future internet

Article

Data Synchronization: A Complete Theoretical Solution for Filesystems

Elod P. Csirmaz [1,*] and Laszlo Csirmaz [1,2]

1 Alfréd Rényi Institute of Mathematics, 1053 Budapest, Hungary
2 Institute of Information Theory and Automation, CZ-182 00 Prague, Czech Republic
* Correspondence: elod@epcsirmaz.com

Abstract: Data reconciliation in general, and filesystem synchronization in particular, lacks rigorous theoretical foundation. This paper presents, for the first time, a complete analysis of synchronization for two replicas of a theoretical filesystem. Synchronization has two main stages: identifying the conflicts, and resolving them. All existing (both theoretical and practical) synchronizers are *operation-based:* they define, using some rationale or heuristics, how conflicts are to be resolved without considering the effect of the resolution on subsequent conflicts. Instead, our approach is *declaration-based:* we define what constitutes the resolution of all conflicts, and for each possible scenario we prove the existence of sequences of operations/commands which convert the replicas into a common synchronized state. These sequences consist of operations rolling back some local changes, followed by operations performed on the other replica. The set of rolled-back operations provides the user with clear and intuitive information on the proposed changes, so she can easily decide whether to accept them or ask for other alternatives. All possible synchronized states are described by specifying a set of conflicts, a partial order on the conflicts describing the order in which they need to be resolved, as well as the effect of each decision on subsequent conflicts. Using this classification, the outcomes of different conflict resolution policies can be investigated easily.

Keywords: data synchronization; conflict resolution; filesystem theory

MSC: 08A02, 08A70, 68M07, 68P05

check for updates

Citation: Csirmaz, E.P.; Csirmaz, L. Data Synchronization: A Complete Theoretical Solution for Filesystems. *Future Internet* **2022**, *14*, 344. https://doi.org/10.3390/fi14110344

Academic Editor: Davide Tosi

Received: 10 October 2022
Accepted: 18 November 2022
Published: 21 November 2022

1. Introduction and Related Work

This work is a comprehensive treatment of filesystem synchronization and conflict resolution on a simple but powerful theoretical model of filesystems. Synchronizing diverged copies of some data stored on a variety of devices and locations is an important and ubiquitous task. The last two decades have seen tremendous advancement, both theoretical and practical, in the closely related fields of distributed data storage [1,2] and group editors [3,4]. This progress has been based on, and has expanded significantly, two competing theoretical frameworks: Operational Transformation (OT) and Conflict-free Replicated Data Types (CRDT). OT appeared in the seminal work [5] and was refined later in [4]. The general idea is that operations are enriched with the context in which they were generated. Before applying them, they are transformed depending on the context of their origin and the context where they are to be applied. Main applications are collaborative editors, the most notable example being Google Docs [6]. The core concept of CRDT is commutativity, see [1,2]. Basic data types with special operators are devised so that executing the operators in different orders yield the same results. Examples are counters or sets with add and delete operators. The basic types are used as building blocks in more complex applications such as collaborative editors [7], the commercial product Riak [8], and others.

Both OT and CRDT have been successfully applied in a variety of synchronization tasks [9,10]. Filesystem synchronization, however, fits very poorly into these frameworks, mainly because it works under a completely different modus operandi: constant, low latency communication in OT and CRDT versus a single data exchange in filesystem synchronization; frequent loose synchronization with eventual convergence requirement versus a single but complete synchronization; small number of differences versus significant structural differences accumulated during an extended time period; and so on. Consequently, for a theoretical investigation of filesystem synchronization, we follow a traditional framework instead, described in e.g., [11] and depicted in Figure 1 adapted from [12].

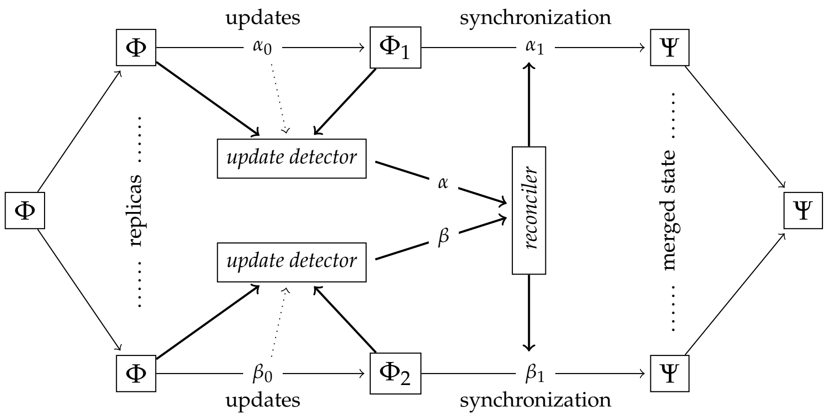

Figure 1. Outline of the synchronization process. Identical replicas of the original filesystem Φ are updated (modified) yielding the divergent replicas Φ_1 and Φ_2. The *reconciler* uses the update information α and β extracted by the *update detector*s, and generates the synchronizing instructions α_1 and β_1. These create the identical merged state Ψ when applied to the replicas. The update detectors determine the update information α and β either by comparing the different states of the replicas (e.g., Φ_1 vs. Φ), or by having access to the update instructions α_0 and β_0 that were applied to Φ.

Two identical replicas of the original filesystem Φ are updated independently, yielding the divergent replicas Φ_1 and Φ_2. In the state-based case the *update detector* receives the original (Φ) and the current (Φ_1 or Φ_2) states, and generates the update information α (or β) describing the differences between the original filesystem and the replica. In the operation-based case the update decoder has access to the performed operations α_0 (β_0) only. The *reconciler* receives the information provided by the update detectors and generates the synchronizing instructions α_1 and β_1, respectively. These instructions, applied to the replicas Φ_1 and Φ_2, create the identical *merged state* Ψ.

In order to reach such a common synchronized state, existing theoretical and practical filesystem synchronizers, such as [10,13–17], apply some conflict-resolution strategy. They identify all (or some) of the conflicts, then apply their strategy to the conflicts one at a time. These algorithms are typically fixed and defined by some rationale or heuristics, or dictated by the underlying technology. In contrast, we *define what comprises a synchronized state.* Intuitively, it is a maximal consistent merger of the replicas, meaning that no further changes can be applied to the merged state from those that were applied to the replicas during the update phase. Then we proceed to prove that every such maximal state (and only these states) can be reached by resolving the conflicts in some order.

In the meticulous survey [16] of existing theoretical and practical filesystem synchronizers it has been observed that "resolving conflicts is an open problem where [. . .] most academic works present arbitrary resolution methods that lack a rationale for their decisions" (page i), and that "a file system can be affected by more than one conflict at once, which has not been discussed in related academic works" (page 65). By presenting, for the first time, a complete analysis of the synchronization process for two replicas of a theoretical filesystem model, we fill these gaps.

Our Contribution

This paper is a complete analysis of the synchronization process of two replicas of a theoretical filesystem. Our filesystem model together with the commands considered—

such as *create*, *rmdir*, or *edit*—are discussed in Secion 3. The changes between the original replica and the locally updated versions are captured by a special command set as defined in Section 4, which also defines algorithms generating this update information.

The important question of which filesystems can, in general, be considered to be a synchronized state of two divergent replicas is tackled in Section 5. Our definition captures this notion in its full generality without prescribing how the synchronized state can be reached. Providing such a declaration-based definition of the synchronized state is one of our main contributions. Section 6 presents the generic synchronization algorithm based on conflict resolution. By using different strategies to resolve conflicts, any of the synchronized states allowed by our definition, and only those, can be the result of the algorithm. Finally, Section 7 summarizes the results and lists open problems and directions for future research.

2. Methodology

Our filesystem model, defined in Section 3, is arguably simple, but it retains all important structural properties of real filesystems. It is this simplicity that allows us to exploit a rich and intriguing algebraic structure [12,18] which eventually leads to the claimed theoretical results. In Section 7 we discuss some possible extensions of the filesystem model.

Depending on the data communicated by the replicas, synchronizers are categorized as either *state-based* or *operation-based* [1,16]. In state-based synchronization replicas send their current versions of the filesystem, or merely the differences (called *diff*s) between the current states and the last known synchronized state [19]. Operation-based synchronizers transmit the complete log (or trace) of all operations performed by the user [13]. The synchronization method of this paper is reminiscent of operation-based one, but can also be considered, with similar overhead, to be state-based. A set of virtual filesystem operations (commands) will be defined in Section 3. Each of these commands have a clear and intuitive operational meaning, but are not necessarily available to the end-user. The current state of the filesystem is described by a special—called *canonical*—sequence of virtual commands, which transforms the original filesystem to the actual replica. The *update detector* can generate this canonical sequence from the operations performed by the user on the replica (operation-based) as well as from the differences between the original and final state (state-based). On Figure 1 these sequences correspond to the information in α and β passed to the reconciler.

Filesystem synchronization must resolve all conflicts between the replicas. Conflict resolution, however, should be "intuitively correct," i.e., discarding all changes made by both replicas is not a viable alternative. The majority of commercially or theoretically available synchronizers do not present a rationale to explain their concrete conflict resolution approach [16]. Two notable exceptions are [10] and [9] which describe high-level consistency philosophies. In [10] the main principles are (1) *no lost update:* preserve all updates on all replicas because these updates are equally valid; and (2) *no side effects:* such as a merge where objects unexpectedly disappear. While these principles intuitively make sense, it is easy to see that neither could possibly be upheld for every conflict; even the authors provide counterexamples. In [9] the relevant consistency requirements are worded as follows:

- R1 *intention-confined effect:* operations applied to the replicas by the synchronizer must be based on operations generated by the end-user; and
- R2 *aggressive effect preservation:* the effect of compatible operations should be preserved fully; and the effect of conflicting operations should be preserved as much as possible.

These requirements are in fact variations of the OT consistency model, see for example the notion of intention preservation in [4]. (We note that the other two OT principles—convergence and causality preservation—do not apply to filesystem synchronizers.)

