diffTree: Robust Collaborative Coding using Tree-Merge

Jedidiah McClurg
CU Boulder *
jedidiah.mcclurg@colorado.edu

Sebastian Burckhardt
Microsoft Research
sburckha@microsoft.com

Michał Moskal
Microsoft Research
michal.moskal@microsoft.com

Jonathan Protzenko
Microsoft Research
protz@microsoft.com

Abstract

Handheld devices and cloud-connected applications are now commonplace, and developers cooperate more than ever, taking advantage of distributed version control systems and online collaborative development environments.

Providing a robust collaborative experience for editing code remains challenging, however. Traditional text-based merge algorithms (e.g. diff3) are unfit for automatic synchronization, as they often require manual resolution of merge conflicts. On the other hand, collaborative editing technologies such as operational transformations (OT) do not support extended offline operation, and are difficult to apply in contexts which must support complex editing operations.

To solve this problem, we introduce diffTree, a new 3-way merge algorithm for tree-structured data, and prove that it satisfies several properties which make it intuitive for users. We show how, by augmenting syntax trees with unique identifiers to track the tree nodes while they are edited, we can implement a failure-free 3-way merge on programs. We then show how to implement real-time collaboration with seamless online and offline support by applying this merge function continuously and automatically.

Our diffTree algorithm and collaborative-coding framework have been implemented in the TouchDevelop online programming environment, where they are publicly available for experimentation. We present preliminary evaluation of diffTree performance using data collected in the field.

* Work performed during an internship at Microsoft Research

1. Introduction

We live in an era where handheld devices and cloud-connected applications are highly prevalent. Even complex software development tasks are moving into the mobile arena. As a result, new forms of collaborative development are gaining traction. We believe there are two major trends in this area.

One is the rapid rise of *distributed version control systems*. Because they are well-suited for both loose and tight collaboration, and often integrate social functionality such as online discussion forums, they foster spontaneous collaboration of developers, sometimes across organizational boundaries. Online portals such as GitHub make it very easy for developers to fork existing code, make their own changes in private, and then reintegrate those changes later if desired.

Another trend is the use of *online integrated development environments* that can replace the traditional workstation as a place to store, compile, run, and test code. One of the benefits is that users can access the code they are currently developing from any device, including mobile devices, which results in a convenient single-workspace experience throughout the development process. Some typical technical challenges in this area are (1) how to adapt the editor experience to small screens and keyboard-less devices, and (2) how to support offline availability by storing eventually consistent replicas of the workspace in local storage [2].

1.1 Merging Code

A central technical challenge in both scenarios (version control systems, and replicated workspaces) is the requirement for some procedure that can reconcile concurrent, possibly-conflicting edits to the code being developed. The traditional solution to this problem is to rely on a per-file, text-based merge function (typically using the diff3 algorithm). However, traditional text-based merges leave much to be desired:

- The merge may fail and ask the user to manually resolve conflicts, which is inconvenient at best, and dauntingly difficult at worst.
- The merge algorithm sees a program as a collection of text files, rather than a tree of declarations/statements. Thus, the moving of code using copy/paste is misinterpreted as a deletion, followed by the addition of new, un-

related code. Thus, reorganizing or refactoring code often introduces merge conflicts requiring manual resolution.

- The diff3 algorithm has no insight into basic syntactic or semantic concepts, such as the difference between comments and code, or the use of identifiers. Common tasks like the global renaming of functions or variables are risky, since they can cause many merge conflicts.
- Since it does not always succeed, a traditional merge cannot be used for fully automatic synchronization.

1.2 The diffTree Algorithm

To achieve a better collaboration experience, we propose and evaluate a solution that represents programs as *trees*, and merges those trees using a novel tree-merge algorithm called *diffTree* which always succeeds (i.e. never requires any manual conflict resolution). A key aspect of our solution is to embed invisible *unique identifiers* in the code that mark the nodes of the tree. These identifiers allow us to (1) track accurately how users rearrange nodes of the tree when they move code within and across functions, and (2) track the relationship of identifier uses to identifier definitions. Thus, our merge function reliably merges changes even if users move code in arbitrary ways or rename identifiers at will.

We call our trees *hybrid syntax trees* (HST), since they do not simply correspond to traditional concrete or abstract syntax trees used in compilers. Rather, they combine information that is typically associated with several different compiler phases (tokenize, parse, symbol). We show that our merge algorithm preserves some basic semantic information about merged trees, and possesses several properties which make it intuitive for developers and allow it to be applied continuously without changing the trees in unexpected ways.

1.3 Implementation in TouchDevelop

We have implemented our system in the TouchDevelop online programming environment [2, 15], which already included a semi-structured editor, hybrid syntax trees, and support for forking (but not merging) scripts online. Using our new diffTree merge function, we were able to implement the following new functionality:

- Scripts that have a common ancestor can be merged. This allows developers to not only freely fork and edit scripts, but to combine those changes at a later point as desired.
- Users can join "group scripts" which are collaboratively edited by all participants. This collaboration is based on automatic, continuous merging, and can transition seamlessly between online, real-time collaborative editing (similar to what is provided by Google Docs, Office Online, etc.), and offline editing (which requires maintaining eventually consistent replicas in local storage).

Our solution requires some integration with the code editor, in order to accurately track the unique identifiers. The amount of additional information maintained is small, however—one unique identifier per line of code. Preserving this type of information is compatible with typical text editing operations like cut-pasting full lines of code or various code-moving refactorings. It is thus conceivable that a smart text-based editor, and not only a semi-structured editor, could also keep track of such identifiers behind the scenes, making our technique applicable to traditional programming languages edited in modern IDEs.

1.4 Contributions

- We develop a general and failure-free 3-way merge algorithm *diffTree* for tree-structured data with embedded unique identifiers, and show that it has desirable properties regarding quality of the merge result (§3).
- We show how to take advantage of diffTree in the context of an IDE that uses a semi-structured editor and stores programs as hybrid syntax trees (§2), thus allowing developers to collaborate by branching and merging programs freely as desired.
- We describe how to use diffTree in a continuous, automatic manner. This enables, through a single mechanism, both online real-time collaboration, as well as support for offline editing of eventually consistent replicas stored in local storage (§4).
- We have implemented diffTree-based collaborative coding as a publicly available feature in the TouchDevelop IDE, and demonstrate its use in a linked video.[1]
- We present a performance evaluation of diffTree, on data automatically collected since we deployed the online collaboration feature (§6).

Overall, we make a definitive step towards enabling both real-time and off-line collaboration in an online software development environment. The results of this research can help make developers more productive, allowing them to edit and collaborate on-the-go. It can also help educators make their programming classes more interactive, allowing an entire classroom of students to participate in a "hands-on" way with software development assignments.

2. Language and Environment

TouchDevelop is a web-based application development environment which allows participants to use a structured code editor (see Figure 1) to write scripts in a simple imperative programming language, and then run these scripts in a browser. Many libraries are made available, allowing users to easily create games, quizzes, interactive forms, and all the way up to RESTful web services utilizing state-of-the-art cloud infrastructure. TouchDevelop users range from students and teachers to enthusiasts and seasoned developers.

[1] The video is available at `http://bit.ly/tdmerge` or directly at `http://youtu.be/e7Pa5pxIyeM`. The curious reader can play with the implementation at `https://www.touchdevelop.com/`.

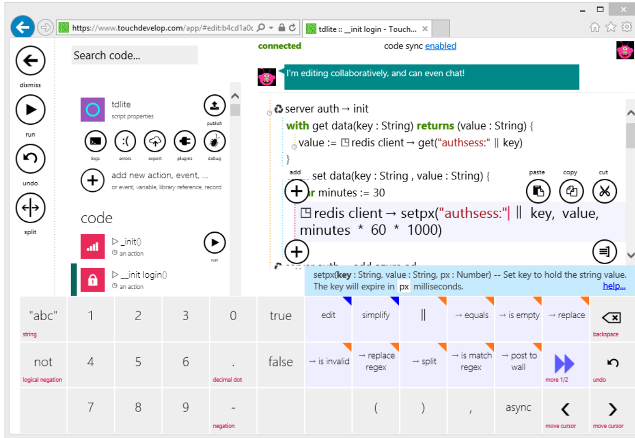

Figure 1: TouchDevelop structured code editor

TouchDevelop scripts are cross-platform, since the runtime and editor are both written in JavaScript/HTML5 and run in a wide range of browsers. Users can easily publish their scripts, allowing the community to read and extend them by "cloning" (forking) published scripts. These features, in combination with the semi-structured code-editor, make TouchDevelop an excellent candidate for experimenting with features for collaborative coding.

2.1 Example: Merging Conflicting Edits

To motivate our use of a tree-based merge function, consider the simple script shown at the top of Fig. 2. This script displays the numbers 0 through 9 (the post-to-wall method is a built-in method that displays the argument). The **for**-loop implicitly declares the loop variable **i** and initializes it to 0.

We now consider two users, Alice and Bob, who independently fork and modify this script. Alice decides to change the printed value from **i** to (2 ∗ **i**), while Bob refactors the print functionality into a separate function **print**. What happens if they try to merge their changes? With a text-based merge function, the conflicts cannot be automatically resolved, and the user would need to determine what happened and manually fix it. However, our tree-based merge function can use embedded unique identifiers to detect that the print statement was *moved* by Bob, and can thus correctly apply Alice's change to the statement in its new location.

2.2 Hybrid Syntax Trees

In TouchDevelop, a program is always stored and edited as a tree, using a structured editor. Figure 3 shows the basic (simplified) syntax tree of TouchDevelop programs, in line with how the user edits them and the environment stores them. While it may look like an abstract syntax tree at first, it is both more concrete in some ways and more abstract in others. We thus call it "hybrid" syntax tree, or HST.

At the top level, a program is a sequence of function declarations (we have omitted data declarations for simplicity). Each function declaration contains a unique node identifier

original program (version O)

```
function main() {
  for var i < 10 {
    i→post to wall
  }
}
```

Alice's change (version A)

```
function main() {
  for var i < 10 {
    (2 * i)→post to wall
  }
}
```

Bob's change (version B)

```
function main() {
  for var i < 10 {
    print(i)
  }
}
function print(i: Number) {
  i→post to wall
}
```

merged changes

```
function main() {
  for var i < 10 {
    print(i)
  }
}
function print(i: Number) {
  (2 * i)→post to wall
}
```

Figure 2: Illustration of concurrent editing operations and the desired outcome of the merge.

(ID) n for the function, its current name *cseq*, a formal parameter list (which contains for each parameter a unique identifier, a name, and a type), and a body which is a list of statements. Node IDs must be unique (the implementation uses randomly generated strings).

All statements start with a node ID n, which is used by the merge function to track how statements in different versions of the scripts correspond. We show five kinds of statements:

- a *comment* statement, containing a character sequence;
- an *expression* statement, containing an expression which is a *token sequence*;
- a *variable declaration*, containing an identifier, as well as an initialization expression;
- a *conditional* statement, containing an expression and two statement sequences, one per clause;
- a *for-loop*, containing an identifier (loop variable), an expression for the bound, and statements for the body.

