COAST: A Conflict-free Replicated Abstract Syntax Tree

Aäron Munsters[a], Angel Luis Scull Pupo[b] and Jens Nicolay[c]
Software Languages Lab, Vrije Universiteit Brussel, Pleinlaan 2, Brussels, Belgium
{*amunster, ascullpu, jens.nicolay*}@*vub.be*

Keywords: Conflict-free Replicated Data Types, Software Evolution, Abstract Syntax Trees, Distributed Collaboration.

Abstract: Remote real-time collaborative text editing enables collaboration of distributed parties which improves an agile workflow, team member availability and productivity. Collaborative source-code editors are often implemented as a variant of regular collaborative text editing with source code highlighting. Such approaches do not use the structural program information to accurately merge concurrent changes on the same portions of code for temporal network partitions. Therefore, these approaches fail to merge concurrent structural changes to the program such as a concurrent move and edit operation. In this paper we propose an approach in which the editor replicates not the program text but the program tree that corresponds with the program text. Propagating source code changes as tree operations enables resolving concurrent tree changes with higher accuracy. We evaluate our approach by reproducing a use case in which we concurrently change source code on existing tools and our approach. We show that existing tools break the lexical structure and come up with an incorrect program while our approach can distinctly apply the changes preserving the program structure.

1 INTRODUCTION

In collaborative real-time editing systems, multiple replicas of a digital resource can be updated concurrently, while the system ensures that all replicas automatically and quickly converge. An important application of collaborative real-time editing is programming. Using collaborative programming, programmers can work together by simultaneously writing and editing source code, while observing edits performed by other programmers in real-time and discussing ideas and issues that emerge from these concurrent activities. Enabling teams to *pair program* in geographically distributed settings is known as *distributed pair programming*, which has been shown to lead to software development with comparable quality and productivity to colocated pair programming (Baheti et al., 2002).

A Conflict-Free Replicated Data Type (CRDT) (Shapiro et al., 2011b) is a data type that enables the replication and maintenance of a digital resource in a distributed system without requiring additional communication to resolve concurrent changes. CRDTs that are used to support state of the art real-time text editors both for commercial and research projects are modelling the shared data structure as a string (Yu, 2012; Yu, 2014). For real-time *program* editing, however, modelling the data structure as a string limits the extent to which the merge strategy can reason about merges. We noticed that modelling the shared program as its syntax tree would allow the CRDT to perform better in terms of merging concurrent program changes.

In this work, we propose *CoAST*, a real-time collaborative source-code editing system supported by an abstract syntax tree (AST) CRDT. By computing AST changes on local replicas and sharing these with others accompanied by a logical timestamp, all replicas can compute an equal view of the AST by maintaining a chronologically ordered log of all changes and querying this log to construct the same AST. By doing so we aim to provide better support for distributed developers who experience concurrent changes which could break the syntax for development on code editors supported by string-based CRDTs. To the best of our knowledge CoAST is the first system that proposes a real-time collaborative editor using a CRDT based on the program structure (i.e., the AST) instead of the program's text.

The contributions of the paper are the following:

- A novel approach that enables real-time code collaboration through modelling the program's AST as a CRDT.

[a] https://orcid.org/0000-0001-5593-1273
[b] https://orcid.org/0000-0003-2083-1285
[c] https://orcid.org/0000-0003-4653-5820

- A proof-of-concept implementation of a real-time collaborative editor for a Lisp-like language based on our AST CRDT.

The rest of the text is organized as follows. Section 2 explains the main challenges in the state-of-the-art collaborative editing systems and why the use of a higher level of abstraction for the program representation may improve the performance of such collaborative systems. Section 3 describes the main characteristics of CoAST. Section 4 describes the design choices of our AST CRDT and Section 5 explains how we used the GumTree algorithm to compute language-agnostic AST changes. In Section 6, we explain the Scala-based implementation of our AST CRDT. First, we explain the properties and the interface of the CRDT. Then, we describe the implementation of the underlying AST data structure. We discuss and validate our collaborative editor in Section 7 for an example concurrent program edit. Finally, we discuss the ordering of operations and the merging granularity as the current limitations of our approach in Section 8.

2 PROBLEM

Over the years, many collaborative editors (Microsoft, 2022; Atom, 2022; Ghorashi and Jensen, 2016; Nicolaescu et al., 2015; Salinger et al., 2010; Yu, 2012; Yu, 2014; Ball et al., 2015; Nichols et al., 1995) for software development have been proposed. However, while surveying the existing approaches, we observed that a considerable number of existing collaborative code editors are string-based. String-based editors handle the consistency of concurrent changes of the source code at the level of sequences of individual characters. Handling concurrent changes at the textual level of a program source code is problematic when replicas disconnects and reconnect while concurrent changes are being made.