In agreement with R1 and R2 we *declare* what a synchronized state is, rather than present an algorithm which generates it. Our declaration can be outlined as follows. Suppose that the two replicas Φ_1 and Φ_2 are represented by the canonical sequences α and β, respectively, that is, $\Phi_1 = \alpha\Phi$ and $\Phi_2 = \beta\Phi$, where $\alpha\Phi$ is the result of applying the command sequence α to Φ. The *synchronized* or *merged state* Ψ is determined by the canonical sequence μ as $\Psi = \mu\Phi$ such that

C0 μ is applicable to the original filesystem Φ,
C1 every command in μ can be found either in α, in β, or in both, and
C2 μ is maximal, i.e., no canonical sequence adding more commands to μ can satisfy both C0 and C1.

Condition C0 is the obvious requirement that the synchronizer must not cause errors. Condition C1 ensures that the synchronization satisfies the *intention-confined effect* (no surprise changes in the merged filesystem) requirement R1 as μ should consist only of commands which were supplied by the replicas. The other consistency requirement R2 is guaranteed by C2 as μ is maximal, therefore it preserves as much of the intention of the users as possible. Note that this definition of a synchronized state is never vacuous. There are only finitely many sequences which satisfy C0 and C1 (as every command in μ comes from either α or β and no repetitions are allowed), and there are at least two, namely α and β. Because of this, the empty sequence (undoing all changes) will not satisfy C2 (assuming one of α and β is not empty), in line with our requirements.

The declaration-based synchronization is intuitively clear, easy to understand, and does not use any predetermined conflict-resolving policy. An operational characterization is proved in Theorem 2. The essence of this theorem is that any merged filesystem Ψ can be generated from the replica, say Φ_1, by

1. rolling back some of the commands in α, followed by
2. applying some commands from the other sequence β, $(*)$

and vice versa for the other replica. The commands rolled back represent a minimal set whose removal resolves all conflicts. These commands also give users a clear understanding of the changes the synchronizer wants to perform on their filesystem. They are also helpful when some of the rolled-back commands should be introduced again after the merging (doing a redo of the undo).

Turning to traditional conflict-based synchronization, Section 6 proves that any declaration-based merged state, and only those states, can be the result of a conflict-based synchronization algorithm. For each pair of canonical sequences describing the replicas to be synchronized we define what the conflicting command pairs are. Their resolution uses the winner/loser paradigm: the winner command is accepted while the loser command is discarded. It turns out that resolving a conflict may automatically resolve some of the subsequent conflicts, but no new conflicts are created. Using this result the outcomes of different conflict resolving policies can be investigated easily. In particular, the *iterative approach* [16] always works, which applies all non-conflicting commands, resolves, in any way, the first conflict (which might automatically resolve other existing conflicts), and iterates.

3. Definitions

This section defines the basic notation which will be used throughout the rest of the paper. The exposition follows [12,18] with some substantial modifications. Some results from these papers are included without proof.

3.1. Namespace and Filesystems

Our filesystem model is arguably simplistic, nevertheless it captures all important aspects of real-word implementations. In spirit, it is a mixture of *identity-* and *path-based* models [10,16]. Objects are stored in nodes, which are uniquely identified by fixed and predetermined *path*s. The set of available nodes is fixed in advance, and no path operations are considered. Filesystems are required to have the *tree-property* at all times: in a given filesystem along any branch starting from any of the root nodes, there must be zero or more *directories*, zero or one *file*, followed by *empty* nodes. Our model does not support *links* (but see Section 7.4), thus the namespace—the set of available nodes or paths—forms a collection of rooted trees. Filesystems are defined over and populate this fixed namespace.

Formally, the *namespace* is a set $\mathbb{N}$ endowed with the partial function $\uparrow : \mathbb{N} \to \mathbb{N}$ returning the parent of every non-root node (it is not defined on roots). If $n = \uparrow m$ then n is the parent of m, and m is a child of n. For two nodes $n, m \in \mathbb{N}$ we say that n *is above* m, or n is an ancestor of m, and write $n \prec m$ if $n = \uparrow^i(m)$ for some $i \geq 1$. As usual, $n \preccurlyeq m$ means $n \prec m$ or $n = m$. As the parent function induces a tree-like structure, $\preccurlyeq$ is a partial order. Two nodes $n, m \in \mathbb{N}$ are *comparable* if either $n \preccurlyeq m$ or $m \preccurlyeq n$, and they are *uncomparable* or *independent* otherwise.

A *filesystem* Φ populates the nodes of the namespace with values. The value stored at node $n \in \mathbb{N}$ is denoted by $\Phi(n)$. This value can be $\mathbb{O}$ indicating that the node is *empty* (no content, not to be confused with the empty file); can be $\mathbb{D}$ indicating that the node is a *directory*; otherwise it is a *file* storing the complete content (which could happen to be "no content"). We use $\mathbb{O}$, $\mathbb{D}$ and $\mathbb{F}$ to denote the *type* of the content corresponding to these possibilities. While there is only one value of type $\mathbb{O}$ and one value of type $\mathbb{D}$ (see Section 7.3 for a discussion on lifting this limitation), there are many different file values of type $\mathbb{F}$ representing different file contents.

3.2. Internal Filesystem Commands

The synchronizer operates using a specially designed and highly symmetrical set of *internal filesystem commands*. They each modify the filesystem at a single node only, and contain additional information usually not thought of as part of a command which allows them to be inverted, and makes them amenable to algebraic manipulation. Inventing such a command set was one of the main contributions of [18].

The commands basically implement creating and deleting files and directories. Modifying a part of a file only is not available as a command; whenever a file is modified, the new content must be supplied in its entirety. Still, commands issued by a real-life user or system can be easily transformed into the internal commands; for example, the *move* or *rename* operation can be modeled as a sequence of a *delete* and a *create*.

The set of the internal commands is denoted by Ω, and each command $\sigma \in \Omega$ has three components, written as $\sigma = \langle n, x, y \rangle$, as follows:

- $n \in \mathbb{N}$ is the node on which the command acts,
- x is the content at node n before the command is executed (precondition), and
- y is the content at node n after the command was executed.

Thus *rmdir*(n) corresponds to the internal command $\langle n, \mathbb{D}, \mathbb{O} \rangle$, which replaces the directory value at n by the empty value. The command $\langle n, \mathbb{O}, \mathbb{D} \rangle$ creates a directory at n, but only if the node n has no content, i.e., there is no directory or file at n (a usual requirement for *mkdir*(n)). For files f_1 and $\mathsf{f}_2 \in \mathbb{F}$ the command $\langle n, \mathsf{f}_1, \mathsf{f}_2 \rangle$ replaces f_1 stored at node n by the new content f_2. This latter command can be considered to be an equivalent of *edit*$(n, \mathsf{f}_1, \mathsf{f}_2)$.

Applying $\sigma \in \Omega$ to a filesystem Φ is written as the left action $\sigma\Phi$. The command $\sigma = \langle n, x, y \rangle$ is *applicable* to Φ if

- Φ contains x at the node n, that is, $\Phi(n) = x$, and
- after changing the content at n to y the filesystem still has the tree property.

If σ is not applicable to Φ then we say that σ *breaks* the filesystem, and write $\sigma\Phi = \perp$. If σ does not break Φ, then $\sigma\Phi$ is the filesystem where every node has the same value as in Φ except for the node n where the new content is y. Command sequences are applied from left to right, thus $(\sigma\alpha)\Phi = \alpha(\sigma\Phi)$. The composition of sequences α and β is written as $\alpha\beta$; occasionally we write it as $\alpha \circ \beta$ to emphasize that β is to be executed after α. A sequence breaks Φ if one of its commands is not applicable. More formally, $\sigma\alpha$ breaks Φ if either σ breaks Φ, or α breaks $\sigma\Phi$.

Two additional commands will be used, which are denoted by ϵ and λ, respectively. In practice they do not occur, and cannot be elements of command sequences, but are useful in algebraic derivations. The command ϵ breaks every filesystem, while λ acts as identity: $\epsilon\Phi = \perp$ and $\lambda\Phi = \Phi$ for every filesystem Φ.

The command sequences α and β are *semantically equivalent*, written as $\alpha \equiv \beta$ if they have the same effect on all filesystems: $\alpha\Phi = \beta\Phi$ for all Φ. We write $\alpha \sqsubseteq \beta$ to denote that β *semantically extends* α, that is, $\alpha\Phi = \beta\Phi$ for all filesystems that α does not break. Clearly, $\alpha \equiv \beta$ if and only if both $\alpha \sqsubseteq \beta$ and $\beta \sqsubseteq \alpha$. The sequence α is *non-breaking* if there is at least one filesystem Φ which α does not break. This is the same as requesting $\alpha \not\equiv \epsilon$.

The *inverse* of $\sigma = \langle n, x, y\rangle$ is $\sigma^{-1} = \langle n, y, x\rangle$. For a command sequence α its inverse is defined in the usual way: α^{-1} consists of the inverse of the commands in α written in reverse order. This inverse has the expected property: if α does not break Φ, then $(\alpha\alpha^{-1})\Phi = \Phi$. In particular, $\alpha\alpha^{-1} \sqsubseteq \lambda$.

3.3. Command Types and Execution Order

A node value has type $\mathbb{O}$, $\mathbb{D}$, or $\mathbb{F}$ depending on whether it is the empty value, a directory, or a file. For a command $\sigma = \langle n, x, y\rangle$ the type of x is its *input type* and the type of y is its *output type*. Depending on their input and output types commands can be partitioned into nine disjoint classes. To make their descriptions clearer, we use *patterns*. The command $\langle n, x, y\rangle$ matches the pattern $(n, \mathsf{P}_x, \mathsf{P}_y)$ if the type of x is listed in P_x and the type of y is listed in P_y. In a pattern the symbol $\bullet$ matches any value. As an example, every command matches the pattern $\langle \bullet, \mathbb{OFD}, \mathbb{DFO}\rangle$.

Structural commands change the type of the stored data, while *transient commands* retain it. In other words, commands matching $\langle \bullet, \mathbb{O}, \mathbb{FD}\rangle$, $\langle \bullet, \mathbb{F}, \mathbb{OD}\rangle$ or $\langle \bullet, \mathbb{D}, \mathbb{OF}\rangle$ are structural commands, while those matching $\langle \bullet, \mathbb{O}, \mathbb{O}\rangle$, $\langle \bullet, \mathbb{F}, \mathbb{F}\rangle$ or $\langle \bullet, \mathbb{D}, \mathbb{D}\rangle$ are transient ones. Commands with identical input and output values are *null commands*. Every null command is transient, and transient commands which are not null match the pattern $\langle \bullet, \mathbb{F}, \mathbb{F}\rangle$. If σ is a null command, then $\sigma \sqsubseteq \lambda$ meaning that σ does not change the filesystem (but can break it), and if $\sigma \sqsubseteq \lambda$ then σ is a null command.

