



City Research Online

City St George's, University of London

Citation: Semenov, G. & Aksenov, V. (2026). Semantic Conflict Model for Collaborative Data Structures. In: UNSPECIFIED (pp. 25-31). ACM. ISBN 979-8-4007-2637-8 doi: 10.1145/3806077.3806700

This is the published version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: <https://openaccess.city.ac.uk/id/eprint/38469/>

Link to published version: <https://doi.org/10.1145/3806077.3806700>

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to [City Research Online policy](#).

Semantic Conflict Model for Collaborative Data Structures

Georgii Semenov
georgii.v.semenov@gmail.com
ITMO University
Saint Petersburg, Russia

Vitaly Aksenov
aksenov.vitaly@gmail.com
City St George's, University of
London, United Kingdom

Abstract

Digital collaboration systems support asynchronous work over replicated data, where conflicts arise when concurrent operations cannot be unambiguously integrated into a shared history. While Conflict-Free Replicated Data Types (CRDTs) ensure convergence through built-in conflict resolution, this resolution is typically implicit and opaque to users, whereas existing reconciliation techniques often rely on centralized coordination. This paper introduces a conflict model for collaborative data structures that enables explicit, local-first conflict resolution without central coordination. The model identifies conflicts using semantic dependencies between operations and resolves them by rebasing conflicting operations onto a reconciling operation via a three-way merge over a replicated journal. We demonstrate our approach on collaborative registers, including an explicit formulation of the Last-Writer-Wins Register and a multi-register entity supporting semi-automatic reconciliation.

CCS Concepts: • **Human-centered computing** → *Computer supported cooperative work*; • **Computing methodologies** → **Distributed algorithms**; • **Information systems** → **Remote replication**.

Keywords: semantic conflicts, optimistic concurrency control, eventual consistency, collaborative editing, CRDT

ACM Reference Format:

Georgii Semenov and Vitaly Aksenov. 2026. Semantic Conflict Model for Collaborative Data Structures. In *13th International Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '26)*, April 27–30, 2026, Edinburgh, Scotland Uk. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3806077.3806700>

1 Introduction

Digital collaboration environments provide a shared workspace that enables multiple users, teams, or organizations to work together toward a common goal [12]. Implementing highly available groupware systems, however, is challenging due to

the need to tolerate network partitions and enable isolated work, as is common in version concurrency control systems. To address these challenges, users are allowed to collaborate asynchronously by producing *optimistic* transactions that are applied locally to their data replicas, and eventually propagated to other replicas [11]. This approach has been revisited as *eventual state-machine replication* [7].

The core part of such systems is the synchronization algorithm, which ensures convergence to a single state once all contributions have been integrated. During synchronization, *conflicts* arise when concurrent operations, i.e., that are not causally related, interfere with one another. This may be captured semantically, for example, as operations violating each other's intents [4, 17] or as operations being non-commutative [21]. The common emergence of conflicts, however, is in the ambiguity of how operations should be integrated into a shared history. Crucially, the multi-dimensionality of this ambiguity elevates conflict resolution to a distinct collaborative subtask, concerned with coordinating and reconciling parallel commitments, a process referred to as *articulation work* [12].

One widely adopted approach to conflict resolution is exemplified by conflict-free replicated data types (CRDTs) [14], which rely on semilattice structures or on the commutativity of operations. CRDTs provide strong eventual consistency through resolution strategies that are embedded directly into the data type definition, as captured in Add-Wins Sets and Last-Writer-Wins Registers [14]. Multi-Value Registers [14] take a different stance by retaining all concurrently written values and thereby deferring their resolution. Nevertheless, conflict resolution in CRDTs generally remains implicit, opaque to users, cannot be supervised, and is generally non-native to application-specific semantics.

A second, more expressive class of approaches is based on reconciliation techniques [16], which treat conflict resolution as an application-level procedure that integrates concurrent transactions in a semi-automatic manner. Typical strategies include computing an execution order that minimizes violated constraints [10] or maximizes the number of preserved operations [13]. These techniques usually rely on a centralized reconciliation phase with global knowledge of all operations, which makes them poorly suited to local-first [6] collaboration scenarios, where replicas must continue to make progress independently on local copies. This limitation motivates the need for reconciliation middleware



This work is licensed under a Creative Commons Attribution 4.0 International License.

PaPoC '26, Edinburgh, Scotland Uk

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2637-8/2026/04

<https://doi.org/10.1145/3806077.3806700>

that operates without a central coordinator, provides explicit convergence guarantees, and treats conflict resolution as a first-class collaborative activity, that is, as a form of articulation work.

This paper presents a novel conflict model for collaborative data structures that enables local-first conflict resolution through the reconciliation of operations. Conflicts are identified using *entails* and *discards* dependencies, which relate each operation to its logical premises, and conflicting operations are rebased onto a new reconciling operation. We illustrate the model through several examples of collaborative registers and their formal definitions, including the Last-Writer-Wins Register as an instance of a CRDT with automated conflict resolution, and a calendar event as an example of a multi-register entity supporting complex operations.