Note that although the statement syntax is recursive as usual, *expressions* are not defined recursively, but rather as a flat sequence of tokens. This is because the editor works

$$
\begin{array}{rcll}
prog & ::= & \overline{decl} & \textit{(program)} \\
decl & ::= & n : \texttt{function}\ cseq(\overline{param})\ \texttt{do}\ \overline{stmt} & \textit{(decl)} \\
param & ::= & n : cseq : \tau & \textit{(param)} \\
stmt & ::= & n : //\ cseq & \textit{(statement)} \\
 & | & n : \overline{token} & \\
 & | & n : \texttt{var}\ cseq := \overline{token} & \\
 & | & n : \texttt{if}\ \overline{token}\ \texttt{then}\ \overline{stmt}\ \texttt{else}\ \overline{stmt} & \\
 & | & n : \texttt{for var}\ cseq\ \texttt{below}\ \overline{token}\ \texttt{do}\ \overline{stmt} & \\
token & ::= & op \mid digit \mid n \mid ☹cseq & \textit{(token)} \\
\tau & ::= & cseq & \textit{(type)} \\
n & \in & NodeId & \textit{(node ID)} \\
cseq & \in & \{c \cdots : c \in Char\} & \textit{(char sequence)} \\
op & \in & \{+, -, *, /, <, >, =\} & \textit{(operator)} \\
digit & \in & \{0, 1, \ldots, 9, .\} & \textit{(digit)}
\end{array}
$$

Figure 3: Basic TouchDevelop HST

differently at the statement level than at the expression level. In terms of Java-like language, the users never type braces, but they do type parentheses.

For example, when the user inserts an if-then-else statement, it appears with three holes for the condition and two branches, and it is impossible to delete the "then" or "else" keywords alone. Thus, programs are never malformed at the *statement* level. However, it is perfectly possible to enter malformed *expressions* like `foo(42`. Allowing malformed intermediate expressions improves the speed at which users edit expressions [5], and helps avoid causing users to feel overly constrained by the structured editor.

Each token is either an operator (including numeric digits), or a node ID that refers to some variable or some function (as determined by the declaration with matching node ID), or an unbound identifier. Unbound identifiers cannot be entered directly, but may occur as result of deleting or moving code (see the discussion in Section 2.4 below).

Besides parse errors and unbound identifiers, token sequences can also contain type errors. The type errors do not affect the merge process in any way, so we ignore them here.

2.3 Example HST

For example, Bob's version B of the program in Fig. 2 has the following HST, where greek letters represent node IDs:

```
α: function main() do
  β: for var i below 1 0 do
    γ: δ ( β )
δ: function print(ϵ : i : Number) do
  ϕ: ϵ →post to wall
```

2.4 Copy/Paste on HSTs

Users can copy (or cut) a sequence of statements (together with their identifiers) and paste it in another part of the program. Bound identifiers are treated as follows:

- When a function or variable declaration is pasted in a context where the same identifier is already used by a different declaration, the identifier is changed.
- When a user copies a reference to a local variable without also copying the corresponding declaration, the reference is changed to an unbound identifier (containing the identifier used in the original declaration). Upon paste, we try to re-bind unbound identifiers in the target context: if a declaration of the corresponding identifier exists, we use the corresponding node ID.
- The same mechanism is applied to global declarations when the copy-paste operation is performed across different programs.

3. Tree Merge Algorithm

We now proceed to present the centerpiece of our solution: the diffTree algorithm for merging trees. To explain this 3-way merge function at an appropriate level of generality, we first introduce a general notion of an (ordered, annotated) tree. This simplifies the description of how diffTree operates, as it abstracts away the language-specific properties of the HST representation introduced in the previous section. The translation from HST to this generic tree is described in §3.7.

3.1 Definition of Trees

A tree is a set T of tuples of the form (n, p, S, Pr), where $n \in NodeId$ is the unique node ID, $p \in NodeId \cup \{\varepsilon\}$ is the parent ID (or ε if no parent), S is the set of subsequent sibling node IDs, corresponding to the siblings of n appearing to the *right* of n in the tree, and $Pr : \mathbb{N} \to String$ is a mapping from property IDs to values. There is a single root node having ID $\mathcal{R}$. When there is no ambiguity, we omit unused elements of the tuple, e.g. (n, p) when referring to n and its parent p. Note that considering S to be the entire set of successor siblings is slightly redundant—although we could just consider the immediate successor sibling, our set notation makes later developments more straightforward.

Tree property retrieval. We denote the set of all trees by $\mathcal{T}$, and we use functions $par : NodeId \times \mathcal{T} \to NodeId \cup \{\varepsilon\}$ and $index : NodeId \times \mathcal{T} \to \mathbb{N}$ to retrieve the parent of a node, and the node index among its siblings respectively. We use $children : NodeId \times \mathcal{T} \to \mathcal{P}(NodeId)$ to retrieve the children of a node (and $descendants$ to retrieve the children, grandchildren etc.), and use $update(Pr, prop, v)$ to change property $prop$ to value v in map Pr. The $ids : \mathcal{T} \to \mathbb{N}$ function obtains all node IDs from a tree, i.e. $ids(T) = \{n : (n, p, S) \in T \text{ for some } p, S\}$. We use notation $(x \to y) \in T$ to indicate that x is a predecessor sibling of y, that is, $\exists (x, p, S) \in T$ s.t. $y \in S$.

3.2 Design Guidelines

The basic idea of this section is to construct a 3-valued merge function $M(T_O, T_A, T_B)$ that takes three versions of a tree: the original version T_O, a first changed version T_A, and a second (independently-) changed version T_B, and computes as its result a tree T_M that contains the merge of those two changes (as shown in the example in Fig. 2). We design our algorithm according to the following general guidelines, applied to various aspects of the tree structure:

1. *Preserve non-change* – if an aspect was changed neither in T_A nor T_B, then it should not be changed in the result.
2. *Prefer change over no change* – if an aspect was changed by T_A but not T_B, or was changed by T_B but not T_A, then the result should contain the changed aspect.
3. *Preserve identical change* – if some aspect was changed by both T_A and T_B in the same way, then the result should be changed in the same way.
4. *Prefer A over B on conflict* – if the same aspect was changed by both T_A and T_B in different ways, then the result should be changed in the same way as in T_A.

Item 4 expresses the fact that the merge function must be asymmetrical. In our collaborative editing implementation, we view A as "mine" and B as "theirs", but we do not impose such semantic requirements on A, B in general.

Our algorithm satisfies the above guidelines as best it can, while always maintaining essential structural invariants, i.e. merging *trees* should result in a tree, and merging *sequences* should result in a sequence.

3.3 Merge Algorithm

Somewhat surprisingly, the guidelines in Section 3.2 are quite specific and provide an excellent blueprint for the construction of a reasonable merge function. The key insight is to apply the above rules *separately and subsequently* to the following aspects of a tree: (1) the **set** of node IDs contained in the result, (2) the **tree** structure (i.e. parent relationship) of nodes in the result, (3) the **sequence** structure of nodes in the result (i.e. the relative order of siblings, meaning nodes with the same parent), and (4) the node **properties** of each node in the result. We now describe the corresponding four steps of our merge algorithm, and show that they satisfy basic expectations, referring to the appendices for details/proofs.

Step #1 – Perform additions and deletions. Let $I_M = f(ids(T_O), ids(T_A), ids(T_B))$, where $f(o, a, b) = ((a \cup b) - o) \cup (a \cap b \cap o)$. We will construct T_M starting with the nodes appearing in the set I_M (see Figure 4).

Note that this definition matches the guidelines of Section 3.2 precisely, if the aspects being considered are the set membership of each node.

Step #2 – Determine parent for each node (Tree Merge). We construct a function $par : I_M \to I_M \cup \{\delta\}$ which maps each vertex to its new parent (or to δ if it has been deleted).

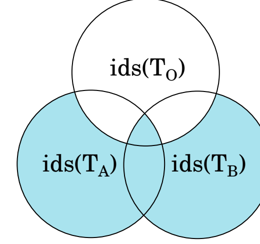

Figure 4: Merge Step #1 – additions/deletions (I_M is shaded)

```
1   function TreeMerge(T_O, T_A, T_B) {
2     var par = ∅ // empty map
3     var V_L = ∅ // empty set
4     var domain = I_M // copy of the I_M set
5     // take changes from A
6     forall x ∈ I_M {
7       if (x,p1) ∈ T_O ∧ (x,p2) ∈ T_A for some p1,p2 {
8         par(x) = p2; if p1 ≠ p2 { V_L = V_L ∪ {x} }
9       }
10    }
11    // take changes from B which have not been
          blocked by a change from A
12    forall x ∈ (I_M − V_L) {
13      if (x,p1) ∈ T_O ∧ (x,p3) ∈ T_B for some p1 ≠ p3 {
14        par(x) = p3
15      }
16    }
17    // removed orphaned nodes resulting from
          deletions
18    while ∃x ∈ I_M : par(x) ∉ I_M {
19      domain = (domain − {x})
20      par(x) = δ
21      forall y ∈ children(x) { par(y) = δ }
22    }
23    // deterministically break any cycles in
          the parent graph (restricted to domain)
24    return makeTree(par, domain)
25  }
```

Figure 5: Tree Merge algorithm (construct the mapping par)

This procedure is shown in Figure 5. At line 23 It is possible for the parent graph par to contain cycles. The `makeTree` function deterministically breaks any cycles in the parent graph par restricted to $domain$, returning the resulting tree.

Note that any cycle in the parent graph must look like the left side of Figure 6 (non-shaded nodes and solid edges).

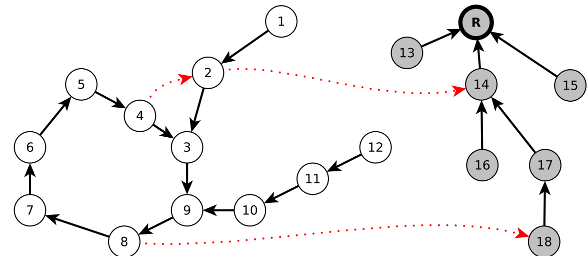

Figure 6: The structure of cycles in parent graph par

Lemma 1 (Properties of an arbitrary parent-graph cycle)**.** *Any cycle induced by the relation par satisfies the following:*

- *The cycle must be* simple*, since no node can have two outgoing edges (i.e. each node only has a single parent).*
- *A node within the cycle may have descendants outside. We call these "tails" of the cycle* descendant-subtrees.
- *For any node x in the cycle, $(\{x\} \cup descendants(x, par))$ contains no descendant of the root node $\mathcal{R}$.*

The `makeTree` function does a depth-first search (DFS) of the parent graph starting from the root $\mathcal{R}$ and using edges from T_A, reconnecting any encountered disconnected components by resetting the parent to a node outside the disconnected component. For example, in Figure 6, if the DFS first arrives at node 8, it removes the edge $8 \rightarrow 7$ by adding the edge $8 \rightarrow 18$. If the DFS arrives at a node such as 4 by way of a descendant(s) of the cycle such as 2 which is *outside* the cycle, this cycle-breaking function must be repeated.

```
1  function makeTree(par, domain) {
2    // compute set of all descendants of root
3    var children = descendants(R, par)
4    // compute strongly-connected components
         of parent graph
5    var components = tarjans(domain, par)
6    DFS(∅, R) // start DFS from the root node
7    function DFS(parents, n) {
8      if n ∈ S for some S ∈ components ∧ |S| > 1 {
9        do {
10         par(n) = head(parents) // immediate parent
11         parents = tail(parents) // older ancestors
12       } while(par(n) ∉ children)
13       // at this point, the cycle is gone.
14       // update the descendants
15       children = children ∪ descendants(par(n), par)
16       // update strongly-connected components
17       components = (components − {S})
18     }
19     forall x ∈ children(n) { DFS(n::parents, x) }
20   }
21   // return the resultant graph (tree)
22   return {(x,p) : x ∈ domain ∧ p = par(x)}
23 }
```

Figure 7: Deterministically breaking cycles in parent graph

Lemma 2 (Correctness of Tree Merge)**.** *The graph produced by the Tree Merge algorithm in Figure 5 is a* tree.