In our work, we use Lisp's S-expression syntax (also used by other Lisp-like languages such as Closure and Julia) for conciseness and simplicity, but our findings are also valid for other syntax families (e.g., C syntax). For example, assume a local and a remote replica of source code that are synchronized (i.e., have the same program text). The content of two synchronized replicas is depicted in the green area at the top of Figure 1. Next, imagine that the two replicas are disconnected, for example because of a network partition, during which two edits happen concurrently. A local edit changes the string "apple" into "banana", while on the remote replica the two expressions in the body of the begin switch places. In Figure 1, this is

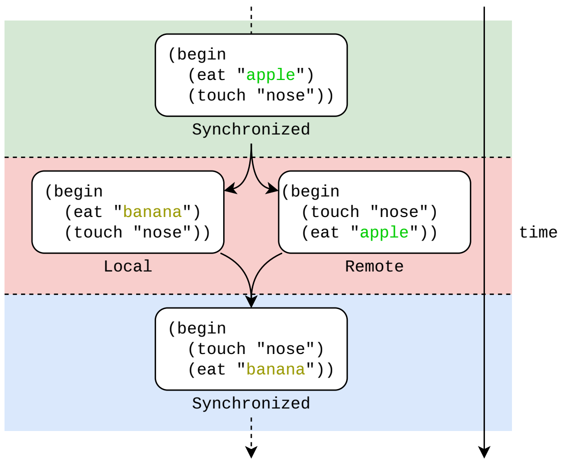


Figure 1: A sample Lisp-like program that evolves through different machines that temporarily break synchronization. This example shows that reasoning over the code structure is required to successfully merge the two separate operations.

depicted in the pink area. Finally, assume that connectivity between the replicas is restored. We believe the desired outcome would be that *both* the local and remote changes are correctly applied at each replica, preserving both replicas their intent as shown in the blue `Synchronized` state of Figure 1.

However, in string-based collaborative code editors such as Yjs(Nicolaescu et al., 2015), the outcome is unpredictable and often wrong (unexpected, sometimes with syntax errors), depending on how and precisely when the two replicas connect again. An example result that can be observed after reconnection for the diverging changes depicted in the pink area in Figure 1 is shown in Listing 1.

Listing 1: An example solution proposed by existing string-based CRTS to the merge conflict shown in Figure 1, showing how the syntactical structure of the program breaks.

```
(beginbanana
  (touch "nose")
  (eat "apple"))
```

In addition to being incorrect with respect to the two changes, the program in Listing 1 is also no longer syntactically correct.

In an attempt to avoid unexpected and incorrect merge results stemming from string-based synchronization of programs, it is possible to synchronize at higher levels of syntactic abstraction, even taking into account the semantic properties of programs. It is for example possible to synchronize at the lexical level (tokens) or at the level of abstract syntax (typically a tree). A system could choose to only synchronize when code is free from certain syntactic or even semantic errors. An example is the work of (Goldman et al., 2012), where synchronization occurs when the code compiles. However, when the level of abstraction of the source code model is high and the conditions for merging are strict, it may be the case that in typical development scenarios there may be long times between replica updates. This may result in a system that is closer to working with a traditional version control system, where programmers are supposed to only commit "working code". Although working with a version control system is also a form of collaborative editing, it is a more asynchronous workflow that does not directly support real-time synchronization between source code replicas.

3 APPROACH

In this paper we explore an approach, called CoAST, for collaborative source code editing that enables near-real-time updates and avoids some of the inconsistencies that result from string-based synchronization mechanisms. More precisely, our approach focuses on several aspects:

- **Structure and Consistency**
  Raising the abstraction level from programs as strings to programs as syntactically structured entities, permits CoAST to make changes to the content of the program while upholding the syntactical integrity ensuring that the program maintains a valid structure by performing merges on a syntactical abstraction level.
- **Near Real-time Distributed Design**
  Through propagating program changes based on the program AST, CoAST is able to make more informed merging decisions at high speeds while abstaining from semantical analysis which we assume would in turn be more costly in terms of performance.
- **Decentralized Setting**
  By building on the existing work of CRDTs, CoAST provides a decentralized implementation of a collaborative editor featuring system scalability, availability and performance.
- **Determinism**
  CoAST ensures a deterministic merging strategy of concurrent operations. This means that replicas that apply concurrent operations will reach the same state, even if the operations arrive in a different order to different replicas.

Real-time collaborative development of text documents requires multiple replicas to agree on an identical view of the shared document while dealing with the problems that are inherent to distributed systems. One model that enables replicas to agree on the same view of a distributed data structure is known as a Conflict-Free Replicated Data Type (CRDT) (Shapiro et al., 2011b).