2 System Model

We consider a shared memory M which is modeled as a set of n pre-initialized registers. Each register M_i has its basis of *actions* A_i , which is formally a set of all allowed atomic operations on the register. We define an *operation* $o = [a_1, a_2, \dots, a_k]$ as a contiguous sequence of action instances $a_i \in \bigcup^n A_i$ on the registers with the transactional semantics, i.e., executing an operation o is equivalent to a sequential uninterruptible execution of all its actions $a_1 \circ a_2 \circ \dots \circ a_k$. The *state* $s_i \in S_i$ of register M_i is deterministically produced by an interpreter I_i after executing a sequence of actions on the initial interpreter context $\perp$. Equality of registers is inferred from the equality of their states, i.e., $x = y \Leftrightarrow s_x = s_y$.

Our distributed system model consists of multiple nodes with their shared memory replicas $\mathcal{R} = \{r_1, r_2, \dots, r_n\}$. Operations issued on the replicas are *applied* locally and eventually *synchronized* between nodes, which are connected with an unreliable network. Local replica r of the shared memory is modeled by a *local history* $\mathcal{H}_r$, which is a sequence of operations, and each register $x = M_i$ has a projection $\mathcal{H}_r(x)$ of the history, which is a sequence of actions corresponding to x . One can provide the notion of *happens-before* [8] relation $\prec$, defined over actions and operations in the history, so that if $o_1 \prec o_2$, then o_1 precedes o_2 in any history $\mathcal{H}_r$ (same definition for $a_1 \prec a_2$ and $\mathcal{H}_r(x)$).

Each memory register $x = M_i$ is initialized with a *constructor* operation c_x , so that $\forall a \in \mathcal{H}_r(x): a \notin c_x \Rightarrow c_x \prec a$. When an operation is *applied* by a replica, it is just appended to the local history $\mathcal{H}_r$. Eventually, the current replica history $\mathcal{H}_r$ is *published*, so that it is available for pull on the other replicas. When ready to reconcile with foreign changes, replica r *synchronizes-with* another replica r' by integrating operations from another replica's published history $\mathcal{H}_{r'}$ into the local history, i.e., $\mathcal{H}_r \leftarrow \mathcal{H}_r \sqcup \mathcal{H}_{r'}$. Complete synchronization is achieved through integrating the histories of all

replicas in an arbitrary order, until they converge to the same *state* once all the conflicts are resolved.

3 Conflict Model

A conflict model provides mechanisms for *identification* and *resolution* of conflicts in collaborative data structures. The strongest conflict model is the *concurrent* conflict model, i.e., operations are conflict-prone if they are concurrent in terms of happens-before relation [8]: $o_1 \parallel o_2 \Leftrightarrow o_1 \not\prec o_2 \wedge o_2 \not\prec o_1$. The weaker one is the *non-commutative* conflict model, i.e., operations conflict if they are concurrent and non-commutative [14]. Usually, conflict resolution is captured by establishing the happens-before order between conflicting operations, so that they can be applied in a consistent manner on all replicas. In this section, we introduce a conflict model based on operation dependencies to empower the synchronization procedure in the outlined system model.

3.1 Conflict Identification

Let us consider two kinds of relations over operations in the history: *entails* ($\vdash$) and *is discarded by* ($\ll$). Intuitively, entailment captures *epistemic preconditions* [19] of an operation: issuing a new operation relies on the presence of effects of its premises, under the assumption that their effects remain valid. Discarding, in turn, captures *epistemic revision* [2]: an operation may invalidate the effect of another operation, thereby operation is no longer a premise for subsequent operations.

Syntactically, upon being applied, o is assigned an immutable set of *premises* $\Gamma = \{o_1, o_2, \dots, o_n\}$ that *entail* o , which is denoted as $\Gamma \vdash o$. The premises of an operation $o = [a_1, \dots, a_k]$ are derived from the premises of its actions, i.e., $\Gamma(o) = \bigcup_{j \in [1, k]} \Gamma(a_j)$, and $a_1 \vdash a_2$ denotes that the corresponding operations $a_1 \in o_1$ and $a_2 \in o_2$ satisfy $o_1 \vdash o_2$. Common premises of several operations are denoted as $\Gamma(o_1, o_2, \dots, o_n) = \bigcap_{i=1}^n \Gamma(o_i)$. Entailment implies happens-before relation, i.e., $o_1 \vdash o_2 \Rightarrow o_1 \prec o_2$. Let $\vdash^*$ define the reflexive transitive closure of $\vdash$. Then, concurrent operations may be redefined in terms of entailment as follows: $o_1 \parallel o_2 \Leftrightarrow (o_1 \not\vdash^* o_2) \wedge (o_2 \not\vdash^* o_1)$.