Lemma 3 (Parent-child preservation of Tree Merge)**.** *The Tree Merge algorithm preserves the parent-child relationship for unchanged nodes, i.e. if p is the parent of x in both T_A and T_B, and p, x appear in the merged result T_M, then p is also the parent of x in T_M.*

Step #3 – Determine sibling order (Sequence Merge). For each $p \in domain$, we can compute the set of children $C = par^{-1}(p)$ in the tree produced by `TreeMerge`. Our task is now to produce an ordering R of the elements in C, and we do this by examining the order of the children of p in T_O, T_A, T_B. Specifically, this is done by calling the $SeqMerge(L_O, L_A, L_B)$ function shown in Figure 8 on the sequences L_O, L_A, L_B which are children of p in O, A, B respectively. This function only produces an order for the intersection of L_O, L_A, L_B, discarding *deleted* nodes, and relegating the insertion of *added* nodes to the helper function `insertAdded` (Figure 9).

```
1  function SeqMerge(L_O, L_A, L_B) {
2    R = ∅ // empty relation
3    C' = L_O ∩ L_A ∩ L_B // the set we need to order
4    added = (L_A ∪ L_B) − L_O // added nodes
5    addMap = ∅ // empty map from node ID to
         list of IDs
6    // take changes from A
7    forall x,y ∈ C' where y ≠ x {
8      if (x → y) ∈ L_A ∧ ((x → y) ∉ L_O ∨ (y → x) ∉ L_B)
9      {
10       R = R ∪ {(x -A-> y)}
11     }
12   }
13   // take changes from B
14   forall x,y ∈ C' where y ≠ x ∧ (x,y) ∉ R ∧ (y,x) ∉ R
         {
15     if (x → y) ∈ L_B ∧ (y → x) ∈ L_O ∩ L_A {
16       R = R ∪ {(x -B-> y)}
17     }
18   }
19   // compute strongly-connected components
         of the relation R
20   components = tarjans(C', R)
21   // go through each of the non-trivial
         components
22   forall S ∈ components where |S| > 1 {
23     // replace each B-edge involved in this
           cycle with an opposing A-edge
24     forall (x -B-> y) ∈ S {
25       R = (R − {(x -B-> y)})
26       R = (R ∪ {(y -A-> x)})
27     }
28   }
29   // sort according to R
30   temp = sort(C', R)
31   // insert the added nodes into the ordered
         sequence
32   return insertAdded(temp, added, L_A, L_B)
33 }
```

Figure 8: Sequence Merge algorithm

Although this sequence merge is quadratic in the worst case, one can envision many optimizations that would offer better performance. Most straightforwardly, we could first compute "hunks" between L_O, L_A, L_B as done in diff3, and coalesce stable (matching) hunks into single nodes, saving us from having n^2 edges connecting all items in the sequence. However, in practice, we have not found such optimizations to be necessary, as the size of straight-line blocks of code is typically small (see performance results in §6).

```
1  function insertAdded(result, toAdd, L_A, L_B) {
2    addMap = ∅ // empty map from node ID to
         list of IDs
3    result = ∅ // empty list
4    function makeAddMap(L) {
5      prev = [ε] // singleton list
6      forall x ∈ L {
7        if x ∈ toAdd {
8          forall y ∈ prev {
9            addMap(y) = addMap(y)@[x] // append
10         }
11       }
12       prev = prev@[x] // append x to list
13     }
14   }
15   makeAddMap(L_A); makeAddMap(L_B)
16   forall x ∈ reverse([ε] :: result) {
17     // prepend x and any outstanding
          successors
18     result = (x :: (addMap(x) ∩ toAdd))@result
19     // mark the elements added to prevent
          re-adding
20     toAdd = toAdd − addMap(x)
21   }
22   return result
23 }
```

Figure 9: Sequence Merge algorithm - adding nodes

Lemma 4 (Correctness of Sequence Merge)**.** *The function $SeqMerge(L_O, L_A, L_B)$ produces a total order on the common elements of L_O, L_A, L_B.*

Lemma 5 (Order preservation of Sequence Merge)**.** *The Sequence Merge algorithm preserves the order relationship for unchanged nodes, i.e. if x appears before y (i.e. $x \rightarrow y$) in both L_A and L_B, and x, y appear in the merged result L_M, then x also appears before y in L_M.*

Step #4 – Merge properties of the nodes (Node Merge). Finally, we need to determine the properties that will appear in each node of the resulting merged program. We do this by simply preferring change over no change for each property. That is, if we are determining the value of property pr in a node $n \in T_M$, we look at the corresponding values pr_O, pr_A, pr_B in T_O, T_A, T_B. If $pr_O \neq pr_A$, we take the value from A, and otherwise we take the value from B.

Overall diffTree algorithm. Given the par map and set of nodes $domain$ generated by `TreeMerge`, along with the relation $R(C)$ generated by `SeqMerge` for each set of children C, we can readily construct the result tree. Specifically, $T_M = \{(n, p, S, Pr) : n \in domain, Pr = newProps(n), p = par(n), S = \{y \in domain : (n, y) \in R(par^{-1}(p))\}\}$, where $newProps$ is a mapping from node IDs to their new properties, as determined by Step #4.

3.4 Simple Tree Merge Examples

We now present some simple examples of the `TreeMerge` portion of diffTree. A-edges are shown in red (solid) with thick lines for those in V_L, and B-edges in blue (dashed), and T_L shows the result before cycle-removal. Figure 10 shows an edit where both users A, B try to set the parent of node D (conflict resolved by choosing A's edit). Figure 11 shows a conflicting edit where user A moves the C, D subtree, and user B moves the A, B subtree. This creates a nontrivial strongly-connected component, and we break the cycle by doing a DFS traversal of T_A (`makeTree` function).

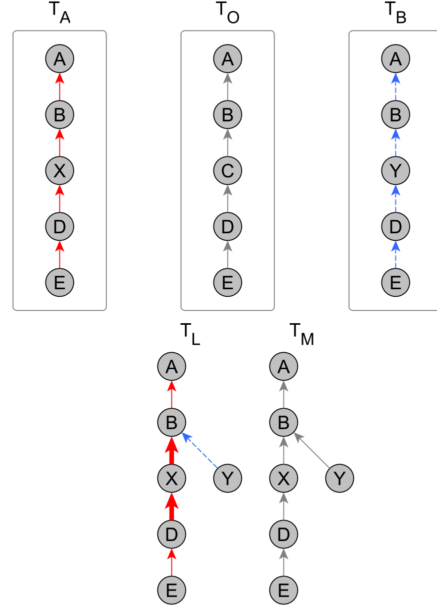

Figure 10: *Tree merge:* change in T_B blocked by change in T_A

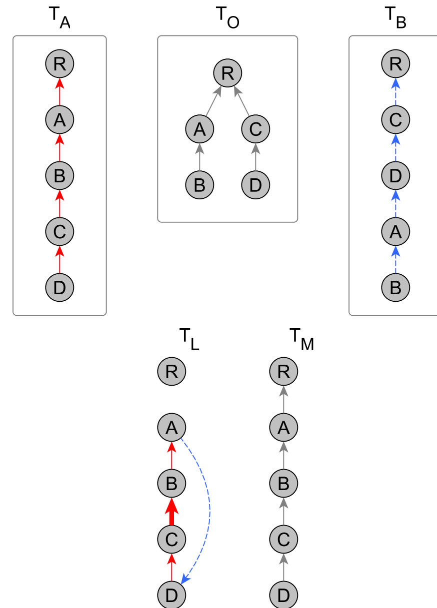

Figure 11: *Tree merge:* one nontrivial strongly-connected component (cycle must be broken)

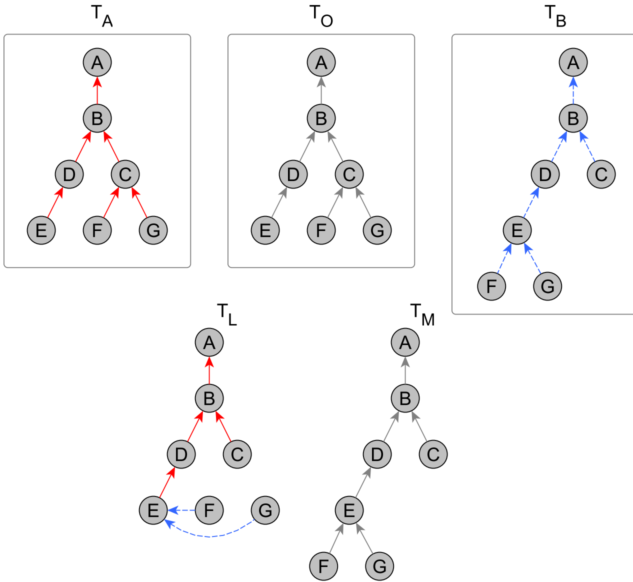

Figure 12: *Tree merge:* T_A is unchanged, allowing all T_B changes

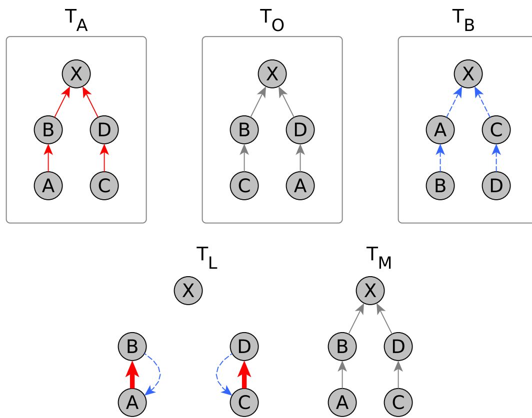

Figure 13: *Tree merge:* two nontrivial strongly-connected components (cycles must be broken)

Figure 12 shows an edit from user B which does not conflict with user A's edits. Finally, Figure 13 shows a highly conflicting edit where users A and B swap nodes in different ways. This creates two nontrivial strongly-connected components, and again cycles are broken deterministically.

3.5 Simple Sequence Merge Examples

We now show some examples of the Sequence Merge portion of diffTree. In Figure 14, no users have made any edits, so the resulting sequence is unchanged. In Figure 15, users have made conflicting edits, with user A swapping the order of $3, 1$, and user B moving 2 between the originally-ordered $3, 1$. This results in a cycle, which is deterministically resolved by reversing all dashed/blue edges (ones caused by user B) in the cycle. Figure 17 shows another example of breaking cycles. Figure 16 shows a non-conflicting change caused by user B.

