Autom Softw Eng (2015) 22:367–397
DOI 10.1007/s10515-014-0151-5

Balancing precision and performance in structured merge

Olaf Leßenich · Sven Apel · Christian Lengauer

Received: 14 September 2013 / Accepted: 20 April 2014 / Published online: 28 May 2014

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 50 real-world JAVA projects, involving 434 merge scenarios with over 51 million lines of code, demonstrate that our approach indeed hits a sweet spot: While largely maintaining a precision that is superior to that of unstructured merge, structured merge with auto-tuning is up to 92 times faster than purely structured merge, 10 times on average.

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.

O. Leßenich (✉) · S. Apel · C. Lengauer
Innstr. 33, 94032 Passau, Germany
e-mail: lessenic@fim.uni-passau.de

Contemporary software-merging techniques can be classified into (1) syntactic approaches and (2) semantic approaches. The former include (1 a) unstructured approaches that treat software artifacts as sequences of text lines and (1 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 suffices that treats all software artifacts equally. 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 (Mens 2002; Apel et al. 2011).

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 (Mens 2002; Westfechtel 1991; Buffenbarger 1995; Apel et al. 2011), which also opens new opportunities for detecting and resolving merge conflicts (Apel et al. 2011)—one of the key problems in this field (Mens 2002). Unfortunately, no practical structured-merge tools for mainstream programming languages are available. Why?

A first problem is certainly that, when developing a structured-merge tool, one must commit to a particular artifact language and, as a consequence, develop and use a distinct tool per language. A second problem is that algorithms that take the structure of the artifacts involved into account are typically at least of cubic, if not even $\mathcal{NP}$-complete time complexity—a major obstacle to their practical application (Mens 2002). The first problem has been addressed, for example, by the technique of *semistructured merge* (Apel et al. 2011) (parts of the artifacts are treated as syntax trees and parts as plain text; see Sect. 5). We strive 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 here 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, line-based approach, as it has more information about the artifacts at its disposal. To cope with the computational 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 occurrence of conflicts in a merge scenario. 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 (Mens 2002).

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 (Mens 2002). 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 434 merge scenarios of 50 software projects, involving over 51 million lines of code. In particular, 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 it 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 92 times faster than purely structured merge, 10 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 discuss properties and trade-offs of several algorithms for ordered and unordered tree matching.
- 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.

This article is an extended version of a prior conference paper presented at the 27th IEEE/ACM International Conference on Automated Software Engineering (Apel et al. 2012). Beside many editorial refinements and more comprehensive discussions of the algorithms that we used (including a new algorithm for unordered tree matching) and the results we obtained, we extended the scope of the empirical study substantially, from 8 to 50 JAVA projects. To increase validity, we selected in the extended study only merge scenarios that occurred in the real world, which was not the case in our prior study, which considered also synthetic merges that seemed "realistic".

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

2 Software merge

In his seminal survey, Mens (2002) provides a comprehensive overview of the field of software-merge techniques. 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 (i.e., the version from which both have been derived) by locating their differences and by 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) (Mens 2002). A major goal is to empower merge tools to detect and resolve conflicts automatically.

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

2.1 Unstructured merge

For illustration, we use the simple example of an implementation of a bag data structure that can store integer values. In Fig. 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 Fig. 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 Fig. 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 Fig. 1 (bottom right).

In practice, most unstructured-merge tools compare and merge versions based on *largest common subsequences* (Bergroth et al. 2000) 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 (2002) 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.

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 (1991) and Buffenbarger (1995) pioneered this field by using structural information, such as the context-free and context-sensitive syntax, during the merge process. Subsequently, researchers proposed a wide variety of structural comparison and merge

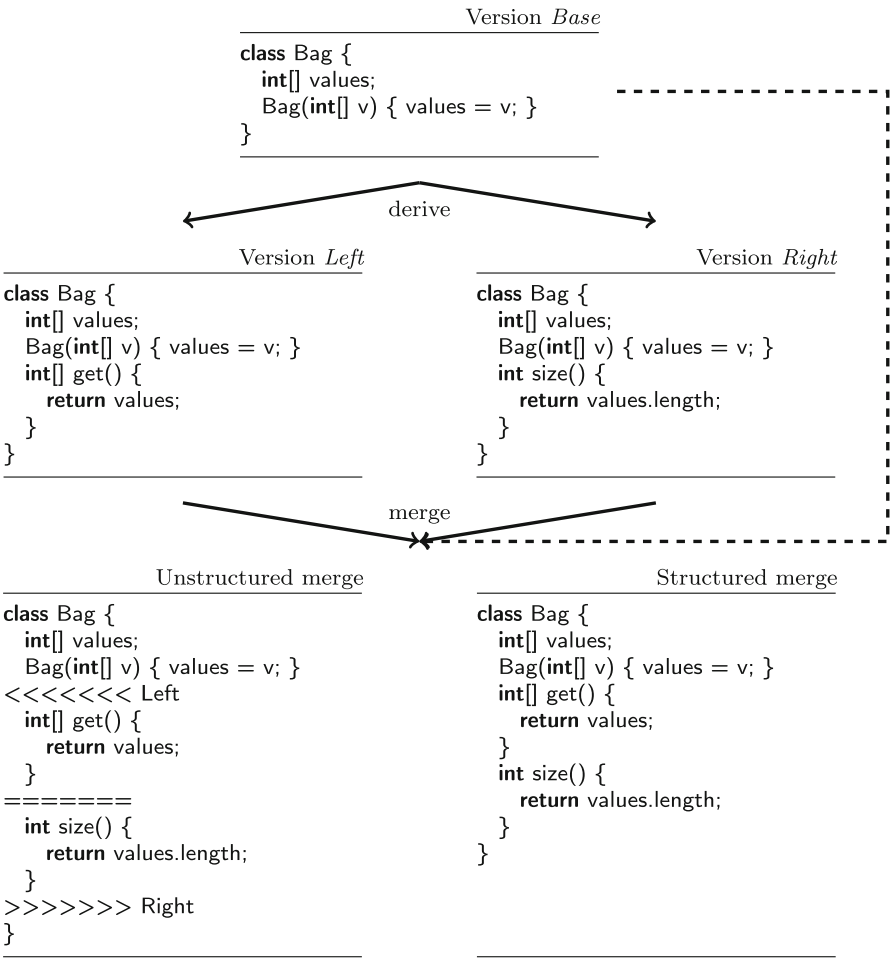

Fig. 1 A conflict resolved with structured merge but not with unstructured merge

tools, including tools for JAVA (Apiwattanapong et al. 2007) and C++ (Grass 1992) (see Sect. 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 proved useful in software merge (Apel et al. 2011). This way, it is almost trivial to merge the two versions of Fig. 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 Fig. 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

