Electronic Communications of the EASST
Volume 82 (2022)

11th International Symposium
on Leveraging Applications of Formal Methods, Verification and Validation
-
Doctoral Symposium, 2022

# Lazy Merging:
# From a Potential of Universes to a Universe of Potentials

Jonas Schürmann and Bernhard Steffen

24 pages

Guest Editors: Sven Jörges, Salim Saay, Steven Smyth
ECEASST Home Page: http://www.easst.org/eceasst/
ISSN 1863-2122

# Lazy Merging: From a Potential of Universes to a Universe of Potentials

**Jonas Schürmann and Bernhard Steffen**

TU Dortmund University

**Abstract:** Current collaboration workflows force participants to resolve conflicts eagerly, despite having insufficient knowledge and not being aware of their collaborators' intentions. This is a major reason for bad decisions because it can disregard opinions within the team and cover up disagreements. In our concept of lazy merging we propose to aggregate conflicts as variant potentials. Variant potentials preserve concurrent changes and present the different options to the participants. They can be further merged and edited without restrictions and behave robustly even in complex collaboration scenarios. We use lattice theory to prove important properties and show the correctness and robustness of the collaboration protocol. With lazy merging, conflicts can be resolved deliberately, when all opinions within the team were explored and discussed. This facilitates alignment among team members and prepares them to arrive at the best possible decision that considers the knowledge of the whole team.

**Keywords:** Collaboration Systems, Version Control, Software Merging, Conflict Handling, Conflict Tolerance, Lattice Theory

## 1 Introduction

When teams collaborate, there is the need to synchronize, but there is also the need to work concurrently and independent of each other. Collaboration systems fulfill both of these needs with the concept of branching & merging. Branches create workspaces isolated from the current main version in which participants can work concurrently without interruption. They are generally used to carry out cohesive tasks, before they are merged back into the main version [BBR+12]. During merging, conflicts may arise from concurrent edit operations. Handling these conflicts is a major challenge in the implementation of collaboration systems [Men02].

Contemporary version control systems (VCSs, e.g., Git[1]) follow what we call an eager merging strategy. All the conflicts that have been built up between branches must be resolved immediately when branches are merged, by the participant who initiated the merge [CS14]. The participant has to stop what they are currently doing and is faced with the difficult task of working out a resolution on their own. To make things worse, conflict representations are often verbose and inaccurate [Men02]. This "integration hell" [PSW11] is a common problem in branching & merging workflows that contributes to the fact that VCSs are predominantly used by technical experts.

To avoid these challenges, many popular applications abandon branches entirely and decide to use simple live collaboration protocols instead. This approach was also chosen for the graphical

[1] https://git-scm.com/

modeling environment Cinco Cloud that is developed at our chair. Participants are required to always work online in a single commonly shared workspace to minimize the possibility for conflicts. If conflicts occur, they are arbitrarily resolved to avoid interrupting participants in their work [ZNS19]. This improves usability and makes the systems accessible to non-experts. However, it gravely restricts the way participants can operate as they can no longer work offline or in isolation from others.

We propose the concept of lazy merging to overcome these problems. Instead of eagerly resolving conflicts during merging or attempting to avoid them altogether, conflicts are regular parts of the collaborative documents. When conflicts arise during merging, they are preserved and aggregated in variant potentials, so that they can be explained to the user and shared with other participants. Moreover, they can stay in the documents and do not block ongoing work. In this way, we can retain usability in the presence of branching & merging, enable participants to communicate conflicts effectively, and grant participants more freedom in their work.

Section 2 motivates the benefits of lazy merging with a real-world scenario. Then Section 3 discusses lazy merging and its advantages more generally. It describes how conflicts are represented by variant potentials and how lazy merging supports version control. Section 4 provides a detailed description and analysis of the lazy merging system, including a mathematical formalization based on lattice theory. After that, Section 5 discusses how lazy merging changes the status quo of collaboration systems. Section 6 compares lazy merging with relevant related work. Finally, Section 7 lists many opportunities for future research before the paper is concluded in Section 8.

## 2 Real-World Scenario

The following exemplary collaboration scenario gives an initial idea of what is possible with lazy merging. Figure 1 shows how Alice and Bob collaboratively design a flow chart for a software quality assurance process. After they both add their own ideas, Alice can bring in Bobs contribution and the conflicting changes are represented in the document. They both wanted to add a node in the same place, so the edges split up and show the two variants. Alice does not need to resolve this conflict immediately, she can keep on working and adds another node to the flow graph. Then she directs her attention to the conflict. Because she is unsure about the best resolution, she shares the conflict with Bob. Bob receives Alice's conflicting version and works out a resolution. Then Bob pushes the resolved version back to Alice, bringing both to the same final state.

With lazy merging conflicts are a regular part of the collaborative document, so Alice and Bob were able to work more freely compared to traditional eager merging systems. Alice was able to bring in Bob's changes without interrupting her work, postponing the conflict resolution to a later time. When she started to work on the resolution, she could transfer the conflicting version over the collaboration system to Bob. This brought them on the same page and Bob could work out a resolution for both of them.

In an eager merging system, Alice would have had to immediately stop her work when she brought in Bob's contribution and resolve the conflict. To discuss the conflict with Bob, she would have to explain it to him out-of-band, e.g., via telephone or screen sharing. And until Bob is available to talk, she would have been blocked from working.

Figure 1: Lazy merging represents conflicts within the collaborative document.

Because this coordination is so cumbersome, Alice could even be compelled to resolve the conflict on her own. She would either pick an arbitrary order as she sees fit, or completely discard Bob's contribution. The result would be erroneous and disregard Bob's opinion and expertise.

# 3 Lazy Merging

It is common to consider conflicts from concurrent editing as abnormal or exceptional. So there is often the attempt to resolve conflicts as fast as possible. VCSs generally force participants to resolve all conflicts before the merge process can be completed. And live collaboration systems usually resolve conflicts arbitrarily (e.g., using last-write-wins semantics). These eager merging strategies can uphold syntactic consistency at all times. But as shown by the previous example, it restricts how participants work, causes interruptions, and does not support collaborative resolution.

## 3.1 Conflicts as Variant Potentials

Lazy merging is a fundamental shift in the perspective on conflicts in collaboration systems. Conflicts are not problems that have to be resolved at the moment they come up. Every conflict is a potential of possible variants, where each variant represents a meaningful opinion deserving proper consideration.

For that reason, variant potentials are preserved and edited as regular elements of the document. The document model is generalized to allow for variant potentials in all units. Merging becomes lazy because the resolution of conflicts is deferred to a later time, and conflicts are resolved when it is required or suitable.

Through this laziness, merging and conflict resolution become two completely independent operations. Both are always available and safe to use, giving participants new freedoms in the way they work. Merging collects all changes without data loss and is fully automated. Conflict resolution discards variants, so it is made explicitly by participants, and it is recorded and applied separately.

## 3.2 Lazy Version Control

Branching & merging is a key feature provided by version control systems. Branching workflows have become popular in software development because they minimize the frequency of task interruptions in concurrent development efforts [BBR+12]. This is critical for teams with more than a few members because task interruptions diminish the productivity of participants [BBR+12]. The big challenge with branching & merging lies in the conflicts that arise when concurrent development efforts are carried out in isolation. They have to be detected, represented, and reconciled after branches are merged back together.

State-of-the-art VCSs employ eager merging strategies to resolve these conflicts, but that incurs severe limitations. Participants are forced to resolve all conflicts they encounter immediately during the merge before they can continue to work. But often they are unaware of the intentions and perspectives of their team members. As a result, they find themselves unable to produce satisfactory resolutions on their own [WFSW11]. Moreover, conflicts arise at points of disagree-

ments, but if they are not preserved participants do not become aware of their differences in opinion [WFSW11].