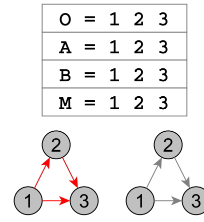

Figure 14: *Sequence merge:* no changes

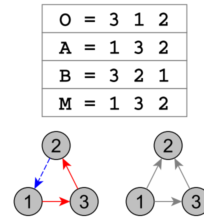

Figure 15: *Sequence merge:* cycle broken by reversing (dashed/blue) B-edge

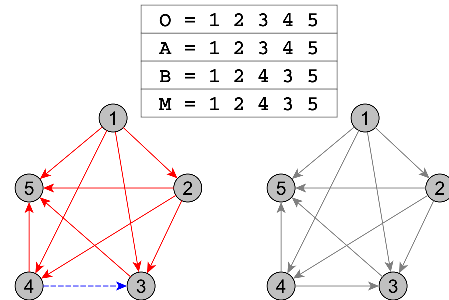

Figure 16: *Sequence merge:* only change in B

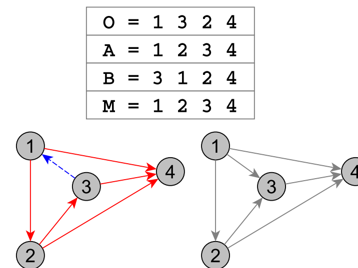

Figure 17: *Sequence merge:* another cycle broken by reversing (dashed/blue) B-edge

3.6 Additional Properties of diffTree

Our merge function is well-defined on any combination of trees, and constructed to satisfy the general guidelines in Section 3.2. Beyond that, we state and prove some additional

properties about the merge function, to gain confidence that its result is acceptable and matches programmer intuitions.

Theorem 1. *If one of the changes is trivial, the result matches the other change:*

- $M(T_O, T_O, T_O) = T_O$
- $M(T_O, T_A, T_O) = T_A$
- $M(T_O, T_O, T_B) = T_B$

The theorem follows from corresponding lemmas for each step of the merge function (as shown in appendix A).

Additional properties we would like to prove are cumulativity (if a programmer makes two changes, the effect of merging after each change should be the same as if merging only after both changes, e.g. Figure 18) and associativity (if three changes T_A, T_B, T_C are made, the effect of merging the merge of T_A, T_B with T_C should be the same as the effect of merging T_A with the merge of T_B and T_C).

3.6.1 Well-Separated Edits

Unfortunately, cumulativity/associativity do not hold without additional preconditions on the trees involved. However, identifying situations in which these properties are guaranteed provides an additional level of validation for our merge.

To this end, we define a notion of *edit* operations on trees, and of *conflicting* edit operations. To keep the presentation light, we give informal definitions here, and include the formal definitions in the appendix containing the proofs.

Edit operations include (1) *additions*: new nodes are added to the tree, (2) *deletions*: nodes are removed from the tree, (3) *move operations*: nodes are (atomically) removed from one position, and inserted at another, and (4) *change operations*: node properties are modified.

Given some set of edit operations, we identify the following conflict situations:

- *Ancestry* conflict – a (sub)set of edits places cyclic parent dependencies on a node.
- *Delete* conflict – an edit deletes a node x while another edit moves a node y into or out of the subtree of x.
- *Ordering* conflict – if an edit reverses the order of some x, y (e.g. $x \to y$ changes to $y \to x$), we call this a *reordering*, and we refer to any node involved in a reordering as *unstable*. An ordering conflict results when a (sub)set of reorderings places cyclic order requirements on a node.
- *Add* conflict – (1) a node is added adjacent to an unstable node, or (2) multiple edits add nodes with identical IDs.

A *well-separated* set of edits is one which has no conflicts. Abusing terminology slightly, if $T_1, \cdots, T_m$ are descendants of common base T_O, we say that T_i are well-separated with respect to T_O if $\bigcup_i edits(T_O, T_i)$ is well-separated.

Theorem 2. *The merge function M can be applied cumulatively (Fig. 18). That is, if T_A, T_B are derived from base T_O, and T'_A is derived from T_A, and edits are* well-separated, *then $M(T_A, T'_A, M(T_O, T_A, T_B)) = M(T_O, T'_A, T_B)$.*

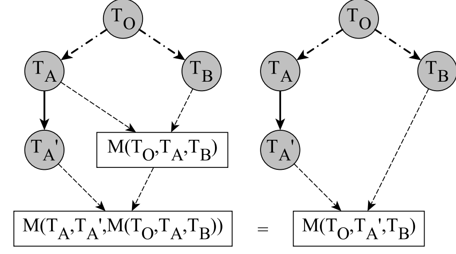

Figure 18: Cumulativity property

Theorem 3. *The merge function M is associative. That is, if T_A, T_B, T_C are derived from base T_O, and edits are* well-separated, *then $M(T_O, M(T_O, T_A, T_B), T_C) = M(T_O, T_A, M(T_O, T_B, T_C))$.*

In practice, these theorems mean that when editing different code regions, the merge function provides particularly strong guarantees. The proofs are in appendix A.

$$
\begin{aligned}
[\alpha\ \overline{\alpha}]_p &= [\alpha]_p^{ids([\overline{\alpha}]_p)} \cup [\overline{\alpha}]_p \\
[]_p &= \{\} \\
a(n, k+1) &= \{(n \cdot k, n, ids(a(n,k)), \varnothing)\} \cup a(n,k) \\
a(n, 0) &= \{\}
\end{aligned}
$$

$$
\begin{aligned}
&[n : \texttt{function}\ cseq(\overline{param})\ \texttt{do}\ \overline{stmt}]_p^S = \\
&\quad \{(n, p, S, \langle name : cseq, category : \texttt{function}\rangle)\} \cup \\
&\quad a(n,2) \cup [\overline{param}]_{n\cdot 0} \cup [\overline{stmt}]_{n\cdot 1} \\
&[n : cseq\ :\ \tau]_p^S = \\
&\quad \{(n, p, S, \langle name : cseq, type : \tau, category : \texttt{param}\rangle)\} \\
&[n : \overline{token}]_p^S = \\
&\quad \{(n, p, S, \langle category : \texttt{expr}\rangle)\} \cup a(n,1) \cup [\overline{token}]_{n\cdot 0} \\
&[n : \texttt{var}\ \ cseq := \overline{token}]_p^S = \\
&\quad \{(n, p, S, \langle name : cseq, category : \texttt{var}\rangle)\} \cup \\
&\quad a(n,1) \cup [\overline{token}]_{n\cdot 0} \\
&[n : //\ cseq]_p^S = \\
&\quad \{(n, p, S, \langle text : cseq, category : \texttt{comment}\rangle)\} \\
&[n : \texttt{if}\ \overline{token}\ \texttt{then}\ \overline{stmt_0}\ \texttt{else}\ \overline{stmt_1}]_p^S = \\
&\quad \{(n, p, S, \langle category : \texttt{if}\rangle)\} \cup \\
&\quad a(n,3) \cup [\overline{token}]_{n\cdot 0} \cup [\overline{stmt_0}]_{n\cdot 1} \cup [\overline{stmt_1}]_{n\cdot 2} \\
&[n : \texttt{for}\ \texttt{var}\ \ cseq\ \texttt{below}\ \overline{token}\ \texttt{do}\ \overline{stmt}]_p^S = \\
&\quad \{(n, p, S, \langle name : cseq, category : \texttt{for}\rangle)\} \cup \\
&\quad a(n,2) \cup [\overline{token}]_{n\cdot 0} \cup [\overline{stmt}]_{n\cdot 1}
\end{aligned}
$$

Figure 19: From HST to a tree

3.7 Translation of HSTs to trees

To conclude this section, we refer to Figure 19 to describe how the HST we defined in Section 2 is translated into the tree representation that we used for defining diffTree.

The function $[\cdot]_p^S$, defined in Figure 19, converts a node from the HST, with parent p and successor set S to a set of tree nodes. The function $[\overline{\alpha}]_p$ where α is any syntactic category converts a sequence of HST nodes with parent p into a set of tree nodes.

Whenever an HST node n has children which are sequences, we construct artificial child nodes using the $a(n,k)$ function, and make the particular nodes in the sequences children of these. The IDs of these artificial children are constructed using the function $n \cdot k$, which needs to be injective and return identifiers not occurring in any of the merged programs (in the implementation, this is easy to achieve by using suffixes to identifiers). Note that the IDs of the artificial children are deterministic and will thus match in all three programs involved in the merge.

The alternative to using deterministic IDs would be storing the IDs in programs. However, they would carry no information about user intention, since there is no way to e.g. move the "else" block around without moving the entire "if".

4. Merge-based Collaborative Editing

To allow users to edit code even when offline, and to keep the editor responsive even when the network is slow, it is important to maintain local copies of the HST on all participant devices. We can then propagate changes asynchronously, meaning that the user can always continue editing without ever having to wait for network responses. As long as we resolve conflicts deterministically, *eventual consistency* is achieved using the approach we present.

In general, implementing protocols for eventual consistency can be challenging when update operations are not idempotent or commutative. However, the existence of a 3-way merge function solves this problem in our context, and allows us to use a wide range of protocols, from simple to sophisticated. We describe two different solutions for implementing collaborative editing using the 3-way merge.

4.1 Cloud Storage Solution

A simple solution is to store a main version S in cloud storage, as well as two versions on each device: the last known server version O and the current client version A. Initially, $S = O = A$. Clients can read and update A, and can try to write changes back to the server (as often as desired and as the connection permits):

$$\text{if } O = S \text{ then } (S, O) \leftarrow (A, A)$$

To avoid losing changes, this writeback is conditional, that is, succeeds only if the server version has not changed in the meantime (most storage APIs support conditional updates using a so-called *e-tag*). If the server version has changed, the client must first fetch the latest server state and merge its changes using the merge function, as in

$$(O, A) \leftarrow (S, M(O, A, S)),$$

Shared Log	
state(T)	$M_0 = T$
delta(O_1,A_1)	$M_1 = M(O_1,A_1,M_0)$
delta(O_2,A_2)	$M_2 = M(O_2,A_2,M_1)$
...	...
delta(O_n,A_n)	$M_n = M(O_n,A_n,M_{n-1})$

Figure 20: Shared Log, represented as a single-column table. The first row contains the state, followed by delta rows.

after which it can retry the writeback. This process can be fully automatic and happen in the background, since our merge operation always succeeds.

This simple scheme works correctly and supports both online and offline collaboration. Its drawbacks are limited throughput for concurrent writes, and the fact that updates are not quickly pushed to all connected clients.

4.2 Shared Log Solution

We can use the existing *cloud types* [3] programming model and implementation, which provides a reliable protocol for keeping replicas in sync with the cloud storage, and uses websocket connections so servers can push changes to the clients quickly. Moreover, since it supports convenient declarations of eventually consistent data consisting of cloud variables, cloud tables, and cloud indexes, it allowed us to easily implement additional functionality such as displaying the current editing location of all participants, a list of users currently connected, and a shared chat window.

We found that the *cloud table* abstraction is expressive enough to implement an eventually consistent HST. The cloud table data type allows clients to append rows at the end of the table, and to update/delete (but not insert) rows anywhere in the table. Also, clients can query the *stable prefix* of the table, within which no more new rows can appear. Thus, we can use cloud tables to implement a "shared log" of updates, as described in the following section.

4.2.1 Implementing a Shared Log

To encode collaborative edits using HSTs, we use a cloud table with a single column,[2] representing a state followed by deltas as depicted in Figure 20. The first row contains an entry of the form state(T), and all the following rows contain entries of the form delta(O_i,A_i). Each delta represents a change as a pair of HSTs, a pre- and a post-state. The table as shown in Figure 20 represents the state M_n defined recursively by $M_i = M(O_i, A_i, M_{i-1})$ and $M_0 = T$.

Using this encoding, we can implement collaborative editing with desired semantics: clients simply push updates by appending delta rows. Whenever the cloud table changes

[2] In our implementation, we actually use four columns to encode entries together with additional instrumentation, but the algorithm is simpler to explain using a single column.

(the cloud table API includes notifications for this), clients recompute the current state and update the GUI accordingly.