While the use of string-based CRDTs has been proven to be an effective strategy for collaborative text editing for text in general, it ignores structural information when the text at hand represents program source code. This omission weakens the consistency of the collaborative code editor. Although a program's source code is stored on computers as a sequence of characters, the programmer and the computer eventually reason about the program structurally in terms of its Abstract Syntax Tree (AST). The program AST is a tree representation of the textual program representation that abstracts from details such as tokens that separate subexpressions from their parent expression.

Designing the CRDT based on the program AST makes it more informed about the underlying program than handling the program as a large string, without the need for semantic reasoning about the program. Replicating the program AST allows us to face the aforementioned weakened consistency from a different perspective. As ASTs manipulation prevents breaking the syntactical structure of the code, querying a CRDT from a concurrently manipulated AST facilitates maintaining the syntactical correctness and shifting the merge conflicts from character-based operations to tree-structure changes. Because our approach relies on a CRDT, there is no need for strong synchronization, which improves the availability of the system (Shapiro et al., 2011a).

4 A REPLICATED CONFLICT-FREE ABSTRACT SYNTAX TREE

This work proposes a new approach for collaborative code editing through modelling a program's abstract syntax tree as an operation-based CRDT. Our approach consists of replicas of the program's AST sharing the state of the code. The CRDT operates in different stages that depart from code changes in one replica to converge on a changed AST on all other replicas. First, changes to the source code text trigger an event that verifies syntactical correctness (ie. an AST can be constructed). Next, for edits resulting in a *valid* AST, the changes are computed in the form of an *edit script* that describes what tree transformations transform the former AST into the latter. In the next stage, these edit scripts are bundled with a logical timestamp and replica identifier, which is stored on a local log and propagated over the network to other replicas. Next, replicas that receive the incoming edit scripts insert these edits in their local log in chronological order and roll back local edits that succeed the incoming operations, after which they replay all edits to end up with an equal view on the program AST. Edit operations arriving in concurrent updates may depend on earlier changes agreed upon by other replicas. We adopt a permissive strategy to not include these operations when computing the final tree as they would invalidate the program, yet they remain in the log. This behaviour prevents adding invalid nodes to the graph that could otherwise break the program. When the state of the local replica is in a valid AST the outcome is then reflected on the code editor. Figure 2 provides an overview of the beforementioned stages that take place for a single syntactically correct code-change to become replayed on other replicas.

The design of the AST CRDT is highly inspired by the design of the paper "A highly-available move operation for replicated trees" by (Kleppmann et al., 2022). Their work discusses the design of a tree CRDT, motivating why move operations become complex for replicated trees. We summarize the discussion on the difficulties:

- **Concurrently Moving the Same Node**
  Say concurrently a move operation takes place for the same node, resulting in the node *green* becoming the child of its sibling *blue* on one replica and becoming the child of its sibling *orange* on another replica as Figure 3 presents. Once these operations merge, different operations are possible, prioritize a single outcome (Figure 3 *(a)* and *(b)*), or duplicate the *green* node so that both *blue* and *orange* can have an instance of *green* as their child (Figure 3 *(c)*). Another option could be to have both nodes *blue* and *orange* have as their child node *green*, however this would break the tree-like structure (Figure 3 *(d)*).
- **Introducing Cycles**
  While move operations may uphold the guarantees of the tree remaining as an acyclic structure on individual replicas, their combination may introduce cycles. Figure 4 shows the distinct outcomes possible. The figure illustrates the possible choices when two sibling nodes are concurrently made a parent of one another. It allows for either move operation to remain (Figure 4 *(a)* and *(b)*), a duplicate instance of both nodes to enable both move operations (Figure 4 *(c)*) or to introduce a cycle to enable both operations without duplication, but this would break the tree requirement (Figure 4 *(d)*).

The work of (Kleppmann et al., 2022) proposes to accompany each operation with a timestamp such that incoming operations that are out of order, allow the CRDT to determine the point at which these should have been applied. It then performs a rollback using the *undo_op* operation up until the point where the incoming operation can be applied after which all latter operations can be replayed using the *redo_op* operation. The authors formally validate their approach by implementing their approach in the Isabelle/HOL language, which allows proving the correctness of the implementation.

Building on their approach we subsequently motivate the correctness of our approach, although we do not include formal correctness.

