Precise Version Control of Trees with Line-Based Version Control Systems

Dimitar Asenov[1], Balz Guenat[1], Peter Müller[1](✉), and Martin Otth[2]

[1] Department of Computer Science, ETH Zurich, Zurich, Switzerland
{dimitar.asenov,peter.mueller}@inf.ethz.ch, guenatb@student.ethz.ch
[2] Ergon Informatik AG, Zurich, Switzerland
martin.otth@ergon.ch

Abstract. Version control of tree structures, ubiquitous in software engineering, is typically performed on a textual encoding of the trees, rather than the trees directly. Applying standard line-based diff and merge algorithms to such encodings leads to inaccurate diffs, unnecessary conflicts, and incorrect merges. To address these problems, we propose novel algorithms for computing precise diffs between two versions of a tree and for three-way merging of trees. Unlike most other approaches for version control of structured data, our approach integrates with mainstream version control systems. Our merge algorithm can be customized for specific application domains to further improve merge results. An evaluation of our approach on abstract syntax trees from popular Java projects shows substantially improved merge results compared to Git.

Keywords: Version control · Trees · Structured editor · Software evolution

1 Introduction

Tree structures such as XML, JSON, and source code are ubiquitous in software engineering, but support for precise version control of trees is lacking. Mainstream version control systems (VCSs) such as Git, Mercurial, and SVN treat all data as sequences of lines of text. Standard diff and merge algorithms disregard the structure of the data they manipulate, which has three major drawbacks for versioning trees. First, standard line-based diff algorithms may lead to *inaccurate and confusing diffs*, for instance when differences in formatting (e.g., added indentation) blend with real changes or when lines are incorrectly matched across different sub-trees (e.g., across method boundaries in a program). Inaccurate diffs do not only waste developers' time, but may also corrupt the result of subsequent merge operations. Second, standard merge algorithms may lead to *unnecessary conflicts*, which occur for incompatible changes to the formatting (e.g., breaking a line at different places), but also for more substantial changes such as merging two revisions that each add an element to an un-ordered list (for instance, a method at the end of the same class). Unnecessary conflicts could be merged automatically, but instead require manual intervention from the developer. Third, standard merge algorithms may lead to *incorrect merges*; for instance, if two developers move the same tree node to two different places, a line-based merge might incorrectly duplicate the node. Incorrect merges lead to errors that developers need to detect and fix manually.

To solve these problems, we propose a novel approach to versioning trees. Our approach builds on a standard line-based VCS (in our case, Git), but provides diff and merge algorithms that utilize the tree structure to provide accurate diffs, conflict detection, and merging. In contrast to VCSs that require a dedicated backend for trees [14,15,21,24,25], employing a standard VCS allows developers to use established infrastructures and workflows (such as GitHub or BitBucket) and to version trees and text files such as documentation in the same VCS. Our diff algorithm relies on the optimized line-based diff of the underlying VCS, but utilizes the tree structure to accurately report changes. Building on the diff algorithm, we designed a three-way merge algorithm that reduces unnecessary conflicts and incorrect merges by using the tree structure and, optionally, domain knowledge such as whether the order of elements in a list is relevant.

Diff and merge algorithms rely on matching different revisions of a tree to recognize commonalities and changes. One option to obtain such a matching is to associate each tree node with a unique ID that remains unchanged across revisions. This approach yields precise matchings and makes it easy to recognize changed and moved nodes, but requires a custom storage format and support from an editor such as MPS [30] or Envision [5]. Alternatively, one can use traditional textual encodings of trees without IDs (e.g., source code to represent an AST) and compute matchings using an algorithm such as ChangeDistiller [10] or GumTree [9]. However, such algorithms require significant time and produce results that are approximate and, thus, lead to less precise diffs and merges. Our approach supports both options; it benefits from the precise matchings provided by node IDs when available, but can also utilize the results of matching algorithms and, thus, be used with standard editors. We will present the approach for a storage format that includes node IDs, but our evaluation shows that even with approximate matchings computed on standard Java code, our approach achieves substantially better results than a standard line-based merge.

The contributions of this paper are:

- A textual encoding of generic trees that enables their precise version control within standard line-based VCSs such as Git.
- A novel algorithm for computing the difference between two versions of a tree based on the diff reported by a line-based VCS.
- A novel algorithm for a three-way merge of trees, which allows the customization of conflict detection and resolution.
- An implementation of the algorithms in the Envision IDE and an evaluation on several popular open-source Java code bases.

The rest of this paper is structured as follows. In Sect. 2, we present a textual encoding of trees and a corresponding diff algorithm, which builds on a line-based diff. We describe a generic algorithm for merging trees and two customizations that improve the merge result in Sect. 3. In Sect. 4, we discuss the results of our evaluation. We discuss related work in Sect. 5 and conclude in Sect. 6. More details of the algorithms can be found in the PhD thesis of the first author [6].

2 Tree Versioning with a Line-Based VCS

The algorithms we designed work on a general tree structure. In order to enable precise version control of trees, we assume, without loss of generality, that each tree *node* is a tuple with the following elements:

– *id*: a globally unique ID. This ID is used to match and compare nodes from different versions and track node movement. IDs can be randomly generated, as long as matching nodes from different versions have the same ID, which can be achieved by using a tree-matching algorithm such as GumTree [9]. We use a standard 128-bit universally unique identifier (UUID).
– *parentId*: the ID of the parent node. The parent ID of the root node is a null UUID. All other nodes must have a parentId that matches the ID of an existing node.
– *label*: a name that is unique among sibling nodes. The label is essentially the name of the edge from parent to child node. This could be any string or number, e.g., $1, 2, 3, ...$ for the children of nodes representing lists.
– *type*: an arbitrary type name from the target domain. For example, types of AST nodes could be `Method` or `IntegerLiteral`. Types enable additional customization of the version control algorithms, used to improve conflict detection and resolution. In domains without different node types, one type can be used for all nodes.
– *value*: an optional value.