We require that each action a is associated with a monotonically non-increasing predicate $vis(a, \mathcal{H}_r(x))$, indicating whether the effect of a is visible in the current history $\mathcal{H}_r(x)$. State is modeled as a set of actions with their visible effects. Next, we allow an action in the history to be *rolled back* by removing it together with all actions that transitively depend on it, i.e., $\mathcal{H}_r(x) \setminus a \equiv [a' \in \mathcal{H}_r(x): a \not\vdash^* a']$. Then, we define *is discarded by* relation as follows: $a_1 \ll a_2 \Leftrightarrow a_1 \vdash a_2 \wedge vis(a_1, \mathcal{H}_r(x) \setminus \{a_2\}) \wedge \neg vis(a_1, \mathcal{H}_r(x))$. Similarly, $o_1 \ll o_2 \Leftrightarrow \exists a_1 \in o_1, a_2 \in o_2: a_1 \ll a_2$, and $vis(o) = \bigwedge_{a \in o} vis(a)$.

A *conflict* arises when a premise of one operation is concurrently discarded by another operation. Formally, for a set

of operations $\{o_1, o_2, \dots, o_n\}$, we define the set of *conflicting premises* as $\hat{\Gamma}(o_1, o_2, \dots, o_n) = \{o' \in \Gamma(o_1, o_2, \dots, o_n) \mid \exists o_i \neq o_j: o' \vdash o_i \wedge o' \ll o_j\}$. Two operations o_1 and o_2 are said to be *compatible* if they do not have conflicting premises, i.e. $o_1 \triangleleft o_2 \Leftrightarrow \hat{\Gamma}(o_1, o_2) = \emptyset$. Similarly, a local history $\mathcal{H}_r$ is compatible with an operation o if all operations in the history are compatible with o : $\mathcal{H}_r \triangleleft o \Leftrightarrow \forall o' \in \mathcal{H}_r: o' \triangleleft o$. A history $\mathcal{H}_r$ is *valid* if it is a topological ordering of the received operations with respect to $\vdash$, and all pairs of operations in the history are mutually compatible: $\forall o_i, o_j \in \mathcal{H}_r: o_i \triangleleft o_j$.

To enable representation of operations with their premises, we use an *entailment graph*, where nodes are the operations of the local history $\mathcal{H}_r$ and directed edges represent the partial order $\vdash$. An edge between o_1 and o_2 exists if and only if $o_1 \vdash o_2$. Then, topological sorting of the entailment graph induces a valid history $\mathcal{H}_r$ [7].

3.2 Conflict Resolution

To support conflict resolution, we allow an operation to be *rebased* with respect to its premises. Then, intuitively, when a conflict occurs, conflicting operations may be rebased to a *merge* operation that reconciles them.

Let us denote by $\hat{o} \hat{\vdash} o$ that an operation o is rebased to $\hat{o}$. Formally, when $\hat{o} \hat{\vdash} o$, we interpret o as an empty sequence of actions and include its premises to the premises of $\hat{o}$, i.e., $\Gamma(o) \subseteq \Gamma(\hat{o})$. Additionally, we close the entailment relation transitively under rebasing: $(o_1 \vdash o') \wedge (o' \hat{\vdash} o_2) \Rightarrow (o_1 \vdash o_2)$. The incoming edges of o may be removed from the entailment graph, as its premises $\Gamma(o)$ are transitively included in $\hat{o}$. That is, rebasing induces causality: $o_1 \hat{\vdash} o_2 \Rightarrow o_1 \triangleleft o_2$.

To support cancellation of operations, we define a distinguished *tombstone* operation $o_\emptyset$. An operation rebased to $o_\emptyset$ is treated as semantically null: it does not contribute effects and need not be merged into replica histories. Such rebased operations act as discontinuity markers and may be omitted from subsequent conflict detection.

Now, we are ready to present the synchronization algorithm empowered by the rebase procedure: a replica r incrementally integrates operations received from another replica r' into its local history $\mathcal{H}_r$.

The procedure $\text{SYNC}(r')$, shown in Figure 1, fetches the recent history $\mathcal{H}_{r'}$ from r' and processes its operations in causal order. For each operation $o \in \mathcal{H}_{r'}$, the algorithm proceeds as follows. First, if o has been rebased at r' , that is, there exists $\hat{o} \in \mathcal{H}_{r'}$ such that $\hat{o} \hat{\vdash} o$, the operation o is rolled back from the local history $\mathcal{H}_r$, i.e., o and all its transitive premises are removed. If o has been rebased to the tombstone operation $o_\emptyset$, it is treated as cancelled and skipped. If an operation is rebased in both replicas to different operations, all the rebases are applied (the corresponding merge operations $\{\hat{o}_i\}$ may be stored in a Grow-only Set [14]). If o is not already present in $\mathcal{H}_r$ and is compatible with the current history, i.e., $\mathcal{H}_r \triangleleft o$, then o is appended

APPLY(o):

$\mathcal{H}_r \leftarrow \mathcal{H}_r \sqcup \{o\}$

SYNC(r'):