Lazy merging solves these problems with the introduction of variant potentials, which aggregate and preserve all conflicts that occur, instead of forcing participants to resolve them immediately. Because variant potentials are regular parts of the document, they can be distributed to all participants. When conflicts are preserved this way, they can help uncover disagreements within the team and guide the discussion to mediate the different opinions. The burden of making a resolution decision does not lie with one unlucky participant. Everyone can take a look at the aggregated variant potentials, creating awareness and understanding within the team. The resolution can be worked out collaboratively with everyone who is affected by the conflicts. This results in a more inclusive conflict resolution process, where all opinions are considered, preventing the omission of important facts or ideas.

With lazy merging, participants can more flexibly work with contributions. They can always bring in new contributions to work with, and postpone the resolution of conflicts. Variant potentials can just stay in the documents without affecting work in other places, so there is no pressure to resolve them. This allows participants to wait for additional information regarding the resolution decision and prevents interruptions when new contributions are brought in.

Finally, lazy merging creates an auditable record of all conflicts that occurred and their resolutions. This is different from current state-of-the-art VCSs where only the resolved version is committed at the end of the merge process, and conflicts are completely ephemeral [WFSW11]. In contrast, lazy merging preserves all conflicts as variant potentials that are committed as a proper new version and records resolution decisions separately. From this history, participants can accurately understand what conflicts occurred and how they were resolved, for the complete history of the project.

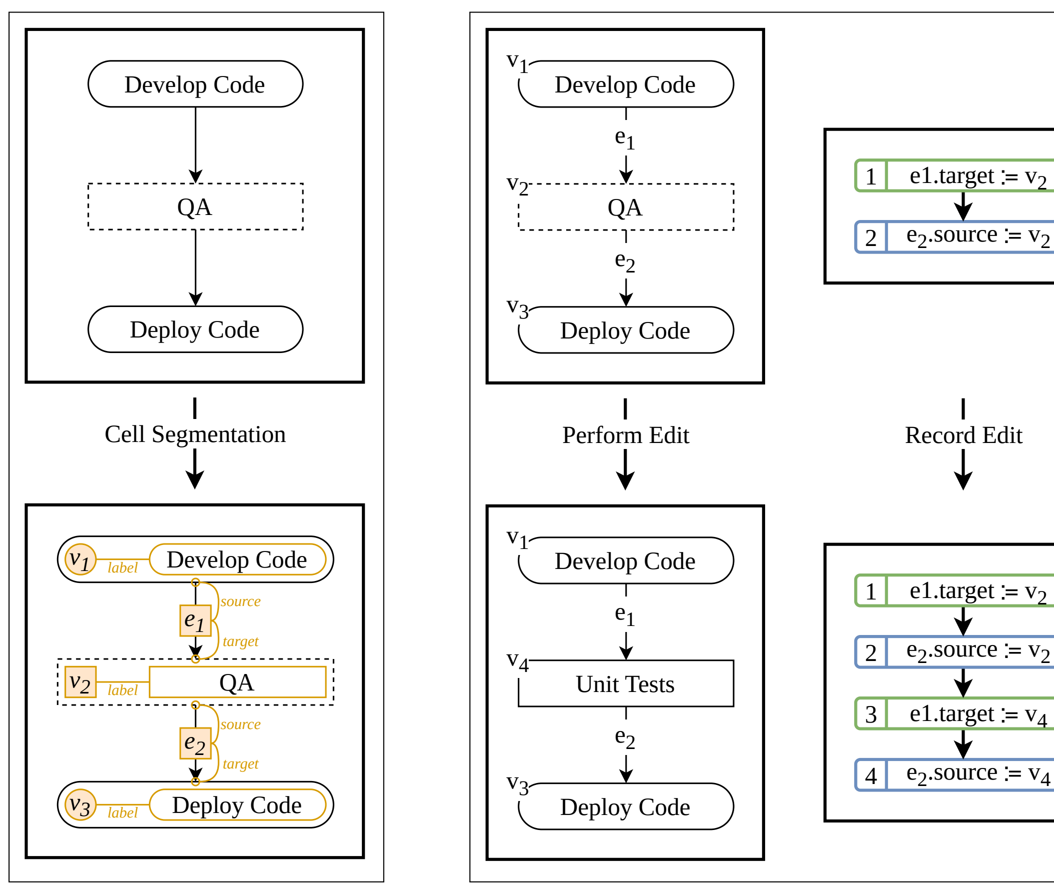
Naturally, participants must balance laziness and eagerness appropriately. While forced eagerness can isolate participants, lead to premature decisions, and cause interruptions, too many variant potentials accumulating in a document can make it incomprehensible.

### 3.3 Cell Segmentation

To ensure the reliable localization of changes, lazy merging applies a segmentation of uniquely identifiable, persistent cells to the collaborative document. These cells represent the granularity at which changes are captured and conflicts are detected. Whenever a cell is affected by the changes of a participant, the new value of the cell is captured as a tiny snapshot. Cells determine what concurrent changes are considered to be in conflict: Concurrent updates to the same cell are a conflict, but concurrent updates to different cells are fine.

Figure 2a shows the segmentation of a graph model. The example uses a fine granularity, where every property of the nodes and edges is captured in its own cell. Because each node and edge is given a unique identifier, the cells can always be reliably distinguished from another. This allows for precise merging and conflict detection. Even when versions with a big time difference are merged, changes are located just as precisely, because cells are persistent over time.

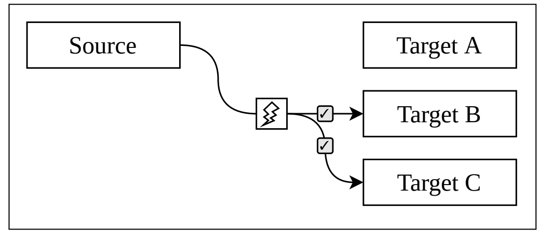
Figure 2b illustrates how edits on the graph model are captured in a causality graph of operations based on this segmentation. The edges are moved from the “QA” placeholder node $v_2$ to the “Unit



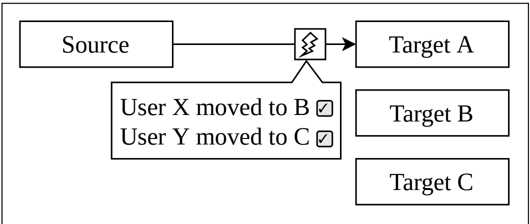

(a) Applying a persistent cell segmentation with unique identifiers.

(b) Recording cell assignments.

Figure 2: The cell segmentation is the basis for change recording.




(a) Showing a generalized view.




(b) Showing the annotated last consensus.

Figure 3: Editors can provide different projections of variant potentials.

Tests" node $v_4$, which causes assignments in the cells for their target and source property. Other operations on graph models could be modeled similarly:

- All simple properties can be captured in cells as well (labels, colors, etc.).
- Moving an object into a different container is an assignment to its parent cell.
- Deleting and restoring an object is an assignment to a boolean-valued cell for its presence.

When we discuss operation causality in the remaining section, we will use the generic variables $x$ and $y$ to identify cells for brevity.

### 3.4 Lazy Editing

The application of a cell segmentation and variant potentials in each cell lead to an enriched document model. Syntactical elements need to be augmented with persistent unique identifiers to support the cell segmentation, and cells must have the capacity to store variant potentials. Plaintext programming languages and editors are neither capable of maintaining persistent identifiers for syntactical elements, nor can they represent variant potentials in their syntax. Therefore, lazy merging needs a structure editor that is able to maintain the cell segmentation when the user edits the collaborative documents. The editor also interleaves the display of the document with visualizations of variant potentials, so that participants can inspect and resolve conflicts in place.

Additionally, this editor can provide different projections on the variant potentials. Figure 3 shows two possible projections for a variant potential in a target cell of an edge. The first projection renders all possibilities simultaneously. The second projection displays the last consensus and lists the variants in a popover. Depending on the situation, participants can choose a projection that supports them best.