A *valid tree* is a set of nodes which form a tree and meet the requirements above.

2.1 Textual Encoding of Valid Trees

In order to efficiently perform version control of trees within a line-based VCS, we encode trees in a specific text format, which enables using the existing line-based diff in the first of two stages for computing the changes between two tree versions. A valid tree is encoded into text files as illustrated in Fig. 1. The key property of the encoding is that a single line contains the encoding of exactly one tree node with all its elements. In Fig. 1, each line encodes a node's label, type, UUID, the UUID of the parent node, and the optional value in that order. A reserved node type *External* indicates that a subtree is stored in a different file (Fig. 1b).

This encoding allows two versions of a set of files to be efficiently compared using a standard line-based diff. The different lines reported by such a diff correspond directly to a set of nodes that is guaranteed to be an overapproximation of the nodes that have changed between the two versions of the encoded tree.

```
2 Method {9c2c..} {e0b6..}
  modifiers Modifier {8842..} {9c2c..} 1
  name Name {3269..} {9c2c..} foo
  body StatementList {1023..} {9c2c..}
    0 If {f3c2..} {1023..}
      condition BinOp {b0a0..} {f3c2..}
        left Text {f7c3..} {b0a0..} two\nlines
```

(a)

```
12 Class {5414..} {425d..}
  methods List {e0b6..} {5414..}
    0 External {e239..} {e0b6..}
    1 External {5db1..} {e0b6..}
    2 External {9c2c..} {e0b6..}
```

(b)

Fig. 1. (a) An example encoding of an AST fragment. For brevity, only the first 2 bytes of UUIDs are shown here. (b) A file that references external files, which contain the subtrees of a class's methods. The last line refers to the file from (a). At most two lines in different files may have the same ID, and one of them must be of type *External*.

For efficient parsing, we indent each child node and insert children after their parents (Fig. 1), enabling simple stack-based parsing. The names of the files that comprise a single tree is irrelevant, but for quickly finding subtrees, it is advisable to include the UUID of the root of each file's subtree in the file name.

2.2 Diff Algorithm

The diff algorithm computes the delta between two versions of a tree (T_{old} and T_{new}). The delta is a set of changes, where each change represents the evolution of one node and is a tuple consisting of:

- *oldNode*: the node tuple from T_{old}, if it exists (node was not inserted).
- *newNode*: the node tuple from T_{new}, if it exists (node was not deleted).
- *kind*: the kind of the change – one of *Insertion*, *Deletion*, *Move* (change of parent and possibly label, type, or value), *Stationary* (no change of parent, but change in at least one of label, type, or value).

These elements provide the full information necessary to report precisely how a node has changed. The encoding from Sect. 2.1 enables an efficient two-stage algorithm for computing the delta between two versions of a tree. The operation of the algorithm is illustrated in Fig. 2.

The **first stage** computes two sets of nodes $oldNodes \subseteq T_{old}$ and $newNodes \subseteq T_{new}$, which overapproximate the nodes that have changed between T_{old} and T_{new}. The sets are computed by comparing the encodings of T_{old} and T_{new} using a standard line-based diff [20,22,29]. Given two text files, a line-based diff computes a longest common subsequence (LCS), where each line is treated as an atomic element. The LCS is a subset of all identical lines between T_{old} and T_{new}. The diff outputs the lines that are not in the LCS, thus overapproximating changes: lines from the "old" file are marked as deleted and lines from the "new" file are marked as inserted. In the middle of Fig. 2, lines B, E, G, H, and D on the left are marked as removed and lines B', E', G, H, and X on the right are marked as inserted. The combined diff output for all files is two sets of removed and inserted lines. The nodes corresponding to these two sets, ignoring nodes of type External, are the inputs to the second stage of the diff algorithm.

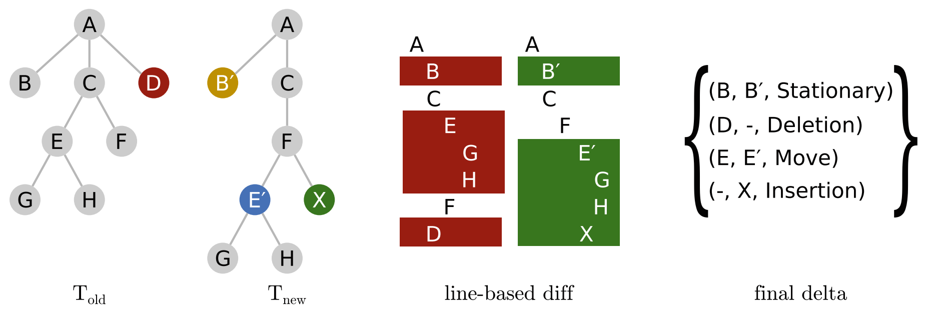

Fig. 2. A tree modification and the outputs of the two stages of the diff algorithm.

The **second stage** (Algorithm 2.1) filters the overapproximated nodes and computes the final, precise delta between T_{old} and T_{new}. The algorithm essentially compares nodes with the same id from *oldNodes* and *newNodes* and if they are different, adds a corresponding change to the delta. A node from oldNodes might be identical to a node from newNodes, for example, if its corresponding line has moved, but is otherwise unchanged. This is the case for nodes G and H in Fig. 2, where the final delta consist only of real changes to the nodes B, D, E, and X. This is in contrast to a line-based diff, which will also report G and H as changed, even though they have not.