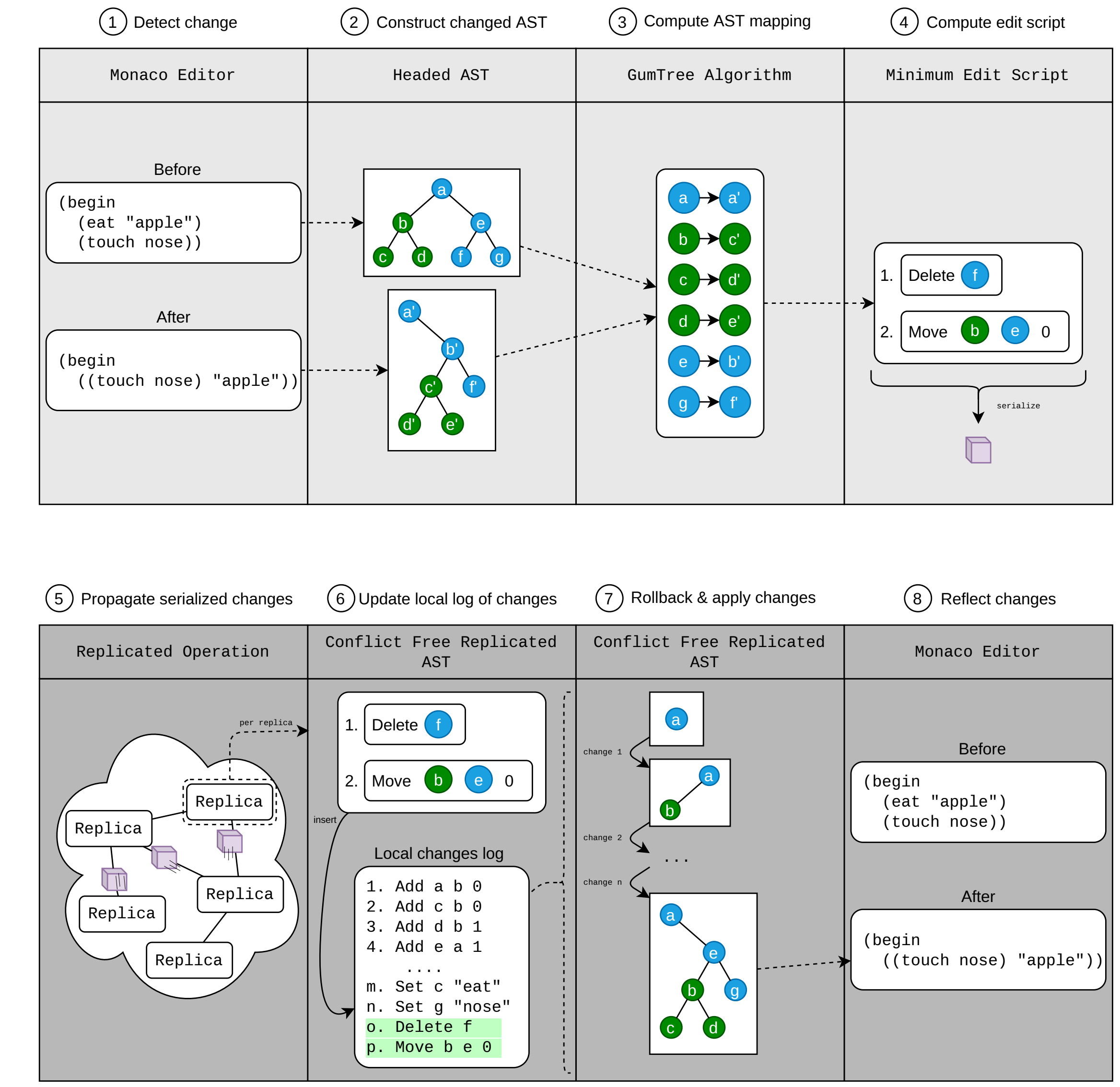
5 COMPUTING AST CHANGES

Before propagating AST changes over the network, they need to be determined locally. AST changes can also be derived from projectional editors, which allow the user to develop their programs through manipulating an AST directly rather than having the programmer type the program in an open space for characters. The benefit of projectional editors is that the programmer can make no syntactic mistakes. The drawback is the number of different building blocks needed to construct a program can become complex for languages with a rich syntax. Another option to compute AST changes is through a tree differencing algorithms (Chawathe et al., 1996; Fluri et al., 2007; Falleri et al., 2014; Huang et al., 2018; Dotzler and Philippsen, 2016; Frick et al., 2018).

Whereas projectional editors come with the disadvantage that they are specialized per language, AST tree differencing algorithms exist with *language agnostic* specifications. This work has adopted the GumTree algorithm (Falleri et al., 2014), a language-agnostic algorithm to compute the changes between two ASTs.