The editor records all user input and saves the resulting assignments to cells in the causality graph. This is a simple and efficient way to record changes that does not require running diffing algorithms on the whole document. The editor could also provide a stream of operations as they happen, to replicate them in real time between participants to enable live collaboration.

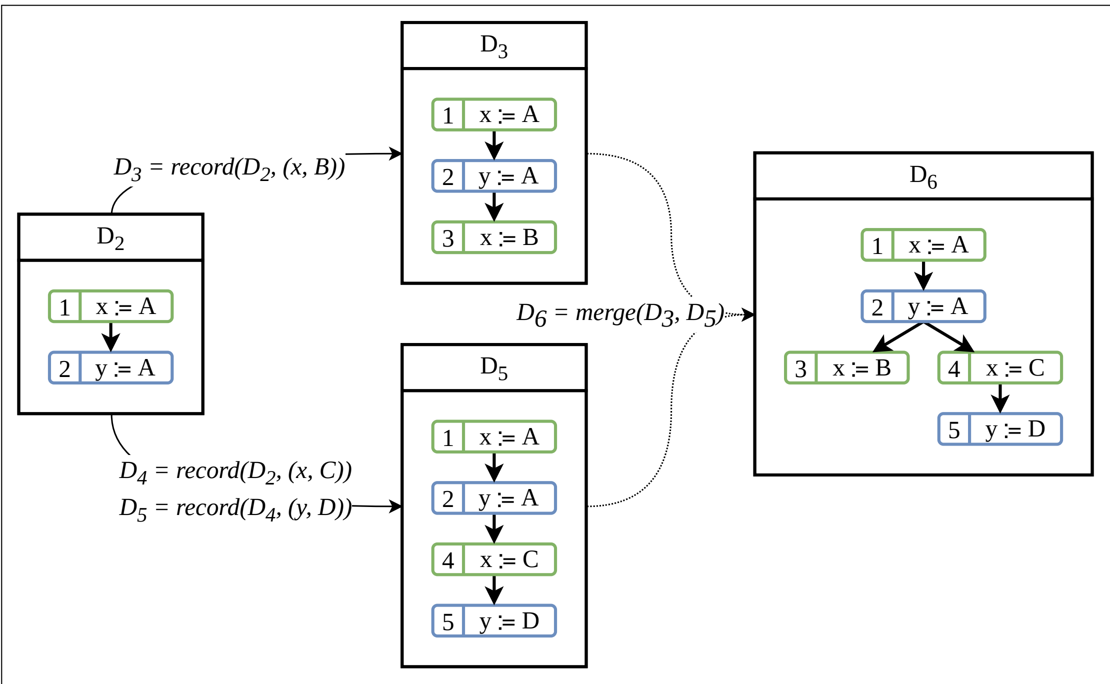



Figure 4: Recording and merging document causality graphs.

# 4 Recording and Aggregating Changes

This section discusses how changes are recorded and aggregated following the causality of edit operations. Important properties are proven, giving strong confidence in the behavior of the system.

## 4.1 Document Causality Graphs

The causality of edit operations is the key to properly capture the intentions of participants. We follow Lamport's definition of the happened-before relation [Lam78] when we record the local perspective on causality for each participant in document causality graphs. Document causality graphs capture the causality of operations in a strict order. When a new operation is recorded, it is considered to have happened after all the existing operations in the local document causality graph. The collaboration system provides globally unique identifiers for each recorded operation. We assume that identifiers are generated in an ascending total order to simplify our proofs, although in practice it is sufficient to use randomly generated identifiers. This preserves the global consistency of the strict order when concurrently evolving document causality graphs are merged with a union. Figure 4 shows the concurrent recording of operations and the merging of the document causality graphs in an example.

**Definition 1** (Globally consistent document causality graphs) If $I$ is a set of totally ordered identifiers, $C$ is a set of cells, and $V$ is a set of values, then $\mathscr{O}_\mathscr{D} = I \times (C \times V)$ is the set of all possible operations, tagged with identifiers.

Document causality graphs $\mathscr{D}$ are a graph $\langle O, \rightarrow \rangle$, where $O \subseteq \mathscr{O}_\mathscr{D}$, and $(\rightarrow) \subseteq O \times O$. Within a collaboration system, they can be constructed according to the following rules:

1. $\langle \emptyset, \emptyset \rangle \in \mathscr{D}$.
2. If $\langle O, \rightarrow \rangle \in \mathscr{D}$, given an assignment $(c, v) \in C \times V$, and the collaboration system generates a fresh globally unique identifier $i$ that is strictly greater than all previously generated identifiers, then $o = (i, (c, v))$ is a new operation record, and so
$$record_\mathscr{D}(\langle O, \rightarrow \rangle, (c, v)) = \langle O \cup \{o\}, (\rightarrow) \cup (O \times \{o\}) \rangle \in \mathscr{D}$$
3. If $\langle O_1, \rightarrow_1 \rangle, \langle O_2, \rightarrow_2 \rangle \in \mathscr{D}$, then
$$merge_\mathscr{D}(\langle O_1, \rightarrow_1 \rangle, \langle O_2, \rightarrow_2 \rangle) = \langle O_1 \cup O_2, (\rightarrow_1) \cup (\rightarrow_2) \rangle \in \mathscr{D}$$

**Lemma 1** *If an operation is in a document causality graph, the strict order includes all of its global lower bounds.*

*Proof.* Operations are recorded with all of their lower bounds, and there is no way to add or remove a lower bound. □

**Theorem 1** *For every $\langle O, \rightarrow \rangle \in \mathscr{D}$, $\rightarrow$ is a strict order.*

*Proof.* The total order of identifiers is a topological order for $\rightarrow$, so it is irreflexive and antisymmetric. $\rightarrow$ is also transitive due to Lemma 1. □

We introduce the idea of *conservative extension* to describe how document causality graphs can evolve over time. Document causality graphs are by construction an append-only data structure, once an operation is recorded it stays in the graph. So conservative extension requires regular inclusion ($\subseteq$) between the operations of document causality graphs. Additionally, causality dependencies of existing operations cannot be changed. No predecessors of an existing operation are allowed to be added or removed. So new assignment operations can only be added after or next to the existing ones, which is in line with the general behavior of causality. One document causality graph is a valid successor of another if and only if it is a conservative extension.

**Definition 2** (Conservative extension of document causality graphs) Let $\langle O_1, \rightarrow_1 \rangle, \langle O_2, \rightarrow_2 \rangle \in \mathscr{D}$. Then the conservative extension of document causality graphs $\sqsubseteq_\mathscr{D}$ is defined as:

$$\langle O_1, \rightarrow_1 \rangle \sqsubseteq_\mathscr{D} \langle O_2, \rightarrow_2 \rangle =_{df} O_1 \subseteq O_2 \wedge (\rightarrow_1) \subseteq (\rightarrow_2) \\ \wedge \forall y \in O_1 . \{x \mid x \rightarrow_1 y\} = \{x \mid x \rightarrow_2 y\}$$