In the absence of unique IDs stored with the tree, it is possible to compute matching nodes using a tree match algorithm, enabling our diff and merge algorithms to be used for traditional encodings of trees, such as Java files. To achieve this, the first stage needs to be replaced so that it parses the input files, computes a tree matching, and assigns new IDs according to the matching.

```
function TreeDiffStageTwo(oldNodes, newNodes)
    changes ← $\emptyset$
    for all {(old, new) ∈ (oldNodes × newNodes) | old.id=new.id ∧ old≠new} do
        if old.parentId = new.parentId then
            changes ← changes ∪ {(old, new, Stationary)}
        else
            changes ← changes ∪ {(old, new, Move)}
        end
    end
    for all {old ∈ oldNodes | old.id ∉ IDs(newNodes)} do
        changes ← changes ∪ {(old, NIL, Deletion)}
    end
    for all {new ∈ newNodes | new.id ∉ IDs(oldNodes)} do
        changes ← changes ∪ {(NIL, new, Insertion)}
    end
    return changes
end
```

Algorithm 2.1. The second stage of the *TreeDiff* algorithm. IDs is the set of all identifiers of nodes from the input set. A more detailed version of this algorithm and a proof of correctness can be found in [12].

The described diff algorithm eliminates (with unique IDs), or greatly reduces (using a tree matching algorithm) inaccurate diffs. This is because the formatting of the encoding is irrelevant, changes are expressed in term of tree nodes, and moved nodes are tracked, even across files. The diff provides a basis for improved merges, discussed next.

3 Merging Trees and Domain-Specific Customizations

Building on the diff algorithm from Sect. 2.2, we designed an algorithm for merging two tree revisions T_A and T_B given their common ancestor T_{base}. At the core of the merge is the change graph – a graph of changes performed by the two revisions, which includes conflicts and dependencies. In this section, we will first describe the change graph and how it is used to merge files, and then we will outline additional merge customizations, which use knowledge about the domain of the tree to improve conflict detection and resolution. Unlike the diff algorithm, the merge does not build on its line-based analog, which is unaware of the tree structure and may produce invalid results. For example, if two revisions move the same node (line) to two different parents, which are located in different parts of a file or in different files, a line-based algorithm would simply keep both lines, incorrectly duplicating the subtree, whereas our algorithm will report a conflict.

3.1 Change Graph and Merge Algorithm

The purpose of the *change graph* (CG) is to bring together changes from two diverging revisions and facilitate the creation of a merged tree. The nodes of the CG are changes, similar to the ones reported by the diff. The changes are connected with two types of edges, which constrain when changes may be applied. A change may require another change to be applied first, expressed as a directed *dependency edge*. For example, a change inserting a node might depend on the change inserting the parent node. Two changes may be in conflict with each other, expressed as an undirected *conflict edge*. For example, if both revisions change the same node differently, these changes will be in conflict. An example change graph is illustrated in Fig. 3.

To merge T_A and T_B into a tree T_{merged}, first an inverse topological ordering of the CG is computed using the dependency edges. Changes are applied according to this ordering, if possible. A change is applicable if it does not depend on any other change and has no conflict edges. Changes that form cycles in the CG may be applied together, in one atomic step, provided that all changes (i) have no conflict edges; (ii) are made by the same revision or by both revisions simultaneously; and (iii) do not depend on any change outside the cycle. Essentially, changes in such cycles are independent of other changes and are compatible with both revisions, making them safe to apply. These restrictions ensure that applying changes preserves the validity of the tree (see Sect. 7.3.1 in [6] for details).

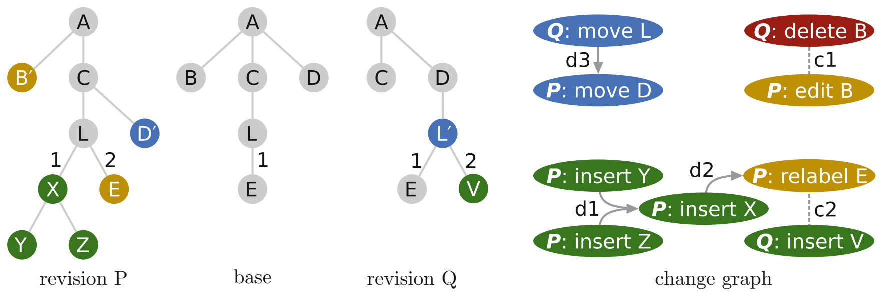

Fig. 3. A base tree with two modifying revisions and the corresponding CG. Each edge in the CG is labeled with the dependency or conflict type that the edge represents.

Applied changes are removed from the CG along with any incoming dependency edges. Once all applicable changes have been applied, any remaining changes represent conflicts and will be reported to the user. Next, we explain how the CG is constructed.

Merge Changes. A *merge change* is a tuple that extends the change tuple from the diff algorithm with one new element, *revisions*, which indicates which revisions make this change: *RevA*, *RevB*, or *Both*. The nodes of the CG are the merge changes obtained by running the diff algorithm twice to compute the delta between T_{base} and T_A and between T_{base} and T_B, respectively. First, each change from the two deltas is associated with either *RevA* or *RevB* to create a corresponding merge change. Then, we organize the elements of a tree node into two *element groups*: (i) parent and label; and (ii) type and value. Each group contains tuple elements whose modification by different revisions is a conflict. Any merge changes that modify both element groups are split into two merge changes: one for each element group. For example, if a node is moved to a new parent and its value is modified, this will appear as two separate and independent merge changes within the CG. This separation reduces conflicts and dependencies in the CG, since the two groups are independent. Finally, any identical changes made by different revisions are combined into a single merge change with *revisions=Both*, which ensures that identical changes are applied only once.