Consider the example in Figure 21. If these edits are well-separated, the cumulativity and associativity properties of the merge give us the semantics shown in 22.

state(T_1)	$M_0 = T_1$
delta(T_1, T_3)	$M_1 = M(T_1, T_3, M_0)$
delta(T_1, T_2)	$M_2 = M(T_1, T_2, M_1)$
delta(T_3, T_5)	$M_3 = M(T_3, T_5, M_2)$
delta(T_2, T_4)	$M_4 = M(T_2, T_4, M_3)$

Figure 21: Example shared log

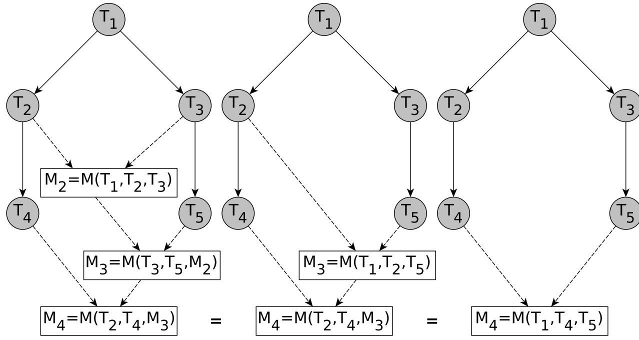

Figure 22: Shared log semantics for well-separated edits

4.2.2 Reducing the Log

The shared log algorithm cannot perform satisfactorily as the number of rows continues to grow. To address this issue, clients reduce the log whenever one or more deltas have become part of the confirmed prefix (as reported by the cloud table API). Specifically, as shown in Fig. 23, they (1) delete all rows in the confirmed prefix but the last, and (2) update that row to contain the corresponding state. Thus, the log only retains deltas that are not stable yet.

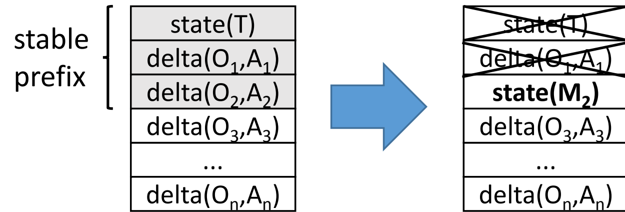

Figure 23: Clients reduce the log by deleting/updating rows.

Note that multiple clients may be concurrently reading the log, and concurrently reducing various prefixes of the log. However, this does not create problems because (1) the effects (deletion of rows and updating of rows) become visible atomically because of the causally consistent transactions in the cloud types model, and (2) deletion of a row is idempotent, and commutes with updates to that row, and (3) when a row is updated multiple times, the last update wins.

4.3 Local *Undo*

This log-based model has practical implications for other common editing scenarios besides simple merging. In a collaborative-editing context where we are editing code alongside other (remote) users, special care must be taken when implementing an *undo* operation. Specifically, each time user A performs an *undo*, she should see *only her own local edits* being reverted, and this should have no effect on changes merged in from remote users. For example, user A performing *undo* of edit T'_A in Figure 24 (left side) should return to the state $M(T_O, T_A, T_B)$ (shown in the right side).

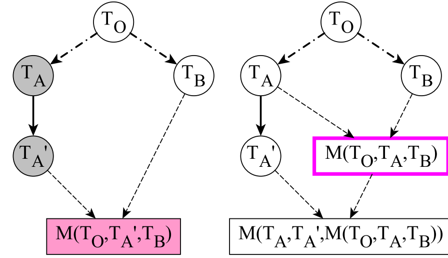

Figure 24: Undo (local + remote edits)

We propose a *local undo* operation implemented via the merge function. As with regular undo, a local history is saved as the user edits, but instead of simply reverting to a saved history item, our *undo* operation pushes an *inverse delta* of the previous local edit. That is, if the user previously made a change from T_A to T'_A, then we push delta(T'_A, T_A).

When collaborators confine their edits to separate regions of the code, this approach works well. We leave for future work investigation of a fully-robust local undo which behaves intuitively even in the case of conflicting edits.

5. Implementation

The implementation described in this section has been deployed in TouchDevelop, and is available for use in the web app. Additionally, TouchDevelop has been released as an open-source project on GitHub,[3] so it is possible to download/build the project, and run the TouchDevelop app locally.

5.1 The diffTree Merge Algorithm

We have implemented the merge algorithm in about 1500 lines of TypeScript. This functionality accepts three TouchDevelop HSTs T_O, T_A, T_B and produces a merged HST T_M.

Before connecting this with the user-facing TouchDevelop functionality, we needed to make several large modifications to TouchDevelop's TypeScript codebase. We first restructured the TouchDevelop HST to make it more straightforward, and added support in HST nodes for node-IDs and multiple base-script IDs. We also made modifications to the *cut/copy* operations in the TouchDevelop Editor, allowing tracking of script/node ID for cut/copied HST nodes.

[3] https://github.com/Microsoft/TouchDevelop

5.2 User interface

The merge function is used in several contexts. First, we synchronize locally-stored data (our cache) with cloud-stored data. It could be that a script has been modified both locally (user modified the script while offline) and remotely (user modified the script using another device). In this situation, we transparently call the merge function so that the locally-stored script incorporates modifications from the cloud, and at the next synchronization, we push the merged script to the cloud. We refer to this as the *cloud storage solution*.

Secondly, we use the merge function in the context of *pull-requests*, i.e. when a user forks a script and asks the original author to integrate her modifications back. To support this, we added a button in the interface which allows the user to manually trigger a call to the merge function. This operation requires computing the least-common-ancestor to determine the base script T_O required for the merge.

Finally, we use the merge function for real-time collaborative editing. In this scenario, multiple users edit the same script at the same time. This is the *shared log solution* we described earlier. This required a significant amount of implementation work to connect the collaboration features with the existing editor in TouchDevelop (see the video in §2).

5.3 Real-time Collaboration

Once the shared log solution was implemented, we enriched it with extra information. We now leverage unique node IDs to record *presence information*: whenever a statement with a given node ID is edited, we push that information to the cloud, allowing other users to notice that someone is editing that statement, and thus helping to coordinate edits. The information is conveyed using a small avatar beside the statement. In practice, we use a diff algorithm to make sure we *move* the DOM nodes for the avatars, rather than add/delete them, in order to eliminate visual flickering. The log reduction for presence information consists of pruning entries that are considered stale—in our case, users who have not reported presence information for the past ten minutes.

We also leverage the shared log to implement a chat interface, which allows users to talk about the code being edited. The log reduction mechanism for this chat feature consists of deleting messages more than ten minutes old.

A few other implementation tweaks were required to enhance the user experience. For example, whenever the user is editing a token (say, a text field), we temporarily disable pulling changes from the cloud—otherwise another user's edit might cause the field to disappear while the user is busy editing it. We also make it possible for the user to manually disable pulling in changes from the cloud. This is useful for postponing any interruptions due to collaborative merging while editing a large piece of code.

6. Preliminary Evaluation

Our collaborative-editing functionality has only recently been deployed, so we do not yet have a large selection of real-world merge benchmarks to use for testing. We continue to collect data on all merges, so as more users experiment with these features, we will be able to build up a more extensive benchmark suite, and identify any performance/usability issues that may be affecting users.

That said, currently we have observed about 50K merges. Of these, 1175 are nontrivial, i.e. $M(T_O, T_A, T_B)$ where $T_O \neq T_A$ and $T_O \neq T_B$. In this Evaluation, we examine merge performance on this nontrivial dataset.

For the experiments, we used a 64-bit Ubuntu workstation with 20 GB RAM and a quad-core Intel i5-4570 CPU (3.2 GHz). We built a test harness for use with the NodeJS JavaScript engine to perform each of the merges using our merge functionality. In order to evaluate our tool's performance relative to other merge tools, we have an interface to both the *node-diff3* JavaScript diff3-based merge,[4] and the Linux diff3 command-line program.

Figure 25 shows the performance of these tools on the dataset, as program size (number of lines) increases:

[4] https://www.npmjs.com/package/node-diff3

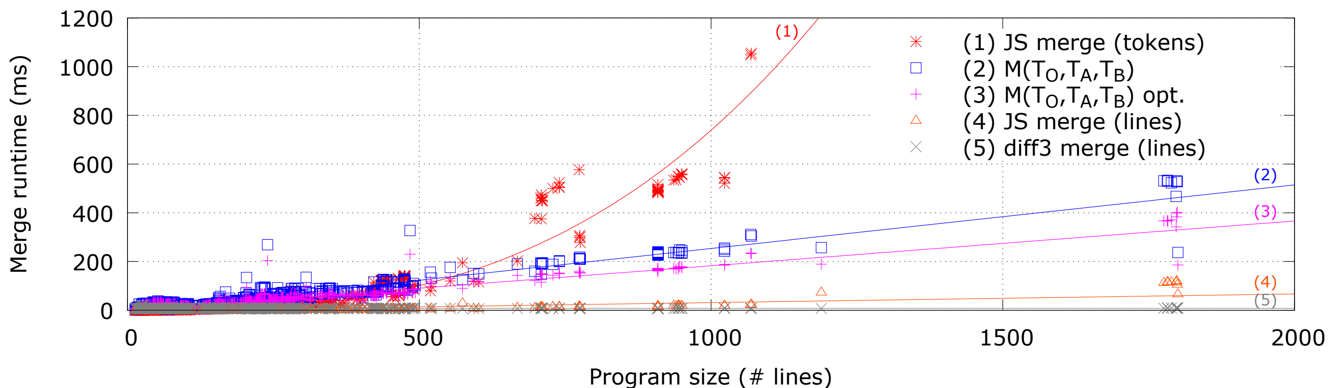

Figure 25: Merge Performance on $M(T_O, T_A, T_B)$

1. *JS merge (tokens)* – diff3-based JavaScript merge tool, operating on whitespace-separated tokens of the program text (avg. time 75.6ms, max 4253ms)
2. $M(T_O, T_A, T_B)$ – our un-optimized merge algorithm operating on the trees (avg. time 56.09ms, max 533ms)
3. $M(T_O, T_A, T_B)$ *opt.* – our merge algorithm with basic optimizations (avg. time 43.06ms, max 402ms)
4. *JS merge (lines)* – diff3-based JavaScript merge, operating on lines of program text (avg. 4.4ms, max 121ms)
5. *diff3 merge (lines)* – Linux diff3 program, operating on lines of the program text (avg. time 7.3ms, max 22ms)

The optimized version of our merge function features some basic adjustments for performance (numeric symbol table for node identifiers, etc.). Our merge performance falls between the two extremes of line-based merge and token-based merge, in the worst case incurring no more than 4-10x penalty versus the JavaScript diff3-based text merge, and never exceeding about 400ms of runtime.

The breakdown of runtime for our optimized version is shown in Figure 26. The runtime components are as follows:

- *initialize* – various startup, initializing data structures, hashing IDs, etc. (overall average 7.9ms, max 106ms)
- *TreeMerge* – total time spent in `TreeMerge` (overall average 6.5ms, max 68ms)
- *SeqMerge* – total time spent in `SeqMerge` (overall average 9.4ms, max 107ms)
- *build HST* – recursively construct final result object from the parent map (overall average 19.3ms, max 158ms)

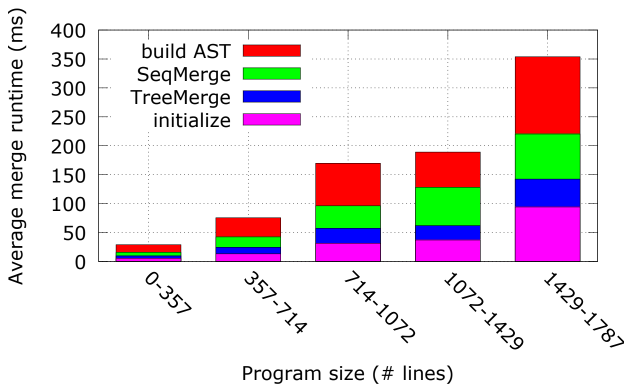

Figure 26: $M(T_O, T_A, T_B)$ runtime breakdown (optimized)

Notice that the core merge functionality (Tree/Seq-Merge) typically accounts for less than half of the runtime.