Structural commands are further split into *constructors* and *destructors*. Constructor commands upgrade the type from $\mathbb{O}$ to $\mathbb{F}$ to $\mathbb{D}$, while destructors downgrade it. That is, constructors are commands matching $\langle \bullet, \mathbb{O}, \mathbb{FD}\rangle$ or $\langle \bullet, \mathbb{F}, \mathbb{D}\rangle$, while the destructors match $\langle \bullet, \mathbb{D}, \mathbb{FO}\rangle$ or $\langle \bullet, \mathbb{F}, \mathbb{O}\rangle$.

Some command pairs on parent–child nodes can only be executed in a certain order. This notion is captured by the binary relation $\sigma \ll \tau$ with the meaning that σ must precede τ in the *execution order*.

Definition 1 (Execution order)**.** *For a command pair $\sigma, \tau \in \Omega$ the binary relation $\sigma \ll \tau$ holds if the pair matches either $\langle n, \mathbb{DF}, \mathbb{O}\rangle \ll \langle {\uparrow} n, \mathbb{D}, \mathbb{FO}\rangle$ or $\langle {\uparrow} n, \mathbb{OF}, \mathbb{D}\rangle \ll \langle n, \mathbb{O}, \mathbb{FD}\rangle$. An $\ll$-chain is a sequence of commands such that $\sigma_1 \ll \sigma_2 \ll \cdots \ll \sigma_k$. An $\ll$-chain* connects *its first and last element.*

The first case in Definition 1 of the $\ll$ relation corresponds to the requirement that a directory cannot be deleted when its descendants are not empty. The second case requires that before creating a file or directory, the directory holding it should exist. Observe that $\sigma \ll \tau$ implies that both σ and τ are structural commands on parent–child nodes, and either both are constructors or both are destructors. It means that elements of an $\ll$-chain match either the pattern

$$\langle n_1, \mathbb{DF}, \mathbb{O}\rangle \ll \langle n_2, \mathbb{D}, \mathbb{O}\rangle \ll \cdots \ll \langle n_{k-1}, \mathbb{D}, \mathbb{O}\rangle \ll \langle n_k, \mathbb{D}, \mathbb{FO}\rangle$$

where n_{i+1} is the parent of n_i (we move up along a branch), or they match

$$\langle n_1, \mathbb{FO}, \mathbb{D}\rangle \ll \langle n_2, \mathbb{O}, \mathbb{D}\rangle \ll \cdots \ll \langle n_{k-1}, \mathbb{O}, \mathbb{D}\rangle \ll \langle n_k, \mathbb{O}, \mathbb{DF}\rangle,$$

where n_i is the parent of n_{i+1} (we move down). Figure 2 illustrates the structure of $\ll$-chains. The circles represent the nodes on which the commands act: the top, input type row denotes the types of their values before executing the commands, while the bottom, output type row shows the types of their values afterwards. The black arrows point towards the parent nodes, and the read arrows indicate the execution order.

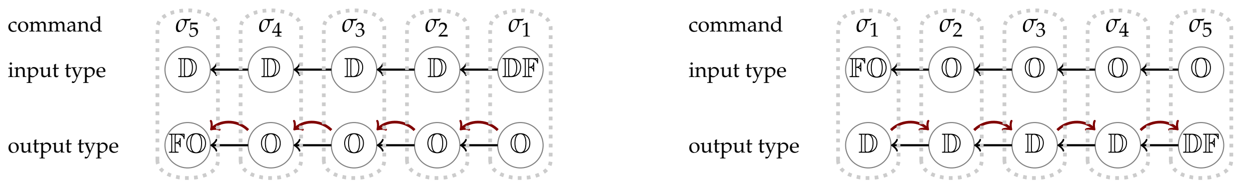
command
input type
output type
σ_5 σ_4 σ_3 σ_2 σ_1
$\mathbb{D}$ $\mathbb{D}$ $\mathbb{D}$ $\mathbb{D}$ $\mathbb{DF}$
$\mathbb{FO}$ $\mathbb{O}$ $\mathbb{O}$ $\mathbb{O}$ $\mathbb{O}$
command
input type
output type
σ_1 σ_2 σ_3 σ_4 σ_5
$\mathbb{FO}$ $\mathbb{O}$ $\mathbb{O}$ $\mathbb{O}$ $\mathbb{O}$
$\mathbb{D}$ $\mathbb{D}$ $\mathbb{D}$ $\mathbb{D}$ $\mathbb{DF}$

Figure 2. Structure of an up (left) and down (right) $\ll$-chain. Black arrows point to the parent node and red arrows show the execution order $\ll$.

4. Canonical Sequences and Sets

Our synchronizer, including the update detector, relies heavily on the algebraic structure of internal command sequences. The theory of this structure has been developed in [12,18]. We provide high-level overviews of the proofs whenever appropriate to make this paper self-contained and also available to non-experts. For full proofs the reader is referred to the original papers.

To understand the semantical properties of command sequences, the first step is to investigate command pairs and determine when they commute, and when they require a special execution order. Proposition 1 covers the case when the commands act on the same node, and Proposition 2 considers command pairs on different nodes.

Proposition 1 ([12], Proposition 4)**.** *Suppose $\sigma, \tau \in \Omega$ are on the same node n. Then one of the following two possibilities hold:*
(a) *$\sigma\tau \sqsubseteq \omega$ for some $\omega \in \Omega$ also on the same node n,*
(b) *$\sigma\tau \equiv \epsilon$, that is, the pair breaks every filesystem.*

Proof. Suppose $\sigma = \langle n, x_1, y_1 \rangle$ and $\tau = \langle n, x_2, y_2 \rangle$. If $y_1 = x_2$ then (a) holds with $\omega = \langle n, x_1, y_2 \rangle$. If $y_1 \neq x_2$ then (b) holds: if σ does not break the filesystem, then it leaves y_1 at node n. Now τ requires x_2 there, but finds a different value, thus breaks the filesystem. $\square$

For two commands we write $\sigma \parallel \tau$ to mean that the nodes they act on are uncomparable, i.e., the nodes are different and neither is an ancestor of the other.

Recall that σ is a null command if it has the same input and output value, that is, it is of the form $\sigma = \langle n, x, x \rangle$.

Proposition 2 ([12], Proposition 5)**.** *Suppose σ and τ are non-null commands on different nodes. Then*
(a) *if $\sigma \parallel \tau$ then $\sigma\tau \equiv \tau\sigma$, and*
(b) *if $\sigma \nparallel \tau$ then $\sigma\tau \not\equiv \epsilon$ if and only if $\sigma \ll \tau$.*

Proof. By checking all possibilities. $\square$

4.1. Canonical Sequences

Informally, canonical sequences are the "clean" versions of non-breaking command sequences. The initial definition is a mixture of syntactical and semantical properties. Proposition 3 defines an algorithm which converts any sequence into a canonical one purification, and Theorem 1 gives a complete syntactical characterization. An important consequence of this characterization is that the semantics of a canonical sequence is determined uniquely by the unordered set of commands it contains. It means that the order of the commands—up to semantical equivalence—can be recovered from their set only.

Definition 2 (Canonical sequences)**.** *A command sequence is canonical if*

(a) *it does not contain null commands, that is, a command of the form $\langle n, x, x\rangle$,*
(b) *no two commands in the sequence are on the same node, and*
(c) *it is non-breaking.*

Conditions (a) and (b) are clearly syntactical, and can be checked by looking at the commands in the sequence. In Theorem 1 below we give a purely syntactical characterization of canonical sequences. Before stating the theorem we show that canonical sequences are the "clean" versions of arbitrary command sequences.

Proposition 3 ([18], Theorem 27)**.** *For every non-breaking command sequence α there is a canonical sequence α^* such that $\alpha \sqsubseteq \alpha^*$, that is, if α does not break Φ, then both α and α^* have the same effect on Φ.*

Proof. Null commands can be deleted from α as they do not change the filesystem (but may break it). Suppose α has two commands on the same node. Choose two which are closest to each other; let these be σ_1 and σ_2, both on node n. The output value of σ_1 must be the same as the input value of σ_2 (as α is non-breaking and the value at node n does not change between σ_1 and σ_2). If σ_1 and σ_2 are not next to each other, then using Proposition 2, either σ_1 can be moved forward by swapping it with the immediately following command (which, by assumption, is on a different node), or σ_2 can be moved backward by swapping it with the immediately preceding command. If none of those swaps can be done, then α would break every filesystem. Eventually σ_1 and σ_2 can be moved next to each other. In this case, however, Proposition 1 implies that $\sigma_1\sigma_2$ can be replaced by a simgle command. □

This proof is constructive and specifies a quadratic algorithm which transforms any non-breaking sequence into a canonical one. To state the purely syntactical characterization of canonical sequences we need further definitions.

Definition 3. *Let α be a command sequence.*
(a) α honors $\ll$, *if for commands $\sigma, \tau \in \alpha$, σ precedes τ whenever $\sigma \ll \tau$.*
(b) α *is* $\ll$-connected, *if for any two commands $\sigma, \tau \in \alpha$, either $\sigma \parallel \tau$, or σ and τ are connected by an $\ll$-chain (see Definition 1) whose elements are in α.*

Theorem 1 (Syntactical characterization of canonical sequences)**.** *A command sequence is canonical if and only if*
(a) *it does not contain null commands,*
(b) *no two commands in the sequence are on the same node,*
(c1) *it honors $\ll$, and*
(c2) *it is $\ll$-connected.*

Proof. Let α be a canonical sequence according to Definition 2. To see that if either (c1) or (c2) fails then α breaks every filesystem, use induction on the length of α. This shows that if α contains commands on nodes n and m which lie on the same branch, then α must contain commands on every node between n and m in the right order.