The output of the GumTree algorithm is an *edit script*, which is an ordered set of edit operations that transform the source AST into the destination AST. An AST edit script is composed of operations of the following kinds (Chawathe et al., 1996):

- $update(n, v)$ to update the content of node n with the value v.
- $add(t, p, i, l, v)$ to add node t with label l and value v to parent p at index i.
- $delete(t)$ to delete node t.
- $move(t, p, i)$ to move node t in tree to be a child of p at index i.



```
(begin
  (eat "apple")
  (touch nose))
```

```
(begin
  ((touch nose) "apple"))
```

```
1. Add a b 0
2. Add c b 0
3. Add d b 1
4. Add e a 1
   ....
m. Set c "eat"
n. Set g "nose"
o. Delete f
p. Move b e 0
```

Figure 2: An overview of the stages that flow through different components working together to make up a distributed conflict-free replicated abstract syntax tree. Changes detected by the javascriptMonaco Editor trigger the construction of a new HeadedAST, which is then mapped onto the previous HeadedAST through the GumTreeAlgorithm allowing to compute the changes that took place as a MinimumEditScript. These AST changes are serialized and propagated over the wire as ReplicatedOperations, as the CRDT is an operation-based CRDT. The incoming operations allow each replica to roll back the local AST to then reapply all known changes in order. The final computed AST is displayed by the javascriptMonaco Editor which then reflects the converging AST to the developer.

A more constrained form of an edit script is a *minimum edit script*, which is the smallest set of edit operations that is still a valid edit script. In the context of our CRDT AST, computing a minimum edit script is desirable since it directly affects the network load.

The GumTree algorithm consists of a top-down phase and bottom-up phase to compute a set of mappings between both ASTs, while it leaves the algorithm to determine an edit script open to the implementor. The GumTree implementation used in this work is based on the work of (Chawathe et al., 1996).

6 IMPLEMENTATION

We used Scala to implement COAST, opting for dependencies such that the implementation can compile to run on the JVM but also to JavaScript such that it supports our prototype browser-based web IDE. The implementation for of the CRDT is discussed in Sec-

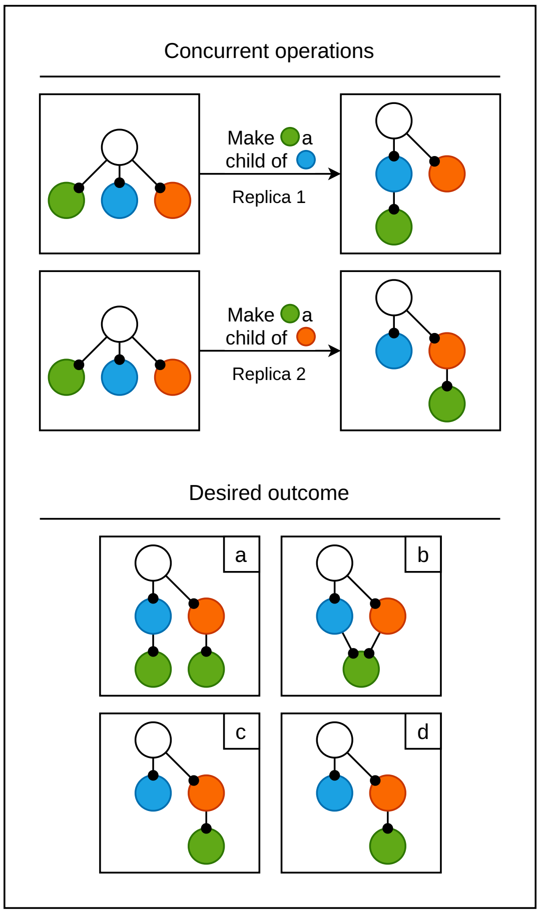


Figure 3: A case in which concurrent move operations take place for a node to become the child of either one of its siblings, including the possible outcomes. The shown outcomes illustrate there is either an operation-discarding choice, a duplicating choice or a tree-structure breaking choice that must be made. Inspired by the illustrations designed by (Kleppmann et al., 2022).