$\mathcal{H}_{r'} \leftarrow \text{FETCH}(r')$ (fetch recent history from r')

for each $o \in \mathcal{H}_{r'}$ in causal order:

if $\exists \hat{o} \in \mathcal{H}_{r'}: \hat{o} \hat{\vdash} o$ **then**

$\mathcal{H}_r \leftarrow \text{ROLLBACK}(o, \mathcal{H}_r)$ (apply recent rebasing)

if $o_\emptyset \hat{\vdash} o$ **then continue** (skip cancelled operation)

if $o \notin \mathcal{H}_r \wedge \mathcal{H}_r \triangleleft o$ **then** **APPLY**(o)

else if $o \notin \mathcal{H}_r \wedge \mathcal{H}_r \not\triangleleft o$ **then**

$\mathcal{H}_r \leftarrow \text{RESOLVE}(o, \mathcal{H}_r)$ (resolve conflict)

Figure 1. Synchronization algorithm to integrate foreign history $\mathcal{H}_{r'}$ into $\mathcal{H}_r$.

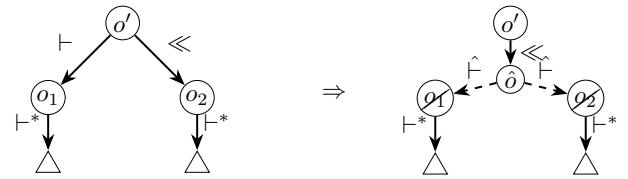


Figure 2. Conflict resolution of o_1 and o_2 with one conflicting premise o' .

to $\mathcal{H}_r$ using **APPLY**, as the premises of o are already in $\mathcal{H}_r$ due to the causal order of $\mathcal{H}_{r'}$. Otherwise, if o is not present in $\mathcal{H}_r$ and conflicts with it, i.e. $\mathcal{H}_r \not\triangleleft o$, synchronization is suspended and conflict resolution is triggered via **RESOLVE** procedure.

The resolution procedure **RESOLVE**, described below, takes as input a local history $\mathcal{H}_r$ and a conflicting remote operation $o_0 \in \mathcal{H}_{r'}$ such that $\mathcal{H}_r \not\triangleleft o_0$:

1. Find all common premises $\Gamma = \bigcup_{i \in [0, k]} \Gamma(o_i)$, where $o_1, \dots, o_k \in \mathcal{H}_r$ and $\forall i \in [1, k]: o_i \not\triangleleft o_0$.
2. Define the set of operations for conflict resolution O as operations entailed by Γ , i.e., $O = \{x \mid \exists \bar{o} \in \Gamma: \bar{o} \vdash x\}$.
3. Optionally, if a user participates in the conflict resolution and authorized to cancel operations, unneeded operations with their respective ancestors can be excluded before merging O . Therefore, O is partitioned into a set of operations to preserve $\mathcal{D}_\delta$ and operations to cancel $\mathcal{D}_\emptyset$. By default, $\mathcal{D}_\delta = O$, $\mathcal{D}_\emptyset = \emptyset$.
4. Create merge operation $\hat{o}$ with premises $\Gamma(\mathcal{D}_\delta)$ and a sequence of actions returned by an external **RECONCILE** procedure applied to the whole $\mathcal{D}_\delta$. In general, **RECONCILE** is an interactive procedure where user manually defines desired order of actions in $\hat{o}$ and, optionally, adds new actions or removes existing ones, as long as no new premise p , such that $p \vdash^* \Gamma(o_i)$, extends $\Gamma(\hat{o})$.
5. Rebase operations in $\mathcal{D}_\delta$ to $\hat{o}$ and operations in $\mathcal{D}_\emptyset$ to $o_\emptyset$.

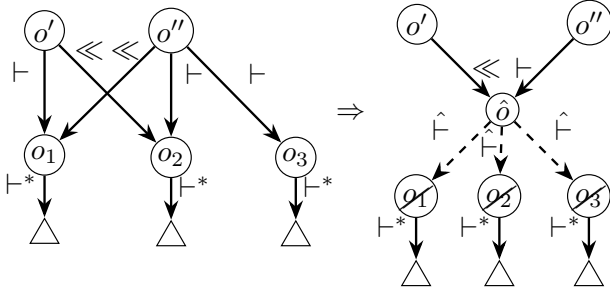


Figure 3. Conflict resolution of o_2 against o_1 and o_3 with two conflicting premises o' and o'' , and the merge operation $\hat{o}$ discarding premise o' .

