359
Jmol	rev11338-11438	221293	546	9	3	2	350	20	10	4721	299620	211114
Jmol	rev11338-11538	235825	570	25	12	11	521	67	57	4786	297964	213165
Jmol	rev11338-11638	258871	646	35	17	16	667	78	68	4893	288328	208357
Jmol	rev11338-11738	273340	701	57	35	34	1070	203	193	4970	281843	203392
Jmol	rev11338-11838	283246	781	78	46	45	1637	333	323	5068	329436	270839
Jmol	rev11338-11938	286014	795	87	48	47	1703	344	334	5080	344811	277888
Jmol	rev11338-12038	292234	826	100	53	53	1869	593	593	5041	355751	289234
PMD	rev5929-6010	111295	1257	1	0	0	5	0	0	4155	42865	4371
PMD	rev5929-6135	125151	1298	3	2	2	22	17	17	4136	43667	4875
PMD	rev5929-6198	129201	1360	12	4	4	784	342	342	3407	34931	4635
PMD	rev5929-6296	132680	1391	14	6	6	734	349	349	3360	33896	4833
PMD	rev5929-6425	145903	1495	14	6	6	727	349	349	2842	29597	4640
PMD	rev5929-6595	149713	1555	20	12	12	670	462	462	2814	29879	5373
PMD	rev5929-6700	150883	1556	24	15	15	1003	539	539	2836	30182	5507
PMD	rev5929-6835	153686	1629	21	16	16	939	506	506	2775	30094	5419
PMD	rev5929-7018	156538	1651	27	18	18	1053	510	510	2793	28456	6169
PMD	rev5929-7073	158929	1674	32	19	19	977	498	498	2817	30117	6400
SQuirrelSQL	rev4007-4051	169033	1734	1	0	0	4	0	0	16938	215394	18358
SQuirrelSQL	rev4007-4103	187571	1827	7	1	1	299	7	7	16968	222290	19273
SQuirrelSQL	rev4007-4212	216842	1997	22	8	8	1384	108	108	16831	213916	21380
SQuirrelSQL	rev4007-4321	237495	2089	31	9	9	3862	164	164	16906	216931	23217
SQuirrelSQL	rev4007-4394	254834	2182	36	12	12	4218	357	357	16902	212638	23996
SQuirrelSQL	rev4007-4516	266666	2292	42	15	15	4258	414	414	16955	224109	25483
SQuirrelSQL	rev4007-4908	309397	2880	50	18	18	6111	420	420	16644	208173	26564
SQuirrelSQL	rev4007-5081	314760	2916	52	20	20	4574	615	615	16524	209951	26406
SQuirrelSQL	rev5082-5155	231115	2398	3	1	1	17	50	50	23351	286927	25940
SQuirrelSQL	rev5082-5351	270787	2620	7	3	3	194	9	9	23275	292464	27264