For the reverse direction assume α satisfies conditions (a), (b), (c1) and (c2). We create a filesystem which α does not break. The values at nodes mentioned in the sequence are set to the input values of the commands. Values at nodes that are ancestors of the ones mentioned in the sequence are set to directories. Other nodes of the filesystem remain empty. This is a valid filesystem, and by property (c2) every command in α has the correct input value. Using property (c1) one can prove that α works on this filesystem. □

4.2. Canonical Sets

An important consequence of Theorem 1 is that the semantics of a canonical sequence is uniquely determined by the commands it contains. This is expressed formally in the following Proposition.

Proposition 4 ([12], Theorem 14)**.** *If the canonical sequences α and β contain the same commands, then they are semantically equivalent, that is, $\alpha\Phi = \beta\Phi$ for every filesystem.*

Proof. In fact, α and β can be transformed into each other using the commutativity rules given in Proposition 2(a), see ([12], Lemma 10). □

Definition 4 (Canonical set)**.** *The set A of internal commands is canonical if*
(a) it does not contains null commands,
(b) no two commands in A are on the same node, and
(c2) the set A is $\lll$-connected, meaning that if two of its commands are on comparable nodes, then they are connected by an $\lll$-chain whose elements are in A.

Commands in a canonical sequence form a canonical set by Theorem 1. In the other direction, a canonical set can be ordered in many ways to become a canonical sequence—the only requirement is (c1) which requires that the order honors the relation $\lll$. This can be achieved by using, e.g., topological sort. Any two such orderings give semantically equivalent sequences, as was proved in Proposition 4. Consequently we can, and will, use canonical sets in places where command sequences are required with the implicit understanding that the commands in it are ordered in an appropriate fashion. For example, we write $A\Phi$ to mean $\alpha\Phi$ where α contains the elements of A in an order which honors $\lll$.

Canonical sets play an essential role in reconciliation (Section 5) and conflict resolution (Section 6). One of their crucial properties is, as we have seen, that their commands can be executed in different orders while preserving their semantics—a variant of the *commutativity principle* of CRDT [20]. Proposition 5 discusses the special case when a subset of the commands should be moved to the beginning of the execution line. Then we define and investigate *clusters* in a canonical set, and observe that the clusters can be freely rearranged, and even intertwined.

Definition 5. *For a canonical set A we write $B \Subset A$ to indicate that B is not only a subset of A but can also be moved to the beginning while keeping the semantics, namely $A \equiv B \circ (A \setminus B)$.*

Proposition 5. *Let A be a canonical set and $B \subseteq A$. Then $B \Subset A$ if and only if $\sigma \in A$, $\tau \in B$ and $\sigma \lll \tau$ imply $\sigma \in B$. If $B \Subset A$ then both B and $A \setminus B$ are canonical.*

Proof. An ordering of A does not break every filesystem if and only if it honors $\lll$, see ([12], Lemma 10). Consequently, if $A \equiv B \circ (A \setminus B)$ then this split must honor $\lll$, which is exactly the stated condition. If this condition holds, then A has an ordering which honors $\lll$ and in which B precedes $A \setminus B$.

For the second part it suffices to check that both B and $A \setminus B$ are $\lll$-connected. However, it is immediate from the condition and from the fact that A is $\lll$-connected. □

The $\lll$ relation connects commands in a canonical set A. The *clusters* of A are the components of this connection graph. All commands in a cluster are constructors, or all commands are destructors, or the cluster has a single element matching $\langle \bullet, \mathbb{F}, \mathbb{F} \rangle$. Accordingly, we call the cluster *constructor*, *destructor*, or *editor*, respectively. Nodes of the commands within a cluster form a connected subtree of the namespace $\mathbb{N}$, and the topmost elements of the subtrees are pairwise incomparable. In a constructor cluster the “being the parent” and the $\lll$ relations coincide, while in a destructor cluster they are the reverse of each other, see Figure 2. As feasible orders of the commands in A must only honor the relation $\lll$, clusters can freely move around. In a constructor cluster a command can be executed only *after* all commands on its ancestors have been executed. In a destructor cluster it is the other way around: a command can be executed only after all commands on its descendants have been executed. Editor commands form single-element clusters, thus they can be executed at any point.

4.3. The Update Detector

Algorithms creating the canonical sets required by the reconciler (see Figure 1) are described in Propositions 6 and 7. The update detector is either *state-based*, in which case it has access to the original and modified filesystems, or it is *command-based*, in which case it has access to the exact sequence of commands used to modify the filesystem. Both update algorithms are highly effective; essentially their runtime is linear in the input size.

Proposition 6 (State-based update detector, ([12], Theorem 19))**.** *Let Φ be the original filesystem and let Φ_1 be the replica. The command set*

$$A^* = \{\langle n, \Phi(n), \Phi_1(n)\rangle : n \in \mathbb{N} \text{ and } \Phi(n) \neq \Phi_1(n)\}$$

is canonical, and $A^*\Phi = \Phi_1$.

Proof. To show that A^* is canonical we need to check that it is $\ll$-connected. By ([18], Lemma 23) if $n \prec m$ and Φ and Φ_1 differ at n and at m, then they must differ at every node between n and m. Moreover, the differences follow one of the the patterns on Figure 2, thus these commands form the required $\ll$-chain. A consequence of this is that A^* does not break Φ, and since after executing the commands every node will have the output value from Φ_1, we know $A^*\Phi = \Phi_1$. □

The set A^* can be generated by performing simultaneous depth-first searches on the visible (non-empty) parts of the filesystems Φ and Φ_1. Such an algorithm runs in constant space, and the running time is linear in the total size of the visible parts of the filesystems.

Figure 3 provides two examples for the result of the state-based update detector. The namespace $\mathbb{N}$ consists of the nodes n_1 to n_9. The original filesystem Φ contains directories at nodes n_1 to n_5 and the empty value at n_6 to n_9. The first replica deletes all directories, setting the empty value everywhere. The second replica stores file contents in n_5 and in the empty nodes. The canonical command set transforming Φ into Φ_1 consists of the five commands $A = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5\}$ where $\sigma_i = \langle n_i, \mathbb{D}, \mathbb{O}\rangle$ corresponding to the five differences between Φ and Φ_1. The second command set also has five commands $B = \{\tau_5, \tau_6, \tau_7, \tau_8, \tau_9\}$ with $\tau_5 = \langle n_5, \mathbb{D}, \mathsf{f}_5\rangle$ and $\tau_i = \langle n_i, \mathbb{O}, \mathsf{f}_i\rangle$ for $i > 5$. The set A has a single execution order honoring $\ll$, namely σ_5, σ_4, σ_3, σ_2, σ_1. Any ordering of the commands in B honors $\ll$, as no two commands in B are $\ll$-related.

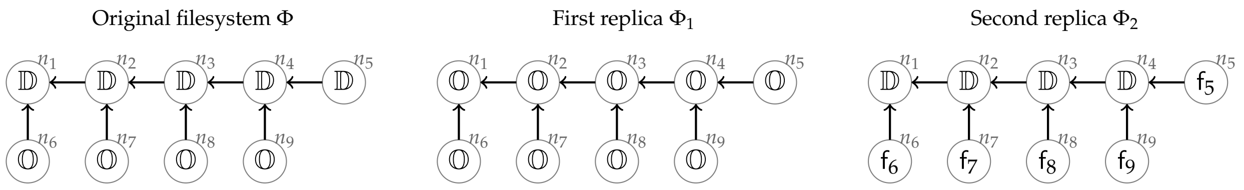

Figure 3. Illustrating the update detector. The nodes are, from left to right, n_1 to n_5 in the top row, and n_6 to n_9 in the bottom row. The first replica deletes all directories, the second replica stores file content f_i at node n_i.

Proposition 7 (Command-based update detector)**.** *Let α be a command sequence that transforms the original filesystem Φ into the replica Φ_1. The following procedure creates a canonical set A^* for which $A^*\Phi = \alpha\Phi = \Phi_1$. Let A^* be empty initially, and iterate over the commands in α from left to right. Let the next command be σ. If there is no command in A^* on the same node as σ, add σ to A^*. If $\sigma^* \in A^*$ and σ are on the same node, then using Proposition 1(a) replace σ^* with the single command corresponding to the composition $\sigma^*\sigma$. Finally, when the iteration ends, delete the null-commands from A^*.*

Proof. It is clear that on each node of the filesystem the sequence α and the set A^* have the same effect. Consequently A^* is equivalent to the command set created in Proposition 6, therefore it is canonical. □

If a hash value of the nodes is also stored, the algorithm can be implented so that its time and space complexity is linear in the length of sequence α.

5. The Reconciler—Synchronizing Two Replicas

The update detector—either state-based or operation-based—passes the canonical sets A and B to the reconciler, see Figure 1. These sets describe, in a straightforward and intuitive way, how the replicas Φ_1 and Φ_2 relate to the original filesystem Φ. As discussed in Section 2, the merged filesystem is also specified by a canonical set M (that of the sequence μ) to be applied to the original Φ. The next definition, which is the formalization of the informal discussion in Section 2 about synchronization, *declares* when the command set M represents an intuitively correct synchronization.

Definition 6 (Merger command set)**.** *The* merger *of the canonical sets A and B is a maximal canonical set $M \subseteq A \cup B$, meaning that no proper superset of M within $A \cup B$ is canonical.*

Note that the merger M is not necessarily unique, and typically many different mergers exist (see the example below). This definition covers every permissible synchronization outcome which satisfies both the *intention-confined effect* R1 (M is a subset of $A \cup B$) and the *aggressive effect preservation* R2 (M is maximal) requirements as discussed in Section 2. When performing the actual file synchronization, one of the mergers should be chosen, either by the system or the user. Section 6 discusses how this choice can be made by successive conflict resolutions.

In this section we prove that every synchronized state defined by a merger M has a clear operational characterization: both replicas can be transformed into the merged state by first rolling back some of the commands executed earlier on this replica, and then applying some commands executed on the other replica. Using the terminology of [12], the canonical sets A and B are called *refluent* if there is filesystem on which both of them work (neither of them breaks). This condition is clearly met when A and B describe two replicas, as both are applied to their common original filesystem.