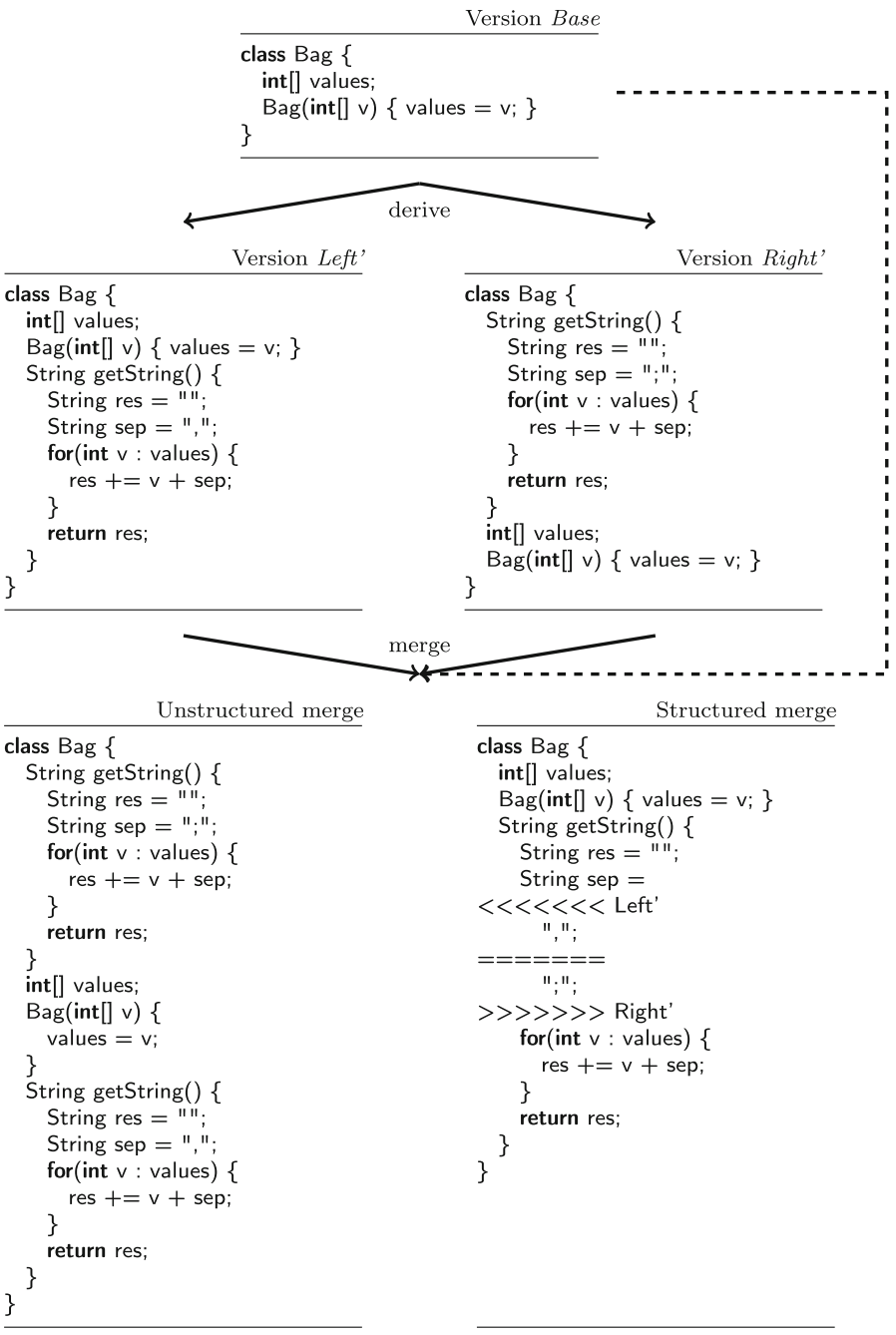

Fig. 2 A conflict detected with structured merge but not with unstructured merge

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 the 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 generation 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 (Apel et al. 2011). Nevertheless, for languages that are widely used, such as JAVA, it is certainly useful to spend the effort and 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 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 auto-tuning, 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 three tailored tree-matching algorithms, two 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 as 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 (Westfechtel 1991). 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 Fig. 1 (inability to resolve a conflict) and Fig. 2 (inability to detect a conflict) arise from the fact that unstructured-merge tools have no informa-

tion 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 merge tool knows whether the corresponding elements can be permuted, and it uses this information during the matching and merge operations. For illustration, we show the tree representing the base revision of our bag example in Fig. 5.

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.

3.2.1 Tree matching

Tree matching takes two trees and computes the *largest common subtree*. In particular, 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. Tree matching based on computing the largest common subtree compares the input trees by level. Algorithms that compare trees across levels are more precise but also more complex, as we discuss in Sect. 6.

The algorithm establishing a match of nodes depends on their syntactic category (e.g., two field declarations are considered equal if their types and names match), and it distinguishes between ordered nodes (which must not be permuted) and unordered nodes (which can be permuted safely).

Ordered nodes For ordered nodes, we use a variation of Yang's algorithm (Yang 1991), which is the tree-equivalent of finding the longest common subsequence of two strings (Hirschberg 1975): 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 that the recursive call invokes TreeMatching, which calls OrderedTreeMatching or UnorderedTreeMatching (Algorithm 2), depending on whether the nodes at the respective level are ordered or unordered. The problem of finding the largest common subtree of ordered trees is *quadratic* in the number of nodes (Yang 1991).

Unordered nodes As shown in Algorithm 2, for unordered nodes, we also 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, an optimization problem also known as the *assignment problem*, which can be solved in *cubic* time (Schrijver 2002).

We implemented and experimented with two solutions for solving the assignment problem: One translates the problem into a linear-programming problem and feeds it into a linear-program solver, and the other is based on the Hungarian method, a polynomial-time, combinatorial optimization algorithm (Kuhn 1955).

Algorithm 1 ORDERED TREE MATCHING

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

The assignment problem can be expressed as linear program as follows[1]:

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

subject to

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

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

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

where $x_{tw} = 1$ means that task t is assigned to worker w (In our case workers and tasks are nodes of the respective abstract syntax trees). 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. For the assignment problem, the constraint matrix is equal to the unoriented incidence matrix of the underlying graph. Therefore, an optimal, integral solution can be computed in polynomial time, in contrast to general exponential time.

Using the linear-programming approach, SOLVEASSIGNMENTPROBLEM in Algorithm 2 creates the constraint matrix and the input matrix, invokes the solver, and returns the maximum number of matches, of which we can compute the actual tree matching.

Our second approach implements the Hungarian method (Kuhn 1955), which provides an optimal solution for the assignment problem in polynomial time. Whereas the original version of the algorithm has a run-time complexity of $\mathcal{O}(n^4)$, we use a modified variant that is proven to run in $\mathcal{O}(n^3)$ (Edmonds and Karp 1972).

We conducted experiments with both implementations and decided to use the LP-based approach as default.

[1] http://www.math.ucla.edu/~tom/LP.pdf

Algorithm 2 UNORDERED TREE MATCHING

```
function UNORDEREDTREEMATCHING(Node A, Node B)
  if A ≠ B then return 0                          ▷ Nodes do not match
  end if
  m ← number of children of A
  n ← number of children of B
  Matrix M ← (m) × (n)                            ▷ Initialize auxiliary matrix
  for i ← 0..m do
    for j ← 0..n do
      M[i, j] ← TREEMATCHING(A_i, B_j)            ▷ Matching of children
    end for
  end for
  sum ← SOLVEASSIGNMENTPROBLEM(M)                 ▷ LP solver or Hungarian method
  return sum + 1                                  ▷ Return maximum number of matches
end function
```

Algorithm 3 TREE AMALGAMATION (MERGE)

```
function MERGE(Node left, Node right)
  merge ← empty tree
  if fixedNumChildren(left, right) ∧ left.hasChanges() ∧ right.hasChanges() then
    merge.add(conflict(left, right))
  end if
  if isOrdered(left, right) then
    merge ← ORDEREDMERGE(left, right)
  else
    merge ← UNORDEREDMERGE(left, right)
  end if
end function
```

3.2.2 Tree amalgamation

Tree amalgamation takes the competing trees (left and right), enriched with matching information, and creates a merged tree as result. The merging process differs for ordered and unordered trees. Based on this distinction, the algorithm fills the merged tree, as specified in Algorithms 3, 4, and 5.

Merging unchanged and consistently changed or added nodes is rather simple, because the left and right versions are not in conflict and the change can be applied safely to the merged tree. The challenge of merging 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 an independent change to the merge tree, we have to check whether it conflicts with a change of the other respective version. We also have to be careful in cases where the language specification demands a fixed amount of child elements.

3.2.3 Conflicts

Next, we explain types of conflicts that can occur while merging. 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 essential: if they over-

Algorithm 4 TREE AMALGAMATION (ORDERED MERGE)

```
function OrderedMerge(Node left, Node right)
  merge ← empty tree
  while ¬(leftdone ∧ rightdone) do
    if left.hasNext() then
      leftChild ← left.next()
    else
      leftdone ← 1
    end if
    if right.hasNext() then
      rightChild ← right.next()
    else
      rightdone ← 1
    end if
    if ¬leftdone ∧ leftChild ∉ right then
      if leftChild ∈ base then
        if leftChild.hasChanges() then
          merge.add(conflict(leftChild, ∅))          ▷ Insertion–deletion conflict
        end if
      else
        if ¬rightdone ∧ rightChild ∉ left then
          if rightChild ∈ base then
            if rightChild.hasChanges() then
              merge.add(conflict(∅, rightChild))     ▷ Insertion–deletion conflict
            else
              merge.add(leftChild)
            end if
          else
            merge.add(conflict(leftChild, rightChild))   ▷ Insertion conflict
          end if
        else
          merge.add(leftChild)
        end if
      end if
    end if
    if ¬rightdone ∧ rightChild ∉ left then
      …                                              ▷ Same for right child
    else if leftChild ∈ right ∧ rightChild ∈ left then
      merge.add(MERGE(leftChild, rightChild))
    end if
  end while
  return merge
end function
```

lap, 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. However, even though the insertion positions are unambiguous, we are still not guaranteed to be safe: Several language elements have a bounded number of arguments. If two insertions result in a violated bound, a conflict has to be reported.

