Towards AST-based Collaborative Editing

(NOTE: decide order of authors! (for now, it is alphabetic))

Sebastian Burckhardt
Microsoft Research
sburckha@microsoft.com

Jedidiah McClurg
University of Colorado Boulder *
jedidiah.mcclurg@colorado.edu

Michał Moskal
Microsoft Research
michal.moskal@microsoft.com

Abstract

Handheld devices and cloud-connected applications are now commonplace. Even complex software development tasks are moving into the mobile arena, as exemplified by "on-the-go" code-editing with applications like Visual Studio Online. Further still, there is a trend towards *real-time* components in collaborative software development, with platforms like Google Docs and MS Office Online enabling limited forms of real-time collaborative document development.

However, these platforms are unattractive for collaborative *software* development since they don't recognize and maintain code structure. Additionally, these systems are built atop ad-hoc algorithms or incredibly complex/fragile techniques for ensuring eventual consistency of documents.

We address these problems by providing a clean framework for realtime collaborative editing of source code within an online structured editor. We first develop a conflict-free three-way merge algorithm for program ASTs, which behaves intuitively to developers, and preserves syntactic correctness. We then show how this merge functionality can be used in conjunction with the Cloud Types eventual consistency model to enable seamless realtime collaboration.

We have implemented our system in the TouchDevelop online programming environment. The UI now allows a user to select another user's code, and merge it into their current application (patch merge). We have also implemented basic realtime collaborative editing, allowing users to edit the same piece of code on multiple devices simultaneously.

* Work performed during an internship at Microsoft Research

[Copyright notice will appear here once 'preprint' option is removed.]

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. Distributed version control systems have become popular, enabling users to develop code locally and then synchronize with the work of other developers at some future point, and even the actual editing of one's local code can happen "on-the-go", due to applications like Visual Studio Online.

This is certainly not the end of the story. It is clear that there is a trend towards social-network and *real-time* components in collaborative software development. Platforms like Google Docs and Microsoft Office Online enable limited forms of development in real-time, allowing users to edit from anywhere, and quickly see changes and messages from other developers. However, the eventual consistency functionality underlying these platforms is sometimes complex and inflexible, and the way it behaves on the surface can be confusing to users, especially when trying to simultaneously edit structured documents, such as source code.

At the heart of collaborative source-code editing is a software merge problem. If we can quickly, accurately, and *automatically* merge source code in a way that developers find intuitive, we can use this functionality to continuously merge changes from other developers to maintain an evolving program text. The research community has provided several possible merge solutions. The classic text-level merge [Khanna et al. 2007] is fast, but suffers from inability to understand the source code that is being merged, and can therefore easily result in confusing conflicts that must be fixed manually. There are syntactic merge approaches [Apel et al. 2012] and semantic ones [Mens 2002], which can improve the quality of the merge, but only at the expense of time.

Once a merge function exists, the next question is how to ensure eventual consistency of the program text across all of the collaborating devices. Existing approaches use Operational Transformations (OT) [Ellis and Gibbs 1989], or Commutative Replicated Data Types (CRDT) [Preguica et al. 2009] [Shapiro et al. 2011]. However, these suffer from a difficult-to-understand programming model, or strong requirements on the datatype operations (or both).

In this paper, we develop a straightforward approach to collaborative editing which attempts to address these problems. Our techniques provide a source-code merge function that is efficient while still preserving the well-formed structure of the code, and we show that this function has properties which make it intuitive to the developers relying upon it. Using this merge function, we build on a recent eventual consistency model which avoids some of the difficulties of the aforementioned eventual consistency approaches.

Our merge function is a pseudo-syntactic (AST-based) three-way merge. We show that by modifying a structured online source-code editor, we can develop an efficient merge algorithm in this context by keeping track of AST node IDs as users are collaborating on the code. This is more useful than a purely syntactic merge, because nodes can be moved around in the program, copy/pasted from other programs, and brought in from previous versions, while still retaining their unique identities. Our merge is always automatic, i.e. conflicts do not cause it to abort, allowing for seamless collaboration. We show that the merge preserves some basic semantic information about merged programs, and possesses several properties which make it intuitive for developers and allow it to be applied continuously without changing the merged source code in unexpected ways.

We adapt the eventual consistency model of Cloud Types [Burckhardt et al. 2012], using our merge function as a log evaluation operator. Cloud Types provide a clean eventual consistency solution that is easy for developers to reason about, and does not place harsh restrictions on the data operations. We show how this approach can be used for collaborative editing. We also show how the merge function can be used within this context to provide an *undo* operation which works sensibly in a collaborative environment.