**Theorem 2** record$_\mathscr{D}$ *and* merge$_\mathscr{D}$ *perform conservative extensions.*

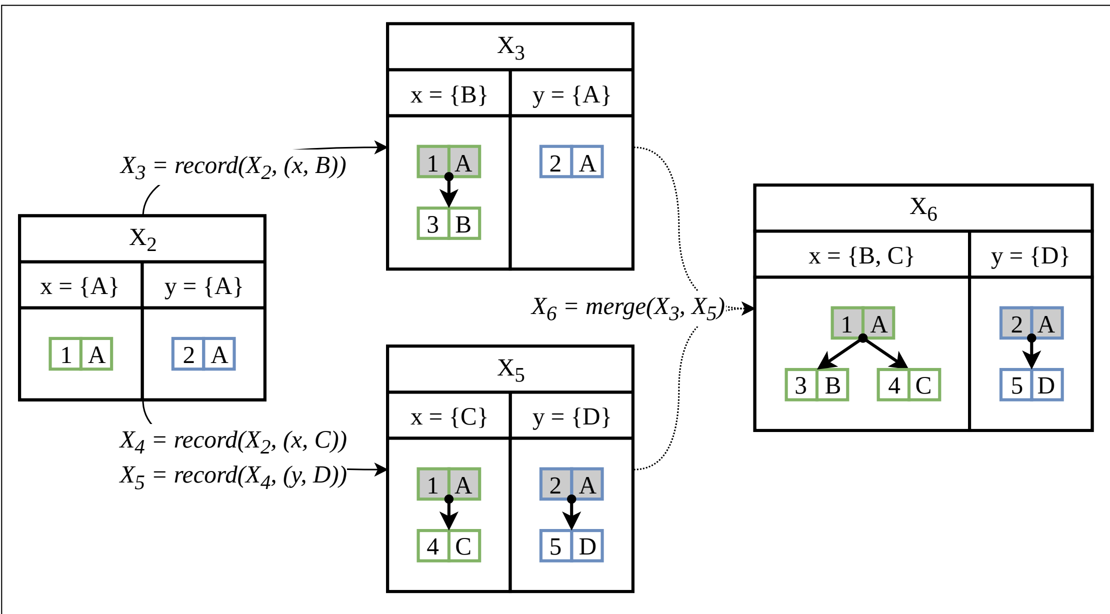



Figure 5: Recording and merging cartesian causality decompositions.

*Proof.* Both recording and merging include the given document causality graphs in the union they form. And due to Lemma 1, the set of predecessors cannot change. □

**Theorem 3** $\langle \mathscr{D}, \sqsubseteq_{\mathscr{D}} \rangle$ *is a partial order and a lattice of document causality graphs with the pairwise union/intersection as the supremum/infimum.*

*Proof.* $\sqsubseteq_{\mathscr{D}}$ is obviously reflexive, and because it requires the inclusion for both sets it is antisymmetric. $\sqsubseteq_{\mathscr{D}}$ is also transitive, because if the predecessors do not change from $a \rightarrow b$ and from $b \rightarrow c$, they neither do change from $a \rightarrow c$. Document causality graphs are pairs of power set elements, so their union is the supremum. And this supremum always exists as the $merge_{\mathscr{D}}$ that we demand in the construction. The intersection of document causality graphs is naturally their infimum. □

**Observation 1.** *Merges of document causality graphs are suprema, and infima are the point at which concurrent evolutions diverged.*

## 4.2 Cartesian Causality Decompositions

Document causality graphs capture the full causal order of all operations, but they do not immediately show the current valuation of each cell. Cartesian causality decompositions record the causal graphs separately for each cell, as shown in Figure 5. They provide a perspective where the current state of the document is easy to determine, because the valuation for each cell is obvious from the decomposed operations.

**Definition 3** (Globally consistent cartesian causality decompositions) If $I$ is a set of totally ordered identifiers and $V$ is a set of values, then $\mathscr{O}_{\mathscr{X}} = I \times V$ is the set of all possible value assignments, tagged with identifiers.

If $C$ is a set of cells, then globally consistent cartesian causality decompositions $\mathscr{X}$ are a tuples of causality graphs $(\langle O_c, \rightarrow_c \rangle)_{c \in C}$, where each $O_c \subseteq \mathscr{O}_{\mathscr{X}}$, and each $(\rightarrow_c) \subseteq O_c \times O_c$. Within a collaboration system, they can be constructed according to the following rules:

1. $(\langle \emptyset, \emptyset \rangle)_{c \in C} \in \mathscr{X}$.

2. If $X \in \mathscr{X}$, given an assignment $(c_1, v) \in C \times V$, and the collaboration system generates a fresh globally unique identifier $i$ that is strictly greater than all previously generated identifiers, then $o = (i, v)$ is a new value assignment record, and so

$$record_{\mathscr{X}}(X, (c_1, v)) = \left( \left\{ \begin{array}{ll} \langle O_c \cup \{o\}, (\rightarrow_c) \cup (O_c \times \{o\}) \rangle & \text{if } c = c_1 \\ \langle O_c, \rightarrow_c \rangle & \text{otherwise} \end{array} \right\} \right.$$
$$\left. \mid \langle O_c, \rightarrow_c \rangle = X_c \right)_{c \in C} \in \mathscr{X}$$

3. If $X_1, X_2 \in \mathscr{X}$, then

$$merge_{\mathscr{X}}(X_1, X_2) = (\langle O_1 \cup O_2, (\rightarrow_1) \cup (\rightarrow_2) \rangle$$
$$\mid \langle O_1, \rightarrow_1 \rangle = (X_1)_c \wedge \langle O_2, \rightarrow_2 \rangle = (X_2)_c)_{c \in C} \in \mathscr{X}$$

**Lemma 2** *If an operation is in one causality graph of a cartesian causality decomposition, its strict order includes all global lower bounds.*

*Proof.* The rationale of Lemma 1 applies the same way for cartesian causality decompositions. □

**Theorem 4** *For every $X \in \mathscr{X}$, $c \in C$, and $\langle O_c, \rightarrow_c \rangle = X_c$, $\rightarrow_c$ is a strict order.*

*Proof.* Because the causality graphs of cartesian causality decompositions are kept completely separate, the rationale of Theorem 1 carries over, considering Lemma 2. □

We establish the rule that within the causality graphs, an operation supersedes all the operations that happened before it. This rule leads to sensible and intuitive behavior, because participants acknowledge all previous operations as they work on the current version. So the leafs in the causality graphs determine the current valuation of each cell, as they are maximal within the strict order. Usually, there is only one leaf which determines the value, unless conflicts occur. Then each of the leafs represents one conflicting variant. In this case, the cell contains multiple values.

**Definition 4** (Cell valuations) Each cell is valued with the set of values in the leaf nodes:

$$valuations : \mathscr{X} \rightarrow \prod_{c \in C} \mathfrak{P}(V)$$
$$valuations(X) = (\{v \mid (i, v) \in O_c \wedge (\forall o \in O_c.(i, v) \not\rightarrow_c o) \wedge \langle O_c, \rightarrow_c \rangle = X_c)_{c \in C}$$

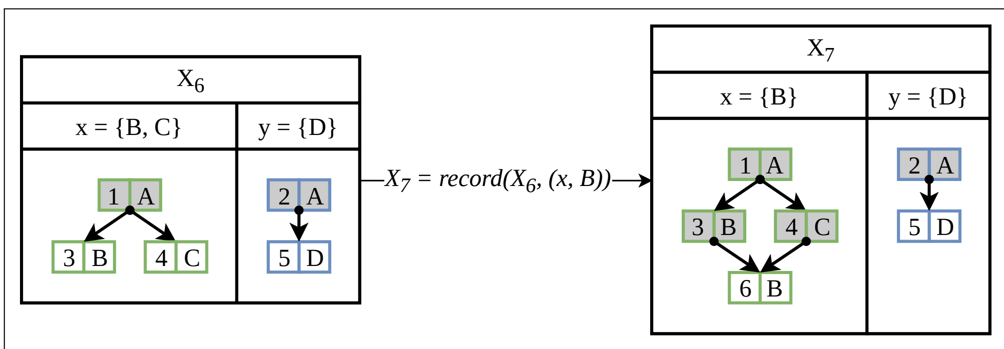



Figure 6: Resolving variant potentials with a regular assignment.

After merging, new variant potentials may arise. To resolve them, participants can simply assign a new value to the respective cell (cf. Figure 6). The assignment will supersede all existing variants and resolve the potential.

Cartesian causality decompositions show that all resolution decisions are permanent. The decision supersedes all previous variants, and because assignments cannot be removed once they are recorded there is no way to bring them back. So no matter how intertwined further merges are, past variants are guaranteed to never accidentally reappear.