Dependencies Between Merge Changes. A dependency $X \rightarrow Y$ means that change X cannot be applied before Y, and is the first of two means that restrict applicable changes. Dependencies prevent three cases of tree structure violations.

(d1) orphan nodes: (a) Before a change IM inserts or moves a node N, N's parent destination node P must exist. If P does not already exist in T_{base}, then there must be an insertion change I, which inserts it. An edge $IM \rightarrow I$ is added to indicate that I must be applied before IM can be applied. In Fig. 3, nodes

Y and Z depend on the insertion of X. (b) Before a change D deletes a node N, all of N's children must be deleted or moved. An edge $D \rightarrow DM$ is added between D and each change DM that moves or deletes a child of N. In both (a) and (b), the changes I and DM are guaranteed to exist if they are necessary, because the merge changes were computed from the deltas of valid trees. Note that dependencies that prevent orphan nodes cannot form cycles on their own.

(d2) clashing labels: Before a change IMR inserts, moves, or relabels (modifies the label of) a node N, there must be no sibling at the destination of N with the same label. If a node with the same label as N's final label exists in T_{base} at the destination parent of N then there must be a change DMR that deletes, moves, or relabels that sibling. An edge $IMR \rightarrow DMR$ is added to the CG. In Fig. 3, node X depends on the relabeling of E. Such dependencies may form cycles. For example, swapping two elements in a list yields two relabel changes, where each change depends on the other.

(d3) cycles: If a change M_N moves a node N, N must not become its own ancestor. Such a situation occurs, for example, if a revision A moves an if-statement IF_1 into an if-statement IF_2, and revision B moves IF_2 into IF_1. To prevent such issues, move changes are applied only if the destination subtree does not need to be moved. This is enforced using dependencies. If M_N moves N to a subtree that needs to be moved, let M_P be the change that moves the subtree. An edge $M_N \rightarrow M_P$ is added to the CG. Move changes from different revisions may create dependency chains that form a cycle in the CG. For example, the move of IF_1 will depend on the move of IF_2, which will itself depend on the move of IF_1. Such a cycle means that the two revisions perform incompatible moves and the changes from the cycle cannot be applied. Move changes from different revisions do not always result in a cycle. For example, in Fig. 3, the move of L depends on the move of D, which is independent.

Conflicting Merge Changes. Conflicts that would result in a node becoming its own ancestor are indirectly represented in the CG in the form of dependency cycles described above. Other conflicts cannot be expressed with dependencies and appear directly as conflict edges, which are the second means for restricting change application. There are three cases of direct conflicts.

(c1) same node: If two revisions make non-identical changes X and Y to the same node, these changes may be conflicting. Deletions conflict with all other changes. Other changes conflict only with changes of the same element group. Conflicting changes are connected with an undirected edge $X \sim Y$ in the CG. An example of such a conflict is the modification and deletion of B in Fig. 3.

(c2) label clash: If a change IMR_N inserts, moves, or relabels a node N, and another change IMR_Q inserts, moves, or relabels a node Q such that N and Q have identical final labels and parent nodes, the two changes are in conflict. An edge $IMR_N \sim IMR_Q$ is added to the CG. In Fig. 3, such a conflict is the relabeling of E and the insertion of V.

(c3) deletion clash: If a change D_N deletes a node N, and another change IM_Q inserts or moves a node Q as a child of N, the two are in conflict. An edge $D_N \sim IM_Q$ is added to the CG.

In contrast to line-based merges, applying changes using the CG prevents incorrect merges by considering the tree structure. The algorithm, as described so far, has no knowledge about the domain of the tree, and misses opportunities for improved merges and better error reporting. Next, we explain customizations, that improve the merge results and report potential semantic issues.

3.2 Domain-Specific Customizations

Merging two tree revisions without any domain knowledge, as described so far, can lead to suboptimal merges. Figures 3 and 4 illustrate one such example, where revision P inserts a node X in the beginning of list L and revision Q inserts a node V at the end. These two changes conflict, because the label of E in P is identical to the label of V in Q. Despite this conflict, intuitively these changes can be merged by relabeling V. To achieve better merge results, we allow the merge process to be customized by taking domain knowledge into account. Customizations use domain knowledge, such as the semantics of specific node types or values, to tweak the CG, eliminating conflicts and dependencies, and thus, enabling additional changes to be applied. Customizations may also produce *review items*, which are messages that inform the user of a potential semantic issue with the final merge. Review items have two advantages over conflicts. First, unlike changes in a conflict or their depending changes (even if not in a conflict), which are not applicable, review items are not part of the CG and do not prevent the application of changes. Applying more changes is desirable because the final merge more closely represents both revisions and the user has to review issues with only a selected group of nodes, instead of manually exploring many unapplied changes. Second, review items provide semantic information to the user, making it easier to take corrective action, unlike conflicts, which represent generic constraints on the tree structure. Similarly, review

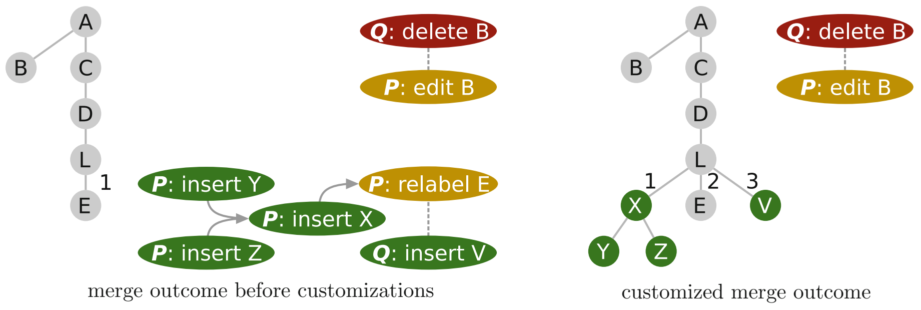

Fig. 4. The resulting T_{merged} and CG after applying all possible changes from Fig. 3 (left) and after additional customizations (right).