We have implemented our system in the TouchDevelop online programming environment [Burckhardt et al. 2013]. This environment already supported basic version history and branching, but not merging. The UI now allows a user to select another user's code, and merge it into their current application (patch merge). It also allows a user to create development branches which she/he can later merge back into a main branch. We have also implemented basic realtime collaborative editing, allowing users to edit the same piece of code on multiple devices simultaneously.

Our contributions can be summarized as follows:

- We show how to exploit a structured editor with version-history/branching to maintain mergeable ASTs (§3).
- We develop a general and conflict-free 3-way merge algorithm for ASTs, and show that the merge function has desirable properties (§4).
- Using the merge algorithm as a log evaluator, we build on Cloud Types to create an eventually-consistent AST datatype. We demonstrate the use of Cloud ASTs for realtime collaboration, and present correctness results for this technique (§5).
- We provide an implementation of this approach, enabling patch-merge and real-time collaboration functionality in a popular online programming environment (§6), and perform a preliminary evaluation of its usability (§7).

Overall, we make a definitive step towards enabling seamless real-time 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. Overview

In a cloud-based programming environment such as TouchDevelop, the editor assists the user in writing well-formed code, and the source code is stored in the cloud in a structured way. Namely, a script is stored in the form of an abstract syntax tree (AST). We can keep track of unique identifiers for each AST node, creating fresh IDs for new code, and propagating current IDs to forked versions of a script. For example, consider this TouchDevelop script and corresponding AST:

```
1   action main() do
2     var x := 10
3     while x ≥ 1 do
4       x→post to wall
5       ▷one(x)
6       x := x - 1
7     end while
8   end action

10  action one(p: Number)
11    p→post to wall
12  end action
```

Figure 1: Simple TouchDevelop script

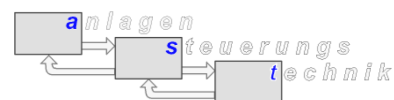


Figure 2: AST for script in Figure 1

This is a simple script that displays the numbers 1 through 10 in decreasing order, repeating each one twice. Let us now assume that two users A, B have forked this script, and they both find themselves wishing to print the factorial of each number. Both users do not wish to re-implement this functionality, so they browse the repository of TouchDevelop scripts, finding a user C who has a straightforward factorial-printing function `fact`. Users A, B select-copy the `fact` function, and paste into their fork, adding a line after 4 which

calls `fact(x)`. However, user A has pasted the function definition *before* line 1, and user B has pasted it *after* line 8.

If users A, B now attempted to merge their code using a Diff3-style line-based merge, or even a smarter syntax-based merge, they would now have a new script with *two* definitions of `fact`, one before and one after the `main` definition. While possibly syntactically correct, this new script is semantically incorrect, and causes a compile error.

We avoid situations like this by merging the ASTs based on the unique IDs of the nodes. In this case, we would recognize that the pasted `fact` functions have the same IDs, making them identical in both the users' forks, and would simply need to decide whether the `fact` function should be placed before or after `main`. In this case, either choice would be semantically correct, but in general, our merge needs to make sure the merged program respects define-before-use statement ordering.

We show that our merge preserves define-before-use and other basic well-formedness properties in the absence of conflicts (where users A, B both change an AST node from its base version in differing ways), and makes a sensible automatic choice in the case of a conflict. This presents the user with a useful and intuitive merge. We also show that in the absence of conflicts, the merge has an associativity property, and a "transitivity" property which equates the merging of two branches with the result of successively cherry-picking each commit from the two branches. These properties ensure that real-time collaboration behaves in a seamless way.

Our real-time collaboration functionality is based on the eventual consistency model of Cloud Types. Cloud Types use a (logically) centralized log of operations which represents the correct global state. Each device stores a local log containing their own local operations, and a *prefix* of the global log. As a device continues operating, it will periodically inform the centralized log server of its newly-performed operations, and possibly receive a longer prefix of the global log. In this way, the devices all progress towards the consistent global state (see Figure 3).