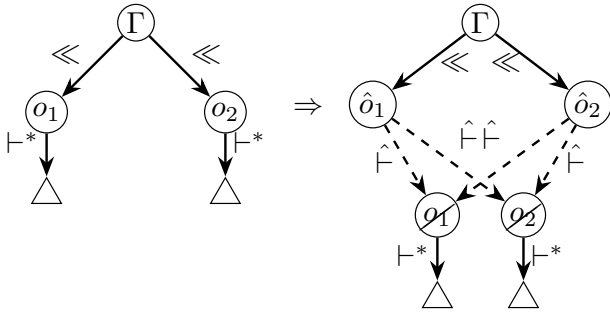


Figure 4. Conflict resolution of o_1 and o_2 with two conflicting premises o' and o'' and a conflict resolved concurrently leading to another conflict between $\hat{o}_1$ and $\hat{o}_2$.

Simple cases with one and two conflicting premises are illustrated in Figures 2 and 3 respectively. For simplicity, common premises may be depicted simply as Γ . When several operations o_1, o_2 discard same premises Γ , it means that $\exists o', o'' \in \Gamma \mid o' \ll o_1 \wedge o'' \ll o_2$.

Conflicts may be resolved concurrently, i.e., it is possible for a single operation o to be rebased to different merge operations $\hat{o}_1$ and $\hat{o}_2$ on different replicas, as illustrated in Figure 4. It is an expected situation, which may lead to necessity of further conflict resolution as an inevitable part of an articulation work.

3.3 Register Definition

Let us discuss requirements on actions and their visibility predicates to define a register suitable for our conflict model. We introduce the notion of *discard-completeness* for registers, which ensures that if an action a_1 is not visible due to another action a_2 , then a_1 is a premise for a_2 .

Definition 3.1. A register x with an action basis A is *discard-complete* if for every pair of its action instances a_1, a_2 in any history $\mathcal{H}_r(x)$ of x the following holds:

$$\text{vis}(a_1, \mathcal{H}_r(x) \setminus \{a_2\}) \wedge \neg \text{vis}(a_1, \mathcal{H}_r(x)) \Rightarrow a_1 \ll a_2.$$

Then, to define a register $x = M_i$ for our shared memory model, it is sufficient to specify:

- Action basis A_i ;
- Interpretation function $I : \mathcal{H}_r \rightarrow S_i$ mapping local history to the state $s \in S_i$;
- Arbitrary entailment rule, defining premises $\Gamma(a)$ for each action $a \in A_i$, depending on its semantics;
- Visibility predicate $\text{vis}(a, \mathcal{H}_r)$, defining when the effect of an action is visible, so that register is discard-complete.

3.4 Correctness

The presented shared memory model is not strongly convergent, as we expect conflicts to may have indefinitely arisen during synchronization. However, we may prove that if no new operations are introduced and all conflicts are resolved, the shared memory strongly converges to the same state on all replicas. We present a proof sketch which is split into two parts: firstly, we prove that the entailment graph as a graph data type satisfying *strong eventual consistency* (SEC) [15] (beyond reconciliation scenarios), and secondly, we show that each register converges to the same state in case all conflicts are resolved (assuming conflicts have been resolved in a centralized manner).

Theorem 3.2. A graph data type, equivalent to the entailment graph G_+ , which supports operations $\text{add}(\Gamma, o)$ (adding a new vertex o with its incoming edges Γ) and $\text{rebase}(o, \hat{o})$ (adding an edge to o from $\hat{o}$ not succeeding it) satisfies SEC [15], i.e., all replicas that have delivered the same updates have the equivalent states.

Proof sketch. In this proof, we will rely on the definition of add-only monotonic directed acyclic graph CRDT in [14], which models partial order graph starting from the initial edge (i_s, i_f) and provides an interface with a single method $\text{addBetween}(a, o, b)$, which adds a new vertex o with edges (a, o) and (o, b) . CRDTs are strongly convergent by their definition, so if we can model a data type using CRDT, it is strongly convergent as well.

Let us show that the operations of G_+ may be expressed via addBetween , so G_+ is a specific case of the state-based CRDT above. Initial state may be expressed as a set of edges $\{(c_i, i_f)\}$, where c_i are the register constructor operations, and i_f is a sentinel. $\text{add}(\Gamma, o)$ may be expressed as executing $\text{addBetween}(o', o, i_f)$ for every premise $o' \in \Gamma$, and $\text{rebase}(o, \hat{o})$ is equivalent to performing $\text{addBetween}(o', \hat{o}, o)$ for every premise $o' \in \Gamma(o)$.

□

We now establish that, once all conflicts are resolved, the shared memory converges to a common state across replicas. Without loss of generality, we assume that conflict resolution is carried out on a single replica, thereby excluding second-order conflicts during this phase.

Theorem 3.3. *Let all replicas freeze their histories $\mathcal{H}_{r_i}$ and have them published, so replica r synchronizes-with all of them, resulting in local state s produced by a local history $\mathcal{H}_r$ (which is induced from the entailment graph G_r). After all replicas synchronize-with published $\mathcal{H}_r$, they obtain the same state s and their histories $\mathcal{H}'_{r_i}$ are inducible from G_r .*

$$\forall i: \exists s: \forall r \in \mathcal{R}: s = \mathcal{I}_i(\mathcal{H}_r(M_i)).$$

Proof sketch. We need to prove that in absence of newly issued operations and centralized reconciliation of all replicas, which took place at r , all the replicas will acquire the same state during synchronization without calling **RESOLVE**, i.e., no further conflict resolution happens.