**Definition 5** (Conservative extension of cartesian causality decompositions) Let $X_1, X_2 \in \mathscr{X}$. Then the conservative extension of cartesian causality decompositions $\sqsubseteq_{\mathscr{X}}$ is the product order of conservative extension on causality graphs:

$$\begin{aligned} X_1 \sqsubseteq_{\mathscr{X}} X_2 = \forall c \in C. O_1 \subseteq O_2 &\wedge (\rightarrow_1) \subseteq (\rightarrow_2) \\ &\wedge \forall y \in O_1. \{x \mid x \rightarrow_1 y\} = \{x \mid x \rightarrow_2 y\}, \\ &\text{where } \langle O_1, \rightarrow_1 \rangle = (X_1)_c \wedge \langle O_2, \rightarrow_2 \rangle = (X_2)_c \end{aligned}$$

**Theorem 5** $\mathrm{record}_{\mathscr{X}}$ *and* $\mathrm{merge}_{\mathscr{X}}$ *perform conservative extensions.*

*Proof.* The rationale of Theorem 2 applies to the product of causality graphs as well, considering Lemma 2. □

**Theorem 6** $\langle \mathscr{X}, \sqsubseteq_{\mathscr{X}} \rangle$ *is a partial order and a lattice of cartesian causality decompositions with the product of unions/intersections as the supremum/infimum.*

*Proof.* The reasoning for $\sqsubseteq_{\mathscr{X}}$ being a partial order is analogous to Theorem 3: It is obviously reflexive, it is antisymmetric because the inclusion is required for all sets, and it is transitive because the preservation of predecessors is transitive. Because $\sqsubseteq_{\mathscr{X}}$ is a product order, the supremum of cartesian causality decompositions is the product of the causality graph suprema. As of Theorem 3, that is the product of the causality graph unions, which always exists as $merge_{\mathscr{X}}$. The product of intersections is naturally the infimum. □

**Observation 2.** *Merges of cartesian causality decompositions are suprema, and infima are the point at which concurrent evolutions diverged.*

### 4.3 Cartesian Lifting

Cartesian lifting can derive the corresponding cartesian causality decompositions from document causality graphs. It separates the causality graph by the cell that each operation is applied to. Figure 7 shows how a whole history of document causality graphs is projected onto a history of cartesian causality decompositions.

**Definition 6** (Cartesian lifting) The set of operations is partitioned by the cell they are applied to, so for each cell $c$ there is a set of operations $O_c$:

$$O_c = \{(i,v) \mid (i,(c_1,v)) \in O \wedge c_1 = c\}$$

Separate causality relations $\rightarrow_c$ are created for each cell $c$, which only capture the relationships between operations on this cell:

$$(i_1,v_1) \rightarrow_c (i_2,v_2) = (i_1,(c_1,v_1)) \rightarrow (i_2,(c_2,v_2)) \wedge c_1 = c_2 = c$$

Then cartesian lifting is defined as:

$$lift : \mathscr{D} \rightarrow \mathscr{X}$$
$$lift(\langle O,\rightarrow\rangle) = (\langle O_c,\rightarrow_c\rangle)_{c\in C}$$

**Theorem 7** *Cartesian causality decompositions produced by cartesian lifting are globally consistent.*

*Proof.* Because the operations are uniquely identified within the whole document causality graph, they are also uniquely identified in each subset that is formed by cartesian lifting. And cartesian lifting maps every construction step of document causality graphs onto a valid construction step of cartesian causality decompositions. □

**Observation 3.** *Cartesian lifting maps the empty document causality graph to the empty cartesian causality decomposition.*

$$lift(\langle\emptyset,\emptyset\rangle) = (\langle\emptyset,\emptyset\rangle)_{c\in C}$$

**Theorem 8** *Cartesian lifting is a record homomorphism (cf. Figure 9). Let $\langle O,\rightarrow\rangle \in \mathscr{D}$ and $(c,v) \in (C \times V)$, then:*

$$lift(record_{\mathscr{D}}(\langle O,\rightarrow\rangle,(c,v))) = record_{\mathscr{X}}(lift(\langle O,\rightarrow\rangle),(c,v))$$

Figure 7: Cartesian lifting derives the cartesian causality decompositions from the document causality graphs.

*Proof.*

$$
\begin{aligned}
& \mathit{lift}(\mathit{record}_{\mathscr{D}}(\langle O, \rightarrow \rangle, (c_1, v))) \\
=\ & \mathit{lift}(\langle O \cup \{(i,(c_1,v))\}, (\rightarrow) \cup (O \times \{(i,(c_1,v))\})\rangle) && \mid \text{Def. } \mathit{record}_{\mathscr{D}} \\
=\ & (\langle (O \cup \{(i,(c_1,v))\})_c, ((\rightarrow) \cup (O \times \{(i,(c_1,v))\}))_c \rangle)_{c \in C} && \mid \text{Def. } \mathit{lift} \\
=\ & \left( \left\{ \begin{array}{ll} \langle O_c \cup \{(i,v)\}, (\rightarrow_c) \cup (O_c \times \{(i,v)\})\rangle & \text{if } c = c_1 \\ \langle O_c, \rightarrow_c \rangle & \text{otherwise} \end{array} \right\} \right. \\
& \mid \langle O_c, \rightarrow_c \rangle = X_c)_{c \in C} && \mid \text{Def. } O_c, \rightarrow_c \\
=\ & \mathit{record}_{\mathscr{X}}(X, (c_1, v)) && \mid \text{Def. } \mathit{record}_{\mathscr{X}} \\
=\ & \mathit{record}_{\mathscr{X}}(\mathit{lift}(\langle O, \rightarrow \rangle), (c_1, v)) && \mid \text{Def. } \mathit{lift} \quad \square
\end{aligned}
$$

**Theorem 9** *Cartesian lifting is a merge homomorphism (cf. Figure 10). Let* $\langle O_1, \rightarrow_1 \rangle, \langle O_2, \rightarrow_2 \rangle \in \mathscr{D}$, *then:*

$$\mathit{lift}(\langle O_1, \rightarrow_1 \rangle \sqcup_{\mathscr{D}} \langle O_2, \rightarrow_2 \rangle) = \mathit{lift}(\langle O_1, \rightarrow_1 \rangle) \sqcup_{\mathscr{X}} \mathit{lift}(\langle O_2, \rightarrow_2 \rangle)$$

*Proof.*

$$
\begin{aligned}
& \mathit{lift}(\mathit{merge}_{\mathscr{D}}(\langle O_1, \rightarrow_1 \rangle, \langle O_2, \rightarrow_2 \rangle)) \\
=\ & \mathit{lift}(\langle O_1 \cup O_2, (\rightarrow_1) \cup (\rightarrow_2) \rangle) && \mid \text{Def. } \mathit{merge}_{\mathscr{D}} \\
=\ & (\langle (O_1 \cup O_2)_c, ((\rightarrow_1) \cup (\rightarrow_2))_c \rangle)_{c \in C} && \mid \text{Def. } \mathit{lift} \\
=\ & (\langle O_{1_c} \cup O_{2_c}, (\rightarrow_{1_c}) \cup (\rightarrow_{2_c}) \rangle)_{c \in C} && \mid \text{Def. } O_c, \rightarrow_c \\
=\ & \mathit{merge}_{\mathscr{X}}((\langle O_{1_c}, \rightarrow_{1_c} \rangle)_{c \in C}, (\langle O_{2_c}, \rightarrow_{2_c} \rangle)_{c \in C}) && \mid \text{Def. } \mathit{merge}_{\mathscr{X}} \\
=\ & \mathit{merge}_{\mathscr{X}}(\mathit{lift}(\langle O_1, \rightarrow_1 \rangle), \mathit{lift}(\langle O_2, \rightarrow_2 \rangle)) && \mid \text{Def. } \mathit{lift} \quad \square
\end{aligned}
$$

**Corollary 1** *Because merges are suprema, cartesian lifting is a supremum homomorphism.*

Cartesian lifting is a homomorphism for all possible construction steps, so it is an accurate projection. It is always possible to switch from document causality graphs to cartesian causality decompositions and continue the construction there. The end result will be the same as if the construction had happened on the document causality graphs and would then have been lifted afterwards.