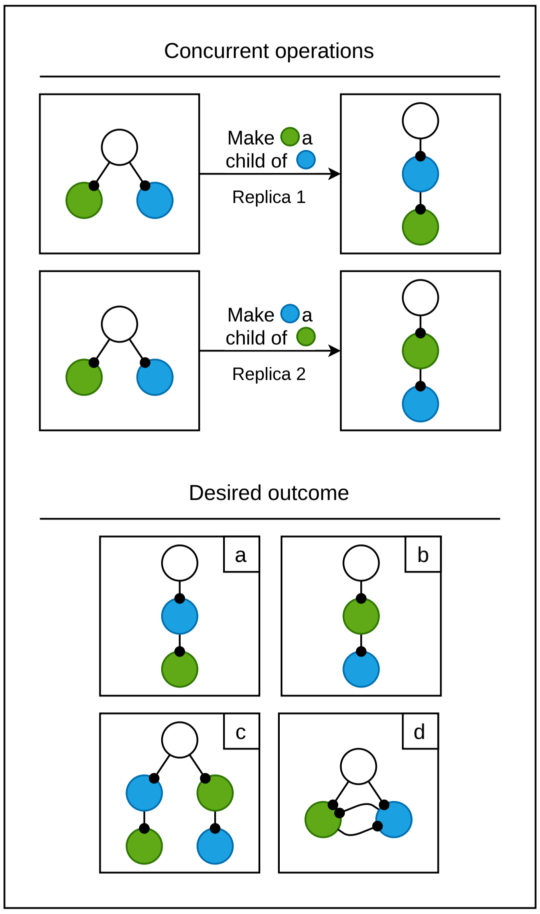


Figure 4: A case in which concurrent move operations take place for two nodes to become each other's descendant, including the possible outcomes. The shown outcomes illustrate there is either an operation-discarding choice, a duplicating choice or a tree-structure breaking choice that must be made. Inspired by the illustrations designed by (Kleppmann et al., 2022).

tion 6.1 and the implementation for modeling the AST is discussed in Section 6.2.

6.1 Implementing the Local CRDT

We define an interface for operation-based CRDT classes that defines three methods:

- update to propagate external changes (operations) to the local replicas model of CRDT
- query to yield the data structure that is known *so far*, modeled by the local replica
- merge for combining incoming operations from other replicas across the network

The concrete implementation of our AST CRDT interface that supports CoAST yields the AST upon invoking the query method by applying all received edit operations in order on an empty AST. Ordering the edit operations is possible by accompanying each with a logical timestamp and replica owner identity, making an arbitrary but deterministic ordering decision possible when the logical timestamp for two edits is equal.

The application of edit operations to a program's AST is deterministic but permissive, thus when the application of operations is not possible (such as moving a node that cannot be found) the operation is ignored. By making both the ordering and the application of the edit operations deterministic, we ensure converging ASTs among all replicas with an equal set of edit operations.

Upon invocation of the update method, the CRDT propagates the local edit operations over the network by publishing these changes on its local Transmitter, which in the prototype is implemented using a third-party peer-to-peer JavaScript networking library using the WebRTC standard. This Transmitter invokes the merge method upon receiving remote edit operations from the network, which expands the local log of edit operations with the new operations.

6.2 Implementing the Local AST

The AST itself is implemented in a functional code style by representing the tree as a mapping from node identities to nodes. This implies that the data structures are immutable which ensures that the replicated application of the same edit script results in the same AST outcome.

An intuitive implementation of the AST would be to implement the tree structure as a root with pointers to other nodes (that in their turn point to other nodes), but opting for this indirection through identities eases serialization of AST nodes, as pointers are avoided in a distributed setting where replicas do not share a memory space.

The prototype implementation limited the types of nodes to *number*s, *string*s, *identifier*s and *expression*s. This set of nodes is relatively small but is sufficiently large to provide a prototype that covers all source-code edit operations for a language supporting primitive values and nodes with descendants, showing support for arbitrary nested setups.

For the browser-based prototype, whenever the code is changed the AST is kept formatted in the browser to ensure all replicas with the same operations have the same view.

7 DISCUSSION AND VALIDATION

We leverage the existing work of (Kleppmann et al., 2022) and (Falleri et al., 2014) to support the correctness of our approach, however, we lay out the high-level editor properties below.

Preventing Interfering Changes: When multiple replicas add new code to the main program during a network partition, the eventually merged program will contain both additions without interleaving these changes. This is guaranteed as the newly added code is logged as an edit script that constructs a subtree (i.e., the AST of the new code) which will contain nodes that uniquely identify the replica that constructed this subtree.

Ensuring Syntactical Correctness: The eventual data structure that is present on each replica is the merged log of edit operations, which when queried will result in an identical AST that represents the merged program. To ensure this program is a syntactically valid program tree, it should (a) be a valid tree and (b) not break constraints that ensure syntactical correctness.

Ensuring the program represents a valid tree requires ensuring the created AST does not introduce cycles, where node a would be the parent of node b while b is also the parent of node a or that it does not add a node without a parent. This requirement is not implemented at the level of the edit script but when constructing the AST, where these requirements are asserted and when not met the operation is ignored, which allows for local resolution of conflicts.

Ensuring syntactical correctness requires the guarantee that the requirements of each node are met. An example would be an *if*-node that requires an expression as its condition field and two more statements, one for its consequent field and another for its alternative field. Consequent edit statements should not result in failing to meet these requirements. The resulting edit scripts from CoAST update nodes exclusively whenever their label matches which is a property that follows from GumTree. In case the labels are not equal the edit script deletes the old node and adds the new node, resulting in two nodes that differ in their identity which avoids interleaving of correctness properties for different types of nodes. Despite we do not provide a formal proof we believe this strategy prevents constructing a tree which is an invalid AST after a merging operation is done.