Let history $\mathcal{H}'$ of a replica r' be *induced*, i.e., generated as a topological sort, from G'_r before replica r *synchronized-with* r' . In terms of graphs, upon r' receives $\mathcal{H}_r$, $G'_r \sqsubseteq G_r$, as no new operations are made on r (and, consequently, no new entailment edges are added). Then, as we previously showed that an entailment graph may be modeled as a state-based CRDT, due to the mononocity $G'_r \leq G_r$ and the idempotence of least-upper-bound operator $\sqcup$ [15], $G'_r \sqcup G_r = G_r$.

The rest we need to prove is that any history induced from G_r produces the same state s which is modeled as a set of visible and invisible action instances. Due to discard-completeness of registers and absence of conflicts in G_r , if a_1 is invisible, then $\exists! a_2: a_1 \ll a_2$, i.e., $a_1 \prec a_2$. Thus, due to the happens-before order on actions that make other actions invisible in any history, all the histories produce the same state. $\square$

4 Examples

In this section, we illustrate the proposed model with several register data type definitions by specifying their actions A and their visibility predicates $vis(a, \mathcal{H}_r)$.

4.1 Register

Register is a data type that stores a single value with an action basis $A = \{mov\}$ and a query val , which is instantiated with the constructor $c_x = [mov\ 0]$. Let the state $s \in S$ be modeled as a pair (v, α) , where v is the current value of the register and α is the last action that set this value. Then an action $a = mov\ v$ is applied to local register replica x_r with $\Gamma(a) = \{\alpha(x_r)\}$ and interpreted by $\mathcal{I}$ as setting the state s to (v, a) . Getting the value is performed as follows: $val = \lambda.v$. Visibility of an action a is defined as the local state $s = \mathcal{I}(\mathcal{H}_r(x)) = (v, \alpha)$ of the register x holds the value v set by this action, i.e., $vis(a, \mathcal{H}_r(x)) \Leftrightarrow v = v \wedge \alpha = a$. Any pair of concurrent operations containing mov actions corresponding to the same register instance x are in conflict.

4.2 Arithmetic Register

Let us extend an integer register data type with arithmetic actions $A = \{mov, add, mul\}$ with an interpreter $\mathcal{I}$ producing the state $s = (\omega, v, \alpha)$, where $\omega \in \{=, +, \cdot\}$ is the current

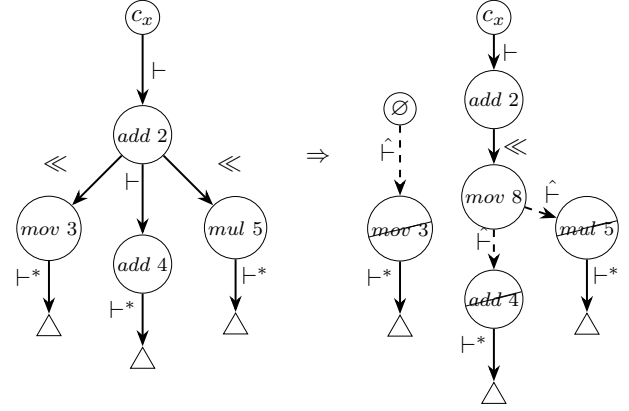


Figure 5. Conflict resolution on arithmetic register within additive context of $add\ 2$, which resulted in $mov\ 8$.

computation mode corresponding to the actions of A , v is the current initial value, and α is a sequence of actions performed on this value. Query val may then be derived as getting the aggregate of the actions in α over v , i.e., $val = \lambda. foldr\ \omega\ v\ \alpha$. Visibility of an action a then may be defined as follows: $vis(a, \mathcal{H}_r(x)) \Leftrightarrow a \in \alpha$, and $\Gamma(a) = \{last(\alpha)\}$.

Let us consider an example with a register x constructed by $c_i = [a_0]$, $a_0 = add\ 0$, which is interpreted as the state $s = (+, 0, \{a_0\})$. A following addition action $a_1 = add\ 2$ is interpreted as appending a_1 to α , i.e., $s = (+, 0, \{a_0, a_1\})$. Similarly, a multiplication action $a_2 = mul\ 5$ is interpreted as setting the computation mode $\cdot$ with the current $v = val$ and overwriting the action sequence with $\{a_2\}$, i.e., $s = (\cdot, 2, \{a_2\})$. An assignment action $a_3 = mov\ 3$ modifies the state to $s = (=, 10, \{a_3\})$, but subsequent assignments are not accumulated in the action set. A similar example is shown in Figure 5.