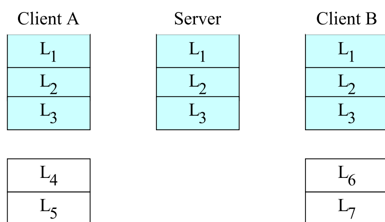


Figure 3: Cloud Types - Global Log of Update Transactions

This eventual consistency mechanism is known as Global Log of Update Transactions (GLUT). In our case, the logged operations are diffs of the AST, represented by pairs of before-after trees. The state seen by each device is represented by a list of these pairs, and the device produces a current view of the AST by "folding" over the local log using the merge function. As devices receive longer log prefixes, this view will change to bring in modifications from other users. We also implement a "local undo" operation by pushing a reverse diff onto the log. The merge causes this to cancel out the previous local edit.

Our work provides a preliminary framework for real-time collaborative source-code editing. Building all of these pieces into TouchDevelop, we have created a working prototype which can be used and tested by a large audience.

3. Language and Environment for Collaborative Editing

TouchDevelop is a web-based application development environment which allows participants to use a structured code editor (see Figure 4) 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 interactive forms, quizzes, games, etc.

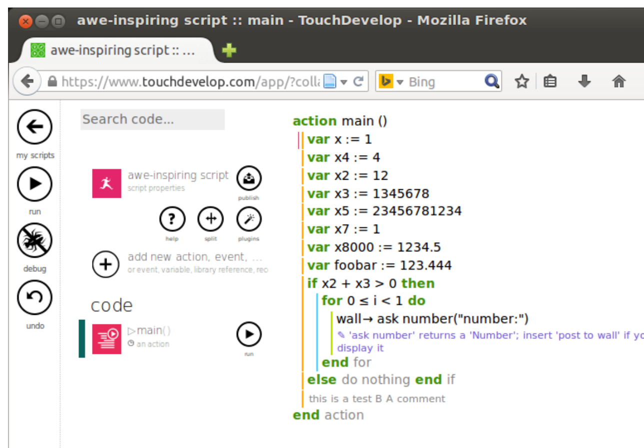


Figure 4: TouchDevelop structured code editor

This environment is cross-platform compatible, since it runs in the browser, and works with a wide range of browsers. Additionally, it allows users to interact by "cloning" (forking) scripts of other users. These features, in combination with the structured code-editor where users write their applications, make TouchDevelop an especially good place to investigate collaborative editing.

In Figure 5, we show the syntax of TouchDevelop programs. TouchDevelop makes a distinction between a *statement* (block-like code structure) and *expression* (single line containing tokens). Statements, which include declarations, loops, comments, etc. are always syntactically well-formed, while expressions such as `a + b + c` may not be.

A TouchDevelop program is stored in the cloud as text, and then parsed and loaded into the structured editor as an AST. Each node in the AST (and in the stored version of the program) has a globally-unique ID which is created on node

v	$\in$	Var	*(variable)*
τ	$\in$	$\{\texttt{Number}, \texttt{Boolean}, \texttt{String}\}$	*(type)*
n	$\in$	$\mathbb{R}$	*(number)*
b	$\in$	$\{\texttt{true}, \texttt{false}\}$	*(boolean)*
s	$\in$	$\{\text{“}c \cdots\text{”} : c \in Char\}$	*(string)*
id	$\in$	$Ident$	*(identifier)*
op	$\in$	$\{+, -, *, /, <, =, \texttt{and}, \texttt{not}\}$	*(operator)*
$prog$	$::=$	$(action \mid decl) \cdots$	*(program)*
$decl$	$::=$	$\texttt{var}\ x ::= expr$	*(declaration)*
$stmt$	$::=$	$decl \mid x ::= expr$ $\mid x \rightarrow id(v, \cdots) \mid \triangleright id(v, \cdots) \mid block$ $\mid comment \mid if \mid for \mid while$	*(statement)*
$expr$	$::=$	$op \mid n \mid b \mid s \mid v$ $\mid expr \cdots$	*(expression)*
$param$	$::=$	$v : \tau$	*(param)*
$block$	$::=$	$stmt \cdots$	*(block)*
$comment$	$::=$	$// \cdots$	*(comment)*
$action$	$::=$	$\texttt{action}\ id(param, \cdots)$ $\texttt{returns}(param, \cdots)\ \texttt{do}\ block$ $\texttt{end action}$	*(action)*
if	$::=$	$\texttt{if}\ expr\ \texttt{then}\ block\ \texttt{else}\ block$ $\texttt{end if}$	*(if)*
for	$::=$	$\texttt{for 0 <=}\ v < n\ \texttt{do}\ block$ $\texttt{end for}$	*(for)*
$while$	$::=$	$\texttt{while}\ expr\ \texttt{do}\ block$ $\texttt{end while}$	*(while)*

Figure 5: Basic TouchDevelop AST

creation, and maintained appropriately as the node is moved around within the script or cut/copied to other scripts.