Theorem 2. *Let A and B be refluent canonical sets and $M \subseteq A \cup B$ be a merger. Then*

(a) $M \sqsupseteq A \circ A_1^{-1} \circ B'$, *where* $A_1 = A \smallsetminus M \subseteq A$ *and* $B' = M \smallsetminus A \subseteq B$,

(b) $M \sqsupseteq B \circ B_1^{-1} \circ A'$ *where* $B_1 = B \smallsetminus M \subseteq B$ *and* $A' = M \smallsetminus B \subseteq A$,

(c) *if both A and B are applicable to a filesystem, then so are M, $A \circ A_1^{-1} \circ B'$ and $B \circ B_1^{-1} \circ A'$.*

In case of synchronizing the replicas $\Phi_1 = A\Phi$ and $\Phi_2 = B\Phi$, let $\Psi = M\Phi$ be (one of) the merged filesystem(s). According to this theorem, we have

$$\Psi = M\Phi = (A \circ A_1^{-1} \circ B')\Phi = B'(A_1^{-1}\Phi_1), \quad \text{and}$$
$$\Psi = M\Phi = (B \circ B_1^{-1} \circ A')\Phi = A'(B_1^{-1}\Phi_2).$$

The first line says that the replica Φ_1 can be transformed into the synchronized filesystem Ψ first by applying A_1^{-1}, that is, rolling back those commands in A which are not in M, and then applying some further commands B' from B. The similar statement for the other replica Φ_2 follows from the second line. This justifies the informal statement $(*)$ in Section 2.

Theorem 2 is illustrated on the filesystems in Figure 3. The canonical sets transforming the original filesystem to the replicas are $A = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5\}$ (where $\sigma_i = \langle n_i, \mathbb{D}, \mathbb{O}\rangle$) and $B = \{\tau_5, \tau_6, \tau_7, \tau_8, \tau_9\}$ (where $\tau_i = \langle n_i, \mathbb{OD}, \mathrm{f}_i\rangle$), respectively. The synchronized states shown on Figure 4 correspond to the mergers $M_1 = \{\sigma_5, \tau_6, \tau_7, \tau_8, \tau_9\}$, $M_2 = \{\sigma_4, \sigma_5, \tau_6, \tau_7, \tau_8\}$, and $M_3 = \{\sigma_2, \sigma_3, \sigma_4, \sigma_5, \tau_6\}$, respectively. Altogether there are six mergers of A and B depending on how many levels of the directory erasures in A are kept. To get Ψ_1 from the replica Φ_1 we first roll back the commands σ_1, σ_2, σ_3 and σ_4 in this order (commands in the set $A \smallsetminus M_1$), which restores the four directories. Then we apply the commands τ_6, τ_7, τ_8, τ_9

from the other command set B to set the file values. Getting Ψ_1 from the second replica requires fewer commands. First roll back τ_5 (the only member of $B \setminus M_1$) restoring the directory at n_5, then apply σ_5 to erase this directory.

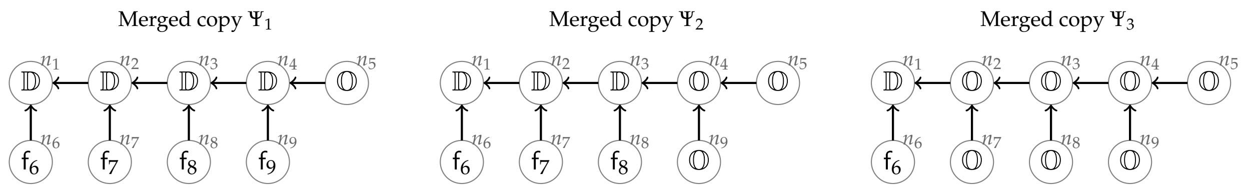

Figure 4. Three possible results of synchronizing the filesystems in Figure 3.

The proof of Theorem 2 uses some intricate properties of canonical sets and mergers which we prove first.

Proposition 8. *Let A and B be refluent canonical sets with $\tau \in A$, $\sigma \in A \cup B$ and $\sigma \ll \tau$. Then $\sigma \in A$.*

Proof. Let σ, τ be on nodes n and m, respectively, and Φ be any filesystem on which both A and B work. The value $\Phi(n)$ is the input value of σ, and $\Phi(m)$ is the input value of τ. Now $\sigma \ll \tau$ matches either the pattern $\langle n, \mathbb{DF}, \mathbb{O}\rangle \ll \langle \uparrow n, \mathbb{D}, \mathbb{FO}\rangle$ or the pattern $\langle \uparrow m, \mathbb{OF}, \mathbb{D}\rangle \ll \langle m, \mathbb{O}, \mathbb{FD}\rangle$, see Definition 1. If A has no command on node n, then executing τ would break Φ as $\Phi(n)$ still has its original value. Thus let $\sigma' \in A$ be on node n. The input value of σ' is the same as that of σ, moreover $\sigma' \ll \tau$ (as A is canonical and σ' and τ are on parent–child nodes). Then the output values of σ' and σ are also equal since they are either both $\mathbb{O}$ or $\mathbb{D}$, thus $\sigma = \sigma'$. □

A similar statement holds for mergers, but the proof is more involved.

Proposition 9. *Let A and B be refluent canonical sets, $M \subseteq A \cup B$ be a merger with $\tau \in M$, $\sigma \in A \cup B$ and $\sigma \ll \tau$. Then $\sigma \in M$.*

Proof. Let σ and τ be a lowest counterexample to the claim, and n and m be their respective nodes. By "lowest" we mean that there is no other counterexample pair σ^*, τ^* where the node of σ^* would be below n.

The argument of the proof of Proposition 8 gives that if M has a command on n (the node of σ), then $\sigma \in M$. Thus σ cannot be added to M only if M already contains a command σ' on a node n' such that n' and n are comparable and σ' and σ cannot be connected by an $\ll$-chain. If m and n' are comparable, then there is an $\ll$-chain in M between τ and σ' as both are in M. As no command in M is on n, the node n cannot be between m and n', but then $\sigma \ll \tau \ll \cdots \ll \sigma'$ is an $\ll$-chain in M connecting σ and σ', a contradiction. (Note that the chain between τ and σ' cannot be in the other direction.)

Therefore m and n' are uncomparable, which means that m and n' are both below n, consequently σ and τ are constructors. Let Φ be any filesystem on which both A and B work. As σ matches $\langle n, \mathbb{OF}, \mathbb{D}\rangle$, $\Phi(n)$ is either empty or is a file. It means that all nodes below n are empty, in particular $\Phi(n') = \mathbb{O}$, and σ' changes this empty value to something else. Now suppose $\sigma' \in A$. As A does not break Φ, it must change the non-directory value at n to a directory, consequently $\sigma \in A$ as well. Then there is an $\ll$-chain between σ and σ' in A starting with σ and ending with $\sigma' \in M$. As σ was chosen to be the lowest counterexample, all elements of this chain are in M, which is a contradiction again. □

Recall from Proposition 5 that $B \Subset A$ if and only if $B \subseteq A$ and there are no $\sigma \ll \tau$ such that $\tau \in B$ and $\sigma \in A \setminus B$.

Proposition 10. *Let A, B be refluent canonical sets, and M be a merger. Then $A \cap M \Subset A$, and $A \cap M \Subset M$.*

Proof. Both statements follow from the combination of Propositions 8 and 9. If $\tau \in A \cap M$ and $\sigma \ll \tau$ with $\sigma \in A \cup B$, then $\sigma \in A \cap M$. □

Claim (a) of Theorem 2 follows immediately from Proposition 10. Let $A_2 = A \cap M$, $A_1 = A \smallsetminus M$, and $B' = M \smallsetminus A$. By Propositions 10 and 5 we have $A = A_2 \circ A_1$, and $M = A_2 \circ B'$. Thus

$$M = A_2 \circ B' \sqsupseteq (A_2 \circ A_1 \circ A_1^{-1}) \circ B' = A \circ A_1^{-1} \circ B',$$

and a similar reasoning gives claim (b).

Recall that the notation $\sigma \parallel \tau$ means that the nodes which the commands σ and τ are acting on are uncomparable (that is, independent). We write $\sigma \parallel B$ to mean that for every $\tau \in B$ we have $\sigma \parallel \tau$, and $A \parallel B$ to mean that for every $\sigma \in A$ we have $\sigma \parallel B$.

Proposition 11. *Suppose that the canonical sets A and B are applicable to the filesystem Φ; moreover, $A \smallsetminus B \parallel B \smallsetminus A$. Then $A \cup B$ is canonical and is applicable to Φ.*

Proof. First we show that $A \cup B$ is canonical. This is so as $A \smallsetminus B$ and $B \smallsetminus A$ do not have commands on comparable or coinciding nodes, thus any two commands in $A \cup B$ on comparable nodes are both in A, or are both in B. Second, $A \cup B$ is applicable to Φ when A and B are disjoint as in this case, by assumption, $A \parallel B$, see also ([18], Lemma 36).

When A and B are not disjoint, let $C = A \cap B$. By Proposition 8, $C \Subset A$, thus $A \equiv C \circ (A \smallsetminus C)$ which means $A\Phi = (A \smallsetminus C)(C\Phi)$. Similarly, $B\Phi = (B \smallsetminus C)(C\Phi)$. Applying the first case of this proof to the disjoint sets $A \smallsetminus C$, $B \smallsetminus C$ and the filesystem $C\Phi$ shows that $(A \cup B) \smallsetminus C$ is applicable to $C\Phi$. Since $C \Subset (A \cup B)$ also holds, we get that $A \cup B$ is applicable to Φ. □

After these preparations claim (c) of Theorem 2 can be proved as follows. Let $A_2 = A \cap M$ and $B_2 = B \cap M$. Then $A_2 \cup B_2 = M$ and $M \smallsetminus A_2 = B_2 \smallsetminus A_2$. Now $A_2 \Subset M$ by Proposition 10, which means that A_2 is canonical and $M \equiv A_2 \circ (M \smallsetminus A_2) = A_2 \circ (B_2 \smallsetminus A_2)$, and similarly $M \equiv B_2 \circ (A_2 \smallsetminus B_2)$. We claim that $A_2 \smallsetminus B_2 \parallel B_2 \smallsetminus A_2$. This is because these sets are disjoint and if σ in the first set and τ in the second set are on comparable nodes, then they are $\ll$-connected (as both are in the canonical set M), thus in any ordering of M which honors $\ll$ they are in a fixed order, but in $A_2 \circ (B_2 \smallsetminus A_2)$ σ comes before τ, and in $B_2 \circ (A_2 \smallsetminus B_2)$ τ comes before σ, which is a contradiction.