An example is illustrated in Fig. 3. An && operation needs exactly two arguments. To merge both insertions correctly, we have to add a second && operation. As it is not clear whether this is the desired behavior, our tool reports a conflict in such cases.

Algorithm 5 TREE AMALGAMATION (UNORDERED MERGE)

```
function UNORDEREDMERGE(Node left, Node right)
    merge ← empty tree
    while ¬(leftdone ∧ rightdone) do
        if left.hasNext() then
            leftChild ← left.next()
        else
            leftdone ← 1
        end if
        if right.hasNext() then
            rightChild ← right.next()
        else
            rightdone ← 1
        end if
        if ¬leftdone ∧ leftChild ∉ right then
            if leftChild ∈ base then
                if leftChild.hasChanges() then
                    merge.add(conflict(leftChild, ∅))          ▷ Insertion–deletion conflict
                end if
            else
                merge.add(leftChild)
            end if
        end if
        if ¬rightdone ∧ rightChild ∉ left then
            if right ∈ base then
                if rightChild.hasChanges() then
                    merge.add(conflict(∅, rightChild))         ▷ Insertion–deletion conflict
                end if
            else
                merge.add(rightChild)
            end if
        else if leftChild ∈ right ∧ rightChild ∈ left then
            merge.add(MERGE(leftChild, rightChild))
        end if
    end while
    return merge
end function
```

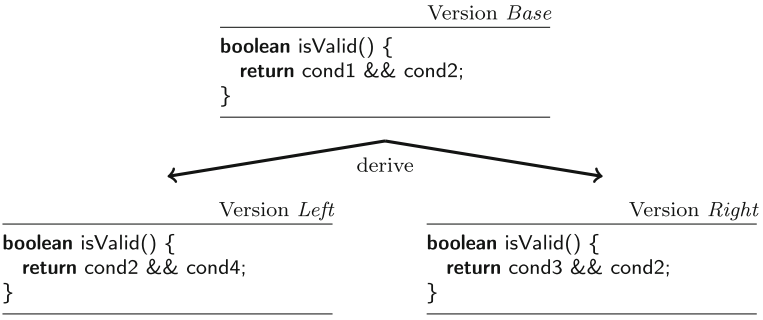

Fig. 3 Two insertions leading to a violation of the language specification

Deletions may also give rise to conflicts when a competing version propagates an insertion at the same position, which results in a *deletion–insertion conflict*. As the simplified example in Fig. 4 illustrates, deletion–insertion conflicts during a structured

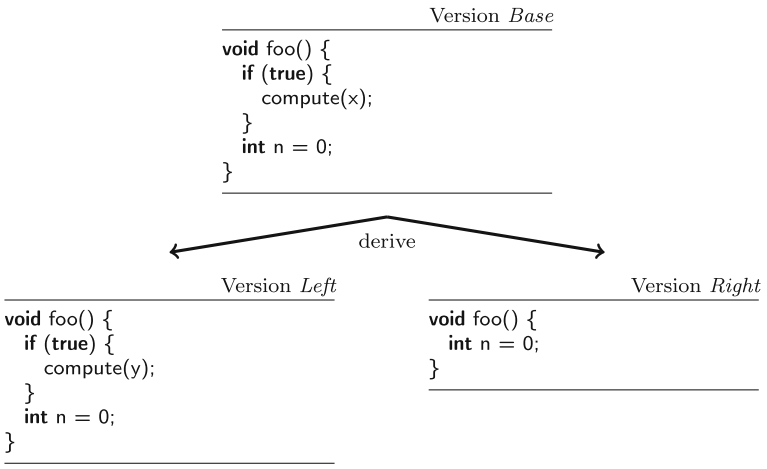

Fig. 4 A conflict between an inner change and a deletion

merge can be difficult to detect. The right revision deletes the conditional expression, whereas the left revision changes an inner node. The merge engine has to detect the conflict between the conditional construct, that includes the actually changed node, and its removal (Fig. 5).

After performing the merge, the pretty printer traverses the abstract syntax tree of the merged program and generates source code. To reduce the overall number of conflicts, we group consecutive conflicts as well as their alternatives while pretty printing, a practice that is also standard in unstructured-merge tools.

3.3 Auto-tuning

The algorithms presented in Sect. 3.2 are computationally complex. In particular, computing the largest common subtree is cubic in the number of (unordered) program elements (Schrijver 2002). This is a limitation of structured merge, compared to the quadratic time complexity of unstructured merge. But we do not want to abandon structured merge entirely. Instead, we strive for a balance between exploiting syntactic information to detect and resolve conflicts and attaining acceptable performance.

The idea of auto-tuning is simple. We use unstructured merge as long as no conflicts have been 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 (Mens 2002). So, for most parts of a program, we can save the effort of 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 Sect. 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.

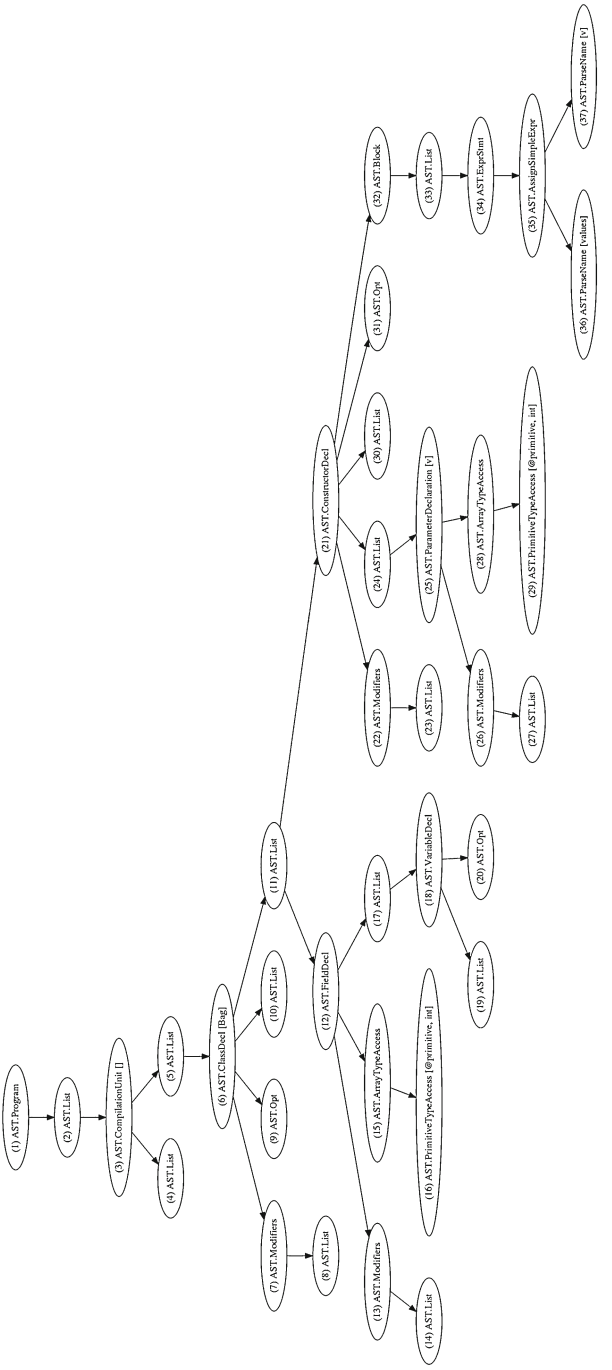

Fig. 5 Version *Base* of the bag example represented as a tree

Algorithm 6 AUTO-TUNING

```
function CombinedMerge(File left, File base, File right)
    result ← UnstructuredMerge(left, base, right)
    if hasConflicts(result) then
        result ← StructuredMerge(left, base, right)
    end if
    PRINT(result)
end function
```

4 Evaluation

To evaluate our approach, especially the balance between precision and performance that we aim to attain with auto-tuning, we implemented a prototype of a structured-merge tool, called JDIME, and we conducted a series of experiments based on 50 real-world, open-source JAVA 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.[2] The implementation is straightforward because JASTADDJ provides excellent extension capabilities. The foundation for our artifact representation is provided by 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 in our linear-programming variant, we used the GLPK solver.[3] 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 one research question:

H$_1$: Unstructured merge reports fewer, but larger conflicts than structured merge.
H$_2$: Unstructured merge is substantially faster than structured merge.
H$_3$: Auto-tuning does not miss many conflicts detected by purely structured merge.
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?