4. AST-based Merge Functionality

Using this ID-enriched TouchDevelop AST, we can build an AST merge function, with the following basic requirements.

Merge requirements: Define a 3-valued merge function $M(T_O, T_A, T_B)$ where T_O is a common ancestor of T_A, T_B. The function returns a new tree T_M, and

1. Keeps invariants of T_A, T_B, i.e. merging *trees* should result in a tree, and merging *sequences* should result in a sequence.
2. Prefers change over no change.
3. Prefers A over B on conflict.

We represent an Abstract Syntax Tree T as a set of tuples of the form (n, p, S), where $n \in \mathbb{N}$ is the unique node identifier, $p \in \mathbb{N} \cup \{\varepsilon\}$ is the parent node (or ε if there is no parent), and S is the set of subsequent sibling nodes, i.e. nodes which are siblings of n and appear to the *right* of n in the AST. We denote the set of all ASTs by $\mathcal{T}$. We want a three-way merge function $M : \mathcal{T} \times \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T}$ such that $M(T_O, T_A, T_B)$ merges T_A, T_B using the base AST T_O, resulting in a new AST T_M. Let $ids : \mathcal{T} \rightarrow \mathbb{N}$ be a function that obtains all node IDs from an AST, i.e. $ids(T) = \{n : (n, p, S) \in T \text{ for some } p, S\}$. We use notation $(x \rightarrow y) \in T$ to indicate $\exists (x, p, S) \in T$ s.t. $y \in S$.

4.1 Merge Algorithm

The merge algorithm works by first computing the set of nodes that will appear in the merged program, taking into account nodes which should be added or deleted. Then, it computes the tree structure of the merged program. Finally, it orders siblings properly in the result tree.

1. ***Perform additions and deletions.*** Let $I_M = ((ids(T_A) \cup ids(T_B)) - ids(T_O)) \cup (ids(T_A) \cap ids(T_B) \cap ids(T_O))$. These are the nodes that will appear in T_M.
2. ***Determine parents (Tree Merge).*** We construct a function $par : I_M \rightarrow I_M$ which maps vertices to parents:
   (a) For all $x \in I_M$, if $(x, p_1) \in T_O \wedge (x, p_2) \in T_A$ where $p_1 \neq p_2$, then require $par(x) = p_2$, add x to V_L.
   (b) For all $x \in I_M$, if $(x, p_1) \in T_O \wedge (x, p_3) \in T_B$ where $p_1 \neq p_3 \wedge x \notin V_L$, then require $par(x) = p_3$.
   (c) Find the strongly-connected components of the graph represented by par. Do a DFS traversal of T_A, and upon first entering a non-trivial (size > 1) component via some vertex n, reset $par(n)$ to the parent of n in T_A.
3. ***Determine ordering (Sequence Merge).*** For each $p \in I_M$, we can compute the set of children $C = par^{-1}(p)$. We wish to produce an ordering $R(C)$ of the elements in C.
   (a) For $x, y \in C$ where $y \neq x$:
      - If $(x \rightarrow y) \in T_A$ and $((x \rightarrow y) \notin T_O$ or $(y \rightarrow x) \notin T_B)$, then add $(x \overset{A}{\rightarrow} y) \in R(C)$.

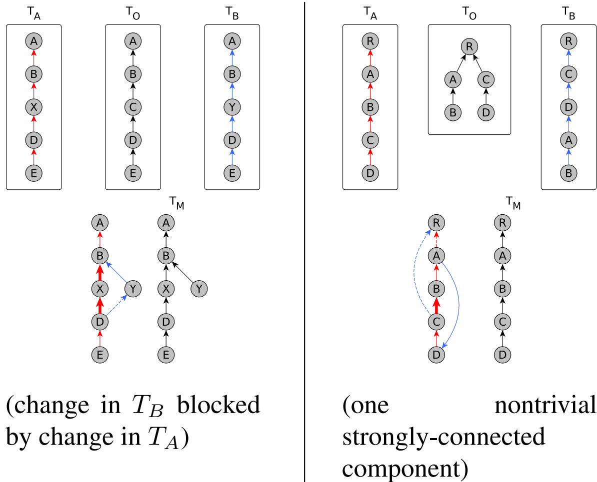
   (b) For $x, y \in C$ with $y \neq x \wedge (x, y) \notin R(C) \wedge (y, x) \notin R(C)$:
      - If $(x \rightarrow y) \in T_B$ and $(y \rightarrow x) \in T_O \cap T_A$, then add $(x \overset{B}{\rightarrow} y) \in R(C)$.