Let Φ be a filesystem to which both A and B can be applied. Again by Proposition 10, $A_2 \Subset A$, thus A_2 is also applicable to Φ, and similarly B_2 is applicable to Φ. By Proposition 11, $M = A_2 \cup B_2$ is also applicable to Φ, which is the first claim in Theorem 2 (c). For the other two claims observe that $A_1 = A \smallsetminus M = A \smallsetminus A_2$ and $B' = M \smallsetminus A = M \smallsetminus A_2$. We know $A_2 \Subset A$, thus $A \circ A_1^{-1} = A_2 \circ A_1 \circ A_1^{-1}$, which is applicable to Φ and gives $A_2\Phi$. Since $A_2 \Subset M$ and M is applicable to Φ, $M \smallsetminus A_2 = B'$ is applicable to $A_2\Phi$. All together, $A \circ A_1^{-1} \circ B'$ is applicable to Φ, as claimed. Similar reasoning gives that $B \circ B_1^{-1} \circ A'$ is also applicable to Φ.

6. Synchronization by Conflict Resolution

In the realm of file synchronization a *conflict* is represented by two filesystem commands which can be executed individually, but not together. Resolving the conflict means choosing one of the following actions (see, among others, [9,16]):

- discard both commands,
- discard one command and keep the other (loser/winner paradigm), or
- replace one or both commands by a new one.

Only the second alternative satisfies both the *intention-confined effect* principle R1 (use only commands issued by the user), and the *aggressive effect preservation* principle R2 (carry over as many operations as possible). Our synchronizer uses this loser/winner paradigm to solve all conflicts. Resolving a conflict might create new ones, or automatically solve—or make irrelevant—others. Thus after each step the set of conflicts has to be regenerated, or at least revalidated/reviewed. Termination of the synchronization process is not automatic and has to be proved, see [16]. Fortunately, in our case no new conflicts are generated, and resolved and disappearing conflicts can be identified easily. As each step eliminates at least one conflict, the synchronizer clearly stops after finitely many steps.

6.1. Theoretical Foundation

In this subsection we fix the refluent canonical sets A and B and the merger M which the synchronization is supposed to reach, but is not necessarily known at the start. First we show that synchronization can be reduced to the case when A and B are disjoint (though they might contain different commands on the same node). Then we define when a command pair represents a *conflict*, and show that this notion has the expected properties: a merger M does not contain conflicting pairs, and if there are no conflicts, then M is the disjoint union of A and B. The main novelty of our conflict resolution algorithm is that instead using the winner to build up the merger, it only discards the losing command from future considerations. In other words, the algorithm thins out the sets A and B until only the merger M remains.

Proposition 12. *Let A, B be refluent canonical sets, and M be a merger. Then (a) $A \cap B \Subset A$ and $A \cap B \Subset B$, (b) $A \cap B \subseteq M$, and (c) $A \cap B \Subset M$.*

Proof. Claim (a) follows from Proposition 8 since if $\tau \in A \cap B$, $\sigma \in A \cup B$ and $\sigma \ll \tau$, then $\sigma \in A \cap B$. From this and from (b) claim (c) follows, too.

To prove (b) it suffices to check that $M \cup (A \cap B)$ is canonical. First, all commands in this union are on different nodes. Both M and $A \cap B$ are canonical, so if their union were not, then there are commands $\sigma \in A \cap B$ and $\sigma' \in M \setminus (A \cap B)$ on comparable nodes which are not connected by an $\ll$-chain in $M \cup (A \cap B)$. Assume $\sigma' \in A$. There is a chain between σ and σ' in A. Since $A \cap B \Subset A$, we know this chain has the direction $\sigma \ll \cdots \ll \sigma'$. From this and because $\sigma' \in M$, we know all members of this chain are in M based on Proposition 9, which is a contradiction. □

Let $C = A \cap B$ and let Φ be a filesystem which shows that A and B are refluent, that is, both A and B work on Φ. Let $A' = A \setminus C$, $B' = B \setminus C$, and $M' = M \setminus C$; observe that the command sets A' and B' are disjoint. By Proposition 12 C is canonical and can be moved to the front of A and B and M; moreover A' and B' and M' are also canonical. Now A' and B' are refluent (both are applicable to $C\Phi$), and M' is a merger for A' and B'. The converse of the last statement is also true: if M^* is any merger of A' and B', then $C \cup M^*$ is a merger of A and B. Consequently it suffices to find the merger M' of the disjoint command sets A' and B', and then adding the intersection $A \cap B$ to M'. For this reason, from this point on we assume that A and B share no commands.

Definition 7. *Let A and B be fixed disjoint, refluent canonical sets. The pair (σ, τ) is a* conflict *if $\sigma \in A$, $\tau \in B$ and $\sigma \not\parallel \tau$, that is, the nodes of σ and τ are comparable (including when they coincide).*

The *conflict graph* is the bipartite graph with the disjoint command sets A and B as the two classes, and an edge between $\sigma \in A$ and $\tau \in B$ if they are in conflict. Figure 5a shows the conflict graph of the synchronization problem depicted on Figure 3. As an example, command σ_5 is connected to τ_5 only, as both σ_5 and τ_5 are on node n_5, while n_5 is independent of the nodes $n_6, \ldots, n_9$ on which the other commands in B are.

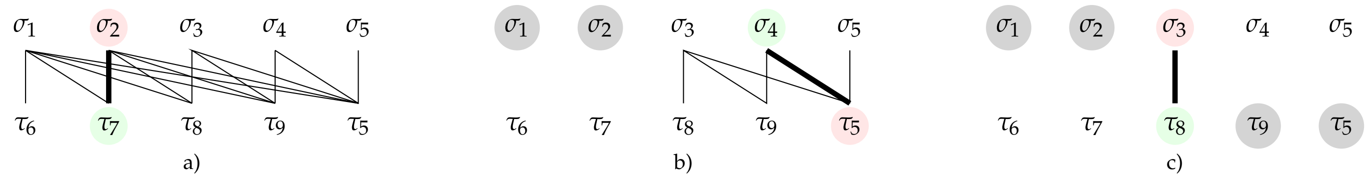

Figure 5. (**a**) The conflict graph of the synchronization problem of Figure 3. The conflict (σ_2, τ_7) is resolved with the winner τ_7. (**b**) The conflict graph after resolving (σ_2, τ_7); commands σ_1 and σ_2 are deleted. The next conflict we resolve is (σ_4, τ_5). (**c**) The conflict graph after resolving (σ_4, τ_5) with the winner σ_4; commands τ_9, τ_5 are deleted. The final conflict graph contains the commands σ_4, σ_5, τ_6, τ_7, τ_8, and no edges.

Proposition 13. *It is impossible that both commands of the conflict (σ, τ) are in M.*

Proof. The meanings of the symbols are as in Definition 7. Suppose by contradiction that both are in M. As they are on comparable nodes, there is an $\ll$-chain in M between σ and τ. Consequently there are two consecutive commands in this chain so that one is in A and the other is in B, say $\sigma' \ll \tau'$, $\sigma' \in A$ and $\tau' \in B$. However, Proposition 8 says that in this case $\sigma' \in B$, contradicting that A and B are disjoint. □

Proposition 14. *Suppose (σ, τ) is a conflict, $\sigma' \in A$ and $\sigma \ll \sigma'$. Then (σ', τ) is also a conflict.*

Proof. Let σ, σ' be on nodes n and n', respectively. If n' is the parent of n, then the node of τ is comparable to n', thus we are done. Therefore n is the parent of n', σ and σ' are constructors, and σ matches $\langle n, \mathbb{OF}, \mathbb{D} \rangle$. If the node of τ is not below n, then we are done. If it is, then the input type of τ is $\mathbb{O}$ (as in the original filesystem Φ, $\Phi(n)$ is not a directory, therefore every node below it is empty). For the same reason the command set B must contain a command on node n (otherwise when executing τ, Φ still has the original content at n, and then τ breaks Φ). This command has the same input type as σ, must create a directory, thus it is equal to σ, contradicting that A and B are disjoint. □

By symmetry, the same claim holds when A and B are swapped. Fix $\sigma \in A$, and split B into B_σ^{ok} and B_σ^{conf} as follows:

$$\begin{aligned} B_\sigma^{\mathsf{ok}} &= \{\tau \in B : \ (\sigma, \tau) \textit{ is not } \text{a conflict}\} = \{\tau \in B : \sigma \parallel \tau\}, \\ B_\sigma^{\mathsf{conf}} &= \{\tau \in B : \ (\sigma, \tau) \textit{ is } \text{a conflict}\} = \{\tau \in B : \sigma \not\parallel \tau\}. \end{aligned}$$

We know $B_\sigma^{\mathsf{ok}} \Subset B$ by Propositions 5 and 14, which means that if Φ is a filesystem B does not break, then neither does B_σ^{ok}. Moreover, if $\sigma \ll \sigma'$, then $B_\sigma^{\mathsf{conf}} \subseteq B_{\sigma'}^{\mathsf{conf}}$. In particular, if $B_{\sigma'}^{\mathsf{conf}}$ is empty, then so is B_σ^{conf}. Next we show that if there are no conflicts, then the synchronization is done; however, we need a stronger statement which will be used to justify the correctness of the synchronization process.

Proposition 15. *Suppose B_σ^{conf} is empty, that is, no $\tau \in B$ is in conflict with $\sigma \in A$. Then $\sigma \in M$.*

Proof. It is enough to show that $M \cup \{\sigma\}$ is canonical. Since $\sigma \parallel B$, it has at most one command on every node. It can only be non-canonical if there is a $\tau \in M$ on a related node that cannot be connected to σ using an $\ll$-chain from the commands in M. This τ must be in A, which has a chain between σ and τ. If this chain goes in the direction $\sigma \ll \cdots \ll \tau$, then $\sigma \in M$ by Proposition 9. Otherwise we know $\tau \ll \cdots \ll \sigma$, and based on the remark above, B_ω^{conf} is empty for all members of this chain and so are independent of B. This means the chain can be added to M, which is a contradiction. □

Corollary 1 (Termination)**.** *If there are no conflicts, then the merger M is $A \cup B$.*

Proof. By Proposition 15 every command in A is in M, and similarly $B \subseteq M$. As M is a subset of $A \cup B$, they must be equal. □

6.2. The Synchronization Algorithm

The input of the synchronization algorithm is (A^o, B^o), which is a pair of refluent canonical command sets. The goal is to find, using successive conflict resolution steps, a merger command set M that fits Definition 6. In addition, we aim to show that our algorithm can produce *any* of the mergers allowed by the definition, and only valid mergers. To achieve this, the algorithm accepts a pre-set merger M^o as an optional argument. Whenever the algorithm needs to make an otherwise arbitrary choice, M^o is used to choose one of the possibilities. While choosing any of the possibilities will generate a valid merger, guiding the choice will guarantee that M^o will be generated.