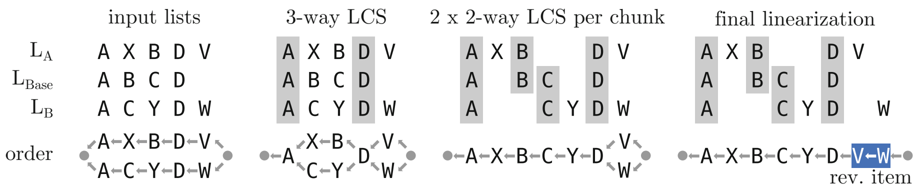

Fig. 5. Computing a total order for the elements of a merged list. Stable chunks have a light-gray background. At the end, V and W are linearized and added to a review item. In the merged list, B and C will be removed to reflect changes from revisions.

items are preferable to conflicts in line-based merges, because review items are more focused and provide semantic information. Next, we present two examples of customizations, which we have found useful for achieving high-quality merges.

List-Merge Customization. Data from many domains (e.g., ASTs, UML models) has list entities. Merging lists is challenging [13,28], as it is not trivial to determine the order of the merged elements and to detect and resolve conflicts. In addition, the CG often contains label clash conflicts in lists (e.g., for nodes E and V in Fig. 4), which are usually easy to resolve automatically. We developed the List-Merge customization, which is crucial for merging list nodes well. Essentially, the customization computes a total order of all list elements from both revisions. This total order is used to relabel all elements, giving each element a unique label. Thus, all conflicts or dependencies due to previously clashing labels are removed from the CG, allowing many more changes to be merged. Next, we describe the computation of the total order and how ambiguities are handled.

The total order is computed in three steps as illustrated in Fig. 5. In the **first step**, a three-way longest common subsequence (LCS) between L_{Base}, L_A, and L_B is computed and used to create an alternating sequence of stable and unstable chunks. The *stable chunks* are a partition of the LCS – elements in a single chunk are adjacent in all lists. There are two stable chunks in Fig. 5: $[A]$ and $[D]$. An *unstable chunk* consists of one element span per list, each span containing elements that are not in the LCS. There are two unstable chunks in Fig. 5: $[XB, BC, CY]$ and $[V, \epsilon, W]$. Elements from different chunks are totally ordered using the order of the chunks, e.g., A before X and D. Elements from the same stable chunk are totally ordered using their order within the chunk. In the **second step**, for each unstable chunk C, two two-way LCSs $lcs_a = LCS(C_{base}, C_a)$ and $lcs_b = LCS(C_{base}, C_b)$ are computed. Elements from lcs_a are totally ordered with respect to elements from lcs_b using the order in C_{base}. In Fig. 5 these are B and C. The remaining elements from C_a and C_b are ordered with respect to elements from lcs_a and lcs_b, respectively. Such elements are totally ordered using the order from one revision, if there are no elements from the other revision in the corresponding chunk (X and Y in Fig. 5). Otherwise, the elements are not totally ordered (V and W in Fig. 5). In the **third step**, unordered elements are linearized in an arbitrary order. If the list represents an ordered collection within the domain, a review item is created to inform the user of the ambiguity.

The List-Merge customization brings essential domain knowledge about lists to the merge algorithm. The customization not only resolves many conflicts automatically, but also reports merge ambiguities on a semantic level. Thus, it lets developers deal with less conflicts and do so more easily, saving time.

Conflict Unit Customization. Our merge algorithm is able to merge changes at the very fine-grained level of tree nodes, which is not desirable in some domains. For example, if $x < y$ in an AST is changed to $x \leq y$ in one revision, and to $x < y + 1$ in another, these two changes can be merged as $x \leq y + 1$, which is not intended. A common case where fine-grained merges might result in semantic issues is when changes affect nodes that are "very close" according to the semantics of the tree domain. We designed the Conflict Unit (CU) customization to detect such situations. In essence, the customization partitions the tree into small regions called CUs and creates review items for each CU that is changed by both revisions. The customization does not alter the change graph.

The CU customization is parametrized by a set of node types – the *CU types*. The *conflict root* of a node is its closest reflexive ancestor of a CU type. The tree root is always a conflict root. The set of nodes that have the same conflict root constitute a CU (see Fig. 6). If two revisions change nodes from the same CU, there is a potential for a semantic issue and this is reported with a review item.

With an appropriate choice of CU types, the CU customization can be useful in identifying potential semantic issues. For example, in ASTs, if statements are CU types, like in Fig. 6, a change in one statement is semantically independent from a change in another statement, but two changes in the subtree of the same statement will result in a review item. In this setting, if a developer changes one part of the $i \geq 0$ expression, while another developer changes another part, these changes will no longer be silently merged, but a semantic issue will be reported.

Structure- and semantics-based review items are more precise and meaningful than the line-based conflicts produced by standard algorithms. A line-based

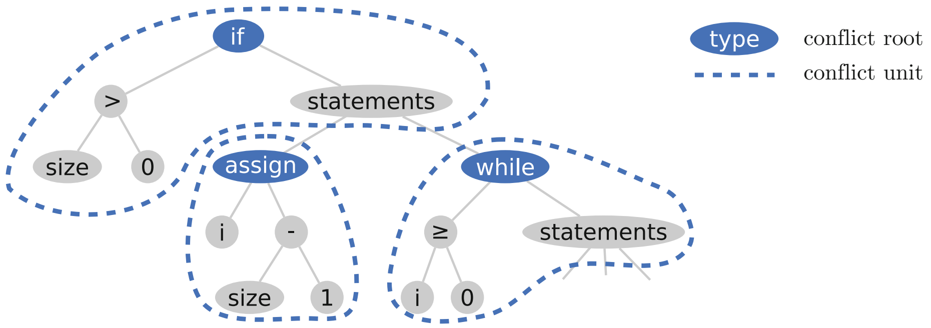

Fig. 6. A tree with three conflict units.

conflict might incorrectly arise due to compatible changes (e.g., moving a declaration in one revision and adding a comment in another revision) or it might be due to formatting (e.g., renaming a method in one revision and moving the opening brace to a new line in another revision). In contrast, our CU approach is precise, predictable, and uses domain knowledge to report issues on a semantic level.