The homomorphism properties provide an opportunity for runtime verification. When the implementation performs the constructions in one world, a runtime monitor can simultaneously do the constructions in the other world. Then cartesian lifting can be used to check if the constructions correspond with each other. This could ensure the correctness of optimizations.

Cartesian lifting is a non-injective projection, as it discards operation causalities between different cells. For every cell, causalities are considered only in isolation. Figure 8 provides an example for this. The document causality graph clearly shows that $(x, y)$ are either $(B, B)$ (left branch) or $(C, C)$ (right branch). On the other hand, the cartesian causality decomposition values the tuple $(x, y)$ with any element of the cartesian product $\{B, C\} \times \{B, C\}$. So the edit context of operations is not yet considered in the current concept of lazy merging.

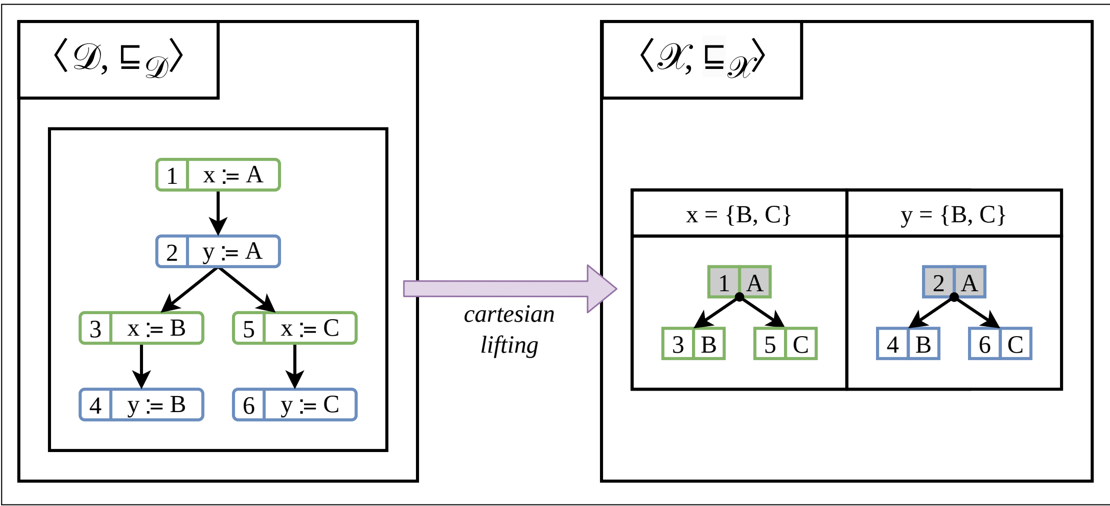



Figure 8: Cartesian lifting is non-injective.

## 5 Impact

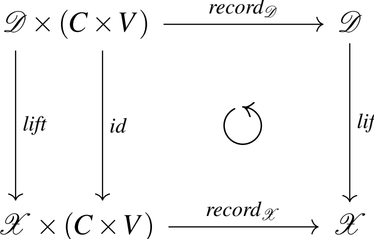



Figure 9: Cartesian lifting is a record homomorphism.

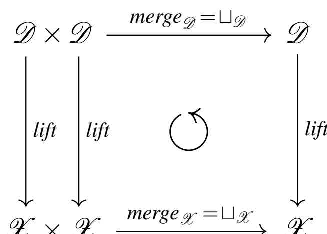



Figure 10: Cartesian lifting is a merge/supremum homomorphism.

Lazy merging tackles the hard problem of branch reconciliation by clearly separating merging and conflict resolution. This separation makes it possible to model merging with lattices and establish invariants that are not possible in systems that intermingle merging and conflict resolution. Because merges are the suprema in their respective lattice they inherit several important properties. First, merges are *complete*, *minimal*, and *unique*, as suprema are the least upper bound. Because they are complete, merges include all operations and all their causal dependencies. No information is lost, and no changes are arbitrarily discarded. Minimality ensures that during merging no additional causal relationships are accidentally introduced. No noise or verbosity can appear from merging. And the uniqueness of merges makes them deterministic and always automatically computable. Thus, participants can always merge safely and be confident to get a good result.

Second, merging is *commutative*, *associative*, and *idempotent*, derived from the algebraic properties of suprema. Commutativity ensures that no matter which participant performs the merge, the result will always be the same. When more than two branches are merged at once, commutativity and associativity make the order in which the branches are merged irrelevant. There is only one canonical result when $n$ branches are merged. And because of idempotence, it does not matter if branches are merged multiple times, the end result will always be as if they were merged only once. Participants can be sure that even in

complex scenarios with many branches merging always works as expected. All these properties hold for the merges of both document causality graphs and cartesian causality decompositions.

The segmentation of documents into persistent, uniquely identifiable cells enables much more precise change recording compared to traditional VCSs. Many VCSs (including recent approaches like Pijul, cf. Section 6.3) operate directly on plain text files and always use text lines as the segments at which they operate, regardless of the content. These text lines have no persistent identifiers in the plaintext files, so they have to rely on complicated and heuristic diffing algorithms to match changes with best effort.

Variant potentials in cells are the key enabler for laziness and completely change the way conflicts are treated in the collaboration system. It relieves the need for constant eager merging and enables the collaboration system to represent and explain conflicts, with the aforementioned benefits (cf. Section 3). Current systems simply have no facility to represent and aggregate conflicts in this way.

The fact that regular assignment operations resolve variant potentials is a significant simplification for the protocol. If another assignment happens on the cell concurrently to a resolution, or if two different resolutions are carried out concurrently, that will be properly aggregated back into a variant potential. In this way, edits and resolution operations are independent of each other and can be used freely. For participants, it is always safe to perform variant potential resolutions.

The granularity of cells can be adjusted to provide the best possible perspective on changes and conflicts in each domain. Highly structured data like graph models can profit from fine-grained cells because edits are easily located. Free-form natural language content would probably best be represented by more coarse-grained cells, perhaps one cell per sentence or even per paragraph. And of course the trivial segmentation where the whole document is one cell is also possible. This would disable the cartesian decomposition and would allow integrating other forms of conflict detection and aggregation into the system, like e.g. traditional diffing algorithms.

Document causality graphs and cartesian causality decompositions are two perspectives that are each valuable for different parts of the collaboration system. Document causality graphs are the source of truth and can provide the full history of operations, with all causality relationships between them. Cartesian causality decompositions on the other hand show the current state and variant potentials, so they are used during the editing of documents. The strong connection between the two provided by the homomorphism properties of cartesian lifting enables the collaboration system to work on any of them interchangeably.

The editor and data structure as presented in this paper have been implemented in interactive prototypes. A live collaboration system for graph models[2] demonstrates a structure editor for lazy merging that is capable of rendering and resolving variant potentials as described in Section 3.4. For a simple key-value store, the behavior of the lazy merging data structure is visualized during branching & merging[3], following the same graphical notation as Section 4. The visualization is accompanied by a simple collaborative editor[4] for the key-value store.

[2] https://graph-models.lazy-merging.jonas-schuermann.name/
[3] https://data-structure.lazy-merging.jonas-schuermann.name/
[4] https://key-value.lazy-merging.jonas-schuermann.name/

# 6 Related Work

In this section we present related approaches that attempt to improve the handling of conflicts or formalize collaboration systems similarly. Wieland et al. explained why conflict representation matters and built a prototype to evaluate their ideas. Conflict-free replicated data types (CRDTs) are another, lower-level formalism that utilizes lattice theory similarly to lazy merging. And Pijul is a distributed VCS that introduced a robust conflict representation for plaintext files.