According to the discussion in Section 6.1, the synchronization algorithm has two main components. The first one, called Algorithm 1, reduces the general problem to the special case when the input command sets have no common elements. Properties of the merger set proved in Proposition 12 provide the foundation of the reduction. The second component, called Algorithm 2, assumes that its input command sets A^o and B^o are disjoint so that it can rely on the intrinsic properties of the conflict graph when generating the merger M. The two algorithms are visalized in Figures 6 and 7, respectively. The detailed description is followed by the proofs of the correctness and a short discussion on time and space complexity.

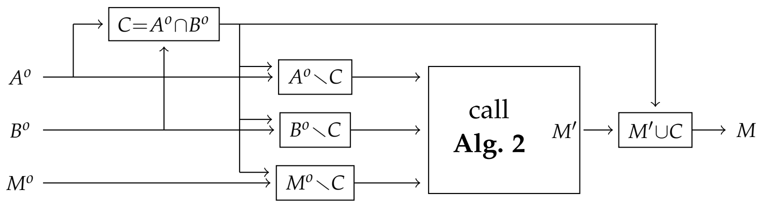

Figure 6. Outline of Algorithm 1. Given the input (A^o, B^o, M^o), extract the common part C of A^o and B^o, and call Algorithm 2 with the reduced sets. Finally, add C to the returned M' to generate the output M.

Algorithm 1 (Synchronization in the general case)**.** Let $C = A^o \cap B^o$. Execute Algorithm 2 with the command sets $A^o \smallsetminus C$ and $B^o \smallsetminus C$. If M^o is specified, then it must satisfy $C \subseteq M^o$; in this case pass $M^o \smallsetminus C$ as the optional input to Algorithm 2. When Algorithm 2 returns M', return $M = C \cup M'$.

Correctness of Algorithm 1. As discussed after the proof of Proposition 12, $A^o \smallsetminus C$ and $B^o \smallsetminus C$ are refluent canonical sets, and C is an initial segment of every merger of A^o and B^o. Moreover, if M^o is a merger of A^o and B^o, then $M^o \smallsetminus C$ is a merger of $A^o \smallsetminus C$ and $B^o \smallsetminus C$. From here the correctness of the algorithm follows easily. □

Complexity of Algorithm 1. Apart from the time and space requirements of Algorithm 2, the five operations here require computing the union, intersection and difference of sets. Using hashing techniques, these operations can be performed both in time and space linear in the input size. □

Inputs of the main Algorithm 2 are the *disjoint* refluent canonical sets A^o and B^o, and the optional target merger M^o of A^o and B^o. The goal is to produce a merger M. If M^o is given, then, at the end, M and M^o must be equal.

The algorithm maintains two command sets A and B, initially set to A^o and B^o. During execution these sets satisfy the following invariants:

1. $A \Subset A^o$, $B \Subset B^o$, consequently A and B are refluent canonical subsets of A^o and B^o;
2. all mergers of A and B are mergers of A^o and B^o;
3. if M^o is specified, then M^o is a merger of A and B.

In each iteration one or more commands are deleted either from A or from B until only the commands of the merger remain.

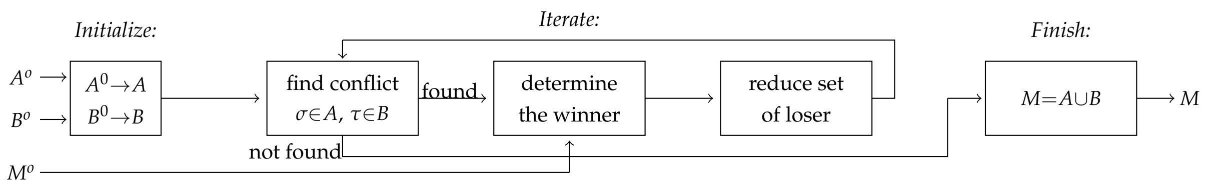

Figure 7. Outline of Algorithm 2. The *Initialize* step sets A and B. The *Iterate* step is executed until no more conflicts are found. Conflicts are resolved using the hint from the additional input M^o, then the command sets A and B are reduced. If no conflict remains, the merger $M = A \cup B$ is returned by the *Finalize* step.

Algorithm 2 (Synchronization of disjunct sets)**.** *Initialize:* Set $A^o \to A$ and $B^o \to B$.

Iterate: Choose $\sigma \in A$ and $\tau \in B$ such that (σ, τ) is a conflict. If no such pair exists, go to Finish. If M^o is given, choose a conflict so that either σ or τ is in M^o.

Conflict resolution: Out of σ and τ, choose the *winner* and the *loser* commands. If M^o is given, let the winner be the one in M^o. If σ is the winner, delete all commands from B which are in conflict with σ (including τ). If τ is the winner, delete all commands from A which are in conflict with τ. Go back to Iterate.

Finish: Return $A \cup B$ as the merger.

Execution of Algorithm 2 is illustrated in Figure 5. The initial command sets are $A = \{\sigma_1, \dots, \sigma_5\}$ and $B = \{\tau_6, \dots, \tau_5\}$. In the first iteration $\tau_7 \in B$ is the winner; therefore, commands in A which are in conflict with τ_7 (σ_1 and σ_2) are removed from A. In the next iteration the winner is $\sigma_4 \in A$, and τ_9 and τ_5 are removed from B. In the last iteration σ_3 is removed from A, thus the remaining sets are $A = \{\sigma_4, \sigma_5\}$ and $B = \{\tau_6, \tau_7, \tau_8\}$. The returned merger is $A \cup B$.

Correctness of Algorithm 2. After the initialization step all three invariants hold. The algorithm terminates as in each iteration the total number of remaining conflicts strictly decreases. When it terminates, the returned set $A \cup B = M$ is a merger of the original command sets A^o and B^o. This is because at that point there are no conflicts between A and B, thus, by Corollary 1, M is a merger of A and B—in fact, the only one—, and then the second invariant guarantees that M is also a merger of A^o and B^o. If M^o was submitted to the algorithm, then the third invariant ensures $M = M^o$. Thus we only need to show that the iteration preserves all three invariants. Suppose there is a conflict (σ, τ) between A and B, and the winner of the conflict is $\sigma \in A$.

Invariant 1. Commands in conflict with σ are deleted from B, thus B is replaced by the set

$$B_\sigma^{\mathsf{ok}} = \{\tau' \in B : \ (\sigma, \tau') \text{ is not a conflict}\}.$$

Proposition 14 (see the discussion following the proof) and the invariant imply $B_\sigma^{\mathsf{ok}} \Subset B \Subset B^o$. Consequently the invariant $B \Subset B^o$ remains true.

Invariant 2. We know that if a merger of A and B contains σ, then it cannot contain any of the discarded elements of B (see Proposition 13). We also know that all mergers of A and B_σ^{ok} contain σ by Proposition 15 as it is not in conflict with any other command. Consequently the set of mergers of A and B_σ^{ok} consists of those mergers of A and B that contain σ, and the invariant continues to hold.

Invariant 3. If M^o was specified for the algorithm, then we chose the winner accordingly, and so $\sigma \in M^o$. A consequence of the discussion above is that M^o remains part of the set of mergers of A and B_σ^{ok}.

Finally we show that if M^o is specified, then one of the conflicting commands can be chosen from M^o. By Proposition 15 if $\sigma \in A$ is not in conflict with any command in B, then

σ is in every merger of A and B; in particular, it is in M^o by the third invariant. Thus, if no conflicting commands remain in M^o, but there is a conflict, then M^o would be a proper subset of the merger returned by the algorithm, which is absurd. □

Complexity of Algorithm 2. Both the *Initialization* and *Finish* steps can be done in time and space that is linear in the input size. Updating the sets A and B in the *Iteration* step discards some of their elements which will never be used again. It means the total time used by the updates is bounded by the number of elements in the original input set $A^o \cup B^o$. Finding conflicting pairs in constant time, however, requires building a structure which stores the bipartite conflict graph. This requires time and space which is proportional to the number of nodes (commands in A^o and B^o) plus the number of conflicting pairs. It means that Algorithm 2 can be implemented with both time and space complexity that is linear in the input size plus the total number of conflicts. While the latter number can be expected to be comparable to the input size, in the worst case it can be quadratic, meaning that this implementation guarantees at most quadratic time and space complexity. □

We conjecture that a more efficient implementation of Algorithm 2 can be created using the structural properties of the conflict graph. Some of those properties are discussed in the next Section.

6.3. Structural Properties of the Conflict Graph

We call a conflict between two commands *structural* if the out types of the commands are different, and a *content conflict* otherwise. An example for a content conflict is when both replicas replace the same directory with different file contents. The best resolution of a content conflict cannot necessarily be achieved by the above winner/loser paradigm, as it uses filesystem-level information only. Fortunately, content conflicts are represented in the conflict graph as isolated edges. It means that all of these conflicts must (and can) be resolved independently of all the other conflicts. We deem it good practice to resolve all content conflicts first.

If (σ, τ) is a conflicting pair and $\sigma' \in A$ is on a node which is above the node of σ, then (σ', τ) is also a conflicting pair. Proposition 14 implies the same when $\sigma \ll \sigma'$. Consequently all elements of a *constructor* cluster (see Section 4.2) of A are in conflict with exactly the same elements of B, and so the whole constructor cluster can be replaced by a single graph node, thus reducing the graph size. Figure 5 shows the typical structure of conflict graphs. Elements of the destructor chain $\sigma_5 \ll \sigma_4 \ll \cdots \ll \sigma_1$ are connected to more and more conflicting commands (this is so as the corresponding nodes go upwards). Choosing τ_5 as a winner automatically resolves all conflicts by wiping out all σ_i.

Automatic conflict-based synchronizers typically give precedence to constructors in conflicts between a constructor and a destructor command [10,16], arguing that it is easier to remove some content again than recreating it. While this is good guidance most of the time, one has to be aware that in the current framework, replacing a directory node with some file content is a *destructor*. Confronted with the conflicting constructor command that creates a file under the same directory, marking this latter command as the winner is not self-evident.

7. Conclusions

We have presented a complete theoretical investigation of filesystem synchronization. Our definition of a filesystem is arguably simplistic, but it captures the important features of real-word implementations. The changes between the original and the updated filesystems are encoded using a sequence of virtual filesystem commands. The synchronizer receives two sequences corresponding to the two replicas to be synchronized, and produces a third sequence which describes the synchronized, or merged state. Definition 6 gives an intuitively correct and convincing description of when the resulting sequence leads to a meaningful merged filesystem. In Section 5 we proved an operational characterization of the possible merged filesystems. The main result of Section 6 is a conflict-based non-deterministic synchronization algorithm which creates exactly those merger sequences which are allowed by Definition 6. The simplicity of this algorithm and the complexity of proving its correctness was a surprise to us. Its success underlines the power of the algebraic framework of file synchronization developed in [18].

Multiple questions and possible extensions of the current work remain which are the subject of future research. Some of them are discussed in the next subsections.

7.1. Synchronizing More Than Two Replicas

The first natural question is to extend the synchronization algorithm from two to three or more replicas. An encouraging observation is that if several canonical sets are pairwise refluent (for any pair there is a filesystem on which both of them work), then they are *jointly refluent*, meaning that there is a single filesystem on which all of them work. The natural extension of Definition 6 for three canonical sets A, B and C is to define their merger as a maximal canonical set $M \subseteq A \cup B \cup C$. This definition has the additional advantage that it is symmetric in the filesystems, and does not give precedence to any of them over the others. We can prove that such a merger also satisfies an operational characterization similar to the one stated in Theorem 2. It seems that such a three-way merger can be generated first by merging A and B to M, and then merging M and C. Is this claim true?

Another question is whether the general task of finding a three-way merger can be reduced to the case when A, B and C are pairwise disjoint. Finally, at least in the case when the sets A, B, C are pairwise disjoint, whether the following variant of the Iteration step of Algorithm 2 generates a 3-way merger, and if yes, whether it generates all possibilities:

Iterate: Pick a winner command σ from $A \cup B \cup C$, and discard those commands from the other two sets which are in conflict with σ.

7.2. Efficiency of the Algorithms

Algorithm 1 can be implemented with a runtime that increases linearly with the size of its input, so it is efficient. The trivial running time estimation of Algorithm 2 is proportional to the product of the sizes of the command sets A and B—assuming that the "above" relation in the node structure $\mathbb{N}$ can be checked in constant time, as generating the conflict graph takes that much time. Section 6.3 discusses some possible improvements by exploiting the structure of the conflict graph. As to whether the running time of the algorithm can be improved significantly, or it is substantially quadratic, our conjecture is that the algorithm can be made sub-quadratic.

7.3. Attributes

Attributes typically store metainformation about a node, such as different timestamps (e.g., creation date, last modification or last access) and permissions (e.g., read and write access), and whether the metainformation refers to the node only or to the whole subtree below it. Attributes are set and modified either automatically, or by special filesystem commands. In our filesystem model attributes can be best modeled by storing them as node content. For file nodes it means that the attributes are added to the actual file content, which means that any change to the attributes can be handled as a change in content. Therefore, file attributes pose no special problems and can be integrated seamlessly into our model. Handling directory attributes, however, is a challenging open problem. A general algebraic framework of filesystems allowing content in directory (and even in empty) nodes was developed in [12] at the expense of more complicated notions and theorems. Unfortunately, with this extension many of the Propositions in Sections 5 and 6 on which the correctness of our method relies do not remain true. To illustrate the problem, suppose A changes some attribute from "private" to "public" at some directory node n, and creates a file under n; while B, under the impression that the directory is still private, creates another file under it. Then the file creation command from B cannot be unconditionally moved to the merging sequence.

One possible solution is to consider and resolve all attribute conflicts (including when A and B modify attributes at the same node differently) as the first stage of the synchronization process, and then proceed according to the algorithms described in this paper. This approach, however, is not necessarily compatible with our declarative Definition 6 of what a synchronized state is. It is not even clear whether all possible mergers satisfying this definition are intuitively correct or not if directory attributes are taken into account.

7.4. Links

From the user's perspective, a *link* between the nodes n and n' is a promise, or a commitment, that the filesystem at and below n is exactly the same as at and below n'. Executing a command below n automatically executes the same command on the corresponding node below n'. A link system must be *loopless* meaning that any node has only finitely many other nodes *equivalent* to it. In particular, n and n' must be uncomparable nodes.

If the original filesystem Φ contains links, but the users do not have tools or permissions to manipulate them, the following synchronization method works. The update detector, knowing the link structure, unfolds it, and replaces each user command by the collection of all "hidden" or "implicit" commands on the equivalent nodes. After creating the canonical command set A, it marks the commands which are on equivalent nodes. For the merger M to be applicable to Φ it must also contain, with each command in M, all of its equivalents. This can be achieved by the following modification to the *Iterate* part of Algorithm 2: if σ is chosen as a winner, then all commands equivalent to it will be winners as well.

It is an open problem to introduce filesystem commands which manipulate the link structure and can be incorporated into our synchronization process.

Author Contributions: All authors contributed equally to this word. All authors have read and agreed to the published version of the manuscript.

Funding: The research of the second author (L.C.) has been supported by the GACR project number 19-04579S and by the Lendulet Program of the HAS, which is thankfully acknowledged.

Data Availability Statement: Not Applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CRDT	Conflict-free Replicated Data Type
CSCW	Computer Supported Collaborative Work
OT	Operational Transformation
$\mathbb{O}, \mathbb{F}, \mathbb{D}$	empty, file, and directory content
Φ, Ψ	filesystem

References

1. Preguiça, N. Conflict-free replicated data types: An overview. *arXiv* **2018**, arXiv:1806.10254.
2. Shapiro, M.; Preguiça, N.; Baquero, C.; Zawirski, M. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 386–400.
3. Sun, C.; Jia, X.; Zhang, Y.; Yang, Y.; Chen, D. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput. Hum. Interact.* **1998**, *5*, 63–108. [CrossRef]
4. Sun, C.; Ellis, C. Operational transformation in real-time group editors. In Proceedings of the Computer Supported Cooperative Work Seattle, Washington, DC, USA, 14–18 November 1998; pp. 59–68.
5. Ellis, C.A.; Gibbs, S.J. Concurrency control in groupware systems. In Proceedings of the SIGMOID conference on Management of Data, Portland, OR, USA, 31 May–2 June 1989; pp. 399–407.
6. Day-Richter, J. What's Different about the New Google Docs: Making Collaboration Fast. Available online: https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html (accessed on 9 September 2022).

7. Nicolaescu, P.; Jahns, K.; Derntl, M.; Klamma, R. Near real-time peer-to-peer shared editing on extensible data types. In Proceedings of the 19th International Conference on Supporting Group Work, Sanibel Island, FL, USA, 13–16 November 2016; pp. 39–49. [CrossRef]
8. Klophaus, R. Riak Core: Building distributed applications without shared state. In Proceedings of the SIGPLAN Commercial Users of Functional Programming (CUFP '10), Baltimore, MD, USA, 1–2 October 2010; Article 14. [CrossRef]
9. Ng, A.; Sun, C. Operational transformation for real-time synchronization of shared workspace in cloud storage. In Proceedings of the 19th International Conference on Supporting Group Work, Sanibel Island, FL, USA, 13–16 November 2016; pp. 61–70.
10. Tao, V.; Shapiro, M.; Rancurel, V. Merging semantics for conflict updates in geo-distributed file systems. In Proceedings of the 15th ACM International Systems and Storage Conference, Haifa, Israel, 13–15 June 2015.
11. Balasubramaniam, S.; Pierce, B.C. What is a File Synchronizer? In Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking, Dallas, TX, USA, 25–30 October 1998; pp. 98–108.
12. Csirmaz, E.P.; Csirmaz, L. Algebra of Data Reconciliation. *Stud. Sci. Math. Hung.* **2022**. [CrossRef]
13. Kermarrec, A.; Rowstron, A.; Shapiro, M.; Druschel, P. The IceCube approach to the reconciliation of divergent replicas. In Proceedings of the ACM Symposium on principles of distributed computing 2001, Newport, RI, USA, 25–27 June 2001; pp. 210–218.
14. Martins, V.; Pacitti, E.; Valduriez, P. Distributed semantic reconciliation of replicated data. In Proceedings of the CDUR, Paris, France, 2–4 November 2005; pp. 48–53.
15. Pierce, B.C.; Vouillon, J. What's in Unison? A Formal Specification and Reference Implementation of a File Synchronizer. *U. of Pennsylvania Technical Reports (CIS) 40*. 2004. Available online: http://repository.upenn.edu/cis_reports/40 (accessed on 9 September 2022).
16. Shekow, M. Syncpal: A Simple and Iterative Reconciliation Algorithm for File Synchronizers. Ph.D. Thesis, Aachen University, Aachen, Germany, 2019.
17. Terry, D.B.; Theimer, M.M.; Petersen, K.; Demers, A.J.; Spreitzer, M.J.; Hauser, C.H. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, Mountain, CO, USA, 3–6 December 1995; pp. 172–182.
18. Csirmaz, E.P. Algebraic File Synchronization: Adequacy and Completeness. *arXiv* **2016**, arXiv:1601.01736.
19. Antkiewicz, M.; Czarnecki, K. Design space of heterogeneous synchronization. In Proceedings of the GTTSE 2007: International Summerschool on Generative and Transformational Techniques in Software Engineering 2007, Braga, Portugal, 2–7 July 2007; Springer: Berlin/Heidelberg, Germany, 2008; pp. 3–46.
20. Preguiça, N.; Marques, J.M.; Shapiro, M.; Letia, M. A commutative replicated data type for cooperative editing. In Proceedings of the 2009 29th International Conference on Distributed Computing Systems, Montreal, QC, Canada, 23–26 June 2009; pp. 395–403.