4 Evaluation and Discussion

To evaluate our approach, we implemented our version control algorithms in the Envision IDE [5]. Even though Envision supports unique node IDs, we use Gumtree [9] to evaluate our approach on large existing Java projects to show its applicability on large trees with a long history. We inspected the default branch of the most popular (having more than 10000 stars) Java projects on GitHub, 19 in total. Six of the projects did not contain any merges of Java files. In the remaining 13 projects, we evaluate each merge of a Java file by comparing the merge results of Git and our implementation. We focus on the merge here since the merge operation depends on the diff and thus, reflects its quality. The results are presented in Table 1. All tests were run on an Intel i7-2600K CPU running at 3.4 GHz, 32 GB RAM, and an SSD.

A *divergent merge* (DM) is one that results in conflicts (C) or one where the automatically merged file is different from the file committed by the developer. One exception are successful automatic merges in Envision that only differ from the developer committed version by the order of methods or import declarations. Since this order is semantically irrelevant, we do not consider such merges divergent – they are counted as *order difference* (OD). For Envision, we also list the number of files whose merge produced review items due to linearized list elements ($\mathrm{RI_l}$) or changes to the same conflict unit by two revisions ($\mathrm{RI_{cu}}$). For conflict unit types we use all statement and declaration node types. The total and average merge times are reported for both tools (merge and avg. merge). Merging Java sources with Envision incurs a significant overhead (ovrhd.) in addition to the merge time, because (i) the sources have to be parsed, (ii) the different revisions have to be matched to the base using Gumtree, and (iii) these two-way matchings are tweaked to enable a three-way merge. Almost all of this overhead can be avoided by using IDs directly stored on disk.

Tree-based merging results in significantly fewer divergent merges, 717, compared to the standard line-based approach, 1100. The difference in conflicts is even more substantial, with 362 for the tree-based approach, and 1039 for Git. Our approach also reports a significant number of files with review items for lists, 222, and conflict units, 662. Unlike textual conflicts, review items describe the semantic issue they reflect and report the minimal set of nodes that are affected, which makes it easier for developers to understand and act on review items.

To get more insight, we manually investigated all 46 cases of Envision's diverging merges in the RxJava project. All of these merges also diverge when using Git. There are 34 merges with real conflicts or merges where the developer

Table 1. Comparison between merges by Git (G) and Envision (E). DM – divergent merge; C – merge with conflicts; OD – merge where only order differs; RI – review item (l – due to linearized list elements, cu – due to multiple changes in a CU).

project	**all**	**number of files**							**merge**		**ovrhd.**	**avg.merge**	
		DM		**C**		**OD**	**RI$_l$**	**RI$_{cu}$**	**[s]**		**[min]**	**[ms]**	
	merges	G	E	G	E	E	E	E	G	E	E	G	E
ReactiveX/RxJava	354	82	46	74	12	19	22	38	3	125	122	9	353
elastic/elasticsearch	2677	863	547	821	281	29	157	525	10	276	266	4	103
square/retrofit	49	14	13	12	10	0	3	12	0	3	4	3	53
square/okhttp	163	4	4	4	1	0	1	11	1	23	22	4	138
nostra13/Android-Universal-Image-Loader	56	15	10	14	6	0	4	8	0	2	4	5	36
iluwatar/java-design-patterns	59	2	1	2	0	0	1	1	0	0	3	5	3
JakeWharton/butterknife	18	0	0	0	0	0	0	0	0	1	2	6	73
greenrobot/EventBus	10	4	3	4	2	0	0	4	0	1	1	7	54
square/picasso	40	0	0	0	0	0	0	3	0	3	4	5	75
PhilJay/MPAndroidChart	169	32	35	24	20	0	9	21	1	13	15	4	76
square/leakcanary	13	1	0	1	0	0	0	2	0	0	1	7	31
bumptech/glide	76	69	54	69	29	2	22	32	0	3	6	3	39
spring-projects/spring-framework	339	14	4	14	1	0	3	5	1	2	19	3	5
total	4023	1100	717	1039	362	50	222	662	16	452	469	5	80

made a semantic change, neither of which can be automatically handled. In the remaining 12 cases, we observed two reasons for divergence in Envision.

First, in six cases the result of a tree merge was, in fact, correct, but the version committed by the developer was incorrect. This occurred when a Git merge results in a conflict which the developer resolves incorrectly, even though the resulting code compiles. For example, a conflict marker (`<<<<<<< HEAD`) inserted by Git was forgotten inside a block comment. Another example is the accidental omission of an `@Test` annotation which appeared just before a conflict marker. This omission potentially disabled one of the test cases in the code and went unnoticed for nearly three years until the RxJava developers accepted our patch for fixing it. Our approach automatically merges all of these cases correctly.

Second, six merges diverge due to the suboptimal matchings produced by GumTree. For example, if a particular Java `import` declaration is present in the base version, but is deleted in both revisions, GumTree may match the deleted `import` to two different newly inserted `imports` from the different revisions. Our merge algorithm detects this as a conflict and fails to merge the file.