[2] http://jastadd.org/web/jastaddj/

[3] http://www.gnu.org/software/glpk/

4.3 Sample projects

To establish a proper set of real-world JAVA projects, we developed a history-analysis crawler for the popular hosting provider GITHUB. We used the crawler to select 50 popular open-source projects that contain several merge scenarios with conflicts in the histories of their repositories. The number of conflicts was determined by re-running the merge scenarios and parsing the output. The crawler is available at the supplementary Web site.

The projects are of substantial but varying sizes, belong to different domains, and have substantial version histories. Of each project, multiple merge scenarios result in conflicts. 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 contrast to the earlier conference version of this article (Apel et al. 2012), we selected only merge scenarios that were actually performed in the projects' histories.

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 50 projects comes with 5–10 merge scenarios. All 434 merge scenarios together consist of more than 51 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 to learn more about the capabilities of unstructured and structured merge.

Overall, we applied our merge tool to each of the 434 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 3.10.9) and Oracle JAVA HotSpot 64-Bit Server VM 1.7.0_25. To avoid side effects on the run times caused by I/O, we copied each scenario into a RAM-based file system before executing the merges.

4.5 Results

Figure 6, shows the average number of conflicts for each project as reported by unstructured merge, purely structured merge, and structured merge with auto-tuning. Similarly, Fig. 7, reveals the respective numbers of lines of code involved in conflicts. Finally, in Fig. 8, we depict the times consumed by three merge approaches. Tables 2 and 3 contain the experimental data for all merge scenarios. All raw data are available at the supplementary Web site, on a per-file-scenario basis.

Table 1 Overview of the sample projects (all from http://github.com/)

Project	Domain	Merge scenarios	Lines of code
ANDROIDANNOTATIONS	Mobile Development Framework	10	353 K
ANDROID_CAMERA	Mobile Application Platform	10	538 K
ANDROID_SETTINGS	Mobile Application Platform	10	972 K
ANKI- ANDROID	Flash Card Application	10	419 K
ATMOSPHERE	Web Applications Framework	10	718 K
BIGBLUEBUTTON	Web Conferencing System	10	1,528 *K*
CASSANDRA	Database	10	4,548 K
CUCUMBER- JVM	Testing Framework	10	279 K
CUKE4DUKE	Testing Framework	10	47 K
GRAILS- CORE	Web Application Framework	10	2,068 *K*
HECTOR	Database Client	10	684 K
HUDSON	Continuous Integration Server	10	2,464 K
JEDIS	Database Client	10	371 K
K- 9	Mail Client	10	1,938 K
KUNDERA	Database Library	10	4,071 K
LOMBOK	Language Extension	10	763 K
MONGO- JAVA- DRIVER	Database Library	10	465 K
MUSTACHE.JAVA	Template Engine	10	122 K
ORIENTDB	Database	10	4,550 K
PRIAM	Database Extension	10	290 K
REXSTER	Graph Server	10	532 K
ROBOLECTRIC	Testing Framework	10	1,619 K
TITAN	Graph Database	10	751 K
TWITTER4J	Twitter Library	10	852 K
USERGRID- STACK	Mobile Applications Platform	10	2,027 K
WILDFLY	Application Server	10	354 K
ZOIE	Search Engine	10	438 K
ANDENGINE	Game Engine	9	514 K
ANDROIDQUERY	Library	9	237 K
JUNIT	Testing Framework	9	494 K
SPRING- ROO	Development Framework	9	1,442 K
ACTIVITI	Business Process Management	8	3,127 K
DROPWIZARD	Web Application Framework	8	197 K
GRAYLOG2- SERVER	Log Management	8	118 K
JBPM	Business Process Management	8	2,258 K
NETTY	Network Application Framework	8	1,306 K
ROBOGUICE	Mobile Development Framework	8	68 K
SOLANDRA	Search Engine	8	227 K
GRADLE	Build System	7	3,146 K
OPENREFINE	Data Management	7	884 K
STORM	Distributed Computation System	7	727 K

Table 1 continued

Project	Domain	Merge scenarios	Lines of code
ANDROID_MMS	Mobile Application Platform	6	358 K
ERJANG	Virtual Machine	6	601 K
H2O	Hadoop Analytics Engine	6	1,010 K
REACTOR	Asynchronous Application Framework	6	159 K
RESTPROVIDER	Caching Proxy	6	80 K
SPOUTCRAFTLAUNCHER	Game Client	6	99 K
COUCHDB- LUCENE	Database Client	5	30 K
GREENHOUSE	Web Application	5	44 K
METRICS	Library	5	121 K

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 109 times faster, 23 times on average. But structured merge with auto-tuning is up to 92 times faster than purely structured merge, 10 times on average.

4.6 Discussion

4.6.1 Hypotheses and research questions

Based on the results, we have to put hypothesis $\mathbf{H}_1$ into perspective: Structured merge tends to produce conflicts of a finer granularity than unstructured merge, but it is also able to avoid certain types of conflicts completely. In the merge scenarios under study, structured merge reports 60 % of the number of conflicts of unstructured merge (average over all merge scenarios), which contradicts our hypothesis. 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. But, as we expected, the size of the conflicts reported by structured merge is smaller. It produced only 21 % of the number of conflicting lines of unstructured merge. Some outliers can be observed in the results of our experiments, where structured merge produced larger conflicts than the unstructured approach. This was the case in the project KUNDERA, where a conflict inside a class declaration was produced, which resulted in two competing classes of about 800 lines when using structured merge. We observed similar cases only in projects ZOIE and K- 9.