   (c) Compute strongly-connected components of $R(C)$, and replace any $(x \overset{B}{\rightarrow} y)$ edge occurring in a non-trivial (size > 1) component with $(y \overset{A}{\rightarrow} x)$.

Now, $T_M = \{(n, p, S) : n \in I_M, p = par(n), S = \{y \in I_M : (n, y) \in R(par^{-1}(p))\}\}$

4.2 Simple Tree Merge Examples

In this section, we present some simple examples of the Tree Merge portion of the algorithm. In Figure 6, the left side



(change in T_B blocked by change in T_A)

(one nontrivial strongly-connected component)

Figure 6: *Tree merge:* automatic conflict resolution

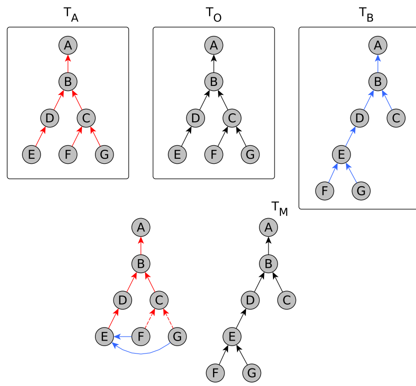


Figure 7: *Tree merge:* T_A unchanged–changes in T_B allowed

shows a conflicting edit where both users A and B try to set the parent of node D (conflict is resolved by choosing A's edit). The right side 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 must break the cycle deterministically by doing a DFS traversal based on user A's tree.

Figure 7 shows an edit from user B which does not conflict with user A's edits. Finally, Figure 8 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.

4.3 Simple Sequence Merge Examples

In this section, we show some examples of the Sequence Merge portion of the merge algorithm. In the left side of Figure 9, no users have made any edits, so the resulting sequence is unchanged. In the right side, 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

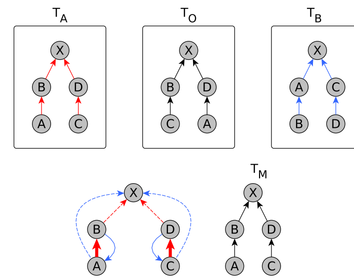
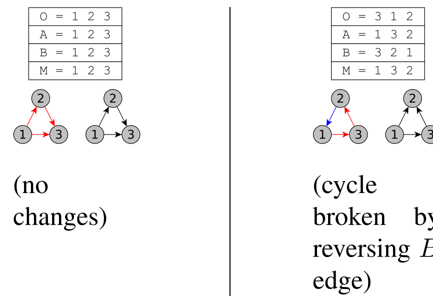


Figure 8: *Tree merge:* two nontrivial strongly-connected components



(no changes)

(cycle broken by reversing B edge)

Figure 9: *Sequence merge:* Non-conflicting and conflicting sequence edits

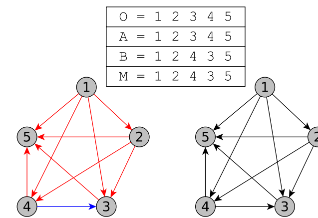


Figure 10: *Sequence merge:* change in B

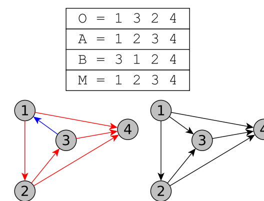


Figure 11: *Sequence merge:* cycle broken by reversing blue (B) edge

by reversing all blue edges (ones caused by user B) in the cycle. Figure 11 shows another example of breaking cycles. Figure 10 shows a non-conflicting change caused by user B.

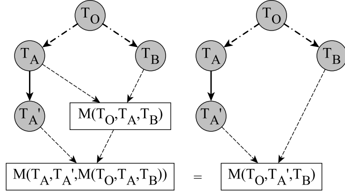
4.4 Properties of Merge Function

In this section, we briefly mention three important properties of the merge function.