In terms of run-time, merging files with Envision is, on average, 16 times slower compared to Git. Nevertheless, Envision still allows merging at a rate of 12.5 files a second, which is significantly faster than manually resolving conflicts.

However, if the files are not stored using the format we described in Sect. 2.1, and require parsing and tree-matching, there is significant overhead, which further slows down Envision by a factor of 60. In this case merging a single file could take about one minute. To further investigate the effect of the matching on the merge result, we implemented a simple tree-matching algorithm and used it instead of Gumtree on the RxJava project. Our tree-matching produces worse matchings compared to GumTree, but incurs less overhead (81 minutes instead of 122). The simpler matcher resulted in more divergent merges (70) and more conflicts (40), compared to using GumTree, but the results are still better than using Git. These results suggest that our approach is most useful for storage formats that include unique node IDs such that matching algorithms are avoided altogether.

Threats to Validity. We evaluated our implementation on 13 Java repositories. Our results might not apply to other projects, other languages, or trees that are not ASTs. Nevertheless, the code bases we used provide a wide variety of tree-merge situations, and we used popular projects in order to increase the ecological validity of the results.

The tool we used to convert Java files into files encoded as we described in Sect. 2.1 omits some rarely-used Java constructs such as multiple type bounds for generic types. It is possible that a conflict in Git is due to a part of the code, which is missing in the new encoding. We are not aware of such cases.

We discard the text formatting and some comments. To handle such unstructured data with our approach the data would have to be encoded as part of the AST, e.g., by attaching a textual prefix node to each AST node.

5 Related Work

Researches have proposed a number of systems for version control of structured data. Molhado [24] is a powerful stand-alone framework for versioning object-oriented data. It is based on an extensible model that could be used to version arbitrary types of objects. Molhado requires deep integration with the development environment, making Molhado the "heart of the environment", in contrast to our more lightweight approach. OperV [25] is another approach for versioning of structured tree data with fine granularity, which, unlike our system, is operation-based, thereby requiring additional data and more complex tool support. Unlike our approach, both Molhado and OperV introduce a custom storage backend and do not integrate with an existing VCS.

Altmanninger has surveyed various systems for versioning models [2]. One of the most popular model repositories is EMFStore [14], part of the Eclipse Modeling Framework. There is continued interest in the research community in improving EMFStore, e.g., by formalizing merging for models [31] or performing semantics-based mering [1]. Odyssey [21,26] is another model VCS, which targets UML models and features advanced merge capabilities. EMFStore, Odyssey, and most systems for versioning models are not often used to version trees, and unlike our approach, they use a custom backend and do not integrate with standard line-based VCSs. Our approach may be applied to graph models, e.g., by expressing them as containment trees, similar to Mikhaiel et al. [19].

Mens [18] provides an overview of different approaches for merging program sources. Newer approaches based on the full [3] or partial structure [4] of source files have been proposed by Apel et al. These approaches improve on the merge results of Git, and can be fast and practical, but unlike our approach they do not work with unique IDs stored as part of the files, and thus may be inaccurate. Other approaches, rely on storing unique IDs, for example, the version control system of TouchDevelop [27] or MolhadoRef [7]. However, TouchDevelop is designed for a specific language and automatically resolves conflicts by ignoring one of the revisions, and MolhadoRef is an operation-based system, in contrast to our approach. Neither of the two integrate with a standard VCS like our approach.

There are also approaches to enhance VCSs for software with additional knowledge about the semantics of code and refactoring in order to improve merging [7,8,23]. Our customization mechanism can also be used to provide similar semantics-based improvements to the merge.

Ghezzi et al. [11] propose that a pluggable framework be built on top of traditional VCSs in order to provide additional services and analysis capabilities. Our algorithms can be seen as an instance of their suggestion.

Lorenz and Rosenan [17] propose a JSON format for storing structured data and integrating it with a traditional VCS. Their proposal however uses the VCS only for storage and performs versioning on its own – one version of the JSON file in the VCS stores itself all previous versions of the objects that comprise it. In contrast, our approach uses the underlying VCS for both storage and versioning.

Lindholm [16] proposes a way to merge XML documents using the XML tree structure. Their approach focuses on the particular class of document-oriented XML files, whereas our approach is designed for arbitrary trees.

MPS [30] is a commercial system which stores programs as XML files and implements custom merge hooks to integrate with traditional VCSs. It relies on IDs for precise merging, but the system does not seem to be customizable or easily usable for other data.

Schwägerl et al. have designed a graph-based algorithm [28] for merging ordered collections. Unlike our List-Merge customization, their algorithm only works with inserted, deleted, and relabeled elements, and there is no treatment for elements which are moved in or out of the list to another subtree and possible conflicts with these operations.

6 Conclusion

We described an approach for accurate version control of tree structures using a mainstream line-based VCS. Our diff algorithm can work with either stored node identifiers or tree matching algorithms. It provides accurate deltas with respect to the input matching, which prevents inaccurate or confusing diffs. Our merge algorithm and domain-specific customizations eliminate incorrect merges, reduce unnecessary conflicts, and report semantic issues, improving the merge result.

We evaluated our approach on traditional Java ASTs with the help of the Gumtree tree-matching algorithm. We observed a substantial reduction in merge conflicts compared to a line-based approach. It will be worth to experiment with trees with stored IDs instead of computing node matchings, which would allow us to further quantify the performance of our approach.

Another promising research direction is the design of additional merge customizations that understand trees at a more semantic level. For example, we have started exploring a customization that can detect renamings of declarations in an AST in one revision and apply them automatically to another on merge. Such high-level customizations might help to further reduce conflicts in particular domains and detect additional semantic incompatibilities between revisions.