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-

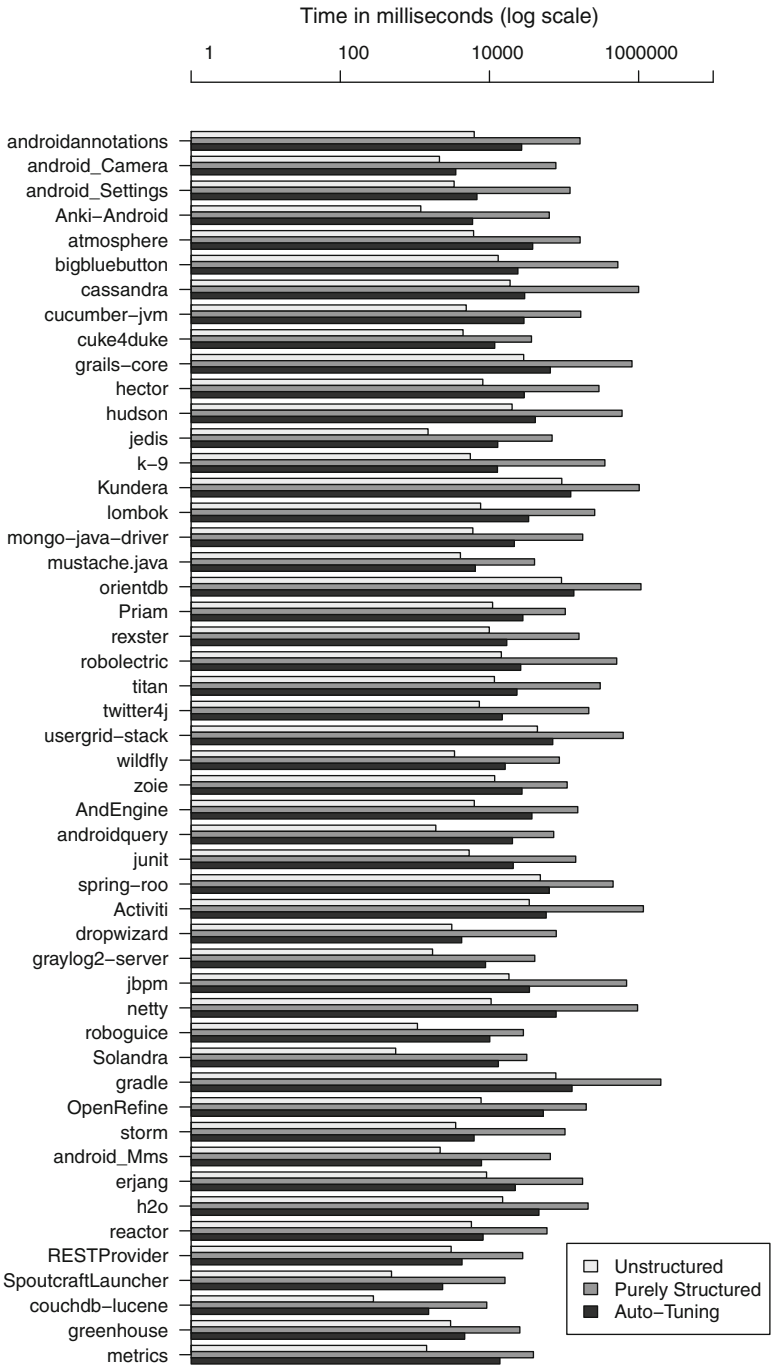

Fig. 6 Number of reported conflicts

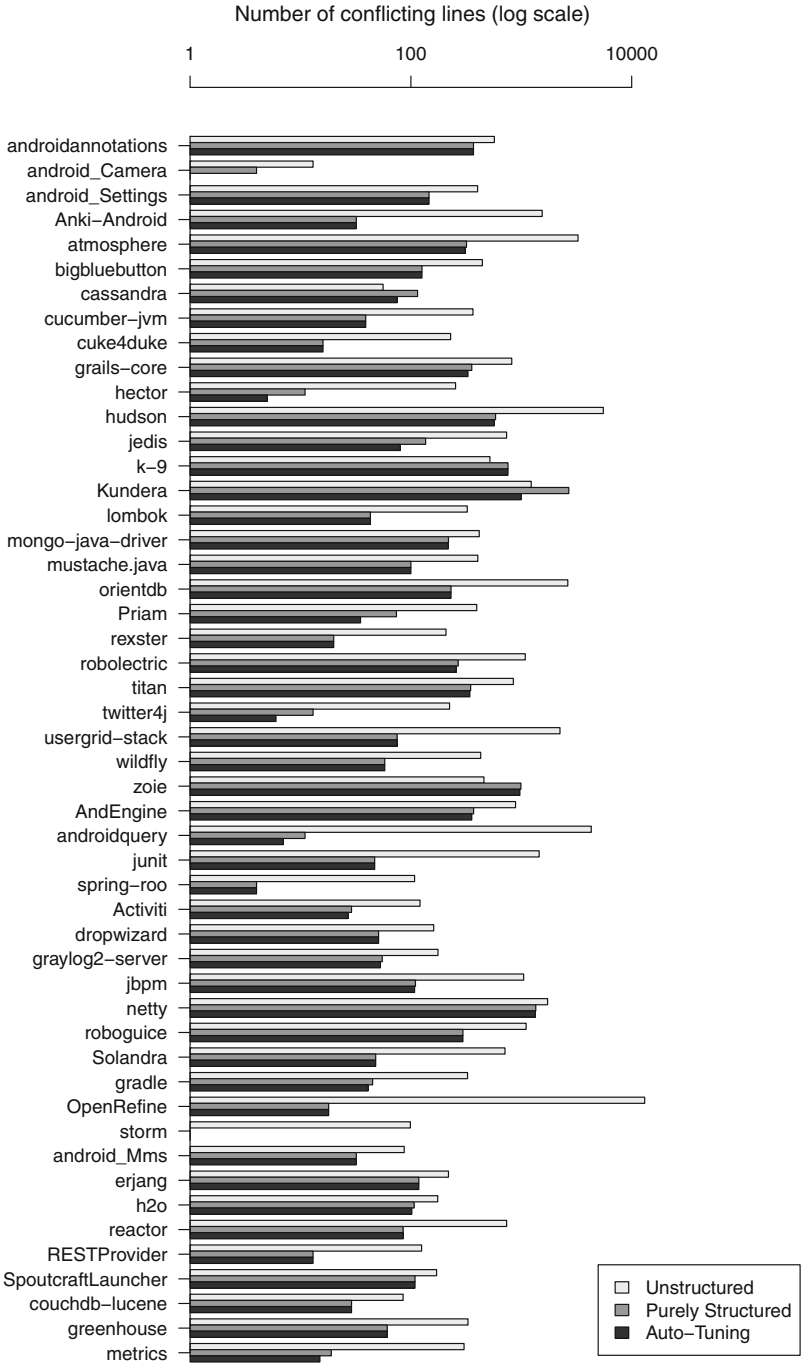

Fig. 7 Number of conflicting lines of code

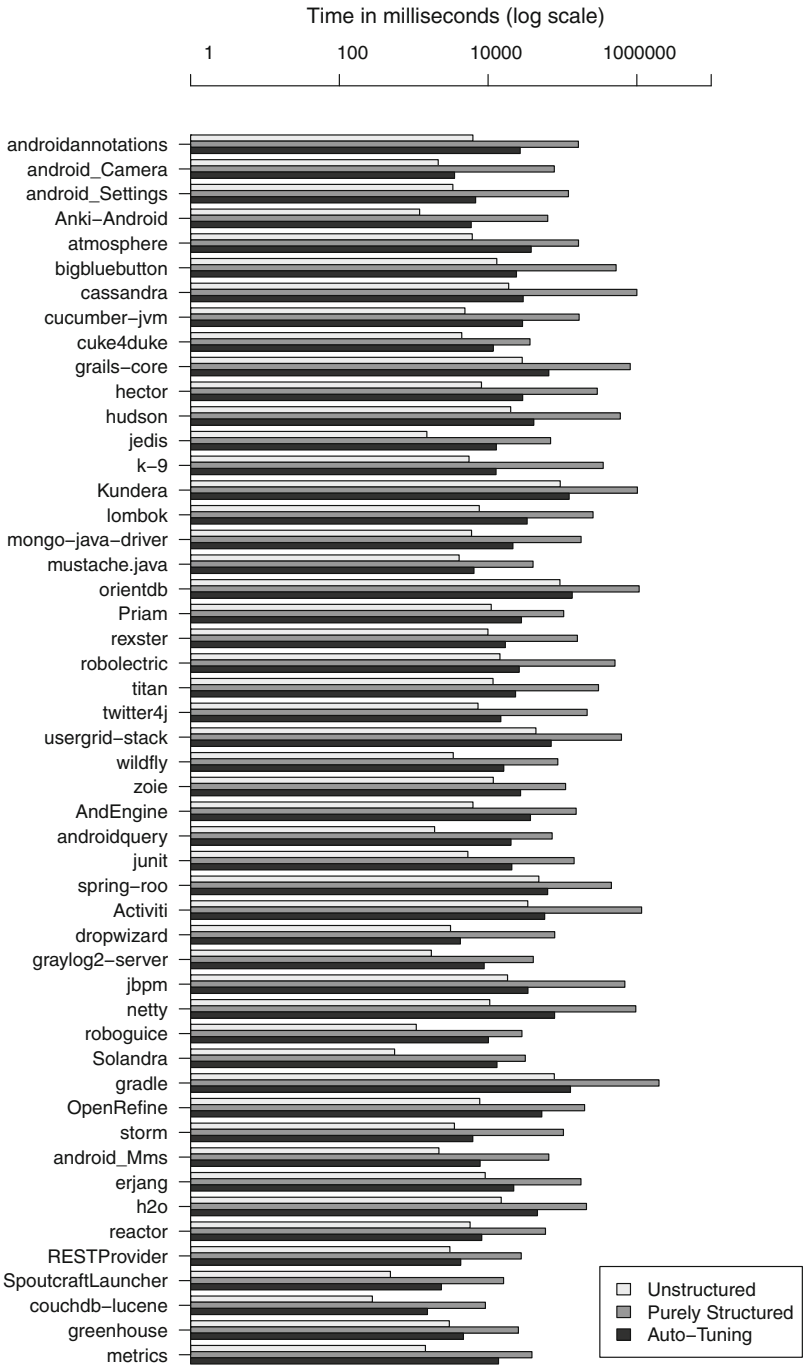

Fig. 8 Merging time in seconds

Table 2 Experimental data

Project	#LOC	#Files	#Conflicts			#Conflicting lines		
			UM	SM	AT	UM	SM	AT
ANDROIDANNOTATIONS	3,52,832	3,125	31	64	64	569	368	368
ANDROID_CAMERA	5,37,849	1,064	3	2	0	12	3	0
ANDROID_SETTINGS	9,72,019	1,678	33	19	19	401	145	145
ANKI- ANDROID	4,19,462	591	62	6	6	1,545	31	31
ATMOSPHERE	7,18,447	3,062	110	41	40	3,263	318	310
BIGBLUEBUTTON	15,27,514	6,548	55	38	38	443	125	125
CASSANDRA	45,48,435	9,314	7	18	5	55	114	74
CUCUMBER- JVM	2,79,383	2,449	19	8	8	364	38	38
CUKE4DUKE	46,957	629	20	7	7	228	15	15
GRAILS- CORE	20,67,689	7,739	26	26	17	819	355	328
HECTOR	6,84,495	4,048	11	2	1	253	10	4
HUDSON	24,64,359	10,056	55	42	34	5,530	584	567
JEDIS	3,71,353	755	69	39	25	735	135	79
K- 9	19,38,410	2,797	41	57	57	519	757	757
KUNDERA	40,71,164	12,983	53	24	22	1,229	2,692	996
LOMBOK	7,62,663	3,933	17	4	4	323	42	42
MONGO- JAVA- DRIVER	4,65,295	1,615	27	9	8	415	218	217
MUSTACHE.JAVA	1,22,071	559	17	7	7	403	99	99
ORIENTDB	45,50,346	13,423	91	39	39	2,637	230	230
PRIAM	2,89,628	1,517	30	12	10	394	73	34
REXSTER	5,31,674	1,916	28	9	9	207	19	19
ROBOLECTRIC	16,19,060	7,187	18	10	5	1,085	267	256
TITAN	7,51,024	3,761	69	41	40	846	347	341
TWITTER4J	8,51,546	3,655	17	4	3	223	12	5
USERGRID- STACK	20,27,302	7,336	95	17	17	2,240	74	74
WILDFLY	3,53,952	1,700	24	12	12	428	57	57
ZOIE	4,37,512	1,611	44	8	7	458	991	970
ANDENGINE	5,14,496	3,132	37	93	86	886	369	355
ANDROIDQUERY	2,36,579	495	15	6	3	4,300	10	6
JUNIT	4,93,628	2,675	28	8	8	1,450	46	46
SPRING- ROO	14,42,323	6,623	20	2	2	107	3	3
ACTIVITI	31,26,627	17,220	13	15	14	120	28	26
DROPWIZARD	1,97,140	1,563	14	4	4	160	50	50
GRAYLOG2- SERVER	1,18,038	863	23	13	12	175	54	52
JBPM	22,58,245	9,286	37	19	18	1,049	109	107
NETTY	13,06,188	5,264	116	105	96	1,730	1,352	1,334
ROBOGUICE	68,211	589	66	70	70	1,105	295	295
SOLANDRA	2,27,265	271	12	6	6	711	47	47
GRADLE	31,46,288	23,253	38	19	17	325	44	40
OPENREFINE	8,83,902	3,489	64	9	9	13,110	17	17

Table 2 continued

Project	#LOC	#Files	#Conflicts			#Conflicting lines		
			UM	SM	AT	UM	SM	AT
STORM	7,26,582	1,756	7	0	0	98	0	0
ANDROID_MMS	3,57,782	1,088	9	7	7	86	31	31
ERJANG	6,00,770	1,270	12	14	14	218	117	117
H2O	10,10,483	2,046	32	28	26	174	106	101
REACTOR	1,58,777	790	30	17	17	736	84	84
RESTPROVIDER	80,210	418	12	6	6	124	12	12
SPOUTCRAFTLAUNCHER	98,587	245	12	3	3	170	108	108
COUCHDB- LUCENE	29,683	141	12	4	4	84	28	28
GREENHOUSE	44,230	419	21	8	8	328	60	60
METRICS	1,20,844	719	15	7	6	302	18	14

LOC lines of code, *UM* unstructured merge, *SM* purely structured merge, *AT* structured merge with auto-tuning

tuning approach. That is, the strategy of auto-tuning 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.