The implementation is by no means fully-optimized. We expect that we can reduce these times significantly as demand for higher performance increases, by pursuing more aggressive optimizations (see Future Work).

7. Related Work

Much work has been done by the research community in terms of merge algorithms. The classic text-level merge [7] is widely used in version-control systems, etc. This is generally fast, but suffers from inability to understand the source code that is being merged, often resulting in confusing conflicts that must be fixed manually by the user(s), with the risk of a malformed merged result.

Syntactic [1] and semantic [10] merge approaches take into account *structure* and *meaning* of the merged programs respectively. This can improve quality of the merge, making sure that the result program is well-formed in certain ways, but can only do so at the expense of time. Additionally, as seen in [6] [13], many of these approaches do not have an ability to recover from merge conflicts, and we need automatic conflict resolution functionality to enable real-time collaboration.

Ensuring eventual consistency across all collaborating devices is another area of related work. One existing approach uses Operational Transformations (OT) [4], which works by transforming operations (add, insert, delete, etc.) based on the order in which they end up executing. This is the approach used by Google for their *Docs* and *Wave* products, but has shown to be extremely complicated to implement/maintain, with a former Google engineer indicating that it took two years to implement OT properly [16].

Additionally, errors have been found in published OT algorithms [12]. Motivated by these difficulties of the OT approach, others have proposed a cleaner approach, Commutative Replicated Data Types (CRDT) [14]. However, this places strong requirements on the datatype operations, namely *commutativity*, which would make our automatic conflict resolution impossible. Instead, we use the similarly-clean Cloud Types [3] eventual consistency model, which does not place this restriction on our merge algorithm.

8. Conclusion and Future Work

We have presented a framework for collaborative editing of source code within a structured online editor. Our approach is based on a conflict-free HST-based merge function, which is used in conjunction with the Cloud Types eventual consistency model to enable both standard branch/merge version control, and real-time collaborative editing. An implementation which allows standard branching/merging, as well as functionality which enables real-time collaboration has been added to the TouchDevelop online programming environment, as shown in our video demonstration.

In the future, we will work on expanding and fully evaluating the multi-user collaborative editing functionality in TouchDevelop. This will likely involve many optimizations. Currently the merge algorithm works on monolithic scripts, so we are interested in allowing the merge/editor to handle small changes *locally*. Additionally, we need to reduce the

communication overhead by compressing the HST diffs that are sent over the network as users collaborate.

Another interesting area of research would be *Longitudinal Program Analysis* in our collaborative editing context. It has been proposed that program analysis approaches take into consideration the full version-history of a program, rather than just individual snapshots [11], and some research has moved in this direction [8] [9], but they focus on analyzing a single diff, i.e. using information from the analysis of program P to analyze modified program P'. It would be interesting to investigate doing this in general, using version history and our carefully-specified merge algorithm.

In summary, we present a new approach to realtime collaborative software development using a tree-merge algorithm. This provides immediately-usable functionality for online development environments, and sets the stage for future research in the area of merge-based collaboration.

References

[1] S. Apel, O. Leßenich, and C. Lengauer. Structured Merge with Auto-tuning: Balancing Precision and Performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 120–129, New York, NY, USA, 2012. ACM.

[2] T. Ball, S. Burckhardt, J. de Halleux, M. Moskal, J. Protzenko, and N. Tillmann. Beyond Open Source: The TouchDevelop cloud-based integrated development and runtime environment. In *Proceedings of Second International Conference on Mobile Software Engineering and Systems*, MOBILESoft 2015, 2015. To appear.

[3] S. Burckhardt, M. Fhndrich, D. Leijen, and B. Wood. Cloud types for eventual consistency. In J. Noble, editor, *ECOOP 2012 Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 283–307. Springer Berlin Heidelberg, 2012.

[4] C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 399–407, New York, NY, USA, 1989. ACM.

[5] A. Harzl, V. Krnjic, F. Schreiner, and W. Slany. Comparing purely visual with hybrid visual/textual manipulation of complex formula on smartphones. In *DMS*, pages 198–201, 2013.

[6] S. Horwitz, J. Prins, and T. Reps. Integrating Noninterfering Versions of Programs. *ACM Trans. Program. Lang. Syst.*, 11 (3):345–387, July 1989.

[7] S. Khanna, K. Kunal, and B. Pierce. A Formal Investigation of Diff3. In V. Arvind and S. Prasad, editors, *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, volume 4855 of *Lecture Notes in Computer Science*, pages 485–496. Springer Berlin Heidelberg, 2007.

[8] S. K. Lahiri, K. Vaswani, and T. Hoare. Differential static analysis: Opportunities, applications, and challenges. In *2010 FSE/SDP Workshop on the Future of Software Engineering Research (Position paper)*. Association for Computing Machinery, Inc., Nov. 2010.

[9] F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear. Verification modulo versions: Towards usable verification. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 294–304, New York, NY, USA, 2014. ACM.

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

[11] D. Notkin. Longitudinal program analysis. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '02, pages 1–1, New York, NY, USA, 2002. ACM.

[12] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia. A Commutative Replicated Data Type for Cooperative Editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 395–403, Washington, DC, USA, 2009. IEEE Computer Society.

[13] T. Reps. Algebraic properties of program integration. *Science of Computer Programming*, 17:139–215, 1991.

[14] M. Shapiro, N. Preguia, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Dfago, F. Petit, and V. Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer Berlin Heidelberg, 2011.

[15] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich. TouchDevelop: programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, ONWARD '11, pages 49–60, 2011.

[16] Wikipedia. Operational transformation, Mar. 2015. URL http://en.wikipedia.org/wiki/Operational_transformation#Critique_of_OT.

A. Merge Algorithm Correctness Proofs

A.1 Precise Definitions for Tree Edits

Defining edits. Conceptually, an "edit" is a single change to the program being modified. We model this by defining an *edit* to be a named tuple having one of the following forms:

- $\texttt{Add}(Pr, p, k)$: add a new node with properties $Pr \in \mathbb{N} \to String$ as a child of $p \in NodeId \cup \{\varepsilon\}$ at index $k \in \mathbb{N}$. The new node's ID is *unique*, i.e. not previously appearing.
- $\texttt{Del}(n)$: delete node having index $n \in NodeId$
- $\texttt{Move}(n, p, k)$: make node $n \in NodeId$ a child of node $p \in NodeId \cup \{\varepsilon\}$, inserting it at position $k \in \mathbb{N}$
- $\texttt{Change}(n, prop, v)$: for node $n \in NodeId$, set the property $prop \in \mathbb{N}$ to value $v \in String$

Applying edits to the AST. When applied to a tree T, an edit produces a new tree T'. We model this by defining the function $apply : \mathcal{T} \times Edits \to \mathcal{T}$ as follows:

$$
\begin{aligned}
&apply(T, \texttt{Add}((n, Pr), p, k)) = \\
&\qquad (T - \{(c, p, *, *) : c \in children(p, T)\}) \cup \\
&\qquad \{(n, p', \{c_i : c_i \in children(p, T) \wedge i \geq k\}, Pr)\} \cup \\
&\qquad \{(c_i, p, S', Pr) : c_i \in children(p, T) \wedge (c_i, p, S, Pr) \in T \\
&\qquad \wedge (i < k \Rightarrow S' = insert(n, S, k)) \wedge (i \geq k \Rightarrow S' = S)\}
\end{aligned}
$$

$$
\begin{aligned}
&apply(T, \texttt{Del}(n)) = \\
&T - \{(c, p, *, *) : c = n \vee p \in \{n\} \cup descendants(n, T)\}
\end{aligned}
$$

$$
\begin{aligned}
&apply(T, \texttt{Move}(n, p, k)) = \\
&\qquad apply(apply(T, \texttt{Del}(n)), \\
&\qquad \texttt{Add}((n, incProps(n, T, p, k)), p, k))
\end{aligned}
$$

$$
\begin{aligned}
&apply(T, \texttt{Change}(n, prop, v)) = \\
&\qquad \{(m, p, S, Pr') : (m, p, S, Pr) \in T \\
&\qquad \wedge Pr' = update(Pr, prop, v)\}
\end{aligned}
$$

Function $incProps(n, T, p, k)$ returns current properties Pr of node n in T, incrementing special properties $treeVer$ if $par(n, T) \neq p$, and $seqVer$ if $index(n, T) \neq k$. These are for keeping track of whether a node has been relocated in the tree, or reordered among siblings.

For simplicity, we have omitted these special properties from our description of the Tree Merge and Sequence Merge algorithms, but it is important to note the effect they have.

- In **`TreeMerge`**, a check such as $par(x, T_A) = par(x, T_B)$ is actually implemented as $(par(x, T_A), treeVer(x, T_A)) = (par(x, T_B), treeVer(x, T_B))$.
- In **`SeqMerge`**, a check such as $(x \to y) \in T_A \cap T_B$ is actually implemented as $seqVer(x, T_A) = seqVer(x, T_B) \wedge seqVer(y, T_A) = seqVer(y, T_B) \wedge (x \to y) \in T_A \cap T_B$.

Version history. If $T' = apply(T, e)$ for edit e, we call T' a *revision* of T, and use the notation $T \to T'$ (solid arrow). If T^k is derived from T via a series of revisions, we call T^k a *descendant* of T, denoted $T \dashrightarrow T^k$ (dashed arrow). Given a T, T^k where $T \dashrightarrow T^k$, we can compute the corresponding sequence of edits, and we denote this $edits(T, T^k)$.

We can define an "is changed after" relation $\leqslant$ on nodes in revisions, e.g. $(x, T) \leqslant (x, T')$ iff T' is a revision of T and some aspect of x differs between T and T'. The $treeVer$ and $seqVer$ properties discussed in the previous section are used to ensure that the relation $\leqslant$ transitive, i.e. $(x, T) \leqslant (x, T') \wedge (x, T') \leqslant (x, T'') \Rightarrow (x, T) \leqslant (x, T'')$. That is, if T' changes x from T, and then T'' changes x from T', then some aspect of x will differ between T and T''.

Conflicting edits. Given a set E of edits, we refer to the set as *conflicting* if any of the following hold:

- *Ancestry* conflict – a (sub)set of edits places cyclic parent dependencies on a node.
- *Delete* conflict – an edit deletes a node x while another edit moves a node y into or out of the subtree of x.
- *Ordering* conflict – if an edit reverses the order of some x, y (e.g. $x \to y$ changes to $y \to x$), we call this a *reordering*, and we refer to any node involved in a reordering as *unstable*. An ordering conflict results when a (sub)set of reorderings places cyclic order requirements on a node.
- *Add* conflict – (1) a node is added adjacent to an unstable node, or (2) multiple edits add nodes with identical IDs.

A *well-separated* set of edits is one exhibiting no conflicts. Abusing terminology slightly, if $T_1, \cdots, T_m$ are descendants of common base T_O, we say that the T_i are well-separated with respect to T_O if $\bigcup_i edits(T_O, T_i)$ is well-separated.