## 6.1 Wieland et al., 2012

In their paper "Turning Conflicts into Collaboration" Wieland et al. presented their vision for a conflict-tolerant collaboration system. They explained how conflicts can help to uncover disagreements and find solutions that work for everyone. The paper relied on expert interviews to show the importance of conflict tolerance, collaborative conflict resolution, and comprehensible evolution. They proceeded to build a prototype collaboration system that fulfills these requirements and conducted a case study to evaluate their results. [WLS+12]

Wieland et al. produced expedient empirical results about collaboration practices, which are also relevant for the lazy merging concept. The overall goals of their work and lazy merging align, what is different is the approach to the problem. They approached the implementation of the collaboration system with object-oriented design methods, while lazy merging emerges from a stringent mathematical model. As demonstrated in this paper, the mathematical approach yields invaluable invariants that prove correctness and robustness properties. Wieland et al. considered properties like completeness and minimality of merges in the case study, but they only showed them for one example. However, their case study examined human behavior within such a collaboration system, this has not yet been done for the lazy merging approach.

Furthermore, Wieland et al. co-opt existing annotation nodes within graph models to represent conflicts textually. In contrast, lazy merging extends the document model with variant potentials as a first-class concept. This more structured approach simplifies the aggregation of variant potentials and allows for different projectional visualizations.

One feature that sets their merge algorithm apart from lazy merging is the consideration the edit context during conflict detection. Concurrent changes in the context can significantly alter the meaning of edits, so it is valuable to capture these kinds of conflicts. Lazy merging does not support this kind of conflict detection yet.

Their prototype also featured a guided conflict resolution process with different states, which included task assignment, resolution proposals and decision-making. This supports the reconciliation efforts of the participants, but it is unclear how the process behaves in the presence of concurrency. Without further measures, the resolution process would have to be performed synchronously and block all affected elements, to prevent further simultaneous edits. This would prevent the usage of branches and offline working during conflict resolution. Lazy merging has a less sophisticated resolution mechanism, but both merging and conflict resolution are independent operations that are always available. There is no additional coordination needed, merging and conflict resolution always work on any branch and when offline. If a resolution conflicts with another concurrent resolution or another concurrent assignment, all variants are properly aggregated into a potential. So with lazy merging, resolution operations can be used more freely.

### 6.2 Conflict-Free Replicated Data Types (CRDTs)

CRDTs[5] introduced a solid theoretical foundation for the design of eventual consistent data structures. In decentralized systems, they can store shared data among multiple replicas, each of which can be optimistically updated without coordination. Despite their decentralized operation, they are guaranteed to eventually converge on a consistent state. State-based CRDTs provide a merge operation, and lattice theory was used to prove convergence after reciprocal merging. [SPBZ11]

The theoretical argument is similar for state-based CRDTs and lazy merging as presented in this paper. Both approaches model merges as suprema and prove important properties in this way. In fact, lazy merging constitutes a state-based CRDT, as shown in Figure 11. So lazy merging inherits properties of CRDTs like guaranteed convergence and the ability for decentralized operation.

payload $\mathscr{D}$ $D$
    initial $\langle \emptyset, \emptyset \rangle$
update record (C c, V v)
    $D := record_{\mathscr{D}}(D, (c, v))$
query project () : $\mathscr{X}$ $X$
    let $X = lift(D)$
compare $(D_1, D_2)$ : boolean $b$
    let $b = D_1 \sqsubseteq_{\mathscr{D}} D_2$
merge $(D_1, D_2)$ : payload $D_3$
    let $D_3 = merge_{\mathscr{D}}(D_1, D_2)$

Figure 11: Specification of lazy merging as a state-based CRDT.

Technically, variant potentials behave similarly to multi-value register CRDTs [SPBZ11]. But while multi-value registers typically rely on version vectors to capture operation causality, variant potentials use the history of operations that is already available. Additionally, multi-value registers only show the current valuation, while variant potentials provide the history of all assignments ever made to their cell.

Conceptually, there is a big difference between CRDTs and lazy merging. CRDTs are general-purpose data structures for decentralized systems. They aim to avoid conflicts from concurrent updates by specifying automatic resolution semantics. While they can be used to implement collaboration systems, they are not concerned with the design of collaboration processes. CRDTs are implementation building blocks, lazy merging is a holistic high-level approach for the design of collaboration systems. Lazy merging builds potentials from conflicts instead of eliminating them arbitrarily. And it provides a rigorous concept from the implementation of the data structure to the user interface design.

### 6.3 Pijul

The distributed VCS Pijul[6] is built on theoretical foundations similar to lazy merging. The Pijul authors argued with category theory and stated that every merge is a pushout, which is the analogous concept to suprema in lattice theory. They, too, generalized the data model to include these pushouts and represent conflicts as regular elements of the documents [MG13]. Pijul operates on plaintext files, which it treats as sequences of lines. Within the repository, files are represented as graphs, to represent the results of concurrent insertions and deletions [Meu].

In principle, Pijul provides a robust and expressive representation for conflicts within the repository. But this representation is severely limited by the usage of plaintext files when participants edit the documents. Only simple conflicts can be adequately expressed with textual

[5] https://crdt.tech/

[6] https://pijul.org/

conflicts markers, and the presence of conflict markers destroys the syntactic consistency of the text document. In contrast, lazy merging considers conflict representation not only in the data structure, but also shows how editors can properly communicate them to the participants.

Although the authors suggested that Pijul could operate on the actual syntax of programming languages instead of lines of text, this is not yet supported. So Pijul currently suffers the same inaccuracies in conflicts detection like traditional VCSs. For example, it cannot differentiate between syntactically irrelevant changes (reformatting, changes in white space) and other syntactically relevant changes. And it cannot operate at the granularity of syntactical elements, as lines are treated as atomic units. Lazy merging requires that the collaboration system operates on the actual syntax of the documents, for accurate conflict detection and representation.

Furthermore, Pijul has no mechanism to reliably track the identity of text lines once they are rendered in plaintext files for participants to work on. Thus, it requires the same heuristic diffing algorithms also used in traditional VCSs. These algorithms are quite complicated and can only guess which blocks of text moved where, because plaintext files do not provide any metadata to track their movements. The generated diffs only describe insertions and deletions and cannot adequately represent the movement of blocks across the document. Lazy merging does not require any heuristic diffing algorithms, because it is integrated with a syntax-aware editor. As all syntactical elements are uniquely identifiable, all changes can be reliably located and easily recorded. And with proper data modeling, movements of elements can be properly captured by respective value assignments.

## 7 Future Work

The next logical step is the implementation of a real-world collaboration system based on the lazy merging concept. Our chair develops the graphical language workbench Cinco Cloud [BBK+22], which currently uses a simple optimistically concurrent live collaboration protocol. We want to use lazy merging to build a new and more powerful collaboration protocol for Cinco Cloud that supports branching & merging with lazy conflict resolution. More operations will be needed for the practical use as a VCS, like reverts, or features like cherry-picking. Change management processes are needed to coordinate incoming changes and perform peer reviews. A lot more work is also needed to explore how participants can interact with the collaboration history. How can the comprehension of the history be supported, how can conflicts be explained and resolved in a guided fashion? Finally, there are many more questions regarding the actual implementation of the system. How is data persisted, how does the network protocol work, how is the user interface implemented, is the system efficient and scalable? This implementation would be a great opportunity for a case study that evaluates the real-world applicability of the lazy merging concept.

Changes in the context of an edit can significantly alter its meaning [WLS+12]. But as discussed previously, cartesian lifting completely discards the context of cells within the document. So the current lazy merging concept does not account for this. One idea would be to introduce certain implications in the cartesian lifting function. An assignment to a property cell of an entity implies the assignment of the value `True` to its presence cell. This would provoke a conflict if the entity is concurrently deleted by assigning `False` to the presence cell. Edit/delete conflicts are accounted

for with such implications, but it is still unclear if this is the best approach and how a more general notion of context-aware conflict detection can be implemented.