4.3 LWW-Register

Last-Write-Wins Register is a conflict-free replicated data type [14] that prioritizes the latest value assigned to the register in case of concurrent writes. To achieve this, each write operation $mov\ x\ t$ includes a total order timestamp $t \in \mathcal{T}$, usually derived from the wall-clock time of the issuing process. A query $val = \{v(a) \mid t(a) = \max_t(\mathcal{H}_r(x))\}$ simply returns the values associated with the greatest timestamp which follows the celebrated Thomas write rule [18].

Despite LWW-Register being seen as conflict-free, still the conflict can happen, which is known as *lost update problem*, when a write operation effect may have never been observed by a read operation due to the presence of a more recent concurrent write operation, i.e., values are silently overwritten during synchronization. Traditionally, this nuance is ignored in favor of simplicity of the data type interface, however, it is easy to resolve such situations in entails-discards model.

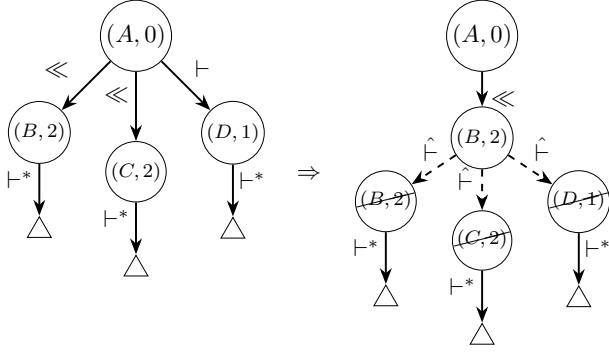


Figure 6. Conflicts in LWW-Register: $(B, 2)$ and $(C, 2)$ are concurrent operations with equal timestamps; $(D, 1)$ is suppressed by its siblings.

To identify lost updates, we specify visibility predicate as $vis(a, \mathcal{H}_r(x)) \Leftrightarrow \{a\} = \{a' \mid t(a') = \max_t(\mathcal{H}_r(x))\}$, query $val = \{v(a) \mid t(a) = \max_t(\mathcal{H}_r(x))\}$, and $\Gamma(a) = \{a \mid t(a) = \max_t(\mathcal{H}_r(x))\}$. Conflicts may be resolved automatically by the replicas in place, when the merge operation $\hat{o}$ simply replays the latest seen write operation. At the same time, equal greatest timestamps may be still defined as a conflict, as shown in Figure 6, or resolved deterministically, e.g., by comparing operation identifiers.

4.4 Multi-Register

Finally, collaboration in shared memory consisting of multiple registers can be considered, which illustrates that operations may contain actions on distinct registers.

Let us introduce a *touch* action to allow definition of custom user-defined premises for operations that span multiple registers. Action *touch* simply forces recent *mov* operation o to be included in the premises Γ of the issuing operation. Roughly speaking, this mechanism enables expression of custom optimistic locks on registers by referencing operations providing the currently observed values.

Let us consider an example with three registers corresponding to fields of a calendar event: M_1 is a string event title, M_2 is a pair of the event start and the end time modeled as unix timestamps, and M_3 is the event location. All the registers are initially instantiated by their respective constructors: $c_1 = \text{mov 'Lunch: Alice x Bob'}$, $c_2 = \text{mov 1pm-1.30pm}$, $c_3 = \text{mov 'Bambi's'}$.

Assume Alice has acquired a time slot to extend the event duration until 2pm, but Bob changes the event location concurrently, i.e., $o_A = [\text{mov } M_2 \text{ 1pm-2pm}]$ exists with $o_B = [\text{mov } M_3 \text{ Meadow's}]$. However, Alice had optimistically assumed that the event location will not be changed by others, because, otherwise, the event duration would not have been extended in o_A .

These operations are compatible, as they operate on different registers, but they can be simply adjusted, so that the

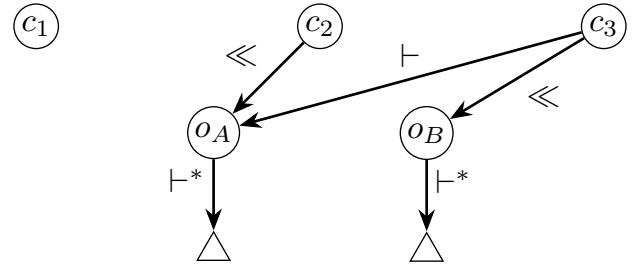


Figure 7. Multi-register example: Alice issues $o_A = [\text{touch } M_3, \text{mov } M_2 \text{ 1pm-2pm}]$, Bob issues $o_B = [\text{mov } M_3 \text{ Meadow's}]$. Conflict arises as the effect of c_3 is concurrently discarded by o_B .

intention conflict and optimistic assumption of Alice about the event location not changed is respected. To achieve this, Alice's operation o_A includes a *touch* M_3 action in o_A , so that Bob's concurrent update o_B will result in a conflict, as soon as Alice receives Bob's update, as shown in Figure 7.