Theorem 1. *The merge function has basic identity properties that one would expect from a sensible merge:*

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

Theorem 2. *The merge function M can be applied "transitively". 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 $M(T_A, T'_A, M(T_O, T_A, T_B)) = M(T_O, T'_A, T_B)$.*



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 $M(T_O, M(T_O, T_A, T_B), T_C) = M(T_O, T_A, M(T_O, T_B, T_C))$.*

As previously mentioned, we wish to use the merge function continuously to enable realtime collaborative editing. While these properties are not needed for *eventual consistency* of collaborative edits when using Cloud Types, as we will see in the next section, they ensure that users' edits are merged in a "sensible" way.

5. Merge-based Collaborative Editing

When multiple remote users are collaborating on a piece of code, we want to ensure that they will all *eventually* see the same AST, a requirement known as *eventual consistency*. As described in the Overview section, Cloud Types form a clean eventual consistency solution. Cloud Types have been used to allow a TouchDevelop program to store data in the cloud for synchronization across other running instances, but now we want the *program currently loaded into the editor* to be synchronized across other running instances. We do this by creating a new cloud type called Cloud AST.

Conceptually, Cloud Types uses the Revision Diagrams eventual consistency model. The actual implementation of this uses a log of operations on the data. As seen in Figure 3, a prefix of the log (shown below in blue) is known to be synchronized, and this prefix is expanded as clients communicate with the server. In the meantime, clients can temporarily see an inconsistent state. Clients compute their current "view" of a Cloud variable by reducing their local log using a function $collapse : L \times L \to L$.

5.1 Encoding ASTs Using Cloud Types

We add a new Cloud AST type by letting each log entry be a *change* (i.e. diff) to the AST, stored as a pair (T_A, T'_A).

T_1	T_3	(entry L_1)
T_1	T_2	(entry L_2)
T_3	T_5	(entry L_3)
T_2	T_4	(entry L_4)
		$\vdots$

Then we can use our merge function for log evaluation: $collapse((T_O, T_M), (T_A, T'_A)) = (T_O, M(T_A, T'_A, T_M))$. This handles *local* edits sensibly, since by the identity properties, $collapse((T_1, T_2), (T_2, T_3)) = (T_1, T_3)$. The "transitive" merge property described in Theorem 2 shows that a local edit (T_A, T'_A) is merged with *remote* edits sensibly.

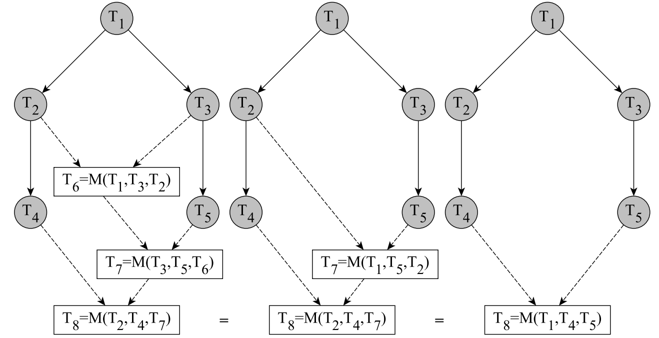


Figure 12: Log Evaluation – Client (right branch) and Server (left branch)

Note that associativity (Theorem 3) ensures that this also behaves properly for more than one client.

5.2 Local *Undo*

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.

We implement this *local undo* operation by utilizing the merge function. As with regular undo, a local history is saved as the user makes edits, but instead of simply reverting to a saved history item, our *undo* operation pushes an *inverse diff* of the previous edit. That is, if the user previously changed the AST from T_A to T'_A, then we push the diff (T'_A, T_A).

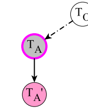


Figure 13: Undo (only local edits)

Notice that this works trivially in the case where the log evaluates to (T_O, T'_A) and there are only local edits. In this case, pushing (T'_A, T_A) to the log clearly restores T_A, since $eval((T_O, T'_A)(T'_A, T_A)) = (T_O, M(T'_A, T_A, T'_A)) = (T_O, T_A)$ by the identity properties (see Figure 13).

If there are remote edits, we should restore $M(T_O, T_A, T_B)$, and this is indeed the case if local/remote edits are well separated. Figure 14 shows an example of this operation.