A.2 Set-Merge Properties

Lemma 6. *The set operation in Merge Step #1 has basic identity properties, i.e. for all o, a, b:*

$$
\begin{aligned}
f(o, o, o) &= o \\
f(o, a, o) &= a \\
f(o, o, b) &= b
\end{aligned}
$$

Proof. Looking at the definition of f, we see that

$$
\begin{aligned}
f(o, o, o) &= ((o \cup o) - o) \cup (o \cap o \cap o) \\
&= (o - o) \cup (o \cap o) \\
&= o \\
f(o, a, o) &= ((a \cup o) - o) \cup (a \cap o \cap o) \\
&= (a - o) \cup (a \cap o) \\
&= a \\
f(o, o, b) &= ((o \cup b) - o) \cup (o \cap b \cap o) \\
&= (b - o) \cup (o \cap b) \\
&= b
\end{aligned}
$$

□

Lemma 7. *The set operation in Merge Step #1 is associative, i.e.* $f(o, a, f(o, b, c)) = f(o, f(o, a, b), c)$ *for any* o, a, b, c.

Proof. Expanding the definition of f, we need to show

$$((a \cup f(o,b,c)) - o) \cup (a \cap f(o,b,c) \cap o) \\ = ((f(o,a,b) \cup c) - o) \cup (f(o,a,b) \cap c \cap o).$$

But looking at Figure 4, we can see that $f(o,a,b) \cap o = (a \cap b \cap o)$, and likewise, $f(o,b,c) \cap o = (b \cap c \cap o)$, meaning we just need to show

$$(a \cup f(o,b,c)) - o = (f(o,a,b) \cup c) - o.$$

First, note that $(x \cup y) - z = (x - z) \cup (y - z)$ for any sets x, y, z. Using this, we can show that $((a \cup f(o,b,c)) - o) =$

$$\begin{aligned} & (a - o) \cup (f(o,b,c) - o) \\ &= (a - o) \cup ((b \cup c) - o) \\ &= (a \cup b \cup c) - o \\ &= ((a \cup b) - o) \cup (c - o) \\ &= (f(o,a,b) - o) \cup (c - o) \\ &= (f(o,a,b) \cup c) - o \end{aligned}$$

This completes the proof. ☐

Lemma 8. *For well-separated edits, the* f *in Step #1 has the cumulative-editing property, in the sense that for all* o, a, a', b, *if* $a = ids(T_A)$ *and* $a' = ids(T'_A)$ *where* T'_A *is an* edit *of* T_A, *then* $f(a, a', f(o,a,b)) = f(o, a', b)$.

Proof. If $a = a'$, it is easy to confirm that this property is true. First note that $x = (x - a) \cup (a \cap x)$ for any x, a. Then we have $f(o, a', b) =$

$$\begin{aligned} &= f(o,a,b) \\ &= (f(o,a,b) - a) && \cup (a \cap f(o,a,b)) \\ &= ((a - a) \cup (f(o,a,b) - a)) && \cup (a \cap f(o,a,b)) \\ &= ((a \cup f(o,a,b)) - a) && \cup (a \cap f(o,a,b)) \\ &= ((a \cup f(o,a,b)) - a) && \cup (a \cap f(o,a,b) \cap a) \\ &= f(a, a, f(o,a,b)) \\ &= f(a, a', f(o,a,b)) \end{aligned}$$

Otherwise, we must recall the allowable ways in which an *edit* can change a into a'. Specifically, if $a \neq a'$, then either $a' \subseteq a$ (a node was deleted or moved), or $a' = a \cup \{x\}$ (a node was added).

We can use the definition of f to confirm that the solid-shaded area in the left side of Figure 27 graphically represents $f(o, a', b)$, and likewise the right side represents $f(a, a', f(o,a,b))$.

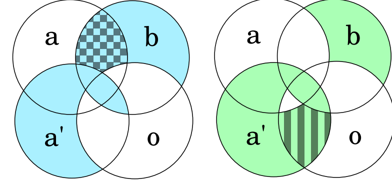

Figure 27: Left = $f(o, a', b)$, right = $f(a, a', f(o,a,b))$

First, note that well-separatedness makes it impossible for both A and B to add an identical node, so $((a \cap b) - o)$ is empty, meaning the cross-hatched subset $(a \cap b) - (a' \cup o)$ must also be empty. Similarly, once a node has been deleted by A, that node cannot reappear in future edits by A, so the striped area $(a' \cap o) - (a \cup b)$ must be empty since it is a subset of $(o - a)$, the nodes deleted by the previous edit.

We have shown that the left and right solid-shaded regions of Figure 27 are equal, completing the proof. ☐

A.3 Tree-Merge Properties

Lemma 1 (Properties of an arbitrary parent-graph cycle)**.** *Any cycle induced by the relation* par *satisfies the following:*

- *The cycle must be* simple*, since no node can have two outgoing edges (i.e. each node only has a single parent).*
- *A node within the cycle may have descendants outside. We call these "tails" of the cycle* descendant-subtrees.
- *For any node* x *in the cycle,* $(\{x\} \cup descendants(x, par))$ *contains no descendant of the root node* $\mathcal{R}$.

Proof. The first two items are clear, based on the knowledge that no node can have more than one parent (outgoing edge). The third is true because (1) the root node $\mathcal{R}$ cannot be part of the cycle, since the parent of the root node cannot be changed, and (2) a series of edges from any node in the descendants of the cycle to the root would cause one of the descendants to have multiple outgoing edges. ☐

Lemma 2 (Correctness of Tree Merge)**.** *The graph produced by the Tree Merge algorithm in Figure 5 is a* tree.

Proof. We first wish to show that the following property **P** is an invariant of the `makeTree` function:

P*: Every node is either (1) a descendant of the root in a cycle-free portion of the graph, or (2) contained in a disconnected cycle-containing component (Figure 6).*

Lemma 1 shows that this property is true before executing `makeTree`. When the DFS first encounters a strongly-connected component at a node n, there are two cases:

1. If the search arrived from a node p in the cycle-free portion of the graph, `makeTree` simply sets the parent of n to p (breaking the cycle), adds all descendants of n to the cycle-free portion, and removes the strongly-connected component from the set, denoting a success for that cycle. Since we're adding a cycle-free subtree to the cycle-free component, the invariant is preserved.

2. If the search arrived from a node p' within the cycle-containing component, simply doing the above can introduce a cycle after eliminating the original one. For example, in Figure 6 if the DFS arrives at node 4 via node 2, breaking the $4 \to 3$ edge and adding the $4 \to 2$ edge creates a new cycle $4, 2, 3, 9, 8, 7, 6, 5$. Thus, `makeTree` repeats the cycle-breaking up the DFS path until it adds an edge into the cycle-free component. The DFS path may cross k descendant-subtrees of the cycle (for example, the $1, 2$ and $12, 11, 10$ subtrees in Figure 6), but each time it does, the edge towards the cycle is broken. Thus, no point along the DFS path can participate in a cycle, meaning no cycles are introduced (invariant preserved).

□

Lemma 3 (Parent-child preservation of Tree Merge)**.** *The Tree Merge algorithm preserves the parent-child relationship for unchanged nodes, i.e. if p is the parent of x in both T_A and T_B, and p, x appear in the merged result T_M, then p is also the parent of x in T_M.*

Proof. If the parent of x is p in both T_A and T_B, the Tree Merge algorithm will clearly set $par(x) = p$ initially, and we only need to examine what happens to $par(x)$ during the `makeTree` routine. If x is in a cycle, the DFS cannot first arrive at x (it must necessarily arrive at p first). Similarly, if x is in a descendant-subtree of the cycle, any DFS arriving at x comes from p, which means that if the parent of x is reset, it can only be reset to p (i.e. no change). □

Lemma 9 (Tree Merge on well-separated edits)**.** *If T_A, T_B are well-separated with respect to T_O, then the parent graph is already cycle-free before the call to* `makeTree`*.*

Proof. The abscence of ancestry conflicts readily implies that the parent graph is cycle-free. □

Lemma 10. *The Tree Merge has basic identity properties, i.e. for all T_O, T_A, T_B:*

$$TreeMerge(T_O, T_O, T_O) = T_O$$
$$TreeMerge(T_O, T_A, T_O) = T_A$$
$$TreeMerge(T_O, T_O, T_B) = T_B$$

Proof. The first case is clearly true, since parents will only be taken from "A" parameter, which in this case is T_O. When $T_A \neq T_O$, it is easy to see that $TreeMerge(T_O, T_A, T_O)$ takes the order of all elements from T_A. Similarly, when $T_B \neq T_O$, we can see that $TreeMerge(T_O, T_O, T_B)$ takes the order of all elements from T_B. □

Lemma 11 (Cumulative-editing property of Tree Merge)**.** *The Tree Merge algorithm can be applied cumulatively, i.e. if T_A, T_B and T'_A, T_B are well-separated with respect to T_O and if T'_A is a revision of T_A, then*

$$TreeMerge(T_A, T'_A, TreeMerge(T_O, T_A, T_B)) = TreeMerge(T_O, T'_A, T_B).$$

Proof. Consider x such that $par(x) = \delta$ in $TreeMerge(T_O, T'_A, T_B)$. This means x was deleted either in T'_A or T_B (or both).

- If it was deleted in T_B, or it was missing in T'_A after being deleted in T_A, then it must also be deleted in $TreeMerge(T_O, T_A, T_B)$, meaning it must be deleted in $TreeMerge(*, *, TreeMerge(T_O, T_A, T_B))$.
- If it was deleted in T'_A and not in T_A, then it is also deleted $TreeMerge(T_A, T'_A, *)$.

Consider x such that $par(x) = \delta$ in $TreeMerge(T_A, T'_A, TreeMerge(T_O, T_A, T_B))$. Then x was either deleted in T'_A (and not in T_A), or deleted by $TreeMerge(T_O, T_A, T_B)$.

- If x was deleted in T'_A, then it is also deleted by $TreeMerge(T_O, T'_A, T_B)$.
- If x was deleted by $TreeMerge(T_O, T_A, T_B)$, then it must have been deleted by T_A (thus absent in T'_A) or T_B, meaning it is also deleted in $TreeMerge(T_O, T'_A, T_B)$.

In other words, the set of deleted nodes is the same for both sides of the equation. For a non-deleted node x, consider the following cases:

1. If $T_O = T_A$, then $TreeMerge(T_O, T_A, T_B)$ assigns the parent of x from T_B, so

$$TreeMerge(T_A, T'_A, TreeMerge(T_O, T_A, T_B)) = TreeMerge(T_O, T'_A, TreeMerge(T_O, T_A, T_B))$$

   has the same effect on x as $TreeMerge(T_O, T'_A, T_B)$.
2. If $T_O \neq T_A$, then $T_O \neq T'_A$ since T'_A is a revision of T_A. Thus, $TreeMerge(T_O, T'_A, T_B)$ assigns the parent from T'_A. Likewise, $TreeMerge(T_O, T_A, T_B)$ assigns the parent from T_A, so $TreeMerge(T_A, T'_A, TreeMerge(T_O, T_A, T_B))$ assigns the parent from T'_A.