Furthermore, our experiments confirm hypothesis $\mathbf{H}_2$: In all projects, structured merge is substantially slower than unstructured merge. Unstructured merge is up to 109 times faster than structured merge, 23 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 92 times and 10 times on average. In projects ANDROID_CAMERA, MUSTACHE.JAVA, DROPWIZARD, and REACTOR, it is even on the order of using unstructured merge. So, auto-tuning seems to be, at least, promising to hit a sweet spot between precision and performance.

Finally, as for research question $\mathbf{R}_1$, we found that 5 % of the changed files cannot be merged with unstructured merge—which is even lower than the value predicted by Mens. With structured merge, this fraction can be reduced to 2 %. This surprised us, as we experienced a way higher number for both approaches in our preliminary study (Apel et al. 2012). The reason for this difference might be the fact that our previous case studies included also merge scenarios that seemed feasible but were not part of the projects histories. All scenarios of the study presented here have been actually performed in the history of the projects.

4.6.2 Run-time complexity

In Fig. 9, we contrast the size of the merge scenarios 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

Table 3 Experimental data

Project	#LOC	#Files	Merge time in milliseconds		
			UM	SM	AT
ANDROIDANNOTATIONS	3,52,832	3,125	6,247	1,63,858	27,029
ANDROID_CAMERA	5,37,849	1,064	2,135	77,719	3,538
ANDROID_SETTINGS	9,72,019	1,678	3,360	1,20,350	6,799
ANKI- ANDROID	4,19,462	591	1,200	63,373	5,935
ATMOSPHERE	7,18,447	3,062	6,126	1,64,365	37,997
BIGBLUEBUTTON	15,27,514	6,548	13,094	5,26,466	24,053
CASSANDRA	45,48,435	9,314	18,905	10,01,244	29,658
CUCUMBER- JVM	2,79,383	2,449	4,882	1,67,547	29,026
CUKE4DUKE	46,957	629	4,425	36,596	11,772
GRAILS- CORE	20,67,689	7,739	28,780	8,15,369	65,528
HECTOR	6,84,495	4,048	8,145	2,93,922	29,174
HUDSON	24,64,359	10,056	20,125	5,99,881	41,192
JEDIS	3,71,353	755	1,502	69,213	12,915
K- 9	19,38,410	2,797	5,540	3,53,205	12,798
KUNDERA	40,71,164	12,983	93,320	10,17,891	1,23,075
LOMBOK	7,62,663	3,933	7,613	2,57,805	33,521
MONGO- JAVA- DRIVER	4,65,295	1,615	5,992	1,77,948	21,555
MUSTACHE.JAVA	1,22,071	559	4,080	40,227	6,457
ORIENTDB	45,50,346	13,423	92,456	10,74,726	1,35,397
PRIAM	2,89,628	1,517	11,019	1,03,872	28,073
REXSTER	5,31,674	1,916	9,949	1,58,759	17,069
ROBOLECTRIC	16,19,060	7,187	14,435	5,08,702	26,259
TITAN	7,51,024	3,761	11,656	3,05,048	23,402
TWITTER4J	8,51,546	3,655	7,322	2,14,671	14,832
USERGRID- STACK	20,27,302	7,336	43,962	6,21,037	70,578
WILDFLY	3,53,952	1,700	3,400	86,558	16,252
ZOIE	4,37,512	1,611	1,1749	1,10,176	27,320
ANDENGINE	5,14,496	3,132	6,261	1,52,800	37,097
ANDROIDQUERY	2,36,579	495	1,909	72,701	20,333
JUNIT	4,93,628	2,675	5,345	1,43,408	20,842
SPRING- ROO	14,42,323	6,623	48,127	4,54,850	63,215
ACTIVITI	31,26,627	17,220	34,167	11,58,996	57,660
DROPWIZARD	19,7140	1,563	3,126	78,658	4,242
GRAYLOG2- SERVER	1,18,038	863	1,726	40,517	8,860
JBPM	22,58,245	9,286	18,266	6,89,762	34,354
NETTY	13,06,188	5,264	10,522	9,68,419	78,332
ROBOGUICE	68,211	589	1,082	28,515	10,096
SOLANDRA	2,27,265	271	553	31,661	13,166
GRADLE	31,46,288	23,253	77,680	19,85,987	1,28,082