References

1. Altmanninger, K., Schwinger, W., Kotsis, G.: Semantics for accurate conflict detection in SMoVer: specification, detection and presentation by example. IJEIS **6**(1) (2010)
2. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. Int. J. Web Inf. Syst. **5**(3), 271–304 (2009)
3. Apel, S., Leßenich, O., Lengauer, C.: Structured merge with auto-tuning: balancing precision and performance. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012. ACM (2012)
4. Apel, S., Liebig, J., Brandl, B., Lengauer, C., Kästner, C.: Semistructured merge: rethinking merge in revision control systems. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE 2011. ACM (2011)
5. Asenov, D., Müller, P.: Envision: A fast and flexible visual code editor with fluid interactions (overview). In: 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), July 2014
6. Asenov, D.: Envision: Reinventing the Integrated Development Environment. Ph.D. thesis, ETH Zurich (to appear, 2017)
7. Dig, D., Manzoor, K., Johnson, R., Nguyen, T.N.: Refactoring-aware configuration management for object-oriented programs. In: 29th International Conference on Software Engineering (ICSE 2007), May 2007
8. Ekman, T., Asklund, U.: Refactoring-aware versioning in Eclipse. Electron. Not. Theor. Comput. Sci. **107**, 57–69 (2004)
9. Falleri, J.R., Morandat, F., Blanc, X., Martinez, M., Montperrus, M.: Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE 2014. ACM (2014)
10. Fluri, B., Wuersch, M., Pinzger, M., Gall, H.: Change distilling: tree differencing for fine-grained source code change extraction. IEEE Trans. Softw. Eng. **33**(11), 725–743 (2007)

11. Ghezzi, G., Würsch, M., Giger, E., Gall, H.C.: An architectural blueprint for a pluggable version control system for software (evolution) analysis. In: Proceedings of the Second International Workshop on Developing Tools As Plug-Ins, TOPI 2012. IEEE Press (2012)
12. Guenat, B.: Tree-based Version Control in Envision. BSc. Thesis, ETH Zurich (2015)
13. Kehrer, T., Kelter, U.: Versioning of ordered model element sets. Technical report 2, University of Siegen (2014)
14. Koegel, M., Helming, J.: EMFstore: a model repository for EMF models. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010, vol. 2. ACM (2010)
15. Koegel, M., Herrmannsdoerfer, M., von Wesendonk, O., Helming, J.: Operation-based conflict detection. In: Proceedings of the 1st International Workshop on Model Comparison in Practice, IWMCP 2010 (2010)
16. Lindholm, T.: A three-way merge for XML documents. In: Proceedings of the 2004 ACM Symposium on Document Engineering, DocEng 2004. ACM (2004)
17. Lorenz, D.H., Rosenan, B.: Source code management for projectional editing. In: Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, Languages & Applications: Software for Humanity, SPLASH 2013. ACM (2013)
18. Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. Softw. Eng. **28**(5), 449–462 (2002)
19. Mikhaiel, R., Tsantalis, N., Negara, N., Stroulia, E., Xing, Z.: Differencing UML models: a domain-specific vs. a domain-agnostic method. In: International Summer School on Generative and Transformational Techniques in Software Engineering IV, GTTSE 2011 (2013)
20. Miller, W., Myers, E.W.: A file comparison program. Softw. Pract. Exp. **15**(11), 1025–1040 (1985)
21. Murta, L., Corrêa, C., Prudêncio, J.G., Werner, C.: Towards Odyssey-VCS 2: improvements over a UML-based version control system. In: Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models, CVSM 2008. ACM (2008)
22. Myers, E.W.: An O(ND) difference algorithm and its variations. Algorithmica **1**(1) (1986)
23. Nguyen, H.V., Nguyen, M.H., Dang, S.C., Kästner, C., Nguyen, T.N.: Detecting semantic merge conflicts with variability-aware execution. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015. ACM (2015)
24. Nguyen, T., Munson, E., Boyland, J.: An infrastructure for development of object-oriented, multi-level configuration management services. In: Proceedings of the 27th International Conference on Software Engineering, (ICSE 2005), May 2005
25. Nguyen, T.T., Nguyen, H.A., Pham, N.H., Nguyen, T.N.: Operation-based, fine-grained version control model for tree-based representation. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 74–90. Springer, Heidelberg (2010). doi:10.1007/978-3-642-12029-9_6
26. Oliveira, H., Murta, L., Werner, C.: Odyssey-VCS: a flexible version control system for UML model elements. In: Proceedings of the 12th International Workshop on Software Configuration Management, SCM 2005. ACM (2005)

27. Protzenko, J., Burckhardt, S., Moskal, M., McClurg, J.: Implementing real-time collaboration in TouchDevelop using AST merges. In: Proceedings of the 3rd International Workshop on Mobile Development Lifecycle, MobileDeLi 2015. ACM (2015)
28. Schwägerl, F., Uhrig, S., Westfechtel, B.: A graph-based algorithm for three-way merging of ordered collections in EMF models. Sci. Comput. Program. **113**(Pt. 1), 51–81 (2015). Model Driven Development (Selected & extended papers from MODELSWARD 2014)
29. Ukkonen, E.: International conference on foundations of computation theory algorithms for approximate string matching. Inf. Control **64**(1), 100–118 (1985)
30. Voelter, M., Siegmund, J., Berger, T., Kolb, B.: Towards user-friendly projectional editors. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) SLE 2014. LNCS, vol. 8706, pp. 41–61. Springer, Cham (2014). doi:10.1007/978-3-319-11245-9_3
31. Westfechtel, B.: A formal approach to three-way merging of EMF models. In: Proceedings of the 1st International Workshop on Model Comparison in Practice, IWMCP 2010. ACM (2010)