As discussed in Section 3.4, lazy merging requires persistent unique identifiers for syntactical elements and the capability to represent variant potentials within collaborative documents. Plain-text programming languages (e.g., Java) do not offer these facilities, so it is not straightforward to apply lazy merging to them. Because the overwhelming majority of coding is currently done in plaintext editor environments, it is important to investigate how lazy merging can be applied to these languages. Structure editors that target textual languages like JetBrains MPS [BCCP21] seem to be promising to bridge this gap. They provide a user experience similar to traditional plaintext editors, but at the same time maintain a structured AST that could be extended with the necessary metadata to support lazy merging. It is not ruled out that plaintext language grammars could be extended to provide the necessary facilities for lazy merging without relying on a structure editor. But it is questionable if this could be achieved in a straightforward and ergonomic way. Furthermore, because textual languages are usually structured hierarchically, the consideration of the edit context becomes even more important.

Conflict resolution in lazy merging is currently very simple, regular assignments select a resolution. More sophisticated mechanisms could further support participants in finding good resolutions. For example, adding and removing single variants without resolving the potential could be useful to collect incomplete information. Assigning the whole set of values in a variant potential would allow for this, and this seems to be easy to do with lazy merging. Another example would be the explicit management of resolution decisions. Variant potentials could be annotated to be postponed for a certain time or to require the attention of specific participants.

Although this paper only discussed version control with state-based branching & merging, the proposed data structure is also suitable for a live collaboration protocol. The fine-grained change recording enables efficient replication and concurrent assignments commute, which easily ensures correctness. A future paper should examine that perspective and the integration of live collaboration with branching & merging. Branches could become convergent workspaces that are replicated live to all interested participants. Lazy merging could provide a fault-tolerant protocol and allow for seamless offline working, because any divergences can be safely merged. This would enable uninterrupted work in the isolation of branches, but still provide the possibility to collaborate in real-time.

When it comes to the versioning of modeling languages, lazy merging opens up possibilities for new generation strategies. Generators could translate variant potentials from the source model into variant potentials in the target model. For example, if an executable application is generated from the models, a variant potential could be translated to a runtime setting. This propagation of laziness is probably not always easy to accomplish, but it would unlock generation to enable testing and execution in the presence of variant potentials.

Another way to handle variant potentials in source models could be a partial generation strategy. The generator analyses the dependencies needed for every target and generates only those targets whose dependencies are free from variant potentials. This would further reduce blocking caused by the occurrence of conflicts.

In [KWHM19] Kleppmann et al. present their vision for local-first software. The goal of local-first software is to give back participants the ownership over their data. The participants' data is primarily stored decentralized on their devices instead of a centralized remote storage. They

find CRDTs to be suitable for the implementation of these decentralized collaboration systems because of their reliability and ease of use. They make a convincing argument, but they also assert that conflicts are not a big problem even after prolonged offline working. The generic resolution mechanisms of CRDTs allegedly suffice to handle conflicts. This is contrary to the arguments presented in this paper, which show the importance of proper conflict preservation and resolution. Furthermore, they recognize the value of branching & merging, but pose its implementation as a future research question. Lazy merging could make offline working safer by properly representing conflicts as variant potentials and bring branching & merging to local-first software.

## 8 Conclusion

Lazy merging is a new approach to the problem of merging and conflict resolution in collaboration systems. We used a real-world example to illustrate new collaboration mechanisms that are enabled by lazy merging. Lazy merging introduces variant potentials to capture conflicts and aggregate them robustly even in complex merging scenarios. They separate merging from conflict resolution, so that merging can be easily automated, while participants are properly supported to resolve conflicts explicitly. Variant potentials capture and hold conflicts, postponing the resolution and supporting communication between participants to uncover disagreements. Contributions can be brought in without interruption and conflicts can be shared and resolved collaboratively. Lazy merging segments documents into persistent, uniquely-identifiable cells for precise change recording and conflict detection. The granularity of this segmentation can be adapted for an optimal perspective on changes and conflicts in any given domain. A structure editor is used to visualize variant potentials and handle the enriched document format. Value assignments to cells are recorded in two different ways. Document causality graphs thoroughly record the complete causality graph between all operations. Cartesian causality decompositions record the causalities separately for each cell, which captures less information but easily shows the current valuation of each cell. Document causality graphs and cartesian causality decompositions both form a lattice regarding their conservative extension. Merges are suprema in these lattices, so they are guaranteed to be complete, minimal, unique, commutative, associative, and idempotent. Finally, cartesian lifting is introduced as a projection from document causality graphs to cartesian causality decompositions. Cartesian lifting is an accurate projection because it is a homomorphism with regard to all of their possible construction steps.

Lazy merging enables powerful features like branching & merging, but it also preserves usability and supports the participants' comprehension of conflicts. Conflict preservation and collaborative conflict resolution foster alignment within teams. It is built on strong theoretical foundations that give confidence in the correctness and robustness of the system. At the same time, lazy merging brings simplicity to the implementation of collaboration systems. This paper explained the fundamental concept of lazy merging, demonstrated its impact, and highlighted many exiting avenues for future research.

## Bibliography


[BBK+22] A. Bainczyk, D. Busch, M. Krumrey, D. S. Mitwalli, J. Schürmann, J. Tagoukeng Dongmo, B. Steffen. CINCO Cloud: A Holistic Approach for Web-Based Language-Driven Engineering. In Margaria and Steffen (eds.), *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering*. Pp. 407–425. Springer Nature Switzerland, Cham, 2022.
doi:10.1007/978-3-031-19756-7_23

[BBR+12] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, P. Devanbu. Cohesive and Isolated Development with Branches. In Lara and Zisman (eds.), *Fundamental Approaches to Software Engineering*. Pp. 316–331. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
doi:10.1007/978-3-642-28872-2_22

[BCCP21] A. Bucchiarone, A. Cicchetti, F. Ciccozzi, A. Pierantonio. *Domain-Specific Languages in Practice: with JetBrains MPS*. Springer International Publishing, 2021.
doi:10.1007/978-3-030-73758-0

[CS14] S. Chacon, B. Straub. *Pro Git*. The expert's voice. Apress, 2014.
https://git-scm.com/book/en/v2

[KWHM19] M. Kleppmann, A. Wiggins, P. van Hardenberg, M. McGranaghan. 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*. Onward! 2019, p. 154–178. Association for Computing Machinery, New York, NY, USA, 2019.
doi:10.1145/3359591.3359737

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21(7):558–565, July 1978.
doi:10.1145/359545.359563

[Men02] T. Mens. A State-of-the-Art Survey on Software Merging. *Software Engineering, IEEE Transactions on* 28:449–462, June 2002.
doi:10.1109/TSE.2002.1000449

[Meu] P.-É. Meunier. The Pijul manual. https://pijul.org/manual/. [Online; last accessed 02-March-2023].

[MG13] S. Mimram, C. D. Giusto. A Categorical Theory of Patches. *Electronic Notes in Theoretical Computer Science* 298:283–307, Nov. 2013.
doi:10.1016/j.entcs.2013.09.018

[PSW11] S. Phillips, J. Sillito, R. Walker. Branching and Merging: An Investigation into Current Version Control Practices. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*. ACM, may 2011.
doi:10.1145/1984642.1984645

[SPBZ11] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011.
https://hal.inria.fr/inria-00555588

[WFSW11] K. Wieland, G. Fitzpatrick, M. Seidl, M. Wimmer. Towards an Understanding of Requirements for Model Versioning Support. *International Journal of People-Oriented Programming* 1:1–23, June 2011.
doi:10.4018/ijpop.2011070101

[WLS+12] K. Wieland, P. Langer, M. Seidl, M. Wimmer, G. Kappel. Turning Conflicts into Collaboration. *Computer Supported Cooperative Work (CSCW)* 22(2-3):181–240, Sept. 2012.
doi:10.1007/s10606-012-9172-4

[ZNS19] P. Zweihoff, S. Naujokat, B. Steffen. Pyro: Generating Domain-Specific Collaborative Online Modeling Environments. In *Proc. of the 22nd Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2019)*. 2019.
doi:10.1007/978-3-030-16722-6_6