Table 3 continued

Project	#LOC	#Files	Merge time in milliseconds		
			UM	SM	AT
OpenRefine	8,83,902	3,489	7,720	1,98,420	52,727
storm	7,26,582	1,756	3,526	1,03,018	6,226
android_Mms	3,57,782	1,088	2,181	65,508	7,815
erjang	6,00,770	1,270	9,144	1,77,236	22,121
h2o	10,10,483	2,046	15,016	2,10,292	46,082
reactor	1,58,777	790	5,728	59,088	8,219
RESTProvider	80,210	418	3,063	27,923	4,290
SpoutcraftLauncher	98,587	245	486	16,153	2,355
couchdb- lucene	29,683	141	278	9,205	1,529
greenhouse	44,230	419	3,012	25,557	4,658
metrics	1,20,844	719	1,437	38,868	13,813

LOC lines of code, *UM* unstructured merge, *SM* purely structured merge, *AT* structured merge with auto-tuning

merge. Apart from two groups of outliers, the merge time grows smoothly with the file size. The outliers can be explained with the cubic run-time complexity of the matching algorithms involved. The most extreme run-time outliers we observed are due to subsequent definitions of large, hard-coded arrays with more than thousand elements that our matching algorithm had to process.

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 Fig. 10. Notice that the outliers of Fig. 9 are farther 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). The most outlying scenario with over 170 s is mainly caused by three files in the project netty. Here, array declarations with over 2000 hard-coded values each resulted in a processing time of more than 30 s for each of these files.

4.6.3 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*

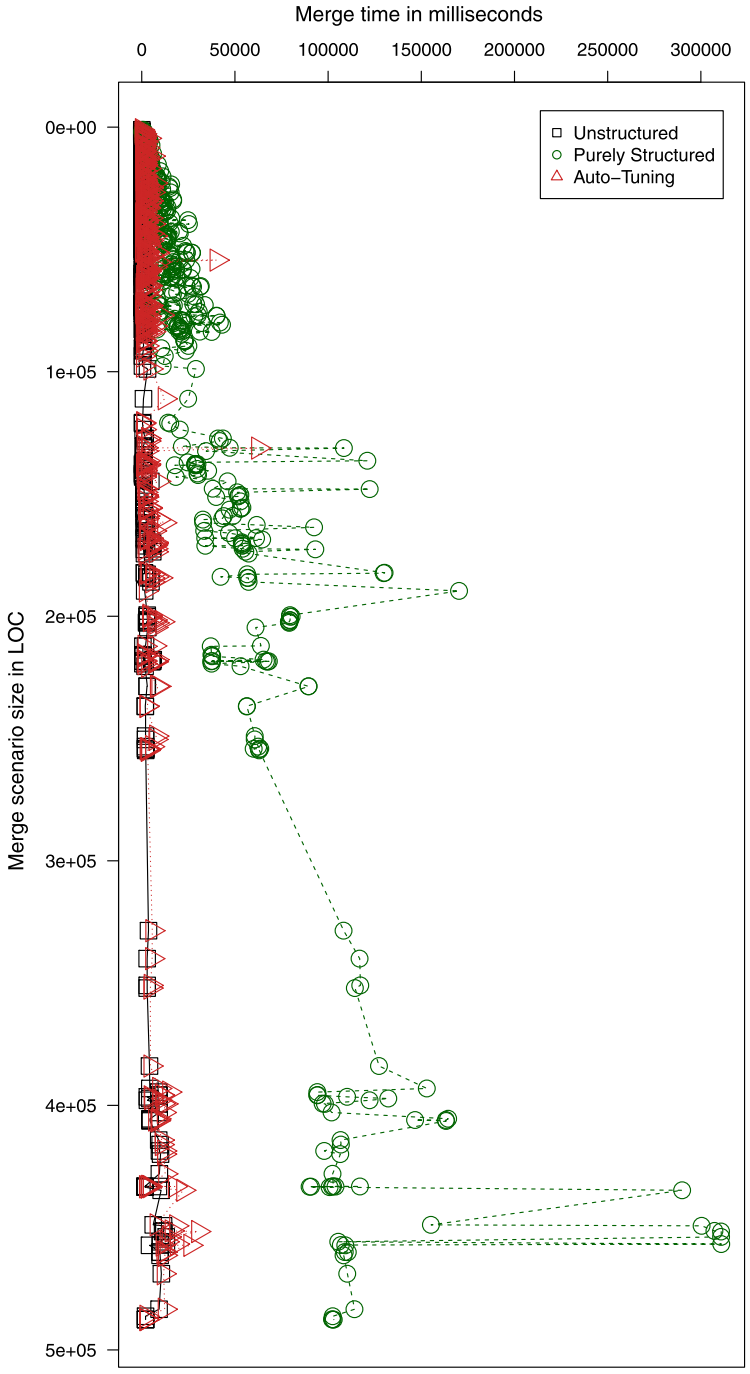

Fig. 9 File size (in number of lines of code) versus merge time (in milliseconds)