Tests:
The implementation of CoAST comes with a test suite of 54 tests to exercise the problems arising from the example described in The IntelliJ Test Coverage Tool reports 100% Scala code coverage for our test suite. Although it is limited, this testing suite paired with the high amount of code coverage aims to demonstrate the effort in verifying the correctness of the codebase.

The test suite includes a setup in which we replicate the behaviour of multiple replicas performing the concurrent changes shown in Figure 1. This test case simulates a temporal network partition and asserts that all permutations of the received operations result in an equal output program for each replica. These preliminary results show that our approach for CoAST can preserve the syntactical structure while including both concurrent changes shown in Figure 1, which the string-based CRDT could not.

8 LIMITATIONS

There are a number of limitations in our current implementation that can be further developed in future work.

Lack of Heuristics to Pick a Winning Operation for Conflicting Operations. The merge strategy of our approach performs an ordering of concurrent operations deterministically. However, the ordering is based on the replica's identity which prevents knowing which replica will perform the winning operations. An improvement to our current ordering of concurrent operations could be the assignment of developer roles through the user interface. The system should order roles such that operations of hierarchically higher roles are prioritized to win over other operations. Alternatively, the system could prompt the user with conflicting changes asking to agree on a single change, which is more in line with the work of traditional version control systems.

Lack of Merge Granularity. Currently, the merging of ASTs happens on a per-node basis. Although this merge strategy is sufficiently powerful, additional merging strategies based on the types of nodes that are being affected can improve our current prototype. For example, for string literal nodes, two concurrent updates on a single string literal will result in an arbitrary choice to dominate based on the replica identity. It would be more optimal to delegate the concurrent changes to the node which could further decide on a better merge strategy, in this case opting for a string-based CRDT. An additional example would be for literal set nodes, two concurrent additions of an expression value would duplicate the content, while this does not make sense for set nodes thus it would be the desired effect to prevent the double addition of the content.

As addressed in Section 7 the interfering changes happen for concurrent modifications to nodes present on both replicas, local changes to locally created nodes introduce no conflicts.

9 RELATED WORK

In this section, we describe recent and close related work to CoAST.

Protzenko et al. (Protzenko et al., 2015) proposed a conflict-free merge algorithm for collaborative editing which has been deployed as a cloud-based integrated development environment (Ball et al., 2015). Similar to our approach, the algorithm works at the AST representation of the program. The algorithm attaches unique identifiers to the AST nodes and tracks the order of nodes by remembering the siblings of a particular node. When a merge occurs, both the local and remote AST replicas are represented as sets. Then, the resulting tree is built by re-attaching the parents of the nodes in the set of the local replica and then, the nodes in the set of the remote replica. Concurrent updates are handled by always accepting the changes of the local replica.

Their approach models the program as an AST directly within the editor, allowing to track tree-edit operations with higher precision. This comes at the cost of specializing the editor to model the AST and accounting for the possible edit operations, while our approach derives the edits in a separate stage. In addition to this specialization, their approach stores the program as text in the cloud while the replicas operate on ASTs that are modelled as Cloud Types.

The *Live Share* (Microsoft, 2022) extension of Microsoft's IDEs enables real-time collaboration sessions between a host replica and guest replicas. *Live Share*'s approach to synchronising replicas is unknown as its source code is not publicly available. In contrast to our model, a *Live Share* collaboration session requires a host replica to be always online to maintain the session of guest replicas. Our experiments show that the replication of Figure 1 does not result in the proposed solution, but breaks the syntactical structure of the program through outputting the code shown in Listing 1, demonstrating the limited syntactical reasoning about the program.

Jimbo (Ghorashi and Jensen, 2016) uses an operational transformation (OP) algorithm to enable real-time collaborative code development. However, the authors do not elaborate on the characteristics of the algorithm or how concurrent updates are handled.

Saros (Salinger et al., 2010) is a collaborative editing plug-in for Eclipse. Concurrent activities within a collaboration session are handled by an implementation of the Jupiter (Nichols et al., 1995) algorithm, which relies on a central server for the coordination of such activities.

The Atom's Teletype (Atom, 2022) plugin implements a string-wise CRDT (Yu, 2012; Yu, 2014) to enable collaborative source code development. In contrast to our work, a string-wise CRDT enables update/insert operations at random places of the source code (i.e., the string). These random updates may enable the propagation of syntactically incorrect updates to the replicas. In contrast, our approach performs its merges on the syntactical abstraction, thus the merged outcome prevents syntactical incorrectness by design.

The Jetbrains' Meta Programming System (?) allows developers to extend IntelliJ-based editors with a projectional editor for a domain-specific language. Their editors furthermore support collaborative editing, however, we are unaware of their approach to synchronising replicas as their source code is not publicly available.