We have shown that the two sides of the equation assign the parents of non-deleted nodes in the same way, and this completes the proof. □

Lemma 12 (Associativity of Tree Merge)**.** *The Tree Merge algorithm is associative, i.e. if T_A, T_B, T_C are well-separated with respect to T_O, then*

$$TreeMerge(T_O, TreeMerge(T_O, T_A, T_B), T_C) = TreeMerge(T_O, T_A, TreeMerge(T_O, T_B, T_C)).$$

Proof. It can be seen that a node x is deleted by $TreeMerge(T_O, TreeMerge(T_O, T_A, T_B), T_C)$

$\iff$ x is deleted by $TreeMerge(T_O, T_A, T_B)$ or T_C
$\iff$ x is deleted by T_A or T_B or T_C
$\iff$ x is deleted by T_A or $TreeMerge(T_O, T_B, T_C)$
$\iff$ x is deleted by $TreeMerge(T_O, T_A, TreeMerge(T_O, T_B, T_C))$.

For any non-deleted node x, consider the following cases:

1. If $T_O = T_A \wedge T_O = T_B$, then $TreeMerge(T_O, T_A, T_B)$ sets the parent of x from T_B, meaning the left-hand-side sets the parent of x from T_C. Likewise, $TreeMerge(T_O, T_B, T_C)$ sets the parent of x from T_C, so the right-hand-side sets the parent of x from T_C.
2. Similarly, if $T_O = T_A \wedge T_O \neq T_B$, then both sides set the parent of x from T_B.
3. If $T_O \neq T_A$, then both sides set the parent of x from T_A.

Thus, non-deleted nodes are processed identically by the left- and right-hand-side merges, completing the proof. □

A.4 Sequence-Merge Properties

Lemma 4 (Correctness of Sequence Merge)**.** *The function $SeqMerge(L_O, L_A, L_B)$ produces a total order on the common elements of L_O, L_A, L_B.*

Proof. We need to show that the relation R is a total order before the sort is called on line 30 in the algorithm in Figure 8. It is easy to see that for every pair of nodes x, y, the relation R contains either $x \rightarrow y$ or $y \rightarrow x$ (but not both). Thus, we only need to show that R is transitive.

We begin by showing that R must be acyclic. First, observing the conditional which allows the addition of B-edges, we note that the reverse of any B-edge is in L_A. Each time the loop on line 22 encounters a non-trivial strongly-connected component S (which contains a cycle), it reverses any contained B edges, meaning only edges from L_A are left within S. Since any subset of the L_A edges is acyclic, we only need to confirm that this edge-reversing operation does not introduce a cycle which extends outside S.

Consider the case where reversing B-edges in a single strongly-connected component introduces a cycle containing some node outside the component. For example, in Figure 28, reversing the $1 \rightarrow 3$ edge in $S1$ introduces a cycle containing node 2. Since $S1$ is a strongly-connected component, all nodes reach node 1 (and hence reach node 2), and all nodes are reachable from node 3 (which is reachable from node 2), meaning node 2 must be a part of $S1$ (contradiction). We reach the same contradiction for any length of continuous path extending out and back into a strongly-connected component.

The only other possibility is that reversing B-edges in multiple disjoint strongly-connected components introduces

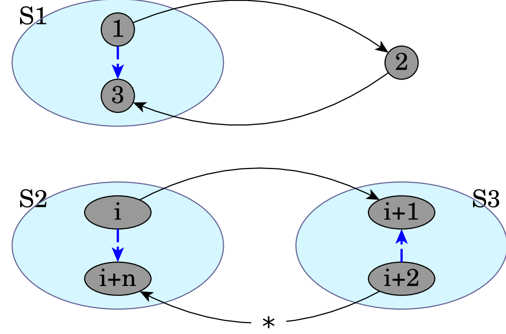

Figure 28: Strongly-connected components in `SeqMerge`

a cycle between the components, e.g. a cycle between $S2$ and $S3$ in Figure 28 which is introduced by reversing both the $(i \rightarrow i+n)$ and $(i+2 \rightarrow i+1)$ edges. Since the relation R contains an edge between every pair of nodes, either $(i+1 \rightarrow i+n)$ or $(i+n \rightarrow i+1)$ exists. In the former case, as happened with $S1$, this forces node $i+1$ to be contained in $S2$ (contradiction), and in the latter case, this forces $i+n$ to be contained in $S3$ (contradiction).

We have shown that R is acyclic. Thus, if we have $(x \rightarrow y) \in R$ and $(y \rightarrow z) \in R$, then $(z \rightarrow x) \notin R$, and since every pair of nodes is joined by some edge, we must have $(x \rightarrow z)$, so this establishes transitivity and completes the proof. □

Lemma 5 (Order preservation of Sequence Merge)**.** *The Sequence Merge algorithm preserves the order relationship for unchanged nodes, i.e. if x appears before y (i.e. $x \rightarrow y$) in both L_A and L_B, and x, y appear in the merged result L_M, then x also appears before y in L_M.*

Proof. If $(x \rightarrow y) \in L_A$ and $(x \rightarrow y) \in L_B$, then $(y \rightarrow x) \notin L_B$, so $(x \rightarrow y)$ will be added as an A-edge because of the check on line 8 of Figure 8. Since A-edges are left unchanged by the cycle-breaking mechanism, the ordering $(x \rightarrow y)$ will be respected in the resulting order. □

Lemma 13 (Sequence Merge on well-separated edits)**.** *If L_A, L_B are well-separated with respect to L_O, then the relation R is already cycle-free before the strongly-connected components operation in $SeqMerge(L_O, L_A, L_B)$.*

Proof. We begin by showing that if the relation R contains a cycle, it must contain a cycle of length three, consisting of a single B-edge and two A-edges. Consider an arbitrary cycle of length $k > 3$ in R, as shown in Figure 29.
The cycle must contain a B-edge (since L_A is cycle-free), e.g. the edge $2 \rightarrow 3$ in the figure. If the edge between nodes 4 and 2 is $4 \rightarrow 2$, then we have found a cycle of length three. Otherwise, the edge is $2 \rightarrow 4$, meaning there is a cycle of length $k-1$ (e.g. $1, 2, 4, 5, \cdots$ in the figure). We can repeat this process as many times as needed, eventually reaching a cycle of length three.

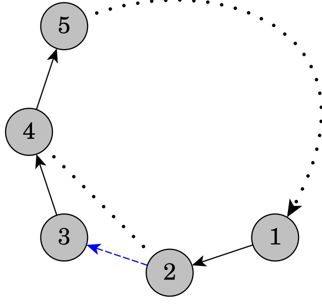

Figure 29: Cycle in the relation R

Call the nodes in this cycle x, y, z, with $z \to x$ as the B-edge. If one of the other edges is a B-edge, then we have a sequence of two B-edges. Without loss of generality, say the second B-edge is $x \to y$. The condition for adding B-edges shows that the existence of these two edges implies $(z \to y) \in L_B$ and $(y \to z) \in L_O \cap L_A$, but this means $(z \to y)$ must be a B-edge (contradicting the fact that $z \to x \to y \to z$ is a cycle). Thus, we have shown that the cycle of length three has one B-edge and two A-edges.

Now, assume L_A, L_B are well-separated with respect to L_O, and consider the R at line 19 of $SeqMerge(L_O, L_A, L_B)$. We wish to show that this R is cycle-free, so assume to the contrary that R has a cycle. As we have seen, this means we have a 3-cycle $z \overset{B}{\to} x \overset{A}{\to} y \overset{A}{\to} z$. Notice that $(x \to y) \notin L_O \cap L_A \cap L_B$ and $(y \to z) \notin L_O \cap L_A \cap L_B$, since otherwise there would be a cycle in L_B. Thus, either $x \to y$ or $y \to z$ differs in order between L_O and L_A. Since $z \to x$ is a B-edge, we already know that it differs in order between L_O and L_B, so we have an ordering conflict between L_A and L_B (contradiction). □

Lemma 14. *The Sequence Merge has basic identity properties, i.e. for all L_O, L_A, L_B:*

$$SeqMerge(L_O, L_O, L_O) = L_O$$
$$SeqMerge(L_O, L_A, L_O) = L_A$$
$$SeqMerge(L_O, L_O, L_B) = L_B$$

Proof. The first case is clearly true, since there are no conflicts, and all orderings will be taken from A, which in this case is L_O.

Now, consider a pair of nodes x, y. If their order differs between L_O and L_A, then $SeqMerge(L_O, L_A, L_O)$ will be taken from L_A. If their order is identical in L_O, L_A, the merge takes the order from L_O (which is the same as the order in L_A). Thus, the second property holds.

The third property is clearly true, since $SeqMerge(L_O, L_O, L_B)$ always takes the order from L_B. □

Lemma 15 (Cumulative-editing property of Sequence Merge)**.** *The Sequence Merge algorithm can be applied cumulatively, i.e. if L_A, L_B and L'_A, L_B are well-separated with respect to L_O and if L'_A is a revision of L_A, then*

$$SeqMerge(L_A, L'_A, SeqMerge(L_O, L_A, L_B)) = SeqMerge(L_O, L'_A, L_B).$$

Proof. Given arbitrary non-added x, y, we wish to show that both sides set the order identically. We consider two cases:

- L_A and L'_A agree on the order of x, y. Then $SeqMerge(L_A, L'_A, SeqMerge(L_O, L_A, L_B))$ will take the order from $SeqMerge(L_O, L_A, L_B)$, which in this case will be the same order as $SeqMerge(L_O, L'_A, L_B)$.
- L_A and L'_A disagree on the order of x, y. Since L'_A is a revision of L_A, it must be the case that L_O and L'_A also disagree on the order, so both sides of the above equation set the order identically.

Now we consider an *added* node α, and we wish to show that both sides insert α at the same position in the output. If $\alpha \in L_B$, well-separatedness guarantees that α will be inserted after the same non-added node in both $SeqMerge(L_O, L'_A, L_B)$ and $SeqMerge(L_O, L_A, L_B)$, and likewise for the left-hand side of the equality. Similarly, if $\alpha \in L'_A$ and $\alpha \notin L_A$, both sides of the equality position α identically.

Finally consider the case where $\alpha \in L_A \cap L'_A$. We know $SeqMerge(L_O, L_A, L_B)$ will contain the same set of nodes having edges to α as L_A contains, so the left-hand side of the equation will take the ordering for α from L'_A, resulting in the same positioning that the right-hand side assigns. □

Lemma 16 (Associativity of Sequence Merge)**.** *The Sequence Merge algorithm is associative, i.e. if L_A, L_B, L_C are well-separated with respect to L_O, then*

$$SeqMerge(L_O, SeqMerge(L_O, L_A, L_B), L_C) = SeqMerge(L_O, L_A, SeqMerge(L_O, L_B, L_C)).$$

Proof. Given arbitrary non-added x, y, we wish to show that both sides set the order identically. We consider two cases:

- L_O and L_A agree on the order of x, y. In this case, the right side takes the order from $SeqMerge(L_O, L_B, L_C)$, and so does the left side, since $SeqMerge(L_O, L_A, L_B)$ takes the order from L_B.
- L_O and L_A disagree on the order of x, y. In this case, the right side will take the order from L_A, and so will the left side, since $SeqMerge(L_O, L_A, L_B)$ takes the order from L_A.

For an *added* node α, if $\alpha \in L_B$, the well-separatedness ensures that $SeqMerge(L_O, L_A, L_B)$ and $SeqMerge(L_O, L_B, L_C)$ place α after the same non-added element. Similarly, if $\alpha \in L_A$, we see that α appears after the same non-added element in $SeqMerge(L_O, L_A, L_B)$ and L_A, and if $\alpha \in L_C$, α appears after the same non-added element in $SeqMerge(L_O, L_B, L_C)$ and L_C. □