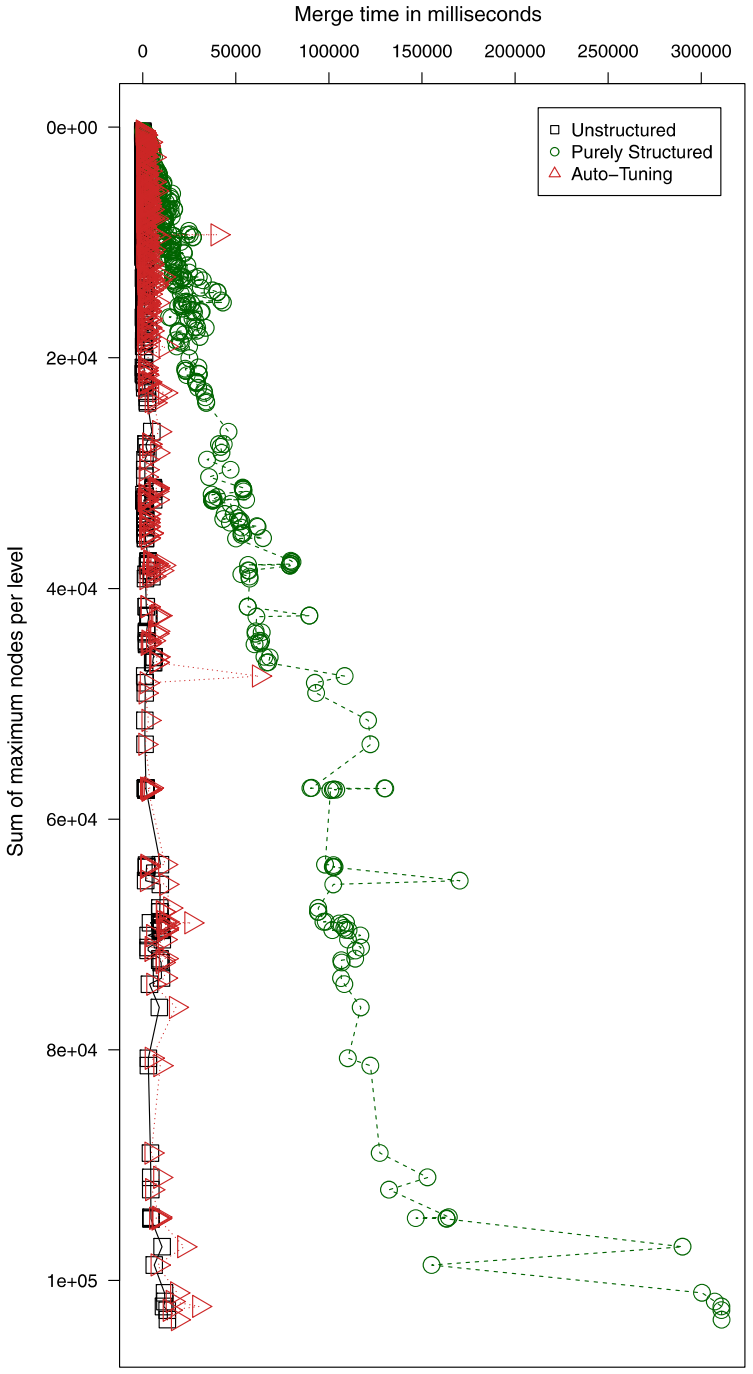

Fig. 10 Sum of maximum nodes per syntax-tree level versus merge time (in milliseconds)

subtrees, but they are generally $\mathcal{APX}$-hard (Zhang and Jiang 1994).[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 Fig. 1, or in terms of uncovering hidden conflicts, as in Fig. 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. However, there are exceptions: Conflicts at nodes in the upper levels of the abstract syntax tree, such as class or method declarations, result in large conflicts. This is also an effect of our implemented matching algorithms and could be avoided by using the cross-level matching approaches already mentioned. In our experiments, we observed such large conflicts several times when generics where involved (and changed by both revisions) in the declaration of a class. An example of this can be found in the project KUNDERA.

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 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 studied. Our findings should even apply to languages that are similar to JAVA (e.g., C#).

We had to exclude 0.13 % of files while conducting our experiment. This was due to technical problems of the underlying framework of our tool (JASTADDJ), which does not handle some language elements (e.g., annotations) very well. However, we do not expect that excluding these files affects the big picture of our study.

Another concern is that we do not know whether the merged code is semantically correct. This is due to the nature of our approach, which guarantees syntactic correctness in purely structured mode, but does not consider semantics. However, this is still an improvement compared to the line-based state-of-the-art merging tools, which do not incorporate syntax. The usage of approaches and tools providing behavioral

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

guarantees is simply not realistic for substantial software projects, such as the ones selected in our study.

5 Related work

5.1 Structured merge

After the seminal work of Westfechtel (1991) and Buffenbarger (1995), 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 (Apiwattanapong et al. 2007) and C++ (Grass 1992). On the other hand, there are many proposals of structured two-way and three-way merge techniques for modeling artifacts (Mehra et al. 2005; Kolovos et al. 2006; Treude et al. 2007)—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, a renaming analysis could be integrated to further improve the precision of tree matching (Hunt and Tichy 2002).

5.2 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 (Apel et al. 2011). 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 worse than structured merge (on average, 5 times 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.

[5] http://pi.informatik.uni-siegen.de/CVSM/

[6] http://fosd.net/SSMerge

5.3 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 (Lippe and Oosterom 1992; Dig et al. 2007; Koegel et al. 2009; Taentzer et al. 2010). 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 (Berzins 1994; Jackson and Ladd 1994), 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 (Maoz et al. 2011).

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 50 real-world projects, including 434 merge scenarios with over 51 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 92 times faster, 10 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 (Kolovos et al. 2006; Treude et al. 2007)—represent software artifacts as graphs, for example, incorporating the context-sensitive syntax (Westfechtel 1991). 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 situations where structured merges produce very large conflicts, as we experienced with KUNDERA.

Acknowledgments We thank Christian Kästner for feedback on the potential of this work, and for the design we used in the Figs. 1 and 2. This work has been supported by the DFG Grants: AP 206/2, AP 206/4, and AP 206/5.

References

Apel, S., Liebig, J., Brandl, B., Lengauer, C., Kästner, C.: Semistructured merge: rethinking merge in revision control systems, pp. 190–200. In: Proceedings of the ESEC/FSE, ACM, (2011)

Apel, S., Leßenich, O., Lengauer, C.: Structured merge with auto-tuning: balancing precision and performance, pp. 120–129. In: Proceedings of the ASE, ACM, (2012)

Apiwattanapong, T., Orso, A., Harrold, M.: JDiff: a differencing technique and tool for object-oriented programs. Autom. Softw. Eng. **14**(1), 3–36 (2007)

Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms, pp. 39–48. In: Proceedings of the SPIRE, IEEE, (2000)

Berzins, V.: Software merge: semantics of combining changes to programs. ACM TOPLAS **16**(6), 1875–1903 (1994)

Buffenbarger, J.: Syntactic software merging. In: Selected Papers from SCM-4 and SCM-5, Springer, LNCS 1005, pp 153–172, (1995)

Dig, D., Manzoor, K., Johnson, R., Nguyen, T.: Refactoring-aware configuration management for object-oriented programs, pp. 427–436. In: Proceedings of the ICSE, IEEE, (2007)

Edmonds, J., Karp, R.M.: Theoretical improvements in algorithmic efficiency for network flow problems. J. ACM **19**(2), 248–264 (1972)

Grass, J.: Cdiff: a syntax directed differencer for C++ programs, pp. 181–193. In: Proceedings of USENIX C++ Conference, USENIX Association, (1992)

Hirschberg, D.: A linear space algorithm for computing maximal common subsequences. Commun. ACM **18**(6), 341–343 (1975)

Hunt, J., Tichy, W.: Extensible language-aware merging, pp. 511–520. In: Proceedings of ICSM, IEEE, (2002)

Jackson, D., Ladd, D.: Semantic Diff: A tool for summarizing the effects of modifications, pp. 243–252. In: Proceedings of the ICSM, IEEE, (1994)

Koegel, M., Helming, J., Seyboth, S.: Operation-based conflict detection and resolution, pp. 43–48. In: Proceedings of the CVSM, IEEE, (2009)

Kolovos, D., Paige, R., Polack, F.: Merging models with the epsilon merging language (EML), vol. 4199, pp. 215–229. In: Proceedings of the MODELS, Springer, LNCS, (2006)

Kuhn, H.: The hungarian method for the assignment problem. Naval Res. Logist. Q. **2**(1–2), 83–97 (1955)

Lippe, E., van Oosterom, N.: Operation-based merging, pp. 78–87. In: Proceedings of the SDE, ACM, (1992)

Maoz, S., Ringert, J., Rumpe, B.: CDDiff: semantic differencing for class diagrams, vol. 6813, pp. 230–254. In: Proceedings of the ECOOP, Springer, LNCS, (2011)

Mehra, A., Grundy, J., Hosking, J.: A generic approach to supporting diagram differencing and merging for collaborative design, pp. 204–213. In: Proceedings of the ASE, ACM, (2005)

Mens, T.: A state-of-the-art survey on software merging. IEEE TSE **28**(5), 449–462 (2002)

Schrijver, A.: Combinatorial Optimization: Polyhedra and Efficiency. Springer, Heidelberg (2002)

Taentzer, G., Ermel, C., Langer, P., Wimmer, M.: Conflict detection for model versioning based on graph modifications, vol. 6372, pp. 171–186. In: Proceedings of the ICGT, Springer, LNCS, (2010)

Treude, C., Berlik, S., Wenzel, S., Kelter, U.: Difference computation of large models, pp. 295–304. In: Proceedings of the ESEC/FSE, ACM, (2007)

Westfechtel, B.: Structure-oriented merging of revisions of software documents, pp. 68–79. In: Proceedings of the SCM, ACM, (1991)

Yang, W.: Identifying syntactic differences between two programs. Softw.: Pract. Exp. **21**(7), 739–755 (1991)

Zhang, K., Jiang, T.: Some MAX SNP-hard results concerning unordered labeled trees. Inf. Process. Lett. **49**(5), 249–254 (1994)