10 CONCLUSION

This paper presented an alternative approach, called CoAST, to implement a real-time collaborative code editor to overcome the limitations that arise for traditional editors that treat source code as sequences of characters. CoAST synchronizes code at the level of the abstract syntax tree (AST) and implements this AST as a conflict-free replicated data type (CRDT). Our proof-of-concept implementation covers a minimal set of nodes, but already demonstrates the capability of the merging strategy to resolve conflicts for different concurrent operations. The strategy is mainly based on the inclusion of a logical timestamp allowing for an ordering of events that allows the data structure to recompute the AST including newly received updates. Even so, certain optimizations are in order to improve conflict resolution, either by the inclusion of developer roles or by shifting the merge granularity.

REFERENCES

Atom (2022). atom/teletype: Share your workspace with team members and collaborate on code in real time in atom. https://github.com/atom/teletype. (Accessed on 03/22/2022).

Baheti, P., Gehringer, E., and Stotts, D. (2002). Exploring the efficacy of distributed pair programming. In Wells, D. and Williams, L., editors, *Extreme Programming and Agile Methods — XP/Agile Universe 2002*, pages 208–220, Berlin, Heidelberg. Springer Berlin Heidelberg.

Ball, T., Burckhardt, S., de Halleux, J., Moskal, M., Protzenko, J., and Tillmann, N. (2015). Beyond open source: The touch develop cloud-based integrated development environment. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, pages 83–93.

Chawathe, S. S., Rajaraman, A., Garcia-Molina, H., and Widom, J. (1996). Change detection in hierarchically structured information. *SIGMOD Rec.*, 25(2):493–504.

Dotzler, G. and Philippsen, M. (2016). Move-optimized source code tree differencing. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 660–671.

Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., and Monperrus, M. (2014). Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 313–324, New York, NY, USA. Association for Computing Machinery.

Fluri, B., Wursch, M., PInzger, M., and Gall, H. (2007). Change distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743.

Frick, V., Grassauer, T., Beck, F., and Pinzger, M. (2018). Generating accurate and compact edit scripts using tree differencing. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 264–274.

Ghorashi, S. and Jensen, C. (2016). Jimbo: A collaborative IDE with live preview. *Proceedings - 9th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2016*, pages 104–107.

Goldman, M. et al. (2012). *Software development with real-time collaborative editing*. PhD thesis, Massachusetts Institute of Technology.

Huang, K., Chen, B., Peng, X., Zhou, D., Wang, Y., Liu, Y., and Zhao, W. (2018). Cldiff: Generating concise linked code differences. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 679–690.

Kleppmann, M., Mulligan, D. P., Gomes, V. B. F., and Beresford, A. R. (2022). A highly-available move operation for replicated trees. *IEEE Transactions on Parallel and Distributed Systems*, 33(7):1711–1724.

Microsoft (2022). Faq - visual studio live share - visual studio live share — microsoft docs. https://docs.microsoft.com/en-us/visualstudio/liveshare/faq. (Accessed on 03/02/2022).

Nichols, D. A., Curtis, P., Dixon, M., and Lamping, J. (1995). High-latency, low-bandwidth windowing in the Jupiter collaboration system. *UIST (User Interface Software and Technology): Proceedings of the ACM Symposium*, (October):111–120.

Nicolaescu, P., Jahns, K., Derntl, M., and Klamma, R. (2015). Yjs: A framework for near real-time p2p shared editing on arbitrary data types. In Cimiano, P., Frasincar, F., Houben, G.-J., and Schwabe, D., editors, *Engineering the Web in the Big Data Era*, pages 675–678, Cham. Springer International Publishing.

Protzenko, J., Burckhardt, S., Moskal, M., and McClurg, J. (2015). Implementing real-time collaboration in touch develop using AST merges. *MobileDeLi 2015 - Proceedings of the 3rd International Workshop on Mobile Development Lifecycle*, pages 25–27.

Salinger, S., Oezbek, C., Beecher, K., and Schenk, J. (2010). Saros: An eclipse plug-in for distributed party programming. *Proceedings - International Conference on Software Engineering*, pages 48–55.

Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. (2011a). A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA.

Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. (2011b). Conflict-free replicated data types. In

Défago, X., Petit, F., and Villain, V., editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Berlin, Heidelberg. Springer Berlin Heidelberg.

Yu, W. (2012). A string-wise CRDT for group editing. *GROUP'12 - Proceedings of the ACM 2012 International Conference on Support Group Work*, (Figure 1):141–144.

Yu, W. (2014). Supporting string-wise operations and selective undo for peer-to-peer group editing. *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work*, pages 226–237.