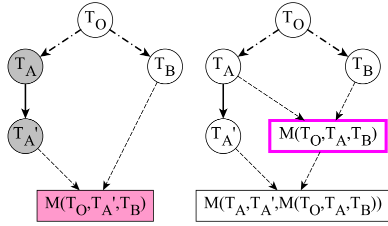


Figure 14: Undo (local + remote edits)

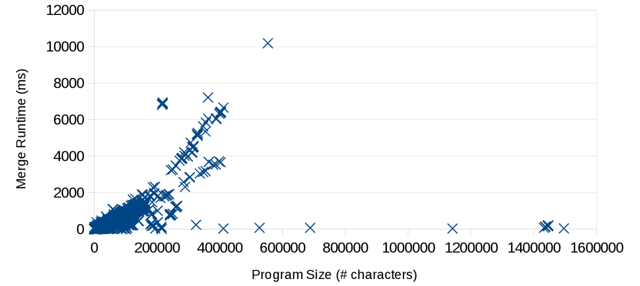


Figure 15: Merge Performance on $M(T_O, T_O, T_O)$ Experiment

6. Implementation

We have implemented the merge algorithm in about 1700 lines of TypeScript. This functionality accepts three TouchDevelop ASTs T_O, T_A, T_B and produces a merged AST 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 AST to make it more straightforward, and added support in AST nodes for node-IDs and multiple base IDs. We also made modifications to the *cut/-copy* operations in the TouchDevelop Editor, allowing tracking of script/node ID for cut/copied AST nodes.

Then, we added Editor buttons for patch merge, allowing users to click "merge" and merge changes from another script. For this, we also implemented least-common-ancestor computation to automatically compute a sensible common ancestor (base) T_O for the patch merge.

To enable realtime collaboration, we modified the Cloud Sessions functionality to add a Cloud-AST datatype, where the log entries are AST diffs (stored as pairs of TouchDevelop ASTs) and our merge function is used for log evaluation. We also modified the Editor to connect to an AST Cloud Session, display the Cloud-AST (which is updated as other users collaborate), and push any local AST changes. Finally, we modified the Editor's *undo* operation to push a reverse diff of the previous local edit.

7. Evaluation

Our TypeScript merge implementation is derived from a Scala baseline implementation. We have run *identity* and *associativity* regression tests with thousands of randomly-generated trees on the Scala implementation, to confirm that we have properly realized the formal specification of the merge algorithm.

As a basic merge performance test, we have iterated through the full collection of 100K+ actual TouchDevelop scripts, merging each one with itself (Figure 15). The results show that the maximum time is almost always below 8 seconds. One script required about 10 seconds to merge due to a 1000-line block of variable declarations, which makes the strongly-connected components call slow. We expect that future optimizations will reduce this time significantly (see Future Work).

8. Related Work

Much work has been done by the research community in terms of merge algorithms. The classic text-level merge [Khanna et al. 2007] 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 [Apel et al. 2012] and semantic [Mens 2002] 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 [Horwitz et al. 1989] [Reps 1991], 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) [Ellis and Gibbs 1989], 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 [Wikipedia 2014].

Additionally, errors have been found in published OT algorithms [Preguica et al. 2009]. Motivated by these difficulties of the OT approach, others have proposed a cleaner approach, Commutative Replicated Data Types (CRDT) [Shapiro et al. 2011]. 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

[Burckhardt et al. 2012] eventual consistency model, which does not place this restriction on our merge algorithm.

9. 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 AST-based merge function, which is used in conjunction with the Cloud Types eventual consistency model to enable both standard branch/merge version control, and realtime collaborative editing. An implementation which allows standard branching/merging, and a preliminary implementation enabling single-user (multi-device) realtime collaboration has been added to the TouchDevelop online programming environment.

In the future, we will work on fully deploying and 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 AST diffs that are sent over the network as users collaborate.

Another interesting area of research would be *Longitudinal Program Analysis* in this collaborative editing context. The paper [Notkin 2002] proposes that program analysis approaches take into consideration the full version-history lifetime of a program, rather than just individual snapshots. A few approaches have moved in this direction [Lahiri et al. 2010] [Logozzo et al. 2014], but they focus on analyzing a single diff, i.e. using information from the analysis of program P to analyze a modified program P'. It would be interesting to see how we could do this in general, using our version history and a carefully-specified merge algorithm.

References

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.

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.

S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. It's Alive! Continuous Feedback in UI Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 95–104, New York, NY, USA, 2013. ACM.

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.

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

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.

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.

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.

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

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.

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.

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

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.

Wikipedia. Operational transformation, Dec. 2014. URL `http://en.wikipedia.org/wiki/Operational_transformation`.