5 Related work

The presented conflict model is influenced by dynamic epistemic logic [19] with a goal to revisit *intention preservation* property [17] firstly captured in consistency model for Operational Transformations [4]. As operation may be applied only when its premises are already present in the history, the conflict model implies a weaker form of causal delivery [8]. Support of multi-register data types accords with wait-free data structures in shared memory [1] and the composition of convergent and commutative replicated data types [3, 20].

The entailment graph is a monotonic semi-lattice, i.e., a *state-based replicated data type* (CvRDT) [14] and, noteworthy, it is used to represent the *operation-based* history of a data type. Mergeable replicated data types (MRDTs) [5] also present a framework for defining arbitrary data types using invertible relational specifications, where *concretization* functions are employed to resolve conflicts by ordering operations in a three-way merge manner [9]. However, conflicts are not explicitly represented neither in CRDTs nor in MRDTs, which complicates reasoning about the data type behavior in presence of concurrent updates in collaboration.

6 Conclusion

In this work, we presented a conflict model for collaborative data types based on custom defined entails-discards semantics of operations. The next step in this research is to refine the correctness proof and explore the expressiveness of the model by revisiting known non-register CRDTs, to define a general procedure for selective undo-redo operations, and implement a software framework to define custom data types and automatically prove their correctness properties.

References

- [1] J. Aspnes and M. Herlihy. 1990. Wait-free data structures in the asynchronous PRAM model. In *Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures* (Island of Crete, Greece) (SPAA '90). Association for Computing Machinery, New York, NY, USA, 340–349. doi:10.1145/97444.97701
- [2] A. Baltag and S. Smets. 2008. A qualitative theory of dynamic interactive belief revision. *Logic and the foundations of game and decision theory (LOFT 7)* 3 (2008), 9–58.
- [3] C. Baquero, P. S. Almeida, A. Cunha, C. Ferreira, et al. 2017. Composition in state-based replicated data types. *Bulletin of EATCS* 3, 123 (2017).
- [4] C. A. Ellis and S. J. Gibbs. 1989. Concurrency control in groupware systems. *SIGMOD Rec.* 18, 2 (June 1989), 399–407. doi:10.1145/66926.66963
- [5] G. Kaki, S. Priya, KC Sivaramakrishnan, and S. Jagannathan. 2019. Mergeable replicated data types. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 154 (Oct. 2019), 29 pages. doi:10.1145/3360580
- [6] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan. 2019. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Athens, Greece) (Onward! 2019). Association for Computing Machinery, New York, NY, USA, 154–178. doi:10.1145/3359591.3359737
- [7] P. Kuznetsov, M. Perion, and S. Tucci-Piergiovanni. 2025. Wait-free Replicated Data Types and Fair Reconciliation. arXiv:2508.18193 [cs.DC] <https://arxiv.org/abs/2508.18193>
- [8] L. Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. doi:10.1145/359545.359563
- [9] T. Mens. 2002. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering* 28, 5 (2002), 449–462. doi:10.1109/TSE.2002.1000449
- [10] N. Preguiça, M. Shapiro, and C. Matheson. 2003. Semantics-Based Reconciliation for Collaborative and Mobile Environments. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, R. Meersman, Z. Tari, and D. C. Schmidt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 38–55.
- [11] Y. Saito and M. Shapiro. 2005. Optimistic replication. *ACM Comput. Surv.* 37, 1 (March 2005), 42–81. doi:10.1145/1057977.1057980
- [12] K. Schmidt and L. Bannon. 1992. Taking CSCW seriously. *Computer Supported Cooperative Work (CSCW)* 1, 1 (1992), 7–40. doi:10.1007/BF00752449
- [13] V. Semenov. 2007. Semantics-based Reconciliation of Divergent Replicas in Advanced Concurrent Engineering Environments. In *Complex Systems Concurrent Engineering*, G. Loureiro and R. Curran (Eds.). Springer London, London, 557–564.
- [14] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages. <https://inria.hal.science/inria-00555588>
- [15] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400.
- [16] M. Shapiro, A. Rowstron, and A. Kermarrec. 2000. Application-independent reconciliation for nomadic applications. In *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System* (Kolding, Denmark) (EW 9). Association for Computing Machinery, New York, NY, USA, 1–6. doi:10.1145/566726.566728
- [17] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. 1998. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.* 5, 1 (March 1998), 63–108. doi:10.1145/274444.274447
- [18] R. H. Thomas. 1979. A Majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.* 4, 2 (June 1979), 180–209. doi:10.1145/320071.320076
- [19] H. Van Ditmarsch, W. van Der Hoek, and B. Kooi. 2008. *Dynamic epistemic logic*. Springer.
- [20] M. Weidner, H. Miller, and C. Meiklejohn. 2020. Composing and decomposing op-based CRDTs with semidirect products. *Proc. ACM Program. Lang.* 4, ICFP, Article 94 (Aug. 2020), 27 pages. doi:10.1145/3408976
- [21] W. E. Weihl. 1988. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.* 37, 12 (1988), 1488–1505. doi:10